# A Lightweight Streaming Layer for Multicore Execution

David Zhang[1] Qiuyuan J. Li[1] Rodric Rabbah[2] Saman Amarasinghe[1]

[1]MIT Computer Science and Artificial Intelligence Laboratory
[2]IBM T. J. Watson Research Center

## Abstract

As multicore architectures gain widespread use, it becomes increasingly important to be able to harness their additional processing power to achieve higher performance. However, exploiting parallel cores to improve single-program performance is difficult from a programmer's perspective because most existing programming languages dictate a sequential method of execution.

Stream programming, which organizes programs by independent filters communicating over explicit data channels, exposes useful types of parallelism that can be exploited. However, there is still the burden of mapping high-level stream programs to specific multicore architectures. The complexities of each architecture's underlying details makes it difficult to schedule the execution of a stream program with high performance.

In this paper, we present the specifications for an intermediate layer between the stream program and the target architecture. This multicore streaming layer (MSL) provides a common level of abstraction that facilitates efficient execution of stream programs by making it easier for compilers to manage computation, and by providing automatic orchestration and optimization of communication when appropriate. We implemented a framework for one such instance of the MSL targeted to the Cell processor and the StreamIt language and achieved greater than 88% utilization on all benchmarks with relatively small amounts of code. The framework can also be applied to other architectures and stream programming languages to enhance generality and portability.

## 1. Introduction

Multicore architectures have become the rule rather than the exception in the changing computing landscape. With single-core performance limited by power consumption, memory latency, and circuit complexity, almost all new architectures (certain mobile and embedded applications excepted) are branching into more cores rather than better cores. Exploiting parallelism has already become absolutely critical if applications wish to make full use of current and future architectures.

Compilers for traditional imperative languages are faced with a daunting task when attempting to aid the programmer in this regard: it is very difficult to automatically extract parallelism from a sequential program written in a von Neumann language such as C. Much of the time, the task of parallelizing a program remains in the hands of the programmer, who must manually convert a single-threaded sequential program into a multi-threaded parallel one. While doing so, programmers must contend with issues specific to the architectures they target, thereby limiting portability. They must also worry about race conditions and a number of other bugs that typically plague multi-threaded programs. Programmers do have access to a number of frameworks such as MPI and OpenMP to aid in their programming; however, parallelization of sequential programs remains a difficult process.

Streaming languages provide a way to alleviate the burden of manually parallelizing applications. In a streaming language, the programmer defines actors that operate on streams of data; the programmer then composes actors and streams into a program. The structure that is explicitly expressed by a streaming language exposes rather than hides the parallelism present in a program, making it much easier for the compiler to automatically extract parallelism. For the programmer, many applications fit within the streaming model and can be naturally expressed in various streaming languages.

Ideally, a compiler for a streaming language is able to focus on high-level scheduling issues: finding parallelism and scheduling actors to obtain the best possible utilization of available computation resources. However, there are generally numerous low-level issues the compiler must contend with, especially when presented with a heterogeneous, distributed-memory architecture.

The Cell multicore architecture is one such example of a heterogeneous, distributed-memory architecture. Its design is a trade-off favoring computing power, ease of manufacturing, and low power consumption at the cost of increased programming complexity. For programmers writing applications that run on Cell and similar distributed-memory architectures, they are left with additional programming complications to contend with: they not only need to effectively orchestrate parallel computation, but must carefully manage communication as well.

The goal of this paper is to create a general runtime framework for streaming applications to alleviate these programming complexities for multicores, especially distributed-memory architectures like Cell. The paper *i*) describes a multicore streaming layer (MSL) that abstracts away the architecture-specific details that otherwise complicate the scheduling of computation and communication in a stream program, and *ii*) demonstrates an implementation of the MSL for the Cell processor. We chose Cell as a test target architecture because we believe that (heterogeneous) multicore architectures with distributed memory will be prevalent in the future, as they scale better [12] than their shared-memory counterparts which are predominant today.

The proposed MSL framework is primarily geared toward compilers rather than programmers. Just as a programmer manually parallelizing a sequential program has access to frameworks like MPI that abstract certain low-level operations, the goal of the MSL framework is to provide similar functionality to streaming language compilers and schedulers that target multicore architectures.

