A Type System for High Performance Communication and Computation

Greg Eisenhauer*, Matthew Wolf*†, Hasan Abbasi† Scott Klasky† and Karsten Schwan*
*College of Computing
Georgia Institute of Technology,
Atlanta, Georgia 30332–0250
{eisen,mwolf,schwan}@cc.gatech.edu
†Oak Ridge National Laboratory
{habbasi,klasky}@ornl.gov

***Abstract*—The manner in which data is represented, accessed and transmitted has an affect upon the efficiency of any computing system. In the domain of high performance computing, traditional frameworks like MPI have relied upon a relatively static type system with a high degree of *a priori* knowledge shared among the participants. However, modern scientific computing is increasingly distributed and dynamic, requiring the ability to dynamically create multi-platform workflows, to move processing to data, and to perform both *in situ* and streaming data analysis. Traditional approaches to data type description and communication in middleware, which typically either require *a priori* agreement on data types, or resort to highly inefficient representations like XML, are insufficient for the new domain of dynamic science. This paper describes a different approach, using FFS, a middleware library that implements efficient manipulation of application-level data. FFS provides for highly efficient binary data communication, XML-like examination of unknown data, and both third-party and *in situ* data processing via dynamic code generation. All of these capabilities are fully dynamic at run-time, without requiring *a priori* agreements or knowledge of the exact form of the data being communicated or analyzed.**

I. Introduction

Scientific computing has always been a data-intensive domain, but with high performance networks, limitations in the scalability of single machines and the natural physical distribution of science teams, it increasingly has turned to distributed computing solutions. Application environments have evolved from tightly-coupled components running in a single location to collaborating components shared amongst diverse underlying computational centers. For example, HPC components include those responsible for data analysis, temporary and long term storage, data visualization, data preprocessing or staging for input or output, and others. Linking such components in ways that are flexible and dynamic but do not compromise performance requires careful attention to the manner in which data is represented, accessed and transmitted. Adding important new capabilities such as moving computation to data also requires careful thought about how such mobile computation can be represented and implemented.

Traditional high performance systems have generally relied upon binary data representations and a high degree of *a priori* knowledge between parties exchanging data. However, in many D3 (data-intensive, distributed and dynamic) science situations, requiring agreement on basic representation is overly restrictive as *a priori* knowledge may be imperfect or non-existent, or the linked components may simply use different representations of semantically similar data. Straightforward alternatives, such as using XML/XSLT[1], name/value schemes or object-based marshaling[2] have significantly higher overhead than traditional approaches[3] and are not readily suitable for data-intensive HPC applications.

This paper describes a different approach, using FFS (Fast Flexible Serialization), an approach to data representation, processing and marshaling that preserves the performance of traditional approaches while relaxing the requirement of *a priori* knowledge and providing complex run-time flexibility. FFS provides for highly efficient binary data communication, XML-like examination of unknown data, and both third-party and *in situ* data processing via dynamic code generation. All of these capabilities are fully dynamic at run-time, without requiring *a priori* agreements or knowledge of the exact form of the data being communicated or analyzed. The efficient binary representation of the data object is also realized in the output format for the FFSfile, an advanced object store-like file format that supports attribute based querying for data objects while providing transparent support for machine portability and *in situ* processing.

FFS addresses the challenges posed by distributed computing requirements without compromising the performance characteristics important for the data intensive nature of the target applications. The FFS approach to handling data has been utilized in a number of scientific middleware solutions by our group and collaborators. We will focus here on two specific instances in this paper, a code coupling middleware with a channelized pub/sub interface [4] based on the EVPath event transport library [5] and a high performance, scalable data extraction library called JITStager[6]. The details of each of these systems are discussed elsewhere, but both of them are completely consistent with the D3 vision discussed earlier.

EVPath is an event processing architecture that supports high performance data streaming in overlay networks with internal data processing. FFS enables EVPath to be a fully type-aware middleware, marshaling and unmarshaling application-level messages as required for its operations and using it

to enact mobile application-level processing in its overlay network. On the other hand, JITStager addresses the performance needs of massively scalable parallel applications for data extraction to staging nodes. In addition to the high performance data extraction mechanism, JITStager uses the self describing nature of the FFS data format in conjunction with the CoD dynamic compiler to allow low overhead data marshaling and data transformations for I/O pipelines and *in situ* data processing.

This paper examines the communication and computational needs of D3 science applications and derives a set of requirements. We then describe the attributes of FFS that address these needs and demonstrate how FFS forms the foundation of the two aforementioned D3 middlewares. We evaluate the FFS framework both independently in a series of microbenchmarks as well as in the context of these middleware libraries.

