Collaborative Research: Programming Models and Storage System for High Performance Computation with Many-Core Processors

Jack B. Dennis, Guang R. Gao and Vivek Sarkar

CAPSL Technical Memo 088
May 11th, 2009

MIT, University of Delaware and Rice University
dennis@csail.mit.edu, ggao@capsl.udel.edu and vsarkar {at} rice.edu

Future generation HEC architectures and systems built using many-core chips will pose unprecedented challenges for users. A major source of these challenges is the storage system since the available memory and bandwidth per processor core is starting to decline at an alarming rate, with the rapid increase in the number of cores per chip. Data-intensive applications that require large data sets and/or high input/output (I/O) bandwidth will be especially vulnerable to these trends.

Unlike previous generations of hardware evolution, this shift in the hardware road-map will have a profound impact on HEC software. Historically, the storage architecture of an HEC system has been constrained to a large degree by the file system interfaces in the underlying Operating System (OS). The basic design of file systems has been largely unchanged for multiple decades, and will no longer suffice as the foundational storage model for many-core HEC systems. Instead, it becomes necessary to explore new approaches to storage systems that reduce OS overhead, exploit multiple levels of storage hierarchy and focus on new techniques for efficient utilization of bandwidth and reduction of average access times across these levels.

The specific focus of this proposal is on exploring a new storage model based on *write-once tree structures*, which is radically different from traditional flat files. The basic unit of data transfer is a *chunk*, which is intended to be much smaller than data transfer sizes used in traditional storage systems. The write-once property simplifies the memory model for the storage system and obviates the need for complicated consistency protocols. We will explore three programming models for users of the storage system, all of which can inter-operate through shared persistent data: 1) a *declarative programming model* in which any data structure can be directly made persistent in our storage system, with no programmer intervention, 2) a *strongly-typed imperative programming model* in which a type system extension will be used to enforce a separation between data structures that can be directly made persistent and those that cannot, and 3) a *weakly-typed runtime interface* that enables C programs to access our storage system. A *compiler* with a high-level *data flow intermediate representation* and lower-level *parallel intermediate representation* will provide the necessary *code generation* support, and a *runtime system* will implements interfaces to the storage system used by compiler-generated code and by the weakly-typed runtime interface. Our proposed research will be evaluated using an *experimental testbed* that can measure counts and sizes of data transfers across the storage hierarchy.

Intellectual Merit: There are several technical challenges in the components outlined above. A number of design choices within the storage system need to be investigated and evaluated. We propose to investigate three different levels of programming models, because while current HEC users are likely to be most comfortable with the lowest level of a weakly-typed runtime interface, future HEC users may find the high level declarative and strongly-typed imperative models more productive. This in turns leads to new compiler challenges for the higher level programming models. Our proposed Storage System will have to achieve a judicious balance of software-hardware co-design between the software runtime system and hardware support in the storage system. Though we will not be able to build a hardware prototype in the scope of this proposal, our experimental testbed will need to contain sufficient instrumentation to help guide the co-design process. In summary, this is a high-risk high-return proposal because of the inherent risks in departing from traditional filesystems, and of the potential for significant improvements in bandwidth utilization and programmability compared to past approaches.

Broader Impact: The demonstration of a new approach to storage systems can pave the way for novel technologies to address the bandwidth crisis facing future HEC applications. The broader impact of this project includes the integration of research and education; we will expose the new concepts and research results from this project to graduate students and upper-division undergraduate students through new courses, with special attention to the needs of underrepresented groups. We will also disseminate research results through the standard academic venues (publication in journals and conferences, conference tutorials, web sites, external talks). Further, advances in HEC storage systems will have a broader scientific and societal impact on all domains that depend on high end computing. Finally, we will leverage our contacts with government labs and industry to encourage adoption of the fundamental technologies in future commercial offerings.

1 Introduction

The High End Computing (HEC) field is at a major inflection point due to the end of a decades-long trend of targeting clusters built out of single-core processors. Future generation HEC architectures and systems built using many-core chips will pose unprecedented challenges for users. While the *concurrency challenge* for these systems is receiving attention from efforts such as past NSF HECURA programs, the DARPA High Productivity Computing Systems (HPCS) program and the centers at UC Berkeley and UIUC funded by Intel and Microsoft, the *storage challenge* for HEC applications has received relatively less attention in Computer Science research. However, the storage challenge is becoming increasingly urgent because of two opposing trends: first, many HEC application domains are becoming increasingly data-intensive in nature, and second, the available data bandwidth and access time per processor core is starting to decline at an alarming rate, with the rapid increase in the number of cores per node.

A major source of overhead in storage systems lies in the file system interfaces in the underlying operating system (OS). The basic design of file systems has been largely unchanged for multiple decades and dates back to an era of multi-programmed uniprocessors. Historically, the file handling part of the OS has been designed as a service to be shared by multiple processes, thereby resulting in additional OS overhead on each data transfer between the storage system and an individual process' data space. As operating systems evolved, the file-system overhead increased with support for richer meta-data and directory structures. This overhead in turn forced the use of large data transfer sizes to amortize the overhead over a larger collection of data[1]. However, transferring long sequences of contiguously located data runs counter to the needs of many emerging HEC applications for two reasons. First, HEC applications increasingly use irregular data structures, which result in wastage of storage bandwidth when only a fraction of the transferred locations is accessed. Second, with increasing parallelism per node (up to 1024 cores/socket projected in future Exascale systems [26]), it becomes important to utilize the available storage bandwidth for larger numbers of concurrent short data transfers rather than fewer long transfers. Our contention is that the basic file system design will no longer suffice as the dominant storage model for many-core HEC systems. Instead, it becomes necessary to explore new approaches to storage systems that reduce OS overhead, exploit manycore processors attached to multiple levels of storage hierarchy, and focus on new techniques for efficient utilization of bandwidth and reduction of average access times across these levels.

The focus of this collaborative proposal is on developing a Storage System that eliminates all OS intervention for data access, with accompanying extensions to programming models, compilers, and runtime systems for improved programmability, building on past contributions by the PI (Sarkar) and the co-PI's (Dennis, Gao) in these areas. Persistent data are represented as *write-once tree structures* in the Storage System and accessed via a pointer to the root node of the tree that represents the intended data object. The basic unit of data transfer is a *chunk*, which is intended to be much smaller than traditional OS page sizes[2]. An additional benefit from the proposed storage model stems from the fact that a chunk will contain information from just one data object. This is in contrast to conventional memory systems where a cache line or virtual memory page may contain information from more than one data object, leading to the problems of false sharing [11]. All data objects in the storage system are required to obey the write-once (single assignment) property. (Data objects in transient memory do not have this restriction.) When a data object is freed via garbage collection, its chunks are available for allocation to other data objects. The write-once

[1]Large data transfers have also been motivated by the characteristics of disk storage devices, especially when I/O was primarily serial.

[2]For concreteness, we assume a 128-byte chunk size in this proposal, but the overall approach is applicable to any chunk size.

property simplifies the memory model for the storage system and obviates the need for complicated consistency protocols. The necessary synchronization between the producer and consumer of a data object is provided by the Storage System Read command, which is delayed, if necessary, until the referenced data object is written by its producer. The Storage System is particularly well suited to runtimes that support fine-grain parallelism, such as work-stealing runtimes for fine-grained tasks[43, 14, 31], that are capable of issuing a large number of concurrent requests to the Storage System.

The proposal has the following components:

1. Design of a *new storage system* with software and hardware support for write-once tree structures (Section 3).
2. A *declarative programming model* based on FunJava, a functional subset of Java [30]. Any data structure in this programming model can directly be made persistent in our storage system, with no programmer intervention and minimal runtime overhead (Section 5.1).
3. A *strongly-typed imperative programming model* based on X10 [10] in which a type system extension will be used to enforce a separation between data structures that can be directly made persistent and those that cannot. No serialization overhead is incurred for the first category. For the second category, serialization consists of data structure transformation (with copying, as needed) to the first category (Section 5.2).
4. A *weakly-typed runtime interface* that enables C programs to access our storage system (Section 5.3).
5. A *compiler* with a data flow intermediate representation (DFIR) that provides the necessary *code generation* support for our storage system (Section 6).
6. An *execution model* and *runtime system* that implements interfaces to the storage system used by compiler-generated code and by the weakly-typed runtime interface (Section 7).
7. Evaluation of the proposed storage system using an *experimental testbed* that can measure counts and sizes of data transfers, so as to compare our storage system design with traditional approaches (Section 8).