A noteworthy aspect of the MSL is its automatic management and optimization of communication between cores. For example, a static (or dynamic) scheduler targeting the MSL can transparently benefit from double-buffering optimizations that can effectively hide communication latencies. The automatic handling of communication lowers the burden on compilers and programmers so that they are not involved in every detail of the parallel computation.

The paper makes the following contributions:

1. A specification of a general runtime framework for streaming applications.
2. An implementation of such a runtime framework for the Cell processor.
3. A dynamic scheduler implemented on top of the runtime library that dynamically schedules computation and communication for streaming programs.
4. A static scheduler implemented on top of the runtime library that statically schedules streaming computation. The static scheduler relies on the runtime library to automatically manage communication.

With our implementation for Cell, we achieve at least 97% utilization on data parallel applications, giving a reasonable 3% overhead, and at least 88% utilization on pipelined applications, giving an acceptable 12% overhead. The amount of code needed is also significantly reduced compared to programming directly at a lower level, thereby simplifying the implementation of a scheduler.

## 2. Multicore Streaming Layer

Emerging multicore architectures provide an excellent target for streaming language compilers for a number of reasons:

- Individual cores are optimized for computation, often supporting short vector operations in the form of SIMD instructions.
- Limited memory capacity on a core is not a severe limitation for streaming actors. In a stream program, actors typically embody computation, are independent of each other, have extremely local data-access patterns, and generally have small code sizes.
- The availability of high-bandwidth and low-latency on-chip communication networks enables a large number of scheduling options which are not generally feasible on other platforms such as computing clusters.

In a multicore setting, a streaming language compiler (or programmer) must address the following challenges:

1. Generating code that explicitly manages data communication (e.g., DMA operations). Architectures that provide an asynchronous communication model also require pipelining the data transfers (e.g., double-buffering) to increase efficiency and throughput.
2. For architectures with a finite local store on a core, the code, input and output buffers, and state required by the computation must be tightly packaged to fit into the local memory. This consideration is akin to locality enhancing optimizations for architectures with caches.
3. Performing high-level optimizations and scheduling to achieve a balanced distribution of work among the cores, avoiding excess communication, transforming code to improve efficiency, and ultimately delivering high processing throughput.

The purpose of the multicore streaming layer (MSL) is to abstract (1) and provide facilities that simplify (2) and (3). The MSL frees a compiler or programmer from the need to deal with the details of the architecture's communication model, allowing it to focus on exploring high-level optimization and scheduling choices. The main goal of the MSL is to provide a generic framework for controlling and dispatching computation to multicores that simplifies scheduling operations. The low-level details that are specific to individual platforms are embedded in the MSL library implementation and hidden from the programmer or the compiler. As a result, the MSL can provide a common platform for mapping streaming computation to multicores and thereby enhance portability.

