A Portable Runtime Interface For Multi-Level Memory Hierarchies

Mike Houston Ji-Young Park Manman Ren Timothy Knight Kayvon Fatahalian
Alex Aiken William J. Dally Pat Hanrahan
Stanford University

Abstract

We present a platform independent runtime interface for moving data and computation through parallel machines with multi-level memory hierarchies. We show that this interface can be used as a compiler target and can be implemented easily and efficiently on a variety of platforms. The interface design allows us to compose multiple runtimes, achieving portability across machines with multiple memory levels. We demonstrate portability of programs across machines with two memory levels with runtime implementations for multi-core/SMP machines, the STI Cell Broadband Engine, a distributed memory cluster, and disk systems. We also demonstrate portability across machines with multiple memory levels by composing runtimes and running on a cluster of SMP nodes, out-of-core algorithms on a Sony Playstation 3 pulling data from disk, and a cluster of Sony Playstation 3's. With this uniform interface, we achieve good performance for our applications and maximize bandwidth and computational resources on these system configurations.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Run-time environments

General Terms Design, Languages, Performance, Experimentation

Keywords Memory Hierarchies, Parallelism, Runtime, Sequoia

1. Introduction

Most parallel programs today are written using a *two-level* memory model, in which the machine architecture, regardless of how it is physically constructed, is abstracted as a set of sequential processors executing in parallel. Consistent with many parallel programming languages, we refer to the two memory levels as *local* (local to a particular processor) and *global* (the aggregate of all local memories). Communication between the global and local levels is handled either by explicit message passing (as with MPI) or by language-level distinctions between local and global references (as in UPC [6] and Titanium [27]).[1] Using a two-level abstraction to program a *multi-level* system, a configuration with more than one level of communication, obscures details of the machine that may be critical to performance. On the other hand, adding support outside of the programming model for moving computation and data between additional levels leads to a multiplicity of mechanisms for essentially the same functionality (e.g., the ad hoc or missing support for out-of-core programming in most two-level systems). It is our thesis that programming abstractions, compilers, and runtimes directly supporting multi-level machines are needed.

[1] The literature on runtime systems, such as MPI, uses slightly different terminology, referring to the two levels as *local* and *remote*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PPoPP '08 February 20–23, 2008, Salt Lake City, Utah, USA.
Copyright © 2008 ACM 978-1-59593-960-9/08/0002…$5.00

Our work is based on the belief that three trends in machine architecture will continue for the foreseeable future. First, future machines will continue to increase the depth of the memory hierarchy, making direct programming model support for more than two-level systems important. Second, partly as a result of the increasing number of memory levels, the variety of communication protocols for moving data between memory levels will also continue to increase, making a uniform communication API desirable both to manage the complexity and improve the portability of applications. Lastly, architectures requiring explicit application control over the memory system, often through explicit memory transfers, will become more common. A current extreme example of the kind of machine is LANL's Roadrunner, which combines disk, cluster, SMP, and the explicit memory control required by the Cell processor.

In this paper, we present an API and a runtime system that virtualizes memory levels, giving a program the same interface to data and computation whether the memory level is a distributed memory, a shared memory multiprocessor (SMP), a single processor with local memory, or a disk system, among other possibilities. Furthermore, our API is composable, meaning that a runtime for a new multi-level machine can be easily constructed by composing the runtimes for each of its individual levels.

The primary benefit of our approach is a substantial improvement in portability and ease of maintenance of a high performance application for multiple platforms. Consider, for example, a hypothetical application first implemented on a distributed memory cluster. Typically such a program relies on MPI for data transfer and control of execution. Tuning the same application for an SMP either requires redesign or reliance on a good shared memory MPI implementation; unfortunately, in most cases the data transfers required on the cluster for correctness are not required on a shared memory system and may limit achievable performance. Moving the application to a cluster of SMPs could use an MPI process per processor, which relies on an MPI implementation with recognition of which processes are running on the same node and which are on other nodes to orchestrate efficient communication. Another option is to use MPI between nodes and Pthreads or OpenMP compiled code within a node, thus mixing programming models and mechanisms for communication and execution. Another separate challenge is supporting out-of-core applications needing access to data from

disk, which adds yet another interface and set of mechanisms that need to be managed by the programmer. As a further complication, processors that require explicit memory management, such as the STI Cell Broadband Engine, present yet another interface that is not easily abstracted with traditional programming techniques. Dealing with mixed mode parallel programming and the multiplicity of mechanisms and abstractions makes programming multi-level machines a daunting task. Moreover, as bandwidth varies through the machine, orchestrating data movement and overlapping communication and computation become difficult.

The parallel memory hierarchy (PMH) programming model provides an abstraction of multiple memory levels [2]. The PMH model abstracts parallel machines as trees of memories with slower memories toward the top near the root, faster memories toward the bottom, and with CPUs at the leaves. The Sequoia project has created a full language, compiler, and a set of applications based on the PMH model [11, 21]. The basic programming construct in Sequoia is a *task*, which is a function call that executes entirely in one level of the memory hierarchy, except for any *subtasks* that task invokes. Subtasks may execute in lower memory levels of the system and recursively invoke additional subtasks at even lower levels. All task arguments, including arrays, are passed by value-result (i.e., copy-in, copy-out semantics). Thus a call from a task to a subtask represents bulk communication, and all communication in Sequoia is expressed via task calls to lower levels of the machine. The programmer decomposes a problem into a tree of tasks, which are subsequently mapped onto a particular machine by a compiler using a separate *mapping* dictating which tasks are to be run at which machine levels.

Although the previous Sequoia work demonstrates applications running on IBM Cell blades and a cluster of PCs, it does not show portability to multi-level memory hierarchies. More importantly, in our view, they also rely on a custom compiler back-end for Cell and a separate, custom runtime for a cluster of PCs which manages all execution and data movement in the machine. The difficulty with this approach is that every new architecture requires a monolithic, custom backend. The Sequoia compiler, along with the bulk optimizations and custom backend used for Cell, is described by Knight et al. [21]. For this paper, we build on the previous PMH and Sequoia work, but we take the approach of defining an abstract runtime interface as the target for the Sequoia compiler, and provide separate runtime implementations for each distinct kind of memory in a system. As discussed above, our approach is to define a single interface that all memory levels support; since these interfaces are composable, adding support for a new architecture only requires assembling an individual runtime for each adjacent memory level pair of the architecture, rather than reimplementing the entire compiler backend.