There are several technical challenges in the components outlined above. There are a number of design choices within the storage system that will need to be investigated and evaluated. We propose to investigate three different levels of programming models, because while current HEC users are likely to be most comfortable with the lowest level of a weakly-typed runtime interface, future HEC users may find the high level declarative and strongly-typed imperative models more productive. This in turns leads to new compiler challenges for the higher level programming models. Finally, our proposed Storage System will have to achieve a judicious balance of software-hardware co-design between the software runtime system and hardware support in the storage system. Though we will not be able to build a hardware prototype in the scope of this proposal, our experimental testbed will need to contain sufficient instrumentation to help guide the co-design process. In summary, this is a high-risk high-return proposal because of the inherent risks in departing from traditional filesystems, and of the potential for significant improvements in bandwidth utilization and programmability compared to past approaches to HEC storage systems.

2 Approach

We propose to explore a computer organization consisting of a collection of processing nodes (the Execution System) coupled to a Storage System embodying a tree-based memory model.

Each processing node (core) of the Execution System executes one or more *threads*. Computation is performed by program *modules* which consist of functions/methods/procedures compiled from a high-level programming language. A module may contain a hierarchy of functions (methods/procedures) with local and heap variables held in stack/heap data structures located in its private address space.

Any running thread is executing one program module and is able to initiate concurrent execution of modules by threads running on other cores. The arguments and results of these calls can include global pointers to data structures in the Storage System. When a thread completes execution of a module, the thread terminates, making its thread execution slot in the core available for assignment to a new thread. Each of the programming models includes a mechanism (either in the language or used by the compiler) for a thread to synchronize with other threads, such as a join feature whereby a thread continues execution only when another thread (or threads) reaches the join control point in the program. A thread may be suspended for any of several reasons: (1) A join point has been reached before other threads; (2) The thread has initiated an I/O operation that has not completed; and (3) The thread has requested a Storage System operation that does not complete immediately.

Transient and Persistent Data. It is intended that the Storage System hold data objects and structures that have lifetimes beyond those of data operated on by individual program modules, and that these data objects and structures are the principal means of intercommunication between threads. These data are called *persistent*. If the Storage System is designed to be fault tolerant, then the persistent data will be preserved in event of a failure of processing cores.

Determinacy. The Storage System as proposed is a determinate system in the sense of Patil/Dennis [40]. A consequence is that if the Execution System is also determinate, then computations performed by the coupled systems are guaranteed to be determinate. This would be true in the case of the proposed functional programming model (in the absence of features to support the programming of transaction-oriented computations). The property of determinacy would eliminate a troublesome aspect of parallel programming.

The Three Programming Models. The three chosen programming models differ in the extent to which the programmer assists in specifying/identifying opportunities for parallel execution. In the purely functional case, parallelism is implicit in the structure of the program, and the compiler must choose both the data distribution and the granularity level at which multithreading is to be applied. In the case of X10, the programmer expresses the computation in terms of places that are intended to be mapped onto distinct processors of the target computer system. In the case of the C-based programming model, the programmer is explicit about the module calls that are to be implemented by invoking a new thread. One result of the project will be a comparison of the three programming models with respect to which applications are served best by each model.

Objectives. Our objectives in pursuing this proposed research are three-fold. First, we wish to demonstrate the benefits of a novel computer system organization employing a write-once storage system for persistent data objects based on a tree-based memory model. Secondly, we wish to compare, in programmability and performance, the merits of three programming models for parallel computing: a declarative model based on functional programming principles in which parallelism is implicit; an imperative programming model derived from X10 that includes programmer means for specifying task and data distribution and is strongly typed; and an imperative programming model, based on the C programming language, that is familiar to users of high performance computing resources. To this end, we will develop means, including compiler tools and run-time support software, for users of each programming model to utilize the write-once tree-based memory model and storage system. Our third goal is to develop and instrument a test bed for experimental studies of the benefits and issues in the proposed system architecture and the three programming models for execution of selected HEC applications.

In the following sections of the proposal, we present the proposed tree-based memory model, and our vision of its implementation in a Storage System. Further details on the three programming models, as well as the compiler and run-time systems needed to support them, are discussed. Our

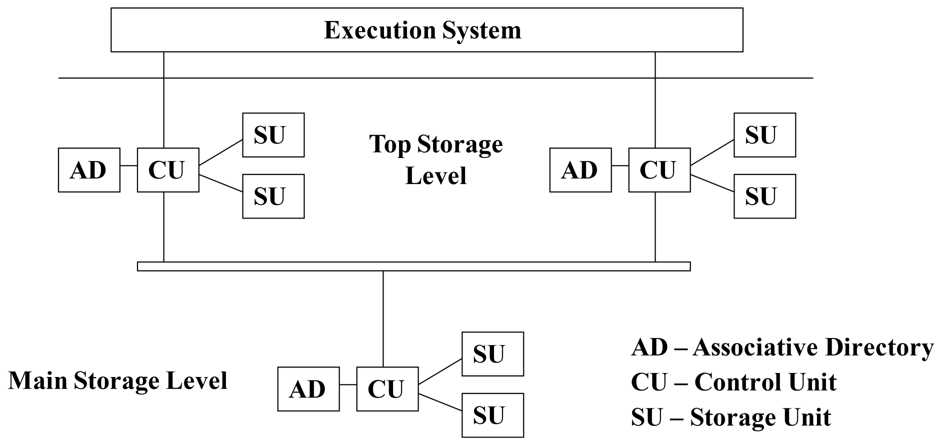

Figure 1: Overview of Storage System

plans for experimentation, related work, prior work by the principal investigators, broader impact, and management plan complete the proposal.

3 The Fresh Breeze Memory Model

In the Fresh Breeze Memory Model[51], information objects and data structures are represented using fixed size chunks, for example 128 bytes. Each chunk has a unique 64-bit identifier, a global pointer, that serves to locate the chunk within the storage system, and is a globally valid reference to the data represented by the chunks contents. Chunks may contain data and/or up to sixteen global pointers. Thus data objects and data structures may be represented by trees of chunks, or more generally, DAGs (directed acyclic graphs). In a nutshell, the life-cycle for a chunk can be summarized as follows: (1) Created in main memory by a producer thread as part of a data object, and immediately written to the lowest level of the Storage System Hierarchy (SSH); (2) Accessed by consumer threads from the lowest level of storage to which both producer and consumer have access; (3) Removed from each level of the memory hierarchy except the lowest (archive) level, when space is needed for other data; (4) Deleted when reference count indicates that no references to the chunk exist.

A key feature of the memory model is that chunks may be created and written by a user of the memory model, but a chunk cannot be made accessible to more than a single computing activity without its content being frozen and read-only. This is the *write once rule* of memory operation. Its adoption leads to some very attractive properties. For one, a memory hierarchy in which the write once rule is honored enables caches to be implemented with less energy and complexity, as evidenced by read-only caches for constant and texture memory in modern GPGPU's. Several users of the memory model may access shared data with no concern that it might be stale. Adopting the write once property requires computations to adopt a functional view of the storage system. A computing step involves accessing existing data values and creating fresh memory chunks to receive the results. Of course, to work efficiently this requires very efficient mechanisms for allocating memory and collecting chunks that no longer contain accessible data. Use of a fixed-size unit of memory allocation makes this feasible.

The write once property implies that a pointer can be incorporated into a chunk only at the time the chunk is allocated. It follows that there is no way that a path can be formed that leads from a reference contained in a chunk to the reference identifier of the chunk itself [18]. This property guarantees that the directed graph consisting of the collection of chunks and the pointers between pairs of chunk will never contain directed cycles[20]. Consequently, low overhead reference counts can be used to collect garbage chunks.

Commands are provided for users of the Fresh Breeze Memory Model to create, read, write and seal chunks. The **Seal** command is used to make the specified chunk read-only. It is provided because a user may need to execute more than one **Write** command to fully define a chunk, and therefore an interface is needed to signal that a chunk is complete. The Execution System must honor the rule that a global pointer is not given to a concurrent activity unless the referenced chunk has been sealed.