II. Motivating Applications & Requirements

A. Distributed Materials Design

For the illustrative purposes of this paper, we consider one particular type of D3 science application, although the core architecture has proven itself to be relevant in a number of other scenarios. Specifically, we want to focus on the capabilities and requirements for large, design-space optimization problems like design-of-materials [7]. These are multi-physics, multi-collaborator, multi-site, and multi-objective projects that require dynamic messaging infrastructures quite different from the highly structured ones used in traditional simulation domains (i.e. MPI). In particular, these sorts of code bases have to deal with very dynamic process models and flexible, higher-level data models that must be maintained in between executions of any individual component.

To be more concrete, consider designing a material to meet a particular tensile strength requirement, such as designing an ideal composite material to be used in a turbine blade. This is not a simple combinatorial design space exploration using a single code – the problem requires utilizing everything from atomic level-codes to study the composite microstructure for crack propagation, all the way up to macroscopic finite element simulations of the entire blade structure. Depending on the type of composite, there may be many intermediate models that best represent the physics at the mesoscale structural level.

From a computer science perspective, this implies that the overall execution cannot be captured in a single, unified execution model. Tasks to explore novel configurations and crack propagation will be dynamically spawned in order to evaluate particular issues as the higher-level codes evolve. Similarly, given the composition of the very different time and length scales implied by this sort of system, there must be the capability to both convert between data representations easily, while simultaneously enabling higher-level experimental concepts (like total error) to be maintained even if the individual component does not understand it.

One approach to solving such problems is to adhere to a componentized framework [8], where each component must have a well-defined interface. Techniques can then be used to automatically generate binding code at compile time to enable the type conversion and inter-process communications [9], [10]. For our approach, however, we wish to enable a more flexible execution environment that is more solidly D3 in nature – adding collaborative and multi-site capabilities to the list of multi-* requirements for the code. The "heavy lifting" can take place in traditional HPC centers, but the overall environment must include the ability to stream such data to secondary sites for further analysis, integration, interpretation, and even visualization and collaborative discussion. Under such requirements, the alternative approaches that require compile-time knowledge of the complete composition, let alone fixed process allocation knowledge (as MPI would require), would clearly not be sufficient.

B. Computer Science Implications

Systems that seek to implement end-to-end real-time processing of scientific data, particularly those that must support distributed processing and dynamic connectivity, have some common needs that drive requirements for their underlying system of data representation and manipulation. As described in the introduction, we focus here on the specific issues associated with the flexible formatting requirements, although we also describe some of the overlapping research we have done in messaging, discovery, and active transport layers that have been enabled by the core FFS capabilities.

It is clear that in an HPC environment, data marshaling and unmarshaling costs must be minimized, avoiding copying data if at all possible. Generally this indicates that the underlying type system for communication must closely mirror the type system of the underlying application, utilizing direct memory access on marshaling and perhaps allowing direct use of incoming data buffers. Additionally, the inclusion of some kind of mobile code system is useful in order to support *in situ* analysis (moving code to data). This support can significantly improve scalability because of the potential for the mobile code to perform filtering, data reduction, and/or feature extraction in order to avoid wasting scarce bandwidth transmitting data that will be later discarded or reduced. Further, mobile code can be invaluable in linking diverse components in data-intensive dynamic workflows because of its ability to transform data representations to "make them right" for the eventual consumer.

In order to maximize the utility of these capabilities, the mobile code must be both efficient and flexible, running at speeds near that of compiled code while operating on data whose exact representation may not be known until run-time. Additionally, because of the dynamic nature of these systems, the ability to recognize and gain information about incoming data types (that is, to access associated metadata or utilize reflective properties) is invaluable. Finally, because of the data-intensive nature of these HPC applications, the inclusion of type meta-information must not have a significant effect upon communication performance.

We can summarize the requirements for a type system for D3 science applications as follows:

- low marshaling/unmarshaling costs, so as not to interfere with delivering communication bandwidth and latency near what the underlying network allows,
- the ability to recognize and gain information about incoming data types that may be unknown (i.e. associated metadata or reflection),
- the ability to enact run-time specified functions to data types that are potentially unknown until run-time, with an efficiency near native code.

III. FFS Design and Architecture

In order to meet the requirements above, FFS is designed around a simple data representation and dynamic code generation for function enactment. A key design decision is that all messages are tagged with a 'metadata token' through which a complete description of the structure of the data can retrieved. This approach allows for the transport of fully-described data without significant overhead on every message. It also allows for the recognition of incoming message types and the customization of handler functions where appropriate.