We begin by presenting our runtime interface (Section 2) and discussing implementations for multi-core/SMP machines, clusters, the STI Cell Broadband Engine, and disk systems (Sections 3.1-3.4). We also show how these runtimes can be composed, allowing us to mix and match runtimes and enabling rapid construction of composite runtimes for machines with multi-level memory hierarchies (Section 4). We present results for applications running on a cluster of SMP nodes, a Sony Playstation 3 running out-of-core applications from disk with 6 SPEs, and a cluster of Playstation 3's; we also show several Sequoia applications on our runtimes and analyze the overheads and efficiency of the implementations and the abstraction (Section 5). For all eight machine configurations that we test (five two-level and three multi-level) we make no source changes at all to our application suite; we simply define an appropriate mapping of each application to the machine and recompile. We believe our work is the first to actually demonstrate portability of the same program to such a wide diversity of machine architectures with good performance.

In summary, the contributions of this paper are:

- a runtime interface for a compiler which allows the compiler to optimize and generate code for a variety of machines without knowledge of the specific bulk communication and execution mechanisms required;
- implementations of our interface on a wide variety of machines demonstrating application portability while maximizing the usage of bandwidth and computational resources; and
- the composition of runtimes to easily allow applications to run on machines with complex, multi-level memory hierarchies with a variety of different communication mechanisms without compiler or source code modifications.

2. The Runtime Interface

A runtime in our system provides three main services for code (tasks) running within a memory level: (1) initialization/setup of the machine, including communication resources and resources at all levels where tasks can be executed; (2) data transfers between memory levels using asynchronous bulk transfers between arrays; and (3) task execution at specified (child) levels of the machine. Each runtime straddles the transition between two memory levels, which we refer to as *top* and *bottom*. There is only one memory at the top level, but the bottom level may have multiple memories; i.e., the memory hierarchy is a tree, where the bottom level memories are children of the top level. An illustration is provided in Figure 1.

The top and bottom interfaces of a runtime have different capabilities and present a different API to clients running at their respective memory levels. A listing of the `C++` public interface of the top and bottom parts of a runtime is given in Figure 2. We briefly explain each method in turn.

We begin with Figure 2(a), the API for the top side of the runtime. The top is responsible for both the creation and destruction of runtimes. The constructor requires two arguments: a table of tasks representing the functions that the top level can invoke in the the bottom level of the runtime, and a count of the number of children of the top. At initialization, all runtime resources, including execution resources, are created, and these resources are destroyed at runtime shutdown.

Our API emphasizes bulk transfer of data between memory levels, and for this reason the runtimes directly support arrays. Arrays are allocated and freed via the runtimes (`AllocArray` and `FreeArray`) and are *registered* with the system using the array's reference (`AddArray`) and *unregistered* using the array's descriptor (`RemoveArray`). An array descriptor is a unique identifier supplied by the user when creating the array. Only arrays allocated using the top of the runtime can be registered with the runtime; registered arrays are visible to the bottom of the runtime via the arrays' descriptors (`GetArray`) and can only be read or written using explicit block transfers.

As mentioned above, tasks are registered with the runtime via a task table when the runtime is created. A request to run a task on multiple children can be performed in a single call to `CallChildTask`. When task f is called, the runtime calling f is passed as an argument to f, thereby allowing f to access the runtime's resources, including registered arrays, transfer functions, and synchronization with other children. Finally, there is a synchronization function `WaitTask` enabling the top of the runtime to wait on the completion of a task executing in the bottom of the runtime.

The API for the bottom of the runtime is shown in Figure 2(b). Data is transferred between levels by creating a list of transfers

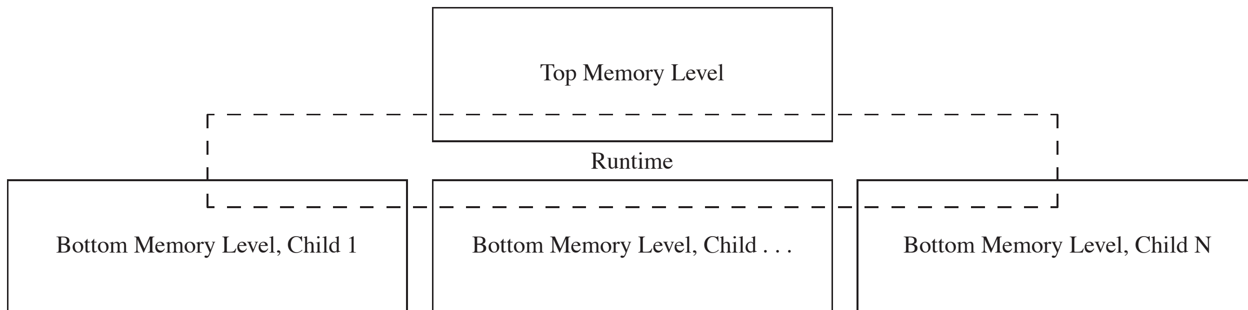

Figure 1. A runtime straddles two memory levels.

```
// create and free runtimes
Runtime(TaskTable table, int numChildren);
virtual ~Runtime();

// allocate and deallocate arrays
virtual Array_t*     AllocArray(Size_t  elmtSize,  int dimensions, Size_t* dim_sizes,
                                ArrayDesc_t descriptor, int alignment) = 0;
virtual void         FreeArray(Array_t* array) = 0;

// register arrays and find/remove arrays using array descriptors
virtual void         AddArray(Array_t array);
virtual Array_t      GetArray(ArrayDesc_t descriptor);
virtual void         RemoveArray(ArrayDesc_t descriptor);

// launch and synchronize on tasks
virtual TaskHandle_t CallChildTask(TaskID_t taskid,
                                   ChildID_t start, ChildID_t end) = 0;
virtual void         WaitTask(TaskHandle_t handle) = 0;
```

(a) Top interface.

```
// look up array using array descriptor
virtual Array_t      GetArray(ArrayDesc_t descriptor);

// create, free, invoke, and synchronize on transfer lists
virtual XferList*    CreateXferList(Array_t* dst,     Array_t* src,
                                    Size_t*  dst_idx, Size_t*  src_idx,
                                    Size_t*  lengths, int count) = 0;
virtual void         FreeXferList(XferList* list) = 0;
virtual XferHandle_t Xfer(XferList* list) = 0;
virtual void         WaitXfer(XferHandle_t handle) = 0;

// get number of children in bottom level, get local processor id, and barrier
int                  GetSiblingCount();
int                  GetID();
virtual void         Barrier(ChildID_t start, ChildID_t stop) = 0;
```

(b) Bottom interface.

Figure 2. The runtime API.

between an array allocated using the top of the runtime and an array at the bottom of the runtime (`CreateXferList`), and requesting that the given transfer list be executed (`Xfer`). Transfers are non-blocking, asynchronous operations, and the client must issue a wait on the transfer to guarantee the transfer has completed (`WaitXfer`). Data transfers are initiated by the children using the bottom of the runtime.

Synchronization is done via a barrier mechanism that can be performed on a subset of the children (`Barrier`). Children can learn their own process id's (`GetID`) and the range of id's of other children (`GetSiblingCount`).