The Fresh Breeze Memory Model has been studied for its ability to achieve competitive performance in linear algebra operations fundamental to much scientific computation, in particular the dot product and matrix multiply [23]. Generation of efficient code for array operations with our storage model will be a key challenge addressed by the proposed research.

4 Storage System Implementation

The Storage System will implement the Fresh Breeze Memory Model. Ultimately, the Storage System will be implemented using hardware memory devices and controllers. In this way, the large portion of the operating system overhead incurred in conventional systems will be absent. However, a hardware implementation is beyond the scope of this proposed project. Therefore emulation of the Storage System will be used to develop, test and demonstrate the merits and projected performance of the envisioned system architecture. Here, we describe the Storage System in its envisioned hardware realization. This will serve to explain the long-range goal of the project and to indicate what will be needed to construct an emulation that will be useful for projecting behavior and performance. The proposed testbed for developing and evaluating the Storage System operating with the three programming models described below is discussed Section 8 of this proposal.

4.1 The Physical Storage System

The Storage System is coupled to the Execution System through a large number of identical ports that communicate independently with processing nodes of the Execution System. This structure is intended to support very large numbers of concurrent transactions with the Storage System so that high bandwidth will be achieved through a high volume of transactions rather than by requiring large individual transactions. This will enable greater exploitation of fine-grain parallelism is application codes.

The Storage System is a hierarchical memory system in which the higher levels (closer to the Execution System) cache data chunks actively involved in on-going computations. In Figure 1, two levels were illustrated for simplicity; the architecture may be extended to further levels as demanded by the device technology available and the capacity required by the system it is part of.

There is no relationship between the 64-bit number that is the global pointer of a chunk and the physical position it is held in the Storage System. This property permits new data to be stored in proximity to the location in the system where they were generated. To support this property *associative search* will be used, as for name resolution in conventional file systems, to map a global pointer to the physical location where the designated chunk is to be found. In the proposed Storage System, this mapping will have a very efficient realization in hardware using the B-tree algorithms favored in many database implementations.

Another function performed by the Storage System is to supply free global pointers to the Execution System for assignment to freshly created chunks. A data structure will be maintained, perhaps a bit map, which keeps a record of available global pointers. Pointers are assigned from the free pool and returned to the pool when the reference count shows they are no longer needed.

The principal components at each level of the Storage System are multiple storage devices to hold data chunks, and an associative directory for mapping chunk identifiers (global pointers) to the locations where chunks reside (Figure 1). At the lowest level (The Main Memory) the set of storage devices is sufficient to hold all data in the computer system. Accordingly, the directory must be able to map to a sufficiently large physical space to accommodate all data, and the directory must be able to handle the anticipated traffic. As usual, it is expected that read requests will far outnumber write requests at this level. At higher levels of the Storage System, the storage devices will be faster and of much less total capacity, but must handle a larger volume of transactions. The directories can be simpler, but must support the expected traffic. In addition, each level includes control logic to handle commands, remove chunks that have fallen out of use, and for garbage collection.

For directory implementation, we have studied hardware implementation of the B-Tree data structure commonly used in software file systems for mapping file names or identifiers to physical locations[3]. The results are very encouraging in that an associative search is guaranteed to complete in a fixed number of clock cycles, and the hardware organization uses RAM memory instead of area and power hungry CAM hardware.

4.2 Benefits and Challenges

Because the Storage System will replace a conventional file system implemented by operating system software, a major improvement in performance is to be expected. In current practice, each access to a file or portion thereof, even access to virtual memory pages, requires intervention of operating system software. Use of the Fresh Breeze Storage System will completely eliminate this operating system overhead. In the proposed Storage System, there is never any semantic conflict among transaction requests. Each request can be processed by the storage hierarchy without regard for other concurrent requests, except for competition for hardware resources. In consequence, there is no impediment to achieving massive levels of concurrency other than limits imposed by hardware organization of the Storage system and the granularity of concurrency supported by the Execution System.

The proposed Storage System will support very large volumes of memory transactions, enabling efficient execution of large computations where fine-grain parallelism can be exposed and exploited. This sets a challenge for the design of programming models and their implementation in future computer systems.

Another benefit of the proposed memory model and Storage System relates to data backup and the checkpointing of long-running computations. To preserve a snapshot of a computation or the state of a database only requires that a snapshot data structure be assembled containing the components to be preserved. The resulting structure, being read-only, will remain intact until the snapshot is released and becomes inaccessible. This will require far less time and resources than writing the complete snapshot to file system memory.

5 Programming Interfaces to the Storage System

5.1 Declarative Programming Interfaces

The Fresh Breeze project at MIT-CSAIL aims to develop a multi-core chip architecture that meets all requirements for composable parallel programming. The programming language chosen for users of a Fresh Breeze system is FunJava, a functional dialect of Java. A compiler for FunJava is under development at MIT-CSAIL and is designed to accept programs as Java Bytecode files and to transform them so to generate optimized Fresh Breeze machine code. In the following paragraphs we discuss: (1) use of data flow graphs as an intermediate program representation by the compiler; (2) our study of the minimal restrictions on Java programs such that FunJava functions meet requirements for composable parallel programming; and (3) the multi-thread model of Fresh Breeze machine code. It is expected that revisions to plans for the Fresh Breeze compiler to generate code for the proposed Execution System/Storage System will be straightforward to implement because the Storage System will implement the Fresh Breeze memory model.

[3]Summer 2008 study by research intern Kumud Bhandari.

Java Restrictions for Composable Parallel Programs. In the absence of input/output operations and other operating system calls, sequential execution of a Java method performs a functional transformation on a space of values that includes the states of any objects touched by the method. This implies that any java method has a representation as a *Data Flow Graph* (DFG). Determining the DFG for a method is made problematic by the possibility of aliasing — the possibility that data components are shared by distinct data structures or objects that are arguments of the method. Aliasing is a violation of program modularity principles because the behavior of a program module (function) will depend on whether components are shared by inputs to the module. Guaranteeing correct execution in the presence of possible aliasing also limits the degree of parallelism that may be exploited.

At present we take FunJava to be a functional subset of Java by which we mean that arguments and results of a method must be immutable values a method must not have any side effects that are visible outside method execution. This eliminates any possibility of aliasing, but may be more restrictive than necessary for achieving composable parallel programming. Determining a weaker set of constraints is a research topic under study.

The Fresh Breeze Multi-Thread Program model. The Fresh Breeze multi-thread program model [22] assumes that threads are the basic and only notion of concurrent activity. A thread corresponds to execution of a single sequential program module – corresponding typically to a Java method. The thread starts with invocation of the module and ends upon termination of module execution. Similar to Cilk [27], a thread may spawn a new thread which performs an independent computation concurrently with its parent, reporting back to the parent by means of a join mechanism when its work has been completed. A thread may spawn many child threads, providing a basis for data parallel computing, as well as concurrent invocations of subsidiary tasks.

5.2 Strongly-Typed Imperative Programming Interfaces

The *strongly-typed imperative programming model* will be based on X10 [10]. Building on the *value type* feature in X10, a type system extension will be used to enforce a separation between declarative vs. imperative data structures *i.e.,* data structures that are write-once and those that are not. Data structures in the first category will be directly mapped to the storage system, without any serialization overhead. For the second category, serialization consists of data structure transformation (with copying, as needed) to the first category.

The parallel constructs in X10 (and extensions in Habanero-Java [53, 54, 55]) are similar to those of Fresh Breeze, except that multiple assignments are permitted to shared mutable imperative data structures. Perhaps the simplest introduction to X10 is to focus on two constructs — async and finish — which can be used to write a large class of parallel programs. An important safety result in X10 is that any program written with these three constructs can never deadlock.

The statement, *async* ⟨*stmt*⟩, causes the parent activity to create a new child activity to execute ⟨*stmt*⟩. Execution of the async statement returns immediately i.e., the parent activity can proceed immediately to the statement following the async. The statement, *finish* ⟨*stmt*⟩, causes the parent activity to execute ⟨*stmt*⟩ and then wait till all sub-activities created within ⟨*stmt*⟩ have terminated globally. There is an implicit *finish* statement surrounding the main program in an X10 application. If async is viewed as a fork construct, then finish can be viewed as a join construct. However, the async-finish model is more general than the fork-join model [10].