A. Data Representation

Some systems used for the transport and processing of typed data differentiate between the representation used 'on-the-wire' and that used for in-memory for processing. XML, for example, relies on a strictly textual representation for transport and storage, but XSLT is a tree-transformation language designed to manipulate Document Object Model (DOM) trees. Because of its intended application domain in high-performance computing, FFS uses an in-memory data representation designed to be simple, but still rich enough to support the classes of data that those applications need to communicate. In particular, the FFS data model is roughly equivalent to a C-style structure or a Fortran Derived type, supplemented with some additional semantics that allow for variably-sized arrays and other pointer-based structures. FFS accommodates nested structures and recursively-defined data structures such as lists, trees and graphs.

In order for FFS to transport an application-defined data structure, the structure must be completely described to FFS. The process for doing this is somewhat similar to constructing a derived datatype in MPI. Figure 1 shows a very simple C-style structure that might be used in a distributed monitoring scenario.

One important features to note here is that the FMField description captures necessary information about application-level structures, including field names, types, sizes and offsets from the start of the structure. MPI derived data types are declared similarly, with the exceptions that FFS also associates a name, rather than just a type, with each field, and FFS separates the root data type (e.g. "`integer`") from the size of field. The `struct_list` associates a name ("message") with the structure described by the field list and provides FFS with the structure's size (which due to alignment requirements may be different than simply the largest field offset and size). In this simple case, the `format_list` contains just a single element

```
typedef struct {
    int cpu_load;
    double mem_use;
    double net_use;
} Msg, *MsgP;

FMField Msg_field_list[] = {
    {"load", "integer", sizeof(int), FMOffset(MsgP, load)},
    {"mem_use", "float", sizeof(double), FMOffset(MsgP, mem_use)},
    {"net_use", "float", sizeof(double), FMOffset(MsgP, net_use)},
    {NULL, NULL, 0, 0},
}
static FMStructDescRec struct_list[] = {
    {"message", Msg_field_list, sizeof(Msg), NULL},
    {NULL, NULL, 0, NULL}
};
```

Fig. 1. Sample FFS data structure declaration in C. The `FMOffset()` macro calculates the offset of the field within the structure.

(and the NULL end record). In more complex situations, the format list includes the transitive closure of all structures that make up the data structure.

The set of built-in FFS supported types includes "integer", "unsigned integer", "float" and "string". The latter is a C-style NULL-terminated string. "Double" is included as an alias for "float", while "enumeration" and "boolean" largely function as integer types. For nested structures, the type field can be the name of another structure from the `struct_list`. For example, Figure 2 shows an FFS declaration for a nested structure that includes a "r3vector" type. Field types may also specify statically-sized arrays (e.g. "`integer[100]`"), or dynamically-sized arrays where the actual size is given by another integer-typed variable in the structure (e.g. "`float[dimen1][dimen2]`").

It is worth noting that while the structure descriptions in Figures 1 and 2 are static, deriving from a structure that is declared at compile-time, FFS does not rely upon (and is not aware of) that aspect of the descriptions. In the context of the requirements laid out in Section II, static FFS structure descriptions match the context of statically-compiled D3 applications and are used to 1) describe the structure of application data to be transmitted, or 2) to describe the structure the incoming data must have in order to be delivered to the receiving application. *FFS does not require that these descriptions be the same.* Rather, FFS' role is to be the arbiter that enables a transformation between the two (or more) potentially different data types used by the communicating applications.

While static FFS structure descriptions generally make sense when interfacing with statically typed languages like C and Fortran, FFS does not rely upon them, and internal type representations and operations are fully dynamic. This flexibility allows FFS to perform processing on application-level types in situations where the computation might not have full *a priori* knowledge of the exact data layout at the point of processing. These situations are common in both *in situ* and in-transit processing and are more fully described in Section III-E.

```
typedef struct {
    double x;
    double y;
    double z;
} r3vector, *r3vectorP;

typedef struct {
    int type;
    double mass;
    r3vector velocity;
} Particle, *PartP;

FMField r3vector_fields[] = {
    {"x", "float", sizeof(double), FMOffset(r3vectorP, x)},
    {"y", "float", sizeof(double), FMOffset(r3vectorP, y)},
    {"z", "float", sizeof(double), FMOffset(r3vectorP, z)},
    {NULL, NULL, 0, 0},
}

FMField particle_fields[] = {
    {"type", "integer", sizeof(int), FMOffset(PartP, type)},
    {"mass", "float", sizeof(double), FMOffset(PartP, mass)},
    {"velocity", "r3vector", sizeof(r3vector), FMOffset(PartP, velocity)},
    {NULL, NULL, 0, 0},
}

static FMStructDescRec struct_list[] = {
    {"particle", particle_fields, sizeof(Particle), NULL},
    {"r3vector", r3vector_fields, sizeof(r3vector), NULL},
    {NULL, NULL, 0, NULL}
};
```

Fig. 2. A more complex data structure declaration in C.