These simple primitives map efficiently to our target machines, providing a mechanism independent abstraction of memory levels. In a multi-level system the multiple runtimes have no direct knowledge of each other. Traversal of the memory levels, and hence runtimes, is done via task calls. The interface represents, in many respects, the lowest common denominator of many current systems; we explore this further in the presentation of runtime implementations in Section 3.

2.1 System Front-end

The front-end of our system is an adaptation of the Sequoia compiler [21]. Input programs are coded in the Sequoia programming language [11], a C variant with extensions for explicit hierarchical bulk operations. The compiler (1) transforms a standard AST representation of input programs into a machine-independent intermediate representation (IR) consisting of a dependence graph of bulk operations, (2) performs various generic optimizations on this IR, and (3) generates code targeting the runtime interface described in this paper. The runtime interface provides a portable layer of abstraction that enables the compiler's generated code to run on a variety of platforms.

The compiler's generic IR optimizations span three main categories:

- locality optimizations, in which data transfer operations are eliminated from the program at the cost of increasing the lifetimes of their associated data objects;
- operation grouping, in which "small", independent operations are fused into larger operations, thereby reducing the relative overheads of the operations; and
- operation scheduling, in which an ordering of operations is chosen to attempt to simultaneously maximize operation concurrency and minimize the amount of space needed in each memory in the machine.

With the exception of the scheduling algorithms, which operate on the entire program at once, all compiler optimizations are local: they apply to a single operation at a time and affect data in either a single memory level or in a pair of adjacent memory levels. The compiler's optimizations require two pieces of information about each memory in the target machine's abstract machine model: its size and a list of its properties, specifically whether the memory has the same namespace as any other memories in the machine model (as happens in the SMP target) and whether its logical namespace is distributed across multiple distinct physical memory modules (as in the cluster target). These specific machine capabilities affect the choice of memory movement optimizations the compiler applies. For example, copy elimination is required on machines with a shared namespace to prevent unneeded transfer overhead. A per-machine configuration file provides this information. Aside from these configuration details, the compiler's optimizations are oblivious to the underlying mechanisms of the target machine, allowing them to be applied uniformly across a range of different machines and also across a range of distinct memory levels within a single machine.

Although the input programs describe a single logical computation spanning an entire machine, the compiler generates separate code for each memory level and instantiates a separate runtime instance for each pair of adjacent levels. Each runtime is oblivious to the details of any runtimes either above or below it.

3. Runtime Implementations

We implemented our runtime interface for the following platforms: SMP, disk, Cell Broadband Engine, and clusters. This section describes key aspects of mapping the interface onto these machines.

3.1 SMP

The SMP runtime implements execution on shared memory machines. A distinguishing feature of shared memory machines is that explicit communication is not required for correctness, and thus this runtime serves mainly to provide the API's abstraction of parallel execution resources and not the mechanisms to transfer data between memory levels.

On initialization of the SMP runtime a top runtime instance and the specified number of bottom runtimes are created. Each bottom runtime is initialized by creating a POSIX thread, which waits on a task queue for task execution requests. On runtime shutdown, a shutdown request is sent to each child thread; each child cleans up its resources and exits. The top runtime performs a `join` on each of the children's shutdowns, after which the top runtime also cleans up its resources and exits.

`CallChildTask` is implemented by placing a task execution request on the specified child's queue along with a completion notification object. When the child completes the task, it notifies the completion object to inform the parent. When a `WaitTask` is issued on the parent runtime, the parent waits for a task completion signal before returning control to the caller.

Memory is allocated at the top using standard `malloc` routines with alignment specified by the compiler. Arrays are registered with the top of the runtime with `AddArray` and can be looked up via an array descriptor from the bottom runtime instances. Calling `GetArray` from the bottom returns an array object with a pointer to the actual allocated memory from the top of the runtime. Since arrays can be globally accessible, the compiler can opt to directly use this array's data pointers, or issue data transfers by creating `XferLists` with `CreateXferList` and using `Xfer`'s, which are implemented as `memcpy`'s.

3.2 Disk

The disk runtime is interesting because the disk's address space is logically above the main processor's; that is, the disk is the top of the runtime and the processor is the bottom of the runtime, which can pull data from and push data to the parent's (disk's) address space. Our runtime API allows a program to read/write portions of arrays from its address space to files on disk. Arrays are allocated at the top using `mkstemp` to create a file handle in temporary space. This file handle is mapped to the array descriptor for future reference. Memory is actually allocated by issuing a `lseek` to the end of the file, using the requested size as the seek value, and a sentinel is written to the file to verify that the memory could be allocated on disk.

Data transfers to and from the disk are performed with the Linux Asynchronous I/O API. The creation of a transfer list (`XferList` in Figure 2) constructs a list of `aio_cb` structures suitable for a transfer call using `lio_listio`. Memory is transfered using `lio_listio` with the appropriate `aio_read` or `aio_write` calls. On a `WaitXfer`, the runtime checks the return status of each re-

quest and issues an aio_suspend to yield the processor until the request completes.

CallChildTask causes the top to execute the function pointer and transfer control to the task. The disk itself has no computational resources, and so the disk level must always be the root of the memory hierarchy—it can never be a child where leaf tasks can be executed.

3.3 Cell

The Cell Broadband Engine comprises of a PowerPC (PPE) core and eight SPEs. At initialization, the top of the runtime is created on the PPE and an instance of the bottom of the runtime is started on each of the SPEs. We use the IBM Cell SDK 2.1 and libspe2 for command and control of SPE resources [18].

Each SPE waits for commands to execute tasks via mailbox messages. For the PPE to launch a task in a given SPE, it signals that SPE's mailbox and the SPE loads the corresponding code overlay of the task and begins execution—SPE's have no instruction cache and so code generated for the SPE must include explicit code overlays to be managed by the runtime. Note that being able to move code through the system and support code overlays is one of the reasons a task table is passed to the runtime at initialization.

The majority of the runtime interfaces for data transfer have a direct correspondence to functions in the Cell SDK. Creating a XferList maps to the construction of a DMA list for the mfc_getl and mfc_putl SDK functions which are executed on a call to Xfer. XferWait waits on the tag used to issue the DMA. Allocation in a SPE is mapped to offsets in a static array created by the compiler, guaranteeing the DMA requirement of 16 byte memory alignment. Synchronization between SPEs is performed through mailbox signaling routines.

The PPE allocates memory via posix_memalign to align arrays to the required DMA transfer alignment. To run a task in each SPE, the PPE sends a message with a task ID corresponding to the address of the task to load as an overlay. Overlays are created for each leaf task by the build process provided by the compiler and are registered with the runtime on runtime initialization.