5.3 Weakly-Typed Runtime Interfaces

Under this research thrust, we will leverage our current research in developing a shared memory programming/execution interface for a computing system based on many-core chip architecture technology such as the IBM Cyclops-64 chip/system [15]. Application programs are written in imperative style parallel programming API (e.g. C with parallel programming extensions

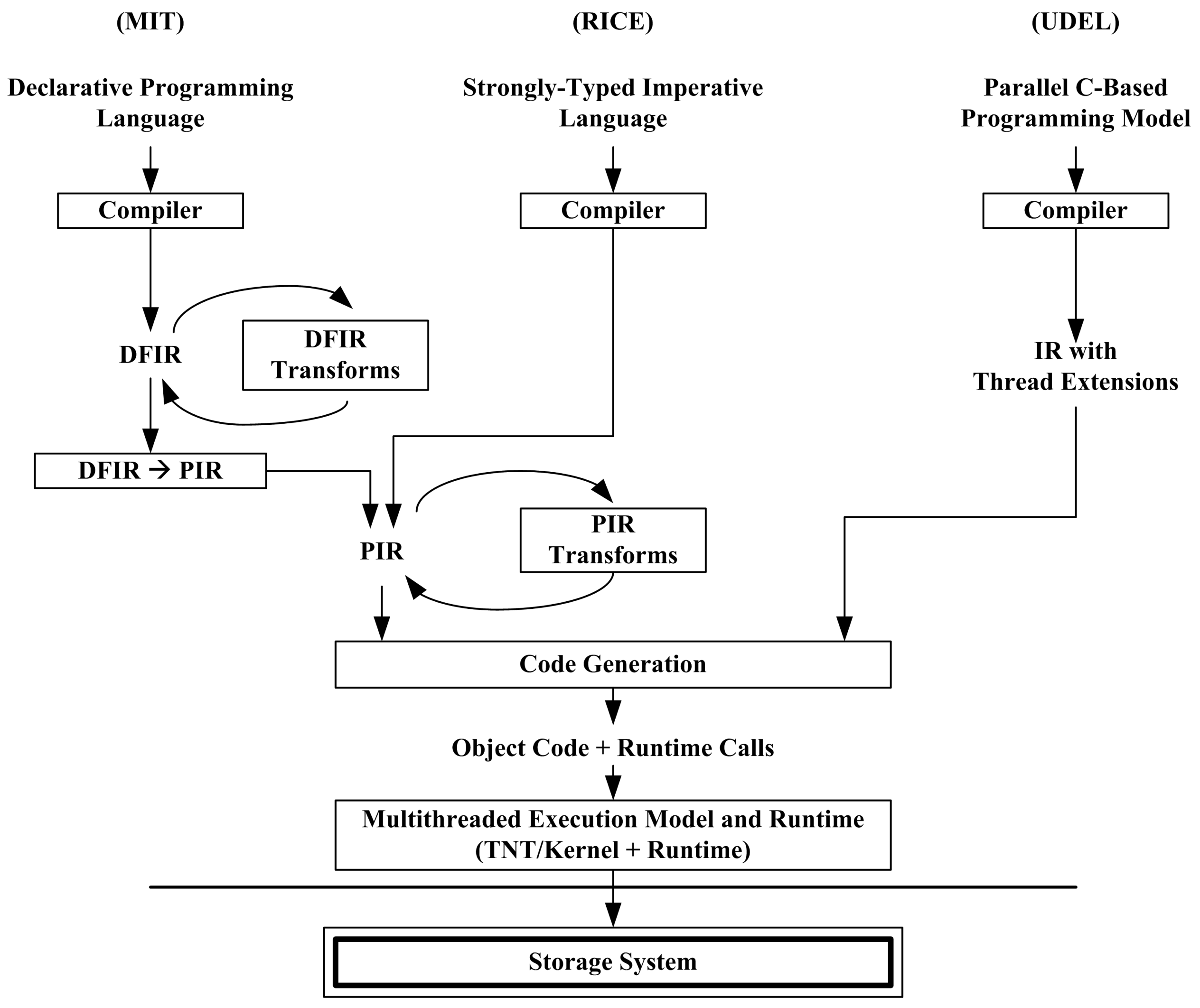
(MIT)
(RICE)
(UDEL)
Declarative Programming Language
Strongly-Typed Imperative Language
Parallel C-Based Programming Model
Compiler
Compiler
Compiler
DFIR
DFIR Transforms
IR with Thread Extensions
DFIR → PIR
PIR
PIR Transforms
Code Generation
Object Code + Runtime Calls
Multithreaded Execution Model and Runtime (TNT/Kernel + Runtime)
Storage System

Figure 2: Compiler support and intermediate representations for the three programming models

e.g. OpenMP, OpenSHMEM, etc. Our OpenMP/C compiler will translate the source code to a multithreaded API – called the TNT (or TiNyThreads).

6 Compiler Support and Intermediate Representations

Figure 2 summarizes the compiler structure proposed for our research. The declarative programming language (FunJava) will be translated to a Data Flow Intermediate Representation (DFIR) such as a Data Flow program Graph (DFG) [16, 17]. Past experience has shown that DFGs are an ideal form for representing functional programs for transformation and code generation by a compiler [57]. DFGs have been used in compilers for VAL [6] and SISAL [36], where most standard optimizations have straightforward implementations that run in linear time and space. In these compilers special data flow representations are provided for data parallel blocks and recurrences recognized by the compiler and are used to generate efficient machine code for these program structures [48, 49, 47]. The DFIR effort in our proposal will be lead by MIT.

Next, we see that the strongly-typed imperative language will be translated to a Parallel Intermediate Representation (PIR), and the DFIR can be lowered to this level as well. The PIR includes explicit constructs for parallel operations such as *foreach*, *async* and *finish*. An early version of the PIR was used in our recent work on chunking parallel loops in the presence of synchronization [55], and the complete PIR will build on our past work on program dependence graphs and parallel program graphs [46, 50, 45]. Like the DFIR, the PIR will expose all known parallelism in the program as well as all control and data dependences. However, unlike the DFIR, the PIR will also have to cope with multiple assignments (imperative updates) to shared locations and is hence at a lower level than the DFIR. Thus, the PIR can also be used to express explicit memory management and copy optimizations that are performed on the DFIR [34, 29]. The PIR effort in our proposal with be led by Rice. Finally, the PIR will be translated to object code (most likely via C code followed by a standard C compiler), which also represents the target level for the weakly-typed runtime interfaces (which will be led by U.Delaware).

While the structure in Figure 2 offers a great deal of flexibility in supporting multiple programming models, it also poses some interesting research challenges. One challenge that is especially relevant to HEC computations is related to optimized code generation for arrays, especially for contiguous array elements stored in the same chunk. This is analogous to optimization of array operations with 128-byte data transfers as in the Cell processor. Another challenge is in locality management to ensure that data and tasks with mutual affinity are located close to each other.

7 Program Execution Model and Runtime System

7.1 Program Execution Model

Our execution runtime features a hierarchical multithreaded program execution model. Our program execution model (PXM) serves as an interface between the architecture and the software. PXM is supported by key features implemented directly in the hardware architecture or indirectly through a combination of hardware features and runtime systems support. The PXM will be defined by a thread virtual machine (TVM) implemented as a user-level library.

7.2 Runtime System

The basis of the program execution model and runtime software for the proposed research will be based on the TiNy Threads runtime system (TNT) developed by the CAPSL group at the University of Delaware, and ETI under the direction of Prof. Gao.

We highlight the features of our runtime as follows:

- Our runtime replaces the conventional OS with a custom-made kernel: Instead of trimming a conventional OS such as Linux, the C64 kernel and the TNT library have been implemented from scratch. Only the functionality that is crucial to achieve and sustain high levels of application performance has been included.
- Our runtime is a non-intrusive runtime system: TNT is implemented as an user-level library that manages directly the hardware resources. TNT supports a non-preemptive thread execution model needed for applications to achieve full resource utilization.

7.3 The TNT Multithreaded program model

The original TiNy Threads (TNT) can be viewed as a C based API of a Thread Virtual Machine for a manycore processor such as the Cyclops-64 architecture.