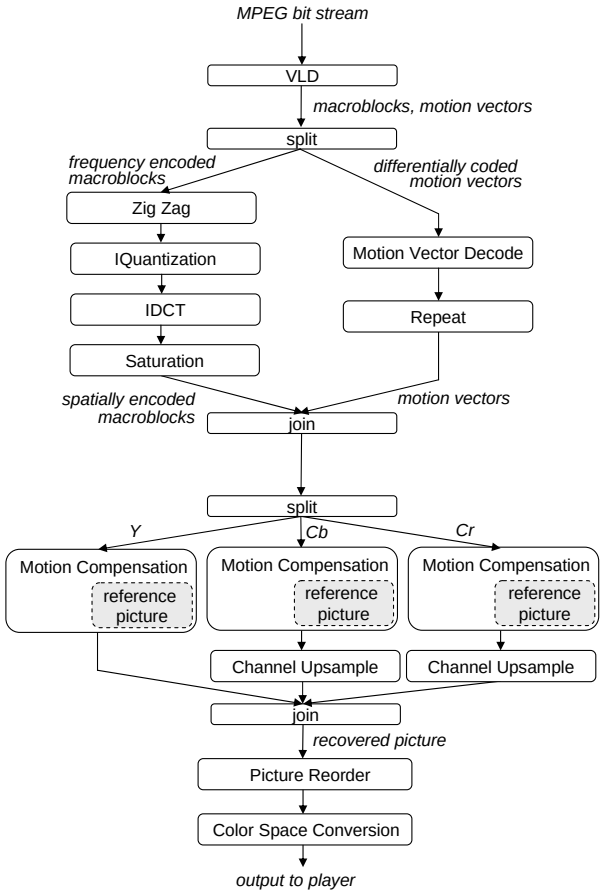


**Figure 1.** MPEG-2 stream graph.

## 3. MSL Constructs

There are a number of stream-oriented languages drawing from domains such as functional, dataflow, CSP, and synchronous programming [18]. The MSL assumes an architecture-independent programming language for high-performance stream programming. It requires that the stream program presented for execution simply consist of a dataflow graph expressing the computation. Nodes in the graph embody computation (e.g., actors, filters, kernels, some encapsulated code block), and edges imply data dependencies between input and output buffers attached to the compute nodes. An example stream graph for an MPEG-2 decoder is shown in Figure 1. Nodes in sequence expose pipeline parallelism, and nodes in parallel expose task (two branches of a split-join that contain different computation) and data parallelism (branches of a split-join that contain the same computation operate on different data).

An execution of a stream program is an ordered sequence of node firings. Each node follows a set of execution steps that consume a number of items from each input channel and produce a number of items onto each output channel.

There are two basic constructs in the MSL: *filters* and *buffers*. A filter represents a generic actor that exposes a work function which conceptually runs infinitely. Filters may be stateful and can read from multiple input buffers and write to multiple output buffers. While a filter can correspond directly to a single node in the pro-

gram, a compiler can also perform optimizations such as fusing multiple nodes into a single coarse grained MSL filter [8]. Work functions are opaque to the MSL.

*Buffers* are contiguous regions of memory that are reserved for temporarily storing input or output data. All buffers are circular, and the MSL library maintains head and tail pointers for each buffer that indicate where data begins and ends. Conceptually, a buffer has front and back ends; data toward the front of a buffer originated earlier in the execution of the program.

Conceptually, a filter consists of two major components, *code* and *state*, as well as basic properties that describe its work function such as the number of input and output buffers. *Code* is a single contiguous block of arbitrary data that may contain constant data and instructions that define multiple functions; the MSL only requires that it contain a function with a specific signature that is used as the work function. Code for a filter is intended to be a single modular component that can be easily relocated to different cores. On a distributed-memory architecture where each core has a dedicated local store (LS), code should not reference absolute addresses (e.g., absolute branches or loads) or modify itself[1].

Furthermore, code should not contain any references to mutable global variables. Instead, code should declare and access mutable state through fields that are local to a filter. *State* contains all mutable data that must be maintained across iterations of the work function. Hence, state for different filters is disjoint, and filters do not share data.

Before a filter can run on a core, it must be loaded onto the core through the MSL library. Although the filter code and state must reside on a core's LS while the filter work function is running, every filter must have a permanent store location in memory. The MSL provides facilities for loading code onto cores and copying state between local store and memory. Note that although we refer to a core's local store, the MSL concepts and constructs are applicable on shared memory multicores. The locality restrictions are generally advantageous for cache-based architectures, NUMA architectures, and distributed-memory architectures.

A user (e.g., compiler or scheduler) provides the library with the properties of the filter and the local store address of its work function; the library initializes a control block that describes the loaded filter in local store. The LS address of the control block identifies the loaded filter in all future operations until it is unloaded. If the filter is stateful, the library also copies its state into local store from its permanent store in memory. Code for the filter must be separately copied into local store through the library, but can be located anywhere as long as the correct work function address is provided to the library. When the user is done with a loaded filter, it can unload the filter through the library, causing the library to copy the filter's state back to its permanent store in memory. Stateful filters can be loaded on at most one core at any time, while stateless filters can be simultaneously loaded on any number of cores (thus facilitating coarse-grained data parallelism).

This separation of code and state allows the user additional control over how and when core local store is used. Since code is constant, the user can preload the code of a filter onto a core even while the filter is loaded on another core (and thus its state is owned by that core) in preparation for loading it on the first core in the future. If multiple (possibly stateful) filters have identical code, only one copy of it needs to reside in memory or a core's local store and it can be shared. When a filter is not being run, its code does not need to be present in core local store, leaving more space free for buffering (local store management is discussed in more detail below).

[1] These restrictions may be ignored if it is acceptable to not relocate filter code, or to pin the code to a single core.

The library provides similar facilities for allocating buffers on cores. The size of a buffer must be a power of two, to allow wrap-around computations to be done with a single bit-wise AND instruction. Buffers are identified by the LS address at which their data region begins in core local store; when allocating a buffer, the library initializes a control block located immediately before the data region that stores the buffer's head and tail pointers and participates in data transfers. The user must specify which buffers the filter refers to before a loaded filter can run.