3.4 Cluster Runtime

The cluster runtime presents the standard two-level memory interface described in Section 1. The aggregate of all node memories is the top (global) level, which is implemented as a distributed shared memory system, and the individual node memories are the bottom (local) level, with each cluster node as one child of the top level. Similar to the disk, the cluster's aggregate memory space is logically above any processor's local memory, and the runtime API allows the local level to read/write portions of the potentially distributed arrays. We implement the cluster runtime with a combination of Pthreads and MPI-2 [24].

On initialization of the runtime, node 0 is designated to execute the top level runtime functions. Other nodes initialize as bottom runtimes and wait for instructions from node 0. Two threads are launched on every node: an *execution thread* to handle the runtime calls and the execution of specified tasks, and a *communication thread* to handle data transfers, synchronization, and task call requests across the cluster.

Bottom runtime requests are serviced by the execution thread, which identifies and dispatches data transfer requests to the communication thread, which performs all MPI calls. Centralizing all asynchronous transfers in the communication thread simplifies implementation of the execution thread and works around issues with multi-threading support in several MPI implementations.

We provide a distributed shared memory (DSM) implementation to manage memory across the cluster. However, unlike conventional DSM implementations, we need not support fully general memory or coherence. All access to memory from the bottom of the runtime must be explicit and in bulk, and the parallel memory hierarchy programming model forbids aliasing. The strict access rules on arrays give us great flexibility in strategies for allocating arrays across the cluster. We use an interval tree per allocated array, which allows specifying a distribution on a per array basis. Because of the copy-in, copy-out semantics of access to arrays passed to tasks in the Sequoia programming model, we can support complex data replication where distributions partially overlap. Unlike traditional DSM implementations where data consistency and coherence must be maintained by the DSM layer, the programming model asserts this property directly. For the purposes of this paper, we use simple block-cyclic data distributions as complex distributions are not currently generated by the compiler.

We use MPI-2 single-sided communication to issue gets and puts on remote memory systems. If the memory region requested is local to the requesting node and the requested memory region is contiguous, we can directly use the memory from the DSM layer by simply updating the destination pointer, therefor reducing memory traffic. However, the response of a data transfer in this case is not instantaneous since there is communication between the execution and communication threads as well as logic to check for this condition. If the data is not contiguous in memory on the local node, we must use memcpys to construct a contiguous block of the requested data.

When the top of the runtime (node 0) launches a task execution on a remote node, node 0's execution thread places a task call request on its command queue. The communication thread monitors the command queue and sends the request to the specified node. The target node's communication thread receives the request and adds the request to the task queue, where it is subsequently picked up and run by the remote node's execution thread. Similarly, to perform synchronization an execution thread places a barrier request in the command queue and waits for a completion signal from the communication thread.

4. Multi-Level Machines With Composed Runtimes

Because the runtimes share a generic interface and have no direct knowledge of each other, the compiler can generate code that initializes a runtime per pair of adjacent memory levels in the machine. Which runtimes to select is machine dependent and is given by the programmer in a separate specification of the machine architecture; the actual "plugging together" of the runtimes is handled by the compiler as part of code generation.

Two key issues are how isolated runtimes can be initialized at multiple levels and how communication can be overlapped with computation. In our system, both of these are handled by appropriate runtime API calls generated by the compiler. Initializing multiple runtimes is done by initializing the topmost runtime, then calling a task on all children that initializes the next runtime level, and so on, until all runtimes are initialized. Shutdown is handled similarly, with each runtime calling a task to shutdown any child runtimes, waiting, and then shutting down itself. To overlap communication and computation, the compiler generates code that initiates a data transfer at a parent level and requests task execution on child levels. Thus a level in the memory hierarchy can be fetching data while lower levels are performing computation.

For this paper, we have chosen several system configurations to demonstrate composition of runtimes. Currently available Cell machines have a limited amount of memory, 512MB per Cell on the IBM blades and 256MB of memory on the Sony Playstation 3, which uses a Cell processor with 6 SPEs available when running Linux. Given the high performance of the processor it is common

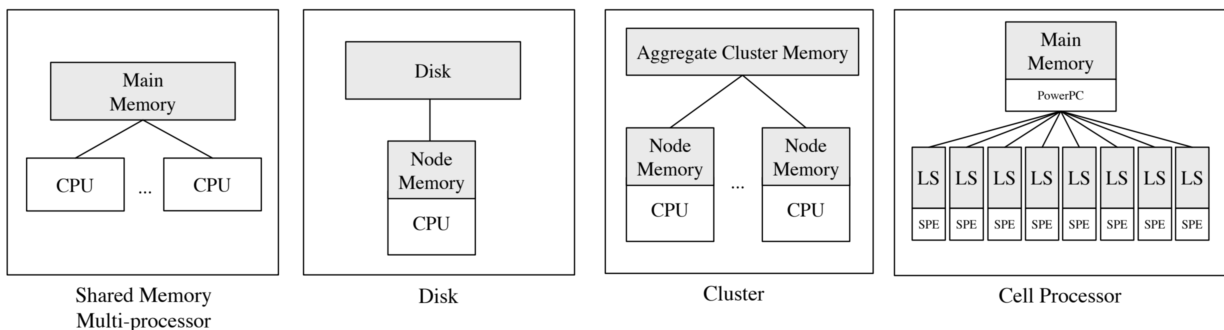

Figure 3. Hierarchical representations of our two-level configurations

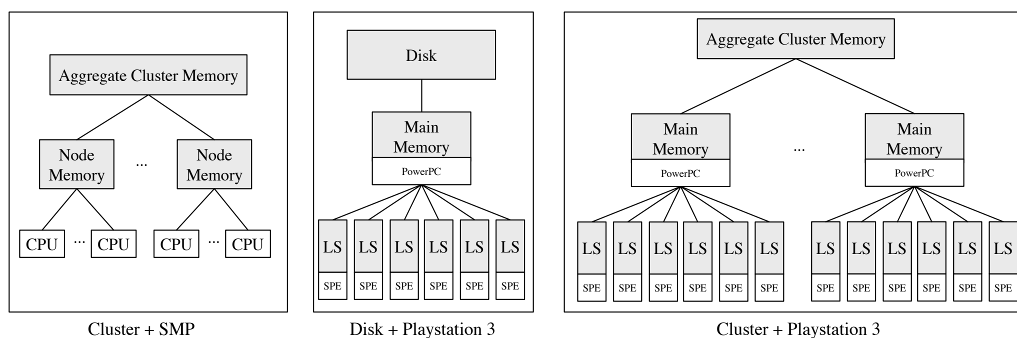

Figure 4. Hierarchical representations of our multi-level configurations

SAXPY	BLAS L1 saxpy
SGEMV	BLAS L2 sgemv
SGEMM	BLAS L3 sgemm
CONV2D	Convolution using a 9x9 filter with a large single-precision floating point input signal obeying non-periodic boundary conditions.
FFT3D	Discrete Fourier transform of a single-precision complex N^3 dataset. Complex data is stored in struct-of-arrays format.
GRAVITY	An $O(N^2)$ N-body stellar dynamics simulation on 8192 particles for 100 time steps. We operate in single-precision using Verlet update and the force calculation is acceleration without jerk [13].
HMMER	Fuzzy protein string matching using Hidden Markov Model evaluation. The Sequoia implementation of this algorithm is derived from the formulation of HMMER-search for graphics processors given in [15] and is run on a fraction of the NCBI non-redundant database.