B. Data Transport

Many communication systems that support typed messages require the sender to convert data to a standard format, such as a pre-defined 'network' representation, before transmission. This can result in an unnecessary up/down translation (e.g. in a homogeneous transfer where the native and network byte order differ). Sometimes negotiation between a sender and receiver can establish a common machine-level representation to be used. FFS takes a different approach. As in its predecessor, PBIO [11], FFS represents data on the network largely as it is in application memory, except that pointers are converted into integer offsets in the encoded message. In particular, FFS does not compact 'holes' in structures that might result from alignment requirements, choosing instead to transport those holes along with the 'real' data so that message data can largely be gathered directly from application memory to avoid additional buffering. One significant advantage of this approach is that FFS need not always copy messages before transport. Instead, FFS can assemble a list of buffer/length values that, when assembled, constitute an encoded message. This list can be passed directly to gather-capable transport routines like `writev()`, largely eliminating data copies from the sending side of the transport. Figure 3 shows a byte-wise representation of the structure and marshaling of the data described by Figure 1.

C. The receiving side

As mentioned above, the same form of structure description that is used by the sender to describe the data being sent is used by the receiver to describe the data that is expected or required. For direct communication between static languages,

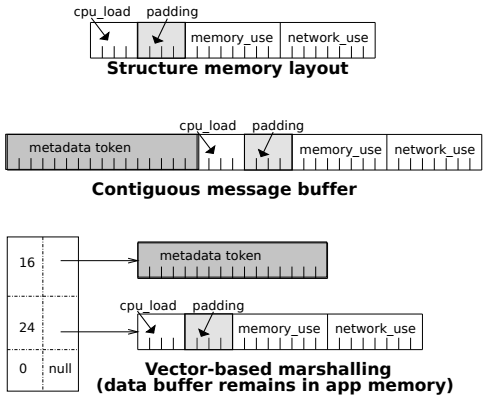

Fig. 3. A simple data structure with in-memory representation, marshaled as a contiguous buffer or marshaled as a vector of buffers (data remains in-place). *N.B. the 'pad' is unused memory which might be added by the compiler to maintain field alignment in the structure.*

the data must exactly match the structure required by compiled code, and the receiving-side structure descriptions are fixed and rigid (particularly with respect to field sizes and offsets). Where FFS is performing in-transit processing, the receiving-side descriptions are more malleable because no statically compiled code must access the data.

The FFS approach of using the sender's native data representation as a wire format can be viewed as an optimistic approach. FFS explicitly does not rely upon *a priori* or negotiated knowledge relating the sender and receiver of data, but in the simplest scenario of a homogeneous transfer between applications that have agreement between the data formats provided and expected, the FFS approach results in zero data copies by either sender or receiver. The received data is directly usable by the application in the network buffer without further manipulation.

However, if the sending and receiving applications differ in their data representation, either because of architecture differences or program-level disagreements on the structure of data, then FFS must *convert* between the on-the-wire data format and a native memory structure. Conversions are also necessary if the message contains embedded pointers (such as for strings or variably-sized arrays) that are represented as message offsets on the wire.

To minimize data copies, FFS differentiates between the circumstance where it the conversion from wire to native format can take place directly in the network buffer and where it cannot. The former case, *in place conversion* allows the minimum data handling and memory use in such situations as an exchange between homogeneous architectures with embedded pointers (only the pointers need to be modified). It is sometimes possible in heterogeneous exchanges, such as an exchange between big-endian and little-endian architectures. However if the conversion between wire and receiving structure formats requires moving data to different offsets, FFS often resorts to using a second buffer for the destination structure.

Because FFS does not rely upon *a priori* knowledge, the exact nature of the conversion between wire and native format is not known until runtime and can vary for each sender/receiver pair based upon their underlying architectures and structure layouts. The conversion itself could be implemented in a table-driven way. Since as the efficiency of the conversion impacts the performance of the data exchange, FFS optimizes it as much as possible by using dynamic code generation to create customized conversion routines that map each incoming data format to a native memory structure. The details of the generated code are similar to that of PBIO as described in [11]. FFS extends the capabilities of PBIO in its ability to handle recursively-defined types such as linked lists, trees and graphs, and its ability to generate conversion routines on architectures that were not supported by PBIO. FFS's dynamic code generation capabilities are based on the Georgia Tech DILL package that provides a virtual RISC instruction set. The DILL system generates native machine code directly into the application's memory without reference to an external compiler. In its use in FFS, DILL provides no register allocation or optimization, but directly translates virtual RISC instructions into instructions for the underlying native architecture in a single pass.

The performance of the marshaling and unmarshaling routines is critical to satisfying the requirements of high-performance applications and plays a role in the evaluation of network performance in real world HPC applications [12], [4].