Theoretically, the number of filters loaded and buffers allocated on a core is limited only by the size of the local store. Allocating local store is completely left to the user, allowing a scheduler to base allocation decisions on scheduling decisions. This is critically important on architectures like Cell that have severely limited local store space, where the scheduler can make much better informed allocation decisions than any other party.

Conceptually, data produced during the execution of a program is contained in exactly one buffer on one core until it is consumed. The MSL library provides facilities for moving data between buffers on different cores.

## 4. MSL Operations

The MSL defines a simple set of operations to ease the mapping of stream programs to multicores. A scheduler dispatches work items to cores by issuing MSL commands, and is notified when cores complete them. Each MSL command encapsulates a specific action to be performed, and has parameters that are specified by the user. The set of operations is divided into three main types.

- *Filter commands*: commands to load or unload filters, copy filter code into local store, attach filters to buffers, and run filters.
- *Buffer commands*: commands to allocate buffers.
- *Data transfer commands*: commands to move data between buffers in the local stores of different cores, or between local store and memory.

As an example, the filter_run command, which runs a loaded filter, takes two parameters: the LS address of a loaded filter's control block and the number of times (iterations) to run the work function. The user is responsible for ensuring that there is sufficient data in input buffers and sufficient space in output buffers for all specified iterations. Other commands have similar requirements. For a complete description of all commands, see [19].

The amount of work specified by a single command varies depending on the command parameters. Typically, work functions are small and thus filter_run commands do not take more than a few hundred microseconds to complete; some other commands, such as allocating and attaching buffers, are auxiliary commands and complete almost immediately. This allows the user to quickly change scheduling decisions and avoids tying a core into any specific long-term action.

When the user issues a command to a core, it assigns the command an ID that must be unique among all commands previously issued to that core that have not yet completed. This ID is used to notify the user when the core finishes executing the command.

### 4.1 Dependencies

In order to keep cores supplied with work at all times, it is necessary to limit round-trips between the scheduler and the cores during which the cores have no commands to execute. The MSL library provides a general facility for queuing and ordering commands on individual cores by allowing each command to specify a set of command IDs on the core on which it depends. Commands issued

to a core are queued and executed only after all dependencies have finished.

At any time, a command that has been issued to a core can be either *queued* (a command with unfinished dependencies), *active* (a command with all dependencies satisfied and currently being executed), or *completed* (a command for which all work has been done, but the user has not yet been notified). From the perspective of the user, all commands that are active on a core are run "concurrently". When a command is issued, all dependency IDs that have not been issued are considered to have already completed and are ignored.

In effect, each core maintains a small dependency graph of commands that represents a subset in time and space of the entire schedule. The scheduler (which may be user code, or a dynamic scheduler running on a control processor) continually adds commands to the dependency graph, while the core continually processes commands that have their dependencies satisfied. To make full use of a core, it is only necessary for the scheduler to ensure the dependency graph on the core is never empty. The scheduler cannot remove commands once issued, but if it keeps the dependency graph shallow in depth, it can quickly change the pattern of work done by a core simply by issuing a different set of new commands.

### 4.2 Command Groups

Each command has a small amount of data associated with it, consisting of command-specific parameters in addition to generic ID and dependency information. Typically, the user will be issuing sets of related commands at once. To avoid the overhead of issuing each command individually, the user can organize commands into groups; the library only allows entire command groups to be issued[2]. Each group specifies a sequence of commands; commands in the group are saved and can be reissued until the group is explicitly cleared.

Since core local store is managed by the user, the user must provide the library with an LS address where command data will be copied to when it issues a command group. For dependency purposes, cores treat commands in a group as having been issued in the order they appear in the group, so later commands in a group can depend on earlier ones (but not vice-versa).

### 4.3 Scheduler Interface

Commands issued to different cores are completely independent; the dependency graph on each core is strictly local. The scheduler serves as the main point of synchronization between cores by adjusting the commands it issues to a core in response to command completion notifications from all cores.

The scheduler is mainly callback-driven. It registers a callback function with the MSL library that is called whenever a command issued to a core completes. The library maintains a per-core bitmap of command IDs that have completed; the user can query this bitmap in the callback to determine which commands have completed and respond accordingly. Bits in the bitmap are set until explicitly acknowledged by the user. After an ID has been acknowledged, it can be reused for a new command issued to the core.