Table 1. Applications tested with runtimes

to have problem sizes limited by available memory. With the programming model, compiler, and runtimes presented here, we can compose the Cell runtime and disk runtime to allow running out of core applications on the Playstation 3 without modification to the user's Sequoia code. We can compose the cluster and Cell runtimes to leverage the higher throughput and aggregate memory of a cluster of Playstation 3's. Another common configuration is a cluster of SMPs. Instead of requiring the programmer to write MPI and Pthreads/OpenMP code, the programmer uses the cluster and SMP runtimes to run Sequoia code unmodified.

5. Evaluation

We evaluate the cost of the generic abstraction layer using several applications written in Sequoia (Table 1. The applications are executed on a variety of two-level systems (Section 5.1) as well as several multi-level configurations (Section 5.2) with no source level modifications, only remapping and recompilation. Our evaluation centers on how efficiently the applications utilize each configuration's bandwidth and compute resources. We find that despite the uniform abstraction, we maximize bandwidth or compute resources for most applications across our configurations.

5.1 Two-level Portability

For the two-level portability tests, we utilize the following concrete machine configurations:

- The **SMP** runtime is mapped to an 8-way, 2.66GHz Intel Pentium4 Xeon machine with four dual-core processors and 8GB of memory.
- The **cluster** runtime drives a cluster of 16 nodes, each with dual 2.4GHz Intel Xeon processors, 1GB of memory, connected with Infiniband 4X SDR PCI-X HCAs. With MVAPICH2 0.9.8 [16] using VAPI we achieve ∼400MB/s node to node.[2] We utilize only one processor per node for this two-level configuration.

[2] MVAPICH2 currently exhibits a data integrity issue on our configuration limiting maximum message length to <16KB resulting in a 25% performance reduction over large transfers using MPI-1 calls in MVAPICH

	SMP	Disk	Cluster	Cell	PS3	Cluster of SMPs	Disk + PS3	Cluster of PS3s
SAXPY	16M	384M	16M	16M	16M	16M	64M	16M
SGEMV	8Kx4K	16Kx16K	8Kx4K	8Kx4K	8Kx4K	8Kx4K	8Kx8K	8Kx4K
SGEMM	4Kx4K	16Kx16K	4Kx4K	4Kx4K	4Kx2K	8Kx8K	8Kx8K	4Kx4K
CONV2D	8Kx4K	16Kx16K	8Kx4K	8Kx4K	4Kx4K	8Kx4K	8Kx8K	8Kx4K
FFT3D	256^3	512^3	256^3	256^3	128^3	256^3	256^3	256^3
GRAVITY	8192	8192	8192	8192	8192	8192	8192	8192
HMMER	500MB	500MB	500MB	500MB	160MB	500MB	320MB	500MB

Table 3. Dataset sizes used for each application for each configuration

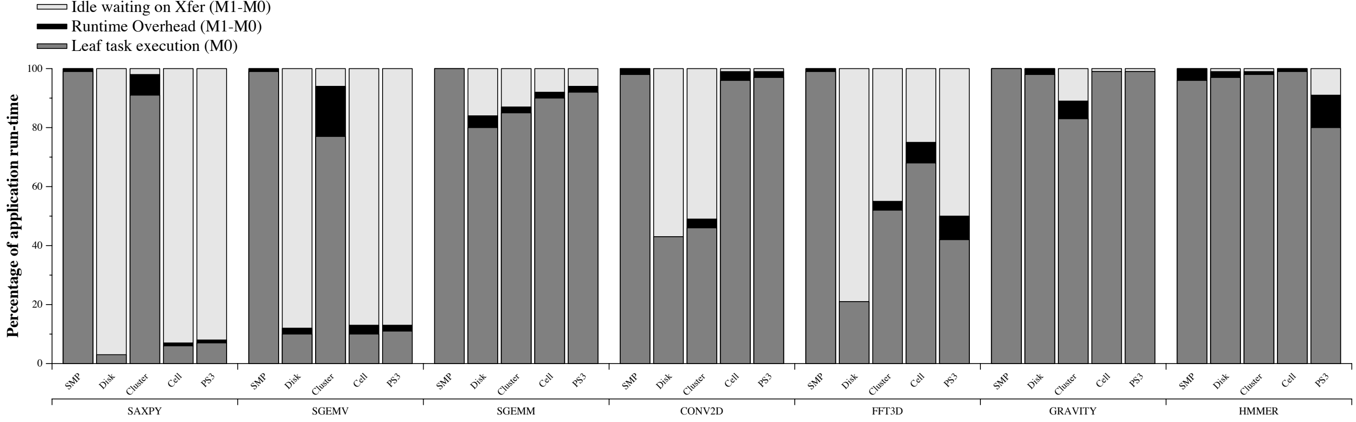

Figure 5. Execution time breakdown for each benchmark when running on SMP, Disk, Cluster, Cell, and Playstation 3 (left to right for each application)

	Baseline	SMP	Disk	Cluster	Cell	PS3
SAXPY	0.3	0.7	0.007	4.9	3.5	3.1
SGEMV	1.1	1.7	0.04	12	12	10
SGEMM	6.9	45	5.5	90	119	94
CONV2D	1.9	7.8	0.6	24	85	62
FFT3D	0.7	3.9	0.05	5.5	54	31
GRAVITY	4.8	40	3.7	68	97	71
HMMER	0.9	11	0.9	12	12	7.1

Table 2. Two-level Portability - Application performance (GFLOPS) on a 2.4GHz P4 Xeon (Baseline), 8-way 2.66GHz Xeons (SMP), with arrays on a single parallel ATA drive (Disk), a cluster of 16 2.4GHz P4 Xeons connected with Infiniband (Cluster), a 3.2GHz Cell processor with 8 SPEs (Cell), and a Sony Playstation 3 with a 3.2GHz Cell processor and 6 available SPEs.

- The **Cell** runtime is run both on a single 3.2GHz Cell processor with 8 SPEs and 1GB of XDR memory in an IBM QS20 bladeserver [17], as well as the 3.2GHz Sony Playstation 3 (PS3) Cell processor with 6 SPEs and 256MB of XDR memory [26].
- The **disk** runtime is run on a 2.4GHz Intel Pentium 4 with an Intel 945P chipset, a Hitachi 180GXP 7,200 RPM ATA/100 hard drive, and 1GB of memory.