The Cyclops-64 thread Virtual Machine (TVM) can be seen as an multi-chip multiprocessor extension of the C64 ISA. It has been designed to replace the OS with a narrow interface layer. Such layer of system software manages directly the hardware resources and provides an interface that shields the application programmer from the complexity of the architecture whenever possible. However, unlike a conventional OS, a TVM exposes those resources that are critical to achieve performance. That is the C64 TVM not only provides an abstraction layer and the application program interface expected by programmers, it also provides the common baseline for future research on program execution models. The C64 TVM includes three components: a *thread model*, a *memory model* and a *synchronization model*, as well as their corresponding APIs. The thread model presents thread management issues. The memory model includes the specification of the consistency model for the C64 system. Last, the synchronization model describes the functionality to implement mutual exclusion regions, perform direct thread-to-thread and barrier type of synchronization.

Two salient features of TNT are as follows:

- TNT provides an efficient coarse-grain thread level programming environment: TNT relocates services to the user-layer to simplify the runtime software environment and to make it more efficient. TNT also supports a familiar fork/join programming API for quick prototyping of parallel applications.
- TNT supports the development of program execution models: TNT does not impose any limitation on the number of threads available for parallel programming models in general and applications in particular. TNT seamlessly orchestrates dozens of hardware thread units and thousands of virtual threads with high efficiency.

7.4 Thread Scheduling

The objective of thread scheduling is to distribute the workload over the processors so that no processor is idle if there is work waiting to be done, and no thread is blocked if there is free processor capacity. It is desirable to do this in a way that exploits/preserves ”locality”, meaning that threads performing closely related computation are executed on processors that are in close communication proximity. The collection of agents performing thread scheduling must maintain collective knowledge of where free capacity resides and must somehow migrate assignments of new threads to areas of low utilization while maintaining locality. There is work by the P2P grid computing community on ”self-organizing grids” that is likely to be relevant to this project.

8 Experimentation

8.1 The Experimental Testbed

An important task is to identify a suitable set of benchmarks to evaluate our proposed storage system and associated programming models. As a starting point, we are considering the following parallel I/O benchmark suites — IOR (http://sourceforge.net/projects/ior-sio), IOzone (http://www.iozone.org/), Xdd (http://www.ioperformance.com/). All these benchmarks allow the user to specify a workload and evaluate the performance of the filesystem. We will leverage the experience gained in ongoing collaboration between ORNL and U.Delaware on porting IOR to C64 in the use of these benchmarks for our proposed research.

We plan to leverage in the following two simulation platforms in our experimental testbed:

- **Cyclops-64 many-core architecture simulator.** FAST [12] is an execution-driven, binary-compatible simulator of a multi-chip Cyclops-64 system developed at U.Delaware. It accurately reproduces the functional behavior and count of hardware components such as thread units, on-chip and off-chip memory banks, and the 3D interconnection network of the system. The FAST many-core architecture simulator plays a critical role in the development of parallel systems since it can accurately model the execution of a binary program in a parallel system. FAST models a program by (1) simulating the execution of instructions, (2) simulating the architecture exceptions (3) reproducing the memory behavior and (4) interconnection network contention and latency. The FAST simulator provides histograms of instructions executed as well as detailed traces of program execution. FAST has been developed following a modular approach that allows smooth integration of new functionalities or architecture characteristics. It will be necessary to adapt FAST for the purposed of our proposed research by designing additional components to model the proposed Storage System and its interaction with processors of the Execution System.

- **Architecture Simulation using HAsim**

  HAsim[42] is a tool for processor architecture experimentation and evaluation through cycle-accurate simulation. Using software simulation tools leads to very long simulation runs when the components of a design that have high logic complexity are modeled in detail. To achieve fast simulations of detailed logic behavior, the HAsim tool is designed to support modeling critical components of an architecture using FPGA technology. The user of HAsim uses a library of modules that model processor componentseither by means of software or through synthesis of FPGA implementations. The user also provides a timing specification similar to the control aspect of synchronous data flow, from which the HAsim software constructs a simulation shell that invokes the components, either software or FPGA modules, honoring the specified timing constraints. The simulation shell also includes provision for multiplexing an FPGA implementation over many instances of a component, thereby making it feasible to simulate larger systems than a single FPGA device could model. Because HAsim is designed for studying processor architecture, it will be necessary to adapt it for the purposed of our proposed research by designing additional components to model the proposed Storage System and its interaction with processors of the Execution System.

8.2 The Experimental Study

Evaluating the benchmark programs using our different programming models on the experimental testbeds listed above will be a major effort in our proposed research. We will implement selected benchmarks on our experimental testbeds and measure their performance and scalability using sizes and counts of data transfers as basic metrics for the storage system. Speedup trends will also be analyzed by varying the number of cores simulated within a chip. Under the simulation environment, we can also run experiments with varying latency and bandwidth to the storage system to study the robustness of our programming models and their implementation.

9 Related Work

The proposed research explores an approach to improving the performance of I/O in HEC systems that has no counterpart in other current research known to the PIs. There are, however, strong relations to earlier work on virtual memory implementation, and our experimental implementation strategies and goals bear similarities to current work on runtime software systems, and work on peer-to-peer cloud computing.

Parallel File Systems. In recent years, a significant amount of research on parallel file systems has been reported, including Lustre [4], GPFS[2], PVFS[35], pNFS[33], PanFS[37] and others [62, 56, 13, 7, 38, 41, 60]. Different APIs to interface with files such as MPI-IO[5], HDF5[3] and NetCDF[44] have gained popularity as an alternative to the basic POSIX API. Object-Based Storage [1] provides a different way to organize data and metadata on the storage medium than file or block methods. Much of the increased functionality of these parallel file systems comes at the cost of increased complexity and overhead of the file system software.

Some recent work seeks to reduce overhead of the file system and its load on the file servers. The Light Weight File System (LWFS) [39], a current project at Sandia National Laboratory, is a parallel file system that implements only essential functionality without any additional functionality that degrades performance and scalability. Implementations of additional features are moved into libraries and the application itself, allowing the application to be optimized to a right-weight solution. In contrast, our proposed Storage System eliminates all software overhead involved in accessing data by replacing software mechanisms used to translate file names or map data identifiers to physical locations with a common associative lookup means distributed across the storage units.

Virtual Memory. The idea of providing hardware supported addressing for very large data structures has been dormant for many years since the early days of the Multics and AS/400 systems. The concept was a key element in the design of the Multics computer system at MIT's Project MAC [8], where any object represented in the file system could be dynamically linked into a user's address space within limits that have not since been approached. The result was a system that offered the most powerful system support for programming in terms of modularity and security that has yet been achieved. Multics, however, did not provide an implementation of global pointers and so its modularity benefits did not extend to parallel programming. The IBM AS/400 series of systems [58] embody an implementation of global pointers, but the benefits were exploited by system programmers and software package developers, and not promoted for the end user. The proposed project will extend the benefits of a global virtual address space to systems built of many-core processing chips and make its benefits available through user programming models.

Runtime Systems: As stated earlier, we will use an extension of TNT as the runtime software system for our proposed research. Other developments of runtime support for small grain parallel computing include threaded function calls under CILK [28, 59], Cascade [9], and work-first and help-first policies for work stealing [32]. Each of these systems replaces OS services for thread scheduling with light-weight software mechanisms. Our proposal goes further in replacing all software mechanisms involved in data access, thereby enabling a major further gain in efficiency and allowing for more asynchrony between data transfers and parallel tasks.

Peer-To-Peer Cloud Computing: Work under way in this area includes the development of concepts and algorithms for organizing clusters of nodes in an open computing cloud in response to requests for resources *e.g.,* [25]. This work is of interest due to its applicability to the problem of scheduling threads on a massively parallel computer. In both situations it is desirable to allocate nodes (processors) in close proximity for computing jobs, so as to improve the locality exhibited by threads within a job.