### 4.4 Data Transfer

Data transfer commands indirectly result in additional points of synchronization between cores. A data transfer conceptually moves data from the front of a source buffer to the back of a destination buffer, and requires two commands: a command to transfer data out of the source buffer, issued to the core containing the source buffer, and a command to transfer data into the destination buffer, issued to the core containing the destination buffer.

[2] To issue a single command, the user can create a group containing only that command.

Splitting data transfers into a pair of commands with one on each core provides the user with explicit control over when the data transfer occurs with respect to both cores. The library ensures that the transfer does not occur until both commands become active on their respective cores. The scheduler must ensure, via the dependency graphs on cores or manually on a control processor, that when a data transfer command becomes active on a core, the local buffer has sufficient data or space to fulfill the transfer.

There are no restrictions on the size of a data transfer (except for the size of the buffers involved), but the same size must be specified by both commands in the pair. Each data transfer command also specifies the address and size of the opposing buffer, since this is information the scheduler (or user) will know in advance; however, buffer head and tail pointers, which are more difficult to track in advance, are handled by the library. In addition, data transfer commands have additional inter-core requirements that the user must ensure are met across all cores.

This "decoupling" of data transfers simplifies the information the scheduler needs to keep track of. When issuing commands to one core, it usually does not need to be concerned with the state of other cores; as long as pairs of data transfer commands are eventually issued with the correct parameters and dependencies, the MSL library will automatically handle synchronization between buffers.

### 4.5 Runtime Checks

The MSL library supports a number of runtime checks that can be enabled or disabled. When enabled, the library can validate buffer accesses to ensure that they contain sufficient data/space, and can perform additional checks to ensure that issued commands are consistent. While this cannot identify all bugs in a schedule or filter work function, it can nonetheless be very useful during the development of a scheduler or programs. These checks can expose bugs that may otherwise appear non-deterministically as hung executions or incorrect output.

## 5. MSL Use-Case Examples

As an example, we will illustrate the commands required to set up and run a filter on a single core. For simplicity, we assume the filter is connected to a buffer that provides a FIFO abstraction over tapes (the input buffer is the input tape, and the output buffer is the output tape). The filter has a single input tape, single output tape, and static rates: its work function pops $i$, peeks $i + e$, and pushes $o$ bytes per iteration.

Before the filter can be run, it must be loaded, its input and output buffers must be allocated, and the filter's tapes must be attached to the buffers. The commands that perform this are illustrated in Figure 2.

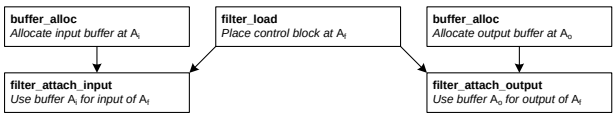


**Figure 2.** Commands to load a filter and to allocate and attach input and output buffers. Lines between commands represent dependencies that must be specified to the library when the commands are issued. These commands may be issued in one or multiple groups.

In addition, input data must be transferred into the input buffer before the filter can be run, and output data must eventually be transferred out of the output buffer. With an initially empty input buffer, the commands to transfer in $n$ iterations of input, run the

filter for $n$ iterations, and then transfer out $n$ iterations of output (assuming that the input and output buffers were sized appropriately) are shown in Figure 3.

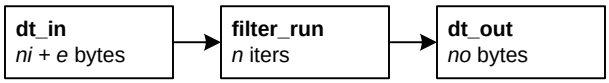


**Figure 3.** Commands to run a filter for the first $n$ iterations, including transferring input and output. The corresponding data transfer commands on other cores are not shown.

A sequence of commands is required to run the filter for a larger number of iterations on a core with a finite local store capacity. This is illustrated in Figure 4.