Application performance in effective GFLOPS is shown in Table 2. Information about the dataset sizes used for each configuration are provided in Table 3. The time spent in task execution, waiting on data transfer, and runtime overhead is shown in Figure 5. In order to provide a baseline performance metric to show the tuning level of our kernels, we provide results from a 2.4GHz Intel Pentium4 Xeon machine with 1GB of memory directly calling our computation kernel implementations in Table 2. Our application kernels utilize the fastest implementations publicly available. For x86, we use FFTW [12] and the Intel MKL[19], and for the Cell, we use the IBM SPE matrix [18] library. All other leaf tasks are best effort implementations, hand-coded in SSE or Cell SPE intrinsics.

Several tests, notably SAXPY and SGEMV, are limited by memory system performance on all platforms but have high utilization of bandwidth resources. SAXPY is a pure streaming bandwidth test and achieves ∼40MB/s from our disk runtime, 3.7GB/s from our SMP machine, 19GB/s from the Cell blade, and 17GB/s on the PS3, all very close to peak available bandwidth on these machines. The cluster provides an amplification effect on bandwidth since there is no inter-node communication required for SAXPY, and we achieve 27.3GB/s aggregate across the cluster. SGEMV performance behaves similarly, but compiler optimizations result in the x and y vectors being maintained at the level of the processor and, as a result, less time is spent in overhead for data transfers. Since `Xfers` are implicit in the SMP runtime, it has no direct measurement of memory transfer time, and shows no idle time waiting on `Xfers` in Figure 5, but these applications are limited by memory system performance.

FFT3D has complex access patterns through memory. On Cell, we use a heavily optimized 3-transpose version of the code similar to the implementation of Knight et al. [21]. On the Cell blade, we run a 256^3 FFT, and our performance is competitive with the large FFT implementation for Cell from IBM [7], as well as the 3D FFT implementation of Knight et al. [21]. On the PS3, 128^3 is the largest cubic 3D FFT we can fit in-core with the 3-transpose implementation. With this smaller size, the cost of a DMA, and therefore the time waiting on DMAs, increases. Our other implementations, running on machines with x86 processors, utilize FFTW for a 2D FFT on XY planes followed by a 1D FFT in Z to compute the 3D FFT. On the SMP system, we perform a 256^3 FFT and get a memory system limited speedup of 4.7 on eight processors. We perform a 512^3 FFT from disk, first bringing XY planes in-core and performing XY 2D FFTs, followed by bringing XZ planes in-core and performing multiple 1D FFTs in Z. Despite reading the largest possible blocks of data at a time from disk, we are bound by disk access performance, with most of the time waiting on memory transfers occurring during the Z-direction

FFTs. For the cluster runtime, we distribute the XY planes across the cluster, making XY 2D FFTs very fast. However, the FFTs in Z become expensive, and we become limited by the cluster interconnect performance.

CONV2D with a 9x9 window tends to be bound by memory system performance on several of our platforms. From disk, we once again achieve very close to the maximum read performance available. On the cluster, we distribute the arrays across nodes, and thus have to read parts of the image from neighboring nodes and we become limited by the network performance. For the Cell platforms, we are largely compute limited, but because we use software-pipelined transfers to the SPEs generated by the compiler to hide memory latency, leading to smaller array blocks on which to compute, the overhead of the support set for the convolution begins to become large and limits our performance.

SGEMM is sufficiently compute intensive that all platforms start to become bound by task execution instead of memory system performance. On our 8-way SMP machine, we achieve a speedup of 6 and observe 5.3GB/s from the memory system, which is close to peak memory performance. Our performance from disk for a 16K by 16K matrix multiply is similar in performance to the in-core performance of a 4K by 4K matrix used for our baseline results. Our cluster performance for a distributed 4K by 4K matrix multiply achieves a speedup of 13. On Cell, we are largely computation bound, and the performance scales with the number of SPEs as can be seen from the performance on the IBM blade vs. the PS3.

HMMER and GRAVITY are compute bound on all platforms. The only noticeable time spent in anything but compute for these applications is GRAVITY on the cluster runtime, where there is idle time waiting for memory transfers caused by fetching updated particle locations each time-step, and HMMER on the PS3, where we can only fit 160MB of the NCBI non-redundant database in memory (the sequences were chosen at random). All other platforms run the same database subset used in Fatahalian et al. [11] for results parity, which, including the surrounding data structures, totals more than 500MB. For disk, we do not bring the database in-core, but instead load the database as needed from disk, and yet performance closely matches the in-core performance. The SMP exhibits super-linear scaling because these processors have larger L2 caches (1MB vs. 512KB) than our baseline machine. The cluster achieves a speedup of 13 on 16 nodes, or 83% of the maximum achievable speedup, with much of the difference due to load imbalance between nodes when processing different length sequences.

In general, for these applications and dataset sizes, the overhead of the runtime implementations is low. The disk and cluster runtimes are the most expensive, the disk runtime because of the kernel calls required for asynchronous I/O and the cluster runtime because of the DSM layer and threading overheads. The overheads are measured as all critical path execution time other than waiting for memory transfers and leaf task execution, and thus accounts for runtime logic, including transfer list creation and task calling/distribution, and time in barriers. The time spent issuing memory transfers is included within the transfer wait times.

The consequences of our implementation decisions for our Cell and cluster runtimes can be seen in the performance differences between our system and the custom Cell backend from Knight et al. [21] and the high-level cluster runtime from Fatahalian et al. [11]. When scaling the performance results from Knight et al. to account for clock rate differences between the 2.4GHz Cell processor used in their paper and our 3.2GHz Cell processor, we see that our runtime incurs slightly more overhead than their system. For example, for SGEMM, scaling the previously reported numbers, they would achieve 128GFLOPS whereas we achieve 119GFLOPS, a difference of 7%. For FFT, GRAVITY, and HMMER, our performance is 10%, 13%, and 10% lower, respectively,

	Cluster of SMPs	Disk + PS3	Cluster of PS3s
SAXPY	1.9	0.004	5.3
SGEMV	4.4	0.014	15
SGEMM	48	3.7	30
CONV2D	4.8	0.48	19
FFT3D	1.1	0.05	0.36
GRAVITY	50	66	119
HMMER	14	8.3	13

Table 4. Multi-level Portability - Application performance (GFLOPS) on four 2-way, 3.16GHz Intel Pentium 4 Xeons connected via GigE (Cluster of SMPs), a Sony Playstation 3 bringing data from disk (Disk + PS3), and two PS3's connected via GigE (Cluster of PS3s).

than previously reported results. This overhead is the difference between our more general runtime and their custom build environment which produces smaller code, thus allowing for slightly larger resident working sets in the SPE, more optimization by the compiler by emitting static bounds on loops, and other similar assistance for the IBM compiler tool-chain to heavily optimize the generated code.