10 Prior Work by PI and co-PIs

Professor Vivek Sarkar is an internationally-known expert in programming languages, program analysis, compiler optimizations and virtual machines for parallel and high performance computer systems. He was inducted as an ACM Fellow in 2008. Prior to joining Rice University, he had nearly 20 years of research experience at IBM, of which 16 years included leadership and management of research teams as follows:

- Sarkar created and led the programming models, tools, and productivity research agenda in the DARPA-funded IBM PERCS project during 2002–2007. Three major outcomes of this effort have been the creation of the X10 language, the creation of the Eclipse Parallel Tools Platform open source project with LANL and other partners, and the creation of a framework for measuring development productivity.

- As Senior Manager of Programming Technologies at IBM Research during 2000 – 2007,Sarkar was responsible for initiating and overseeing research projects carried out by a department with approximately 50 researchers working in the areas of Programming Models, Tools, and Optimized Execution Environments. Systems).

- Sarkar led a 15-person team at IBM Research in development and open source release of Jikes Research Virtual Machine in October 2001 (jikesrvm.org). Since its release, the open source Jikes Research Virtual Machine has been used for research and teaching in 90+ universities worldwide, and contributed to 27+ PhD dissertations and 155+ research publications. The Jikes RVM dynamic optimizing compiler design built on his personal research in the areas of Linear Scan register allocation, optimization of heap accesses using Array SSA form, BURSbased register-sensitive instruction selection, and Array Bounds Check elimination on Demand (ABCD).

- Sarkar led a 10-person team during 1991–1994 in the design and implementation of ASTI, IBM's first high-level product optimizer component. ASTI has been used in the XL Fortran product compilers since 1996 for loop and data transformations for locality and parallelism, and for efficient conversion of array statements to scalar code.

Since his arrival at Rice University in 2007, Sarkar has created the Habanero Multicore Software project at Rice University which spans the areas of programming languages, optimizing and parallelizing compilers, virtual machines, and concurrency libraries for homogeneous and heterogeneous multicore processors. He is the PI (with Prof. Gao as co-PI) on a recently awarded NSF grant CCF-0833166 in the HECURA program for three years starting September 2008. The work under way in this grant is focused on Programming Models, Compilers, and Runtimes for High-End Computing on Manycore Processors, which addresses a different and complementary research area from the storage system research in this proposal. However, a connection between NSF grant CCF-0833166 and this proposal is that the Habanero-C language being developed in that grant is related to the Habanero-Java language which is one of three programming models being explored in this proposal as interfaces to our proposed Fresh Breeze storage system.

Professor Jack Dennis is the Principal investigator of two recent NSF awards to the MIT Computer Science and Artificial Intelligence Laboratory:

- NSF Award 05-09386: September 1, 2005 to August 31, 2008: $250,000, CSR-AES: Multiprocessor Chip for Modular Software, Jack B. Dennis, Principal Investigator.
- NSF Award 07-19753: September 1, 2007 to August 31, 2010: $420,000, CSR-AES: User Support Software for a Fresh Breeze Computer System, Jack B. Dennis, Principal Investigator.

This funding is being used to support Project Fresh Breeze, which aims to develop and evaluate a multi-core chip architecture with the goal of supporting composable parallel programming, meaning that any parallel program written to run on the chip may be used, without change as a component of a larger parallel program. The programmability of multi-core chips is a serious issue that, in our view, stems from inadequacies of existing core and chip architectures that fail to recognize opportunities for a basic rethinking of the relation between hardware and software. With current hardware designs it is close to impossible to use a software component that makes use of parallel computing as a module in a larger parallel program. A redesign of the overall computation is usually required.

A vision of an architecture that promises to realize the goal of composability of parallel programs was put forward in [24] and presented in greater detail in [21]. Under the first NSF Grant, the conceptual architecture was developed into a concrete design and a cycle-accurate simulator was written in Java to test and evaluate concepts of the envisioned Fresh Breeze chip. The simulator has been used to verify the design of a SMT core processor by running several simple test problems. All components of the chip architecture have been made concrete by writing their descriptions as Java simulation models. The results have yielded several iterations of the design from revelations in the process of filling in design details. An important contribution to the project has been a graphical user interface (GUI) for the simulator, developed by a team of undergraduates. The GUI has proven a very useful tool for testing the simulator and running experiments.

The proposed research on the envisioned Storage System and its use in support of several parallel programming models is a natural extension of this work at MIT-CSAIL. In fact the use of tree structures as a universal model for data in computer systems has been advocated by Prof. Dennis since 1968 [19], and was part of the data flow programming model presented in 1974 [16].

Professor Guang R. Gao has a long track record working on HEC systems: from computer architecture, system software (from programming models, compilers, down to runtime and tools), multi threading models, and applications. Gao has been elected as both ACM and IEEE Fellows in 2007, and cited "for contributions to architecture and compiler technology of parallel computers".

- Guang Gao is the PI for Award No. 0708856 ($50,000), "CRI: Planning a Research Compiler Infrastructure Based on Open64." (8/1/07 - 7/31/08) This collaborative project lays the groundwork for building, deploying, and demonstrating the usage of a robust and extensible parallelizing/optimizing compiler and runtime software infrastructure (mainly for high-end computing), facilitating the preparation of a future community resource for a variety of computer science research. The project extends work based on the Open64 compiler suite, a robust, industry quality, state-of-the-art optimizing and parallelizing compiler that permits end-to-end experimentation and compiler research at all levels, including advanced computer architectures, parallel programming languages, compiler/runtime software, system software, application development, performance modeling/tuning, as well as grid computing. The work involves assessing the needs of other groups to address the challenges posed by increasingly complex programming paradigms and architectures.
- Gao is also the PI for Award No. 720531 ($80,000), "CSR-AES: Optimizations for Optimistic Parallelization Systems." (8/1/07- 7/31/08) This project is focused on the exploitation of a particular kind of data parallelism in irregular applications that arises from the use of unordered and ordered work-lists. Optimistic parallelization is the key mechanism for obtaining parallelism in such applications. A runtime system is used to manage the optimistic parallelism, and compiler analysis are used to optimize parallel execution. programs are further optimized using dynamic code specialization as the program executes. To investigate the scalability of this approach, the project uses multi-core hardware prototypes based on field-programmable gate-arrays. If successful, the project will go a long way toward solving the pressing problem of writing software for multi-core processors.
- Gao is also the PI for Award No. 0702244 ($299,999), "A High Throughput Massive I/O Storage Hierarchy for PETA-scale High-end Architectures." (5/01/07 - 4/30/10). This research is aimed at addressing both the limited bandwidth and the performance of today's I/O storage systems. The main idea is to replace traditional rotary disks with a 'memory pool' - a massive grid of solid-state (flash) memories, and to introduce a new memory hierarchy model based on the improved access time and bandwidth of such grid. We aim to develop of a novel I/O architecture model for a class of high-end petascale computing systems; develop of a corresponding RAS model; and perform an experimental study on the new I/O architecture and software model developed herein.

11 Broader Impact

The impact of the proposed work is hard to overstate. The demonstration of a new approach to storage systems can pave the way for novel technologies to address the bandwidth crisis facing future HEC applications. This research will be carried out as a collaborative effort among Rice University, MIT, and the University of Delaware. The broader impact of this project includes the integration of research and education; we shall expose the new concepts and research results from this project to graduate students and upper-division undergraduate students through new courses on modern approaches to HEC software. The teaching materials that result from these courses will also be used in external workshops and tutorials, and made available to other educators through the Connexions system at Rice so as to help train a new generation of researchers and students in our proposed approach to storage systems. Further, advances in HEC storage systems will have a broader scientific and societal impact on all domains that depend on high end computing. We will also leverage our contacts with government labs and industry to encourage adoption of the fundamental technologies in future commercial offerings.

The research proposed in this team proposal will impact all graduate students and postdoctoral researchers affiliated with current and future projects in the research groups led by Dennis, Gao and

Sarkar, since the proposed storage system and its accompanying programming model, compiler and runtime will enable research for follow-on parallel software and hardware. We will also disseminate research results through the standard academic venues (publication in journals and conferences, conference tutorials, web sites, external talks). We will also apply for REU grants to involve undergraduate students in the proposed research projects.