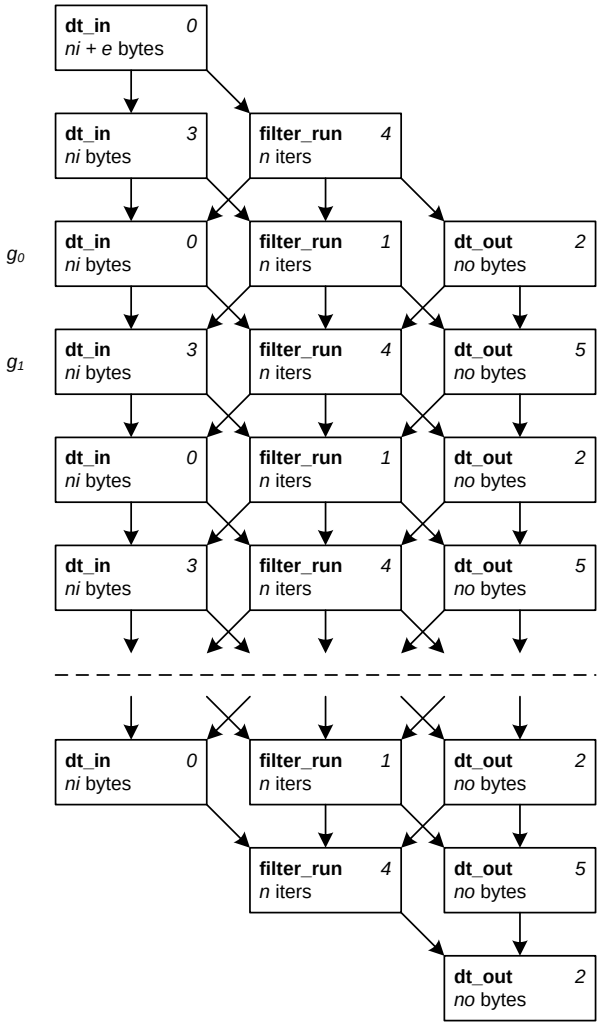


**Figure 4.** Sequence of commands to run a filter for a large number of iterations. Command IDs are indicated in the upper right. Each row is issued as a different group.

Provided that the input buffer is at least $2ni + e$ bytes and the output buffer is at least $2no$ bytes, the dependencies among the commands in the sequence ensure that:

- When a dt_in command becomes active, there are at most $ni+e$ bytes of data in the input buffer, and thus enough space to transfer in an additional $ni$ bytes.
- When a dt_out command becomes active, there are at least $no$ bytes of data in the output buffer, and thus enough data to transfer out.
- When a filter_run command becomes active, there are at least $ni + e$ bytes of data in the input buffer and at most $no$ bytes of data in the output buffer. This is enough input data and output space to run the filter for $n$ iterations.

This sequence of commands effectively "pipelines" the basic operation from Figure 3. Double-buffering is accomplished when the data transfer commands in a group complete before the filter_run does. In this case, the following filter_run has no outstanding dependencies once the current filter_run completes, and can become active immediately.

The user or scheduler can keep the core continually supplied with work by initially issuing the first two groups and thereafter issuing the next group whenever a group completes. In this case, the core almost always has two groups of commands issued, with one group active and the other queued. In addition, with the exception of the first two and last two groups, the command parameters, IDs and dependencies in every other group are identical. This allows the user to initially set up two groups ($g_0$ and $g_1$ in Figure 4) and repeatedly issue them for a majority of the execution. If executions are relatively long, the overhead of the first and last group, where no filter is being run, will be amortized effectively. Alternatively, the user can load another filter and run it during those gaps.

In practice, situations such as the above, where a static-rate filter is run for a large number of iterations and large amounts of input and output data are transferred, are very common in streaming applications. To avoid requiring the user to manually issue groups and deal with command completion callbacks in every such case, the MSL library also provides extended operations that encapsulate this pattern. In an extended operation, the user provides the library with filter rates, the addresses of opposing buffers on other processors for data transfers, and the number of iterations to run for; the library issues and responds to all commands internally and notifies the user when the entire operation is complete. Extended operations greatly simplify setting up pipelines of any length where all filters in the pipeline have static rates.

## 6. Scheduling Stream Programs Using the MSL

Streaming programs typically allow for a lot of freedom in terms of orchestrating the parallel execution of the stream graph. This freedom is afforded by the dataflow models of computation that many streaming languages are founded on. In a stream program, the rules governing the execution of a node or filter in the graph are often simple, and usually reduce to having sufficient buffering on the input to a filter, and sufficient buffering to store the output.

The execution of a stream program requires mapping and ordering filters to cores, allocating buffers, and managing data transfers between buffers. Collectively, mapping, ordering, and buffer management are embodied in a schedule of execution.

It is possible to devise a schedule statically (e.g., at compile time or graph creation time) or dynamically (e.g., during runtime). The goal of a scheduler is simply to maximize the throughput of the streaming application. In the age of multicore architectures, a scheduler will need to utilize multiple cores to increase concurrency and hence improve the throughput of a given streaming application.