The differences between our cluster runtime implementation and that of Fatahalian et al. [11] is in their implementation, much of the work performed dynamically is now performance at compiler time. Since we have a much thinner layer, we have less runtime logic overhead in general, and for some applications we achieve better performance as the generated code has static loop bounds and alignment hints. SAXPY, SGEMV, and GRAVITY are faster than the previous cluster runtime implementation mainly due to to these improvements. FFT3D performance is lower on our implementation as compared to their implementation due to the lower achievable bandwidth when using MPI-2 single-sided communication through MVAPICH2, as noted above.

5.2 Multi-level Portability

We compose runtimes to explore multi-level portability. By composing the cluster and SMP runtimes, we can execute on a cluster of SMP nodes comprised of four 4-way Intel 3.16GHz Pentium4 Xeon machines connected with GigE; we utilize two out of the four processors in the node for our tests. Using MPICH2 [3], we achieve ∼80MB/s node-to-node for large transfers. By composing the disk and Cell runtimes, we can overcome the memory limitations of the PS3 to run larger, out-of-core datasets from the 60GB disk in the console. Further, we can combine the cluster and Cell runtimes to drive two PS3's connected via GigE, achieving a higher peak FLOP rate and support for larger datasets.

The raw GFLOPS rates for our applications are shown in Table 4. Figure 6 shows a breakdown of the total execution time, including the task execution time in the lowest level (M0), the overhead between the bottom two levels (M1-M0), the time idle waiting on `Xfer`'s between the bottom levels (M1-M0), overhead between the top two memory levels (M2-M1), and time idle waiting on `Xfer`'s between the top levels (M2-M1). Memory system performance of the slowest memory system dominates the memory limited applications, whereas the compute limited applications are dominated by execution time in the bottom-most memory level. On all three configurations, SAXPY, SGEMV, CONV2D, and FFT3D become bound by the performance of the memory system, while GRAVITY and HMMER, which are very math intensive, are compute bound.

For SAXPY and SGEMV on the cluster of SMPs, we get a bandwidth amplification effect similar to the cluster runtime from above. Since the data is local to the node, there are no actual memory transfers, only the overhead of the runtime performing this optimization. SAXPY and SGEMV also exhibit a larger overhead for M1-M0 which can be attributed to larger scheduling differences

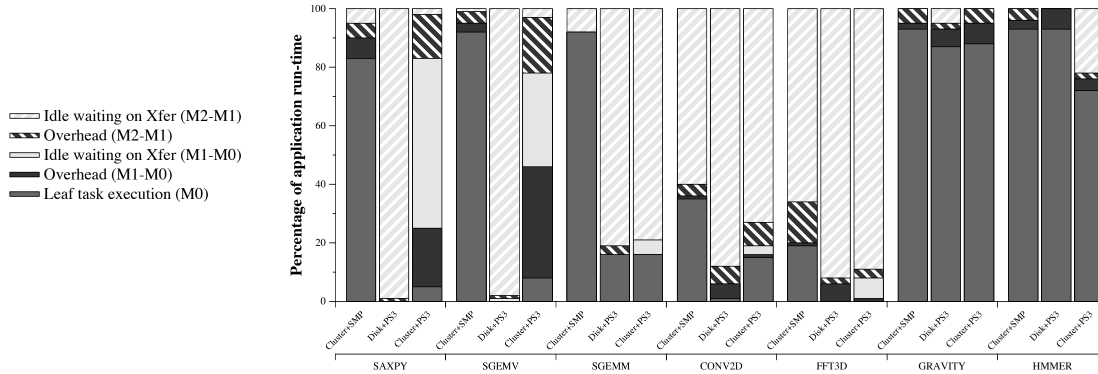

Figure 6. Execution time breakdown for each benchmark when running on Cluster+SMP, Disk+PS3, and Cluster+PS3 (left to right for each application)

and differing start and completion times of the executing tasks. CONV2D has the scaling behavior of the 8-way SMP from Section 5.1 but with the bandwidth limitations of the GigE interconnect for transmission of support regions. FFT3D becomes bound by the interconnect performance during the FFTs in the Z direction, similar to the cluster results from Section 5.1. SGEMM using 8K by 8K matrices is compute bound but we are able to hide most of the data transfer time. HMMER and GRAVITY are insensitive to the memory performance of this configuration and scale comparably to the 8-way SMP system when clock rate differences are taken into account.

By composing the disk and Cell runtimes, we are able to overcome the memory size limitations on the PS3 and handle larger datasets. However, attaching a high performance processor to a 30MB/s memory system has a large impact on application performance if the compute to bandwidth ratio is not extremely high. Only HMMER and GRAVITY achieve performance close to the in-core versions, with performance limited mainly by overheads in the runtimes. We ran HMMER with a 500MB portion of the NCBI non-redundant database from disk. As with the disk configuration from Section 5.1 for GRAVITY, at each timestep, the particles are read from and written to disk. For SGEMM, there is not enough main memory currently available to allow us to work on large enough blocks in-core to hide the transfer latency from disk and we currently spend 80% of our time waiting on disk. All the other applications perform at the speed of the disk, but we are able to run much larger instances than possible in-core.

We are also able drive two PS3's connected with GigE by combining the cluster and Cell runtimes. HMMER and GRAVITY nearly double in performance across two PS3's compared to single PS3 performance, and HMMER can run on a database twice the size. The combined runtime overhead on GRAVITY is $\sim$8% of the total execution time. For HMMER, we spend $\sim$15% of the execution time waiting on memory due to the naive distribution of the protein sequences. SGEMM scalability is largely bound by interconnect performance; with the limited available memory, we cannot hide the transfer of the B matrix with computation. FFT3D is limited to network interconnect performance during the Z direction FFTs, similar to the other platforms. SAXPY and SGEMV are bound by M1-M0 DMA performance between the SPE's LS memory and node memory as well as runtime overheads. CONV2D are largely limited by the GigE interconnect when moving non-local portions of the image between nodes.

6. Related Work

Previous efforts to design portable languages and runtimes focus mostly on two-level systems with either a uniform or partitioned global address space and fine-grain communication. Examples include Co-Array Fortran [25], UPC [6], Titanium [27], ZPL [9], and OpenMP [8]. Our runtime interface focuses on bulk communication and composability, allowing runtimes for multi-level machines to be assembled from runtimes for each adjacent pair of memory levels.

Cilk [4] provides a language and runtime system for lightweight threading, which is suited to cache-oblivious algorithms implicitly capable of using a hierarchy of memories. The execution model is fine-grain and thus efficient memory access on machines without shared memory, or virtualized shared memory, is difficult as Cilk has no notion of explicit or bulk data transfers. However, Cilk runtimes have the ability to better adapt to irregular computation load through work-stealing, which is difficult for our current implementation as the compiler statically assigns work to processors.