Sarkar's team is part of the compiler group at Rice University which has a history of success in Education, Outreach, and Training programs to build on. These efforts have included programs for undergraduate and graduate students, training programs for K-12 teachers and students, and programs aimed specifically at increasing the number of women and minority students in the computational sciences. We will take on efforts that address both graduate and undergraduate education, with special attention to the needs of underrepresented groups. We will expand the existing AGEP (Alliance for Graduate Education and the Professoriate) program at Rice University. In the AGEP program, women and underrepresented minority undergraduate students from various US universities come to Rice for a summer research experience with established faculty. In 2008 Summer at Rice, Sarkar personally supervised two minority women undergraduate students who had only completed their freshman year in college (one from Johnson C. Smith University and one from Case Western Reserve University) through the AGEP program. A paper on their summer research was presented at the Richard Tapia Celebration of Diversity 2009 Conference. Following this trend, we will use our undergraduate student funding to support two AGEP interns in each summer during the course of this proposed project. Because graduate education is one of the most important and effective ways for a research effort to create long-term impacts, it will be a special emphasis in our project. Producing Ph.D.'s from underrepresented groups is a particular priority. Sarkar's first PhD student after arriving at Rice was an African-American student who graduated in September 2008. (While at IBM, Sarkar supervised multiple women and minority graduate student interns, and also participated in the 2007 CRA-W Programming Languages summer school.) Finally, we propose to fund travel for women and underrepresented minority undergraduate and graduate students to attend the biannual Grace Hopper and Richard Tapia conferences. Doing so will enhance the students' knowledge, reputations, and contacts as they pursue their careers in computer science.

Finally, we will leverage our research groups' contacts with industry to influence the design of future product compilers and multicore processors based on the results of this research. We will work with leading HEC vendors to move these technologies into their systems. The PIs have a long history of designing algorithms that are adopted by industry. We understand both the technical and organizational constraints that often prevent academic developments from appearing in commercial products and will work to mitigate those issues.

12 Management Plan

The work of this proposal will build upon extensive prior work in compiler, languages, runtime, operating system and architecture research performed by the PIs, whose areas of research are synergistic and complementary. All three PIs have significant management experience as outlined below.

As Senior Manager of Programming Technologies at IBM Research during 2000 2007, Dr. Sarkar was responsible for initiating and overseeing research projects carried out by a department with approximately 50 researchers working in the areas of Programming Models (X10, XJ, Collage), Tools (Eclipse Parallel Tools Platform, Advanced Re-factorings in Eclipse, Scalable And Flexible Error detection, Security analysis, Scripting analysis, WALA), and Optimized Execution Environments (Jikes RVM, Metronome, Progressive Deployment Systems). Since joining Rice University in July 2007, Sarkar leads a group of approximately 15 researchers working on the Habanero Multicore Software Research project and related efforts.

Professor Dennis founded the Computation Structures Group within the Laboratory for Computer Science at MIT in 1964 and supervised research sponsored by DARPA, NSF, NASA, and DOE, leading to completion of 27 doctoral theses on both theoretical and practical topics relating to many aspects of computer system operation. His work developed principles of the novel virtual memory system of the Multics computer system [8], and he is widely known for seminal work on data flow models of computation and their application to computer architecture.

Professor Gao has extensive experience in managing large research groups. Under his direction, the Computer Architecture and Parallel Systems Lab (CAPSL) has produced solid research work in several areas, including architecture, system software, and compilers/run-time systems. CAPSL has been the gate keeper of Open64 and uses Open64 as a research code base to develop both parallelization and optimization technology for a number of target architectures from superscalar/VLIW architectures to parallel multi-threaded architectures.

We will decompose the proposed research into tasks focused on the design and implementation of the storage system, programming models, compilers, and runtime systems. Many of these tasks will be pursued concurrently and collaboratively. Though all PIs will be closely involved in all aspects of the project, it is expected that some tasks will be led by specific PIs with extensive consultations and discussions among all three teams. Figure 2 outlines the proposed division of responsibility among the three teams. In particular, the declarative programming model based on FunJava will be led by MIT, the strongly-typed imperative programming model based on X10 will be led by Rice, and the weakly-typed runtime interface based on TNT will be led by Delaware. Within the compiler, the DFIR implementation will be led by MIT and the PIR implementation will be led by Rice. The implementation of the experimental testbed will be led by Delaware. The software-hardware co-design of the runtime system and the storage system will be performed collaboratively by all three teams.

12.1 Mentoring of Postdoctoral Researchers

The Rice University budget includes support for a postdoctoral researcher, and the PI, Sarkar, has significant experience with mentoring postdoctoral researchers. While at IBM (prior to July 2007), Sarkar mentored two postdoctoral researchers, Igor Peshansky and Mandana Vaziri, both of whom obtained permanent positions in IBM Research. At Rice University, Sarkar currently has three postdoctoral researchers in his group — Jun Shirako, Yonghong Yan, and Jisheng Zhao. The mentoring activities include weekly one-on-one meetings with each of them, guidance in improving teaching and presentation skills by having them give technical presentations to the entire research group, experience in research supervision by assigning to each of them at least one graduate or undergraduate student whom they co-supervise, co-authoring of research papers with postdoctoral researchers as the lead authors [52, 53, 54, 55, 61], training in research ethics in experimental results and research writing for publications, career counseling, and exposure to career opportunities by inclusion in Sarkar's meetings and conference calls with sponsors and collaborators in industry and academia.

References

[1] *Draft OSD Standard. T10 Committee.* Storage Networking Industry Association (SNIA)., July 2004.

[2] *GPFS V3.2.1 Concepts, Planning, and Installation Guide.* IBM Corporation., August 2008.

[3] *HDF5 User's Guide, HDF5 Release 1.8.2.* HDF Group., November 2008.

[4] *Lustre 1.6 Operations Manual.* Sun Microsystems., Nov 21 2008.

[5] *MPI: A Message-Passing Interface Standard, Version 2.1.* Message Passing Interface Forum., June 2008.

[6] W. B. Ackerman and Jack. B. Dennis. *VAL: A value-oriented algorithmic language(preliminary reference manual).* M.I.T. Laboratory for Computer Science, 1979.

[7] Pavan Balaji, Wu-chun Feng, Jeremy Archuleta, Heshan Lin, Rajkumar Kettimuthu, Rajeev Thakur, and Xiaosong Ma. Semantics-based distributed i/o for mpiblast. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 293–294, New York, NY, USA, 2008. ACM.

[8] A. Bensoussan, C. T. Clingen, and R. C. Daley. The multics virtual memory. *SOSP '69: Proceedings of the second symposium on Operating systems principles*, pages 30–42, 1969.

[9] D. Callahan, B.L. Chamberlain, and H.P. Zima. The cascade high productivity language. pages 52–60, April 2004.

[10] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[11] Jyh-Herng Chow and Vivek Sarkar. False Sharing Elimination by Selection of Runtime Scheduling Parameters. *International Conference on Parallel Processing*, August 1997. (Recipient of Best Paper Award).

[12] d Cuvillo, J. Zhu, W. Hu, and Z. Gao. FAST: A functionally accurate simulation toolset for the Cyclops64 cellular architecture, 2005.

[13] Willem de Bruijn and Herbert Bos. Pipesfs: fast linux i/o in the unix tradition. *SIGOPS Oper. Syst. Rev.*, 42(5):55–63, 2008.

[14] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. Tiny threads: A thread virtual machine for the cyclops64 cellular architecture. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 14*, page 265.2, Washington, DC, USA, 2005. IEEE Computer Society.

[15] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. TiNy Threads: A Thread Virtual Machine for the Cyclops64 Cellular Architecture. *ipdps*, 15:265b, 2005.

[16] J. B. Dennis. First version of a data flow procedure language. *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 362–376, 1974.

[17] J. B. Dennis, J. B. Fosseen, and J. P. Linderman. Data flow schemas. In *Proceedings of the International Sympoisum on Theoretical Programming*, pages 187–216, London, UK, 1974. Springer-Verlag.

[18] J. B. The Paradigm Compiler: Mapping a functional language for the Connection Machine. Dennis. *In Scientific Applications of the Connection Machine, Horst Simon, ed. Singapore*, pages 301–315, 1989.

[19] Jack B. Dennis. Programming generality, parallelism and computer architecture. *In Information Processing 68, North-Holland Publishing Co.*, 1969.

[20] Jack B. Dennis. General parallel computation can be performed with a cycle-free heap. *Proceedings of the 1998 International Conference on Parallel Architectures and Compiling Techniques.*, pages 96–103, 1998.