A dynamic scheduler is conceptually easy to understand. The scheduler maintains an internal representation of a given stream

graph (e.g., list or priority queue). In a multicore architecture, when there is a core available, the scheduler scans its internal representation of the stream graph, and determines which filter is ready to fire. The scheduler assigns the filter to the available core. The core is also informed where the input buffer for the filter resides in memory, and where in memory to commit (buffer) the output of the filter for its successors. The scheduler then updates its internal representation to indicate the filter firings and implement a fairness policy to assure overall progress (e.g., a round-robin scheduler).

For stream graphs that are "predictable", dynamic scheduling generally does not present any advantages over static scheduling. Dynamic scheduling inevitably involves additional communication and scheduling overhead due to extra filter loading and unloading, buffer management, and scheduling computation. When all filters in a program are data-parallel, a static scheduler can make full use of all cores by simply executing each filter in turn on all cores, with a sufficient coarsening of the steady state to amortize filter load/unload and synchronization overhead. The optimal situation results when the compiler can fuse all filters into a single data-parallel filter; this produces the minimum possible communication. This situation would also be optimal for a dynamic scheduler.

Even when filters are stateful and thus cannot be data-parallelized, static software pipelining techniques [7] can make full use of cores when the compiler has an accurate static work estimator and can divide filters in a steady state evenly across cores. A single stateful filter with a heavily imbalanced work function creates a bottleneck, but dynamic schedulers are also faced with this problem.

Dynamic scheduling becomes beneficial when filters are not "predictable": when it is difficult to statically balance loads across cores, when it is difficult to estimate the amount of work done by filter work functions, or when work functions perform widely varying amounts of work through the execution of the program. In these situations, dynamic scheduling may be able to deliver better load-balancing than static scheduling.

A dynamically scheduled program can be run on varying numbers of processors without requiring recompilation or the reanalysis that complex static schedulers would need to perform, and is also tolerant of changes in the availability of processors while the program is running.

We implemented static and dynamic schedulers that use the MSL to facilitate the mapping of stream programs to a multicore. Our methodology and results are discussed next.

### 6.1 Methodology

We evaluated dynamic and static scheduling schemes for the Cell multicore architecture. The Cell processor has 9 cores [10]: a PPE that serves as a general purpose processor, and 8 SPEs that are designed to perform the bulk of the computation. Each SPE has 256KB of local store and a DMA processor to manage data in and out of the SPE. In our methodology, the schedule is run on the PPE and the filters run on the SPEs.

We used the StreamIt [1] programming language to generate the stream graphs that are presented to the MSL, and to the static and dynamic schedulers. StreamIt is an architecture-independent stream programming language that allows a programmer to focus on describing the dataflow in their algorithm, without much concern for how the computation is refined to an implementation. Stream graphs are described programmatically and algorithmically, and the program description does not commit to a specific implementation, buffering, or scheduling strategy. Rather, the program exposes the communication and its characteristics to a compiler or a scheduler that can then decide on the best implementation choices depending on the target architecture.

The basic unit of computation in StreamIt is a *filter*. There are three basic constructs for composing filters into a communicating network: a *pipeline*, a *splitjoin*, and a *feedbackloop*. A pipeline behaves as the sequential composition of all its child streams. A splitjoin is used to specify independent parallel streams that diverge from a common *splitter* and merge into a common *joiner*. A feedbackloop provides a way to create cycles in the stream graph.

The Cell backend for the StreamIt compiler maps StreamIt programs to C code that is run on the Cell processor through the MSL. The backend is built upon the robust StreamIt compiler infrastructure [7]. Briefly, StreamIt code is converted into a high-level stream IR which then undergoes a series of optimizing transformations.

### 6.2 Static Scheduler Implementation

In the case of static scheduling, the compiler generates the MSL filters, allocates the buffers, and attaches the buffers to filters. It relies on the MSL library to manage the communication thereafter. Programs are compiled in four phases.

- In the scheduling phase, a steady state schedule, which maintains a constant number of data items in each input and output buffer after every execution, is calculated based on each filter's declared I/O rates. This steady state schedule can then be run in an infinite loop. Additionally, an initialization schedule may be generated to prime the buffers or to initialize state.
- In the partitioning phase, filters are fused or duplicated to achieve the best level of granularity for the best load balancing.
- In the layout phase, filters are assigned to virtual cores on which they are to be run.
- In the code generation phase, appropriate code that encapsulates the schedule, partitioning, and layout are generated and run on the target architecture.