D. Format Identification and Handling

As mentioned above, marshaled FFS messages do not contain the full metadata required to interpret them, but are instead accompanied by a 'metadata token' (or *format ID*) that can be used to retrieve the full metadata (or *format description*). The format description contains everything required to decode a marshaled message, including the name, type, size and offset of every field (essentially everything specified in the field and format lists of Figs 1 and 2), and also architecture-specific details, such as the byte-order, floating point format, and address size.

Generally speaking, the receiver of FFS-encoded data requires the full format description in order to utilize the incoming message. FFS caches format descriptions whenever possible, so the acquisition of a format description is a startup cost that incurred at most once per message type/process for each pair of communicating FFS clients. Because one-time costs tend to be unimportant in data-intensive communications, we have focused our efforts on making sure that format descriptions are available when needed, rather than trying to optimize their delivery.

The standard FFS mechanism for disseminating format information is to use a third-party *format server* running at a well-known address. When a format description (such as those of Figs 1 and 2) is processed, FFS marshals the description into an internal canonical representation and creates a format ID whose main component is an 8-byte hash of that representation. Then this format ID / format description pair (format pair) is registered with the third party server. When a message is received by an FFS process, FFS first checks to see if it has the format pair in its format cache. If it does not, it sends the format ID to the third-party server and requests the matching format description. Once the description is received, it is entered into the format cache where it will be found in subsequent searches. In practice, the number of format lookups will be fewer than the number of communicating pairs because many communicating peers will be of the same architecture. So, for example, if two 64-bit x86 machines processes operating with identical structure declarations send data to a third process, that process will only do one format lookup because its two peers will produce identical format pairs.

Alternatively, FFS allows higher-level middleware to directly handle format distribution. For example, EVPath tracks the FFS formats of its application-level messages and, before it sends message encoded with FFS format X to peer Y, it sends the format pair associated with format X to that peer (once only, per peer). Peer Y, upon receiving a format pair, enters the pair into its format cache where it will be found in subsequent searches. This technique is also employed in special circumstances, such as in the work described in [12] where EVPath was used on tens of thousands of nodes on the Cray "Jaguar" supercomputer at Oak Ridge National Labs. In order not to have tens of thousands of nodes simultaneously hitting the format server to register their (identical) format descriptions, EVPath transported them on the wire to collection sites.

From an efficiency point of view, the demand-based approach employed with an external format server minimizes format lookups because many peers will end up with identical format pairs, but the requirement to have a well-known server shared among all communicating pairs might be onerous. EVPath's approach of directly sending the format description before any message of that format on every connection incurs a larger cost in terms of one-time communication overhead. However, both costs are quickly amortized over the repeated data exchanges that are typical in D3 science applications.

E. Mobile Functions and the CoD Language

A critical issue in the implementation of FFS and its ability to meet the requirements laid out in Section II is the nature of its mobile code support. There are several possible approaches to this problem, including:

- severely restricting mobile functions to preselected values or to boolean operators,
- relying on pre-generated shared object files, or
- using interpreted code.

Having a relatively restricted mobile code language, such as one limited to combinations of boolean operators, is the approach chosen in other event-oriented middleware systems, such as the CORBA Notification Services and in Siena [13]. This approach facilitates efficient interpretation, but the restricted language may not be able to express the full range

```
{
    if ((input.trade_price < 75.5) ||
         (input.trade_price > 78.5)) {
        return 1; /* pass event over output link */
    }
    return 0; /* discard event */
}
```

Fig. 4. A specialization filter that passes only stock trades outside a pre-defined range.

of conditions useful to an application, thus limiting its applicability. To avoid this limitation, it is desirable to express F in the form of a more general programming language. One might consider supplying F in the form of a shared object file that could be dynamically linked into the process of the stone that required it. Using shared objects allows F to be a general function, but requires F to be available as a native object file everywhere it is required. This is relatively easy in a homogeneous system, but it becomes increasingly difficult as heterogeneity is introduced, particularly if type safety is to be maintained.

In order to avoid problems with heterogeneity, one might supply F in an interpreted language, such as a TCL function or Java byte code. This would allow general functions and alleviate the difficulties with heterogeneity, but it would impact efficiency. Because of our intended application in the area of high performance computing, and because many useful filter and transformation functions are quite simple, D3 applications need a different approach that would maintain the highest efficiency.

FFS builds upon its use of dynamic code generation for FFS conversion functions, preserving the expressiveness of a general programming language and the efficiency of shared objects while retaining the generality of interpreted languages. Functionality such as *in situ* and in-transit processing are expressed in CoD (C On Demand), a subset of a general procedural language, and dynamic code generation is used to create a native version of these functions on the host where the function must be executed. CoD is currently a subset of C, supporting the standard C operators and control flow statements.