[21] Jack B. Dennis. Fresh breeze: a multiprocessor chip architecture guided by modular programming principles. *SIGARCH Comput. Archit. News*, 31(1):7–15, 2003.

[22] Jack B. Dennis. The fresh breeze model of thread execution. *Workshop on Programming Models for Ubiquitous Parallelism, with PACT-2006*, September 2006.

[23] Jack B. Dennis. Linear algebra with the fresh breeze processor. *Unpublished Memo*, 2008.

[24] J.B. Dennis. A parallel program execution model supporting modular software construction. *Massively Parallel Programming Models, 1997. Proceedings. Third Working Conference on*, pages 50–60, Nov 1997.

[25] Deger Cenk Erdil, Michael J. Lewis, and Nael B. B. Abu-Ghazaleh. Adaptive approach to information dissemination in self-organizing grids. In *ICAS '06: Proceedings of the International Conference on Autonomic and Autonomous Systems*, page 55, Washington, DC, USA, 2006. IEEE Computer Society.

[26] Peter Kogge et Al. Exascale computing study: Technology challenges in achieving exascale systems. *Technical Report, AFRL contract Number FA8650-07-C-7724*, 2008.

[27] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, 1998.

[28] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, 1998.

[29] K. Gharachorloo, V. Sarkar, and J. L. Hennessy. Efficient Implementation of Single Assignment Languages. *ACM Conference on Lisp and Functional Programming*, pages 259–268, July 1988.

[30] I. Ginzburg. Compiling array computations for the fresh breeze parallel processor. *Thesis for the Master of Engineering degree, MIT Department of Electrical Engineering and Computer Science*, May 2007.

[31] Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. *(to appear in IEEE International Parallel and Distributed Processing Symposium, IPDPS 2009)*, 2009.

[32] Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. Work-first and help-first scheduling policies for terminally strict parallel programs. In *23rd IEEE International Parallel and Distributed Processing Symposium*, May 2009. (To appear).

[33] Dean Hildebrand and Peter Honeyman. Exporting storage systems in a scalable manner with pnfs. In *MSST '05: Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 18–27, Washington, DC, USA, 2005. IEEE Computer Society.

[34] Paul Hudak and Adrienne Bloss. The aggregate update problem in functional programming systems. *Proceedings of the Twelfth Annual ACM Conference on the Principles of Programming Languages*, pages 300–313, January 1985.

[35] W. B. Ligon and R. B. Ross. Implementation and performance of a parallel file system for high performance distributed applications. In *HPDC '96: Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, page 471, Washington, DC, USA, 1996. IEEE Computer Society.

[36] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. Sisal: Streams and iteration in a single assignment language: Reference manual version 1.2. *Technical Report M-146, Rev. 1. Lawrence Livermore National Laboratory*, 1985.

[37] David Nagle, Denis Serenyi, and Abbie Matthews. The panasas activescale storage cluster: Delivering scalable high bandwidth storage. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 53, Washington, DC, USA, 2004. IEEE Computer Society.

[38] Jarek Nieplocha, Holger Dachsel, and Ian Foster. Distant i/o: One-sided access to secondary storage on remote processors. In *HPDC '98: Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, page 148, Washington, DC, USA, 1998. IEEE Computer Society.

[39] R.A. Oldfield, P. Widener, A.B. Maccabe, L. Ward, and T. Kordenbrock. Efficient data-movement for lightweight i/o. *Cluster Computing, IEEE International Conference on*, 0:1–9, 2006.

[40] Suhas S. Patil. Closure properties of interconnections of determinate systems. pages 107–116, 1970.

[41] Swapnil V. Patil, Garth A. Gibson, Sam Lang, and Milo Polte. Giga+: scalable directories for shared file systems. In *PDSW '07: Proceedings of the 2nd international workshop on Petascale data storage*, pages 26–29, New York, NY, USA, 2007. ACM.

[42] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer. A-port networks: Preserving the timed behavior of synchronous systems for modeling on fpgas. *ACM Transactions on Reconfigurable Technology and Systems, Vol V, No. 11*, pages 1–25, September 2008.

[43] K. Randall. Cilk: Efficient multithreaded computing. Technical report, Cambridge, MA, USA, 1998.

[44] Russ Rew, Glenn Davis, Steve Emmerson, Harvey Davies, and Ed Hartnett. *The NetCDF Users Guide. NetCDF Version 4.0.1.* Unidata Program Center., March 2009.

[45] Vivek Sarkar. Analysis and Optimization of Explicitly Parallel Programs using the Parallel Program Graph Representation. In *Languages and compilers for parallel computing. Proceedings of the 10th international workshop. Held Aug., 1997 in Minneapolis, MN.*

[46] Vivek Sarkar. A Concurrent Execution Semantics for Parallel Program Graphs and Program Dependence Graphs (Extended Abstract). *Springer-Verlag Lecture Notes in Computer Science*, 757:16–30, 1992. Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing, Yale University, August 1992.

[47] Vivek Sarkar and David Cann. POSC — a Partitioning and Optimizing Sisal Compiler. *Proceedings of the ACM 1990 International Conference on Supercomputing*, pages 148–163, June 1990. Amsterdam, the Netherlands.

[48] Vivek Sarkar and John Hennessy. Compile-time Partitioning and Scheduling of Parallel Programs. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, 21(7):17–26, June 1986.

[49] Vivek Sarkar and John Hennessy. Partitioning Parallel Programs for Macro-Dataflow. *ACM Conference on Lisp and Functional Programming*, pages 202–211, August 1986.

[50] Vivek Sarkar and Barbara Simons. Parallel Program Graphs and their Classification. *Springer-Verlag Lecture Notes in Computer Science*, 768:633–655, 1993. Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing, Portland, Oregon, August 1993.

[51] B. Schmidt. A shared memory system for fresh breeze. *Thesis for the Master of Engineering degree, MIT Department of Electrical Engineering and Computer Science*, May 2008.

[52] Jun Shirako, Hironori Kasahara, and Vivek Sarkar. Language extensions in support of compiler parallelization. In *The 20th International Workshop on Languages and Compilers for Parallel Computing (LCPC'07)*, 2007.

[53] Jun Shirako, David Peixotto, Vivek Sarkar, and William Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS '08: Proceedings of the 22nd International conference on Supercomputing*, June 2008.

[54] Jun Shirako, David Peixotto, Vivek Sarkar, and William Scherer. Phaser accumulators: a new reduction construct for dynamic parallelism. In *23rd IEEE International Parallel and Distributed Processing Symposium*, May 2009. (To appear).

[55] Jun Shirako, Jisheng Zhao, V. Krishna Nandivada, and Vivek Sarkar. Chunking parallel loops in the presence of synchronization. In *ICS '09: Proceedings of the 23rd International conference on Supercomputing*, June 2009. (To appear).

[56] Stephen C. Simms, Gregory G. Pike, S. Teige, Bret Hammond, Yu Ma, Larry L. Simms, C. Westneat, and Douglas A. Balog. Empowering distributed workflow with the data capacitor: maximizing lustre performance across the wide area network. In *SOCP '07: Proceedings of the 2007 workshop on Service-oriented computing performance: aspects, issues, and approaches*, pages 53–58, New York, NY, USA, 2007. ACM.

[57] S. Skedzielewski and J. Glauert. IF1 – An Intermediate Form for Applicative Languages. Technical report, Lawrence Livermore National Laboratory, 1985. No. M-170.

[58] F. G. Soltis. *Inside the AS/400.* Duke Press, 1996.

[59] Voon-Yee Vee and Wen-Jing Hsu. A scalable and efficient storage allocator on shared memory multiprocessors. In *ISPAN '99: Proceedings of the 1999 International Symposium on Parallel Architectures, Algorithms and Networks*, page 230, Washington, DC, USA, 1999. IEEE Computer Society.

[60] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Miller, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[61] Yonghong Yan, Max Grossman, and Vivek Sarkar. Jcuda: a programmer-friendly interface for accelerating java programs with cuda. Submitted to Europar 2009 conference.

[62] Shujia Zhou, Amidu Oloso, Megan Damon, and Tom Clune. Application controlled parallel asynchronous io. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 178, New York, NY, USA, 2006. ACM.