The Parallel Virtual Machine (PVM) [14] and MPI [23] are perhaps the oldest and most widely used systems for programming parallel machines and are supported on many platforms. Both systems concentrate on the explicit movement of data between processors within one logical level of the machine. The Pthreads library allows direct programming of shared-memory systems through a threading model and also assumes a uniform global address space. Other two-level runtime systems include Charm++ [20], Chores [10], and the Stream Virtual Machine (SVM) [22]. None of these systems are designed for handling more than two levels of memory or parallel execution in a unified way. MPI-2 [24] adds support for abstracting parallel I/O resources but uses a different API than the core communication API functions.

Our emphasis on bulk communication is shared by MPI, PVM, and SVM. Centering our API around bulk operations allows us to simply and directly map onto machine primitives such as DMAs on Cell, RDMA and other fast operations on network interconnects, and asynchronous I/O to disk systems. Also, by requiring the programmer to allocate data structures through the runtime interfaces, the runtime is able to hide alignment requirements for correctness and performance on machines such as Cell, as well as the use of distributed and remote memories, from the application.

7. Conclusion

We have presented a runtime system that allows programs written in Sequoia, and more generally in the parallel memory hierarchy model, to be portable across a wide variety of machines, includ-

ing those with more than two levels of memory and with varying data transfer and execution mechanisms. Utilizing our runtime abstraction, our applications run on multiple platforms without source level modifications and maximize available bandwidth and computational resources on those platforms.

One of the most interesting features of our design is that virtualization of all memory levels allows the user to use disk and distributed memory resources in the same way that they use other memory systems. Out-of-core algorithms using disk fit naturally into our model, allowing applications on memory constrained systems like the Sony Playstation 3 to run as efficiently as possible. Programs can make use of the entire aggregate memory and compute power of a distributed memory machine using the same mechanisms. And, despite the explicit data transfers in the programming model, through a contract between the runtime and compiler we also run efficiently on shared memory machines without any extra data movement.

All of our runtimes are implemented using widely used APIs and systems. Many systems, like those underpinning the languages and runtime systems from Section 6, could be adapted relatively easily to support our interface. Conversely, our interface and implementations are also easily adaptable for systems that use explicit memory transfers and control of parallel resources. And, although we have presented the runtime as a compiler target, it can also be used directly as a simple programming API.

There are also other systems for which it would be useful to develop an implementation of our API. For example, GPUs use an explicit memory management system to move data and computational kernels on and off the card. The BrookGPU system [5] has a simple runtime interface which can be adapted to our interface. Having an implementation of our runtime for GPUs would, in combination with our existing runtimes, immediately enable running on multiple GPUs in a machine, a cluster of nodes with GPUs, and other, more complex compositions of GPU systems. However, it should be mentioned that generating efficient leaf tasks for GPUs is nontrivial; our runtime and system would aim to solve data movement and execution of kernels on the GPUs, not the development of the kernels themselves.

Scalability on very large machines, which we have not yet demonstrated, is future work. Previous successful work on distributed shared memory implementations for large clusters can be adapted to our runtime system. Dealing with load imbalance is also a problem for the current implementation. However, since our runtimes use queues to control task execution, adapting previous work on work-stealing techniques appears to be a promising solution, but will require support from the compiler for dynamic scheduling of tasks by the runtime and consideration of the impact of rescheduling tasks on locality as discussed in Blumofe et al. [4] and explored further in Acar et al. [1]. Scaling to machines with many more processors as well as even deeper memory hierarchies is the next goal of this work.

References

[1] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *SPAA '00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 1–12, New York, NY, USA, 2000. ACM.

[2] B. Alpern, L. Carter, and J. Ferrante. Modeling parallel computers as memory hierarchies. In *Proc. Programming Models for Massively Parallel Computers*, 1993.

[3] ANL. MPICH2. `http://www-unix.mcs.anl.gov/mpi/mpich2`, 2007.

[4] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, 1995.

[5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[6] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. University of California-Berkeley Technical Report: CCS-TR-99-157, 1999.

[7] A. Chow, G. Fossum, and D. Brokenshire. A programming example: Large FFT on the Cell Broadband Engine, 2005.

[8] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.

[9] S. J. Deitz, B. L. Chamberlain, and L. Snyder. Abstractions for dynamic data distribution. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 42–51. IEEE Computer Society, 2004.

[10] D. L. Eager and J. Jahorjan. Chores: Enhanced run-time support for shared-memory parallel computing. *ACM Trans. Comput. Syst.*, 11(1):1–32, 1993.

[11] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[12] M. Frigo. A fast Fourier transform compiler. In *Proc. 1999 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, volume 34, pages 169–180, May 1999.

[13] T. Fukushige, J. Makino, and A. Kawai. GRAPE-6A: A single-card GRAPE-6 for parallel PC-GRAPE cluster systems. *Publications of the Astronomical Society of Japan*, 57:1009–1021, dec 2005.

[14] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing. Cambridge, MA, USA, 1994. MIT Press.

[15] D. R. Horn, M. Houston, and P. Hanrahan. ClawHMMER: A streaming HMMer-search implementation. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, page 11, Washington, DC, USA, 2005. IEEE Computer Society.

[16] W. Huang, G. Santhanaraman, H.-W. Jin, Q. Gao, and D. K. Panda. Design and implementation of high performance MVAPICH2: MPI2 over InfiniBand. In *International Symposium on Cluster Computing and the Grid (CCGrid)*, May 2006.

[17] IBM. IBM BladeCenter QS20. `http://www.ibm.com/technology/splash/qs20`, 2007.

[18] IBM. IBM Cell Broadband Engine Software Development Kit. `http://www.alphaworks.ibm.com/tech/cellsw`, 2007.

[19] Intel. Math kernel library. `http://www.intel.com/software/products/mkl`, 2005.

[20] L. Kalé and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[21] T. J. Knight, J. Y. Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. J. Dally, and P. Hanrahan. Compilation for explicitly managed memory hierarchies. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 226–236, Mar. 2007.

[22] F. Labonte, P. Mattson, I. Buck, C. Kozyrakis, and M. Horowitz. The stream virtual machine. In *Proceedings of the 2004 International Conference on Parallel Architectures and Compilation Techniques*, Antibes Juan-les-pins, France, September 2004.

[23] MPIF. MPI: A message passing interface standard. In *International Journal of Supercomputer Applications*, pages 165–416, 1994.

[24] MPIF. MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville, 1996.

[25] R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[26] Sony. Sony Playstation 3. `http://www.us.playstation.com/PS3`, 2007.

[27] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, Stanford, California, 1998.