Like the DCG used for FFS format conversions, CoD's dynamic code generation capabilities are based on the Georgia Tech DILL package, to which are added a lexer, parser, semanticizer, and code generator. The CoD/Dill system is a library-based compilation system that generates native machine code directly into the application's memory without reference to an external compiler, assembler or linker. Only minimal optimizations and basic register allocation are performed, but the resulting native code still significantly outperforms interpreted approaches.

Some *in situ* or in-transit filters may be quite simple, such as the example in Figure 4. Applied in the context of a stock trading example, this filter passes on trade information only when the stock is trading outside of a specified range. This filter function requires 330 microseconds to generate on a 2Ghz x86-64, comprises 40 instructions and executes in less than a microsecond. Transformational functions (those that

```
{
    int i, j;
    double sum = 0.0;
    for(i = 0; i<37; i= i+1) {
        for(j = 0; j<253; j=j+1) {
            sum = sum + input.wind_velocity[j][i];
        }
    }
    output.average_velocity = sum / (37 * 253);
    return 1;
}
```

Fig. 5. A specialization filter that computes the average of an input array and passes the average to its output.

are not restricted to simply passing or suppressing the flow of data) extend this functionality in a straightforward way. For example, the CoD function defined in Figure 5 averages wind data generated by an atmospheric simulation application, thereby reducing the amount of data to be transmitted by nearly four orders of magnitude.

IV. FFS Use in High Performance Middleware

The capability for data introspection supplemented by the just-in-time transformations of CoD allow the FFS middleware package a great deal of flexibility for D3 applications. In particular, it allows placement and data conversion decisions in higher-level middleware to be delayed to run-time, so that a more effective overall strategy can be computed. This capability allows one to support dynamic cases where the strategy might change in the middle of execution, subject to whatever problem-based performance metrics the higher-level data management protocols establish.

In the following subsections, we detail specific examples of how this just-in-time derivation and introspection capabilities of FFS are applied to the targeted materials science application described earlier. Performance evaluation numbers and analysis are provided for each of the scenarios.

A. Data Morphing for Streaming Clients

In large multi-collaborator environments, and even in single collaborator instances, it is not uncommon to have separate code analysis and code coupling instances that have a forced interface standard. It is also quite common that for some new version of a code, that messaging interface will need to be upgraded, modified, etc. In supporting our more dynamic mode of connection for the materials design applications, we could not rely on traditional compile-time verification of signature. Instead, we constructed a scientific messaging middleware capable of recognizing type and/or message incongruence on the fly. Utilizing FFS (and through it CoD) to implement the appropriate data morphing routines to "make the data right" on the receiver's side allows for the minimum of complexity in development costs.

Figure 6 shows an instance of a simple transformation that a developer would need to register with the system if he or she decided that the current interface needed to be extended by adding the average and standard deviation of an array of data. This interface function would only need to be registered once, and *all* messages published to the original interface would

```
{
    ... /* initializations */
    for (i=0; i<input.num_recs; i++) {
      sum = sum + input.array[i];
      sum2 = sum2 +   \
           (input.array[i])*(input.array[i]);
    }
    output.avg = sum/input.num_recs ;
    output.std_dev = sqrt(sum2/input.num_recs \
                        - output.avg*output.avg);
    ... /* any other value setting operations */
    return 1;
    }
}
```

Fig. 6. An example of a Data Morphing transformation for a case where a format has been extended to include the average and standard deviation of an array of values. Details such as initialization and copyout of the array values are elided for simplicity.

immediately become available as inputs to any codes with the upgrades. The details of the messaging and coordination layer can be seen in [14], [4], [12], [6].

As a concrete instance of such a data morphing requirement from the materials design application, we have implemented and measured the impact of a transformation dictated by one of the analysis codes utilized for understanding molecular dynamics output data. As a first phase of interpreting the raw atomic coordinates output by the lower-level code, a connectivity graph is generated to represent the nearest neighbor bonds between atoms. Subsequent analysis routines consume this connectivity graph as an input in order to determine things like crystalline faces, location or existence of cracks, etc.

From a data layout perspective, this graph can be represented in several different forms, and different components wish to consume it in the most natural format for them. In particular, one natural data representation is as a list of ordered integer indices where the first index is strictly less than the second (i.e. 1,34 means the atom at index #1 is adjacent to atom #34). We refer to this as the "pair" data structure. This representation is natural for any analysis that iterates over the edges in a consecutive fashion.

However, some analyses iterate over the adjacency list of each atom in turn. With the pair data structure, analyzing atoms with very high index values may require iterating through a large number of irrelevant data points to determine what atoms are adjacent. In the extreme case of the last index value, one would need to consume the entire list of edges in order to select those few that are its neighbors. Here, an "adjacency" format is preferable, where the data is represented as an array of structs, each of which contains a list of those atoms adjacent to the atom with the equivalent index value. (i.e. atom_adjacency[0] contains the adjlist for atom #0.)

We have implemented exactly this conversion routine in both C for static compilation and in CoD for dynamic insertion. In Table I, we demonstrate the timing for data morphing between codes publishing and subscribing using the two different interfaces. For two input data sizes (1013 and 77728 atoms), we give the execution time for a compiled conversion and a dynamically code generated conversion. We also give execution times that demonstrate our DCG routine's ability to operate on marshaled data (that is, data straight off the wire that has not undergone unmarshaling).

TABLE I
DATA MORPHING TIMES BETWEEN 'PAIR' AND 'ADJACENCY' DATA STRUCTURES. "HOMOGENEOUS" REFERENCE PLATFORM IS A 64-BIT LITTLEENDIAN MACHINE. TIMES ARE MILLISECONDS/MESSAGE.

input data # of atoms	Compiled	Generated	Marshaled data		
			Homo.	32-bit	Bigend.
1013	0.546ms	0.608ms	0.608ms	0.608ms	0.612ms
77728	40.1ms	49.3ms	49.5ms	49.5ms	49.7ms

The data in the table demonstrate that our DCG data morphing routines are only 10-20% slower than the statically compiled routine. This is not a significant penalty to pay in return for the ability to relocate the computation to any in-transit site and without regard to *a priori* knowledge!

Further, FFS is able to operate nearly as efficiently on marshaled data as on unmarshaled data. In this example, our processing requires reading all the incoming data, so avoiding an unmarshal operation is not so significant. But in situations where only a small amount of the incoming data needs to be examined in order to perform routing or filtering, the ability to operate on data in marshaled form offers significant performance improvements. Dynamic code generation for this 34-line routine required roughly 0.64 milliseconds and generated 400 x86-64 machine instructions.

B. In Situ *Data Analysis*

Building an *in situ* data morphing capability for D3 science can be accomplished by coupling FFS with advanced network scheduling, as described and evaluated in detail in [6]. One key issue is that the marshaling and unmarshaling operations described above do not interfere with standard network operations. Measurements in [15] show that FFS-based marshalling imposes only a 15% overhead at message sizes of 10Kb and essentially no overhead at sizes of 100Kb.

Using dynamic code generation can also enable significant reductions in the data transfer times for data generated in an *in situ* analysis function. For example, in a molecular dynamics analysis pipeline, the atomic output is analyzed for the appearance of a plastic deformation event. Once such an event is detected the molecular data is then processed with a different analysis function to understand the crystalline structure near the deformation. Instead of building this support in the application the data can be analyzed using a dynamically generated code fragment. This allows the pipeline to only output data to the next stage in the pipeline when the deformation event is detected. Table II shows the performance benefits of taking this dynamic approach.

V. FUTURE AND RELATED WORK IN I/O

Although this paper has so far highlighted the network-centric nature of FFS, the techniques proposed have an equal applicability to storage-driven solutions. Existing solutions include those that adopt an intermediate format with a flexible type descriptor, like XDR[16]. The advantage is that data on disk or on the wire will always be a consistent format, but it does mean that conversions may be required even in single host situations.

TABLE II
DATA TRANSFER TIME REDUCTION BY MOVING THE ANALYSIS OPERATION TO THE DATA

	No in situ reduction	In situ analysis
Transfer time (ms)	79.82	1.73
Marshalling Time (ms)	5.99	1.66

Another common approach is to utilize a fixed format, like netCDF [17] or HDF5 [18]. Although these formats have a fixed, hierarchical view of data, the flexibility of metadata specification allows for a considerable variety of utilization for storing and describing complex scientific data. In contrast, the FFS approach offers more flexibility in construction of data structures while maintaining the ability to describe hierarchical structures. However, the focus of FFS on data structure description currently makes the creation of formats based on the sorts of higher-level data model concepts more tedious for scientific end users.

As part of on-going and future work, we have been exploring the integration of FFS with the ADIOS data file format BP [19] to address some of these data model concerns. ADIOS is a componentized library interface that allows a user to plug in different I/O transports (MPI I/O, Posix, JITStager, or others). ADIOS currently can utilize an external XML specification to bind I/O writes to a particular global format descriptor which will then be reflected in the resulting BP format. FFS has already been utilized inside ADIOS as an intermediate format representation when moving data into a staging area.

We see a synergistic opportunity by utilizing the support for recursive and tree-based data structures from FFS to extend BP's current set of supported applications while using ADIOS/BP's higher-level format descriptors to provide better end-user access to the data morphing capabilities of FFS. Indeed, the ADIOS XML construction allows end users to extend the data description easily at run-time to specify data morphing requirements. This future work will result in a much easier transition path for end users to access FFS's active data morphing and management capabilities.

VI. CONCLUSION

The ability to examine, manipulate and operate upon user-level data, both in-transit and *in situ*, and without restrictive assumptions about *a priori* knowledge, is critical to a holistic approach to end-to-end processing and management of scientific data. The JITStager and EVPath systems that we used to drive the discussion and evaluation of FFS both seek to meet different higher-level needs of D3 science applications, yet they shared the need for a basic ability to manipulate application data.

The capabilities and measurements above demonstrate the viability and utility of FFS-based communication in D3 science scenarios. We have shown that FFS' low-overhead marshaling and unmarshaling and efficient dynamic code generation allow significant flexibility in communication, including supporting *in situ* and in-transit processing, without compromising performance. All of these capabilities are fully dynamic at run-time, without requiring *a priori* agreements or knowledge of the exact form of the data being communicated or analyzed.

While no single paradigm or middleware is likely to meet the needs of every application under the D3 science umbrella, FFS provides a powerful set of enabling capabilities for supporting higher-level D3 services.

REFERENCES

[1] W. W. W. Consortium, "Xsl transformations (xslt)," 1999, http://www.w3.org/TR/xslt.

[2] P. Eugster, "Type-based publish/subscribe: Concepts and experiences," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 1, p. 6, 2007.

[3] A. Haeberlen, J. Liedtke, Y. Park, L. Reuther, and V. Uhlig, "Stub-code performance is becoming important," in *WIESS'00: Proceedings of the 1st conference on Industrial Experiences with Systems Software*. Berkeley, CA, USA: USENIX Association, 2000, pp. 4–4.

[4] M. Wolf, H. Abbasi, B. Collins, D. Spain, and K. Schwan, "Service Augmentation for High End Interactive Data Services," in *Proceedings of Cluster 2005*, 2005.

[5] G. Eisenhauer, M. Wolf, H. Abbasi, and K. Schwan, "High performance event communications," in *Attainging High Performance Communications: A Vertical Approach*, A. Gavrilovska, Ed. CRC Press, 2010.

[6] H. Abbasi, G. Eisenhauer, S. Klasky, K. Schwan, and M. Wolf, "Just in time: Adding value to the io pipelines of high performance applications with jitstaging," in *Proceedings of the International Symposium on High Performance Distributed Computing 2011 (HPDC'11)*, June 2011.

[7] D. McDowell, J. Panchal, H.-J. Choi, C. Seepersad, J. Allen, and F. Mistree, *Integrated Design of Multiscale, Multifunctional Materials and Products*. Elsevier, 2009.

[8] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, and J. A. Kohl, "The cca core specification in a distributed memory spmd framework," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 5, pp. 323–345, 2002.

[9] T. Epperly, S. R. Kohn, and G. Kumfert, "Component technology for high-performance scientific simulation software," in *IFIP TC2/WG2.5 Publications*, 2000, pp. 69–86.

[10] Y. Gil, V. Ratnakar, E. Deelman, G. Mehta, and J. Kim, "Wings for pegasus: Creating large-scale scientific applications using semantic representations of computational workflows," in *National Conference on Artificial Intelligence*, 2007, pp. 1767–1774.

[11] G. Eisenhauer and L. K. Daley, "Fast heterogenous binary data interchange," in *Proc. of the Heterogeneous Computing Workshop (HCW2000)*, May 3-5 2000.

[12] H. Abbasi, M. Wolf, F. Zheng, G. Eisenhauer, S. Klasky, and K. Schwan, "Scalable data staging services for petascale applications," in *Proceedings of the International Symposium on High Performance Distributed Computing 2009 (HPDC'09)*, June 2009.

[13] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Challenges for distributed event services: Scalability vs. expressiveness," in *Proc. of Engineering Distributed Objects (EDO '99), ICSE 99 Workshop*, May 1999.

[14] H. Abbasi, M. Wolf, and K. Schwan, "Live data workspace: A flexible, dynamic and extensible platform for petascale applications," in *Proceedings of Cluster 2007*, 2007.

[15] G. Eisenhauer, M. Wolf, H. Abbasi, and K. Schwan, "Event-based systems: opportunities and challenges at exascale," in *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '09. New York, NY, USA: ACM, 2009, pp. 2:1–2:10. [Online]. Available: http://doi.acm.org/10.1145/1619258.1619261

[16] R. Srinivasan, "Xdr: External data representation standard," *Network Working Group*, 1995.

[17] R. Rew and G. Davis, "Netcdf: an interface for scientific data access," *IEEE Computer Graphics and Applications*, vol. 10, pp. 76–82, 1990.

[18] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the hdf5 technology suite and its applications," in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, 2011, pp. 36–47.

[19] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible io and integration for scientific codes through the adaptable io system (adios)," in *Challenges of Large Applications in Distributed Environments*, 2008, pp. 15–24.