For our data parallel applications (i.e., stream graph that are comprised of stateless filters), our partitioning phase fuses all filters into a single filter. We generate all code, both control and filter work function code, through the backend. We then run this fused filter in parallel across all SPEs. Many real-world applications are stateless and can therefore be data-parallelized in this fashion.

### 6.3 Dynamic Scheduler Implementation

The Cell architecture's communication network provides very high memory bandwidth. The design of the dynamic scheduler assumes that memory bandwidth will never be a bottleneck, and the dynamic scheduler buffers all output produced on SPEs to memory. The scheduler performs dynamic coarse-grained software pipelining on the stream graph; if sufficient data can be buffered in all channels at all times, pipeline stalls can be avoided and all SPEs can be fully utilized. While SPE–SPE communication is more efficient than SPE–memory communication, SPE local store is generally too limited to store the buffering needed for software pipelining, and thus the scheduler never executes SPE–SPE pipelines; this avoids having to deal with work imbalances between pairs of adjacent filters. At any time, any two SPEs will typically be operating on data from different iterations of the program.

At startup, the dynamic scheduler allocates a large[3] buffer in memory for each channel; this is used to buffer the output of the upstream filter to provide input for the downstream filter. At any time, for any specific filter, the amount of data available in its input channels and amount of space available in its output channels, along with its rates, determines the maximum number of iterations that the filter can be run for.

[3] 1 MB in the current implementation, but this can be adjusted.

The scheduler selects filters to run on SPEs based on a metric computed from the maximum number of iterations and certain filter properties (see below). When a filter is selected to run on an SPE, it is scheduled for a limited but fairly large number of iterations in order to amortize the cost of loading it. Filters run for their entire allotment of iterations; however, allotments are kept small to allow the scheduler to quickly schedule another filter if necessary in response to the changing amount of data in different parts of the stream graph.

Using command completion notifications, the scheduler determines when the current filter scheduled on an SPE has almost finished running for all of its allotted iterations and selects a replacement filter for the SPE. While the current filter is still running, the scheduler issues additional commands to load the new filter, allocate its buffers, and transfer data into its input buffers from memory; this communication is overlapped with the computation done by the current filter. When the replacement filter is the same as the current filter, this additional work can be avoided. Finally, the first command to run the new filter is queued after the last command to run the current filter. When the replacement filter is selected early enough, it will be set up on the SPE before the current filter finishes running, ensuring that it can start running as soon as the current filter finishes. When the current filter has completed all of its allotted iterations, it is unloaded and can then be scheduled on another SPE.

The dynamic scheduler can run multiple instances of data-parallel filters on multiple SPEs at the same time. A data-parallel filter is still selected by the same metric as other filters; it will only be run on more than one SPE at once if it is determined to be significantly better than other filters.

The current metric implemented is very simple: it prioritizes filters based on the amount of data the state of their input and output channels allow them to consume and produce, respectively. However, the filter that is currently running on an SPE is prioritized when considered for scheduling on the same SPE; this effectively causes filters to be run for as long as possible on an SPE, with no load overhead, while no other filters are significantly better. Other more complex metrics can be easily substituted. We have yet to make a full analysis of the design of metrics and their effects on the propagation of data through the stream graph; we plan to use the dynamic scheduling framework to investigate this.

When the dynamic scheduler encounters a pipeline, the filter selection metric quickly causes all filters in the pipeline to be run sufficiently to generate some data in every channel buffer. Thereafter, the sequence of filter executions selected by the scheduler appears to perform software pipelining, although without a recognizable steady state.

### 6.4 Code Generation

For both static and dynamic scheduling cases, we must generate the code that is to be run on the multicore. In the case of Cell, we generate C code that utilizes the library and compile it with Cell's GCC. The C output from the StreamIt compiler is scalar code, and is currently not vectorized by GCC. We run all control code on the PPE and all filter initialization and work functions on the SPEs.

We set up filter description parameters specifying how many inputs and outputs a filter has, which input and output buffers the filter reads from and writes to, and how many bytes the filter reads and writes in one execution. These parameters are then used by the library to handles the necessary data transfers and executions of the work function of the filter.

In the case of dynamic scheduling, scheduling, partitioning, and layout are handled at run-time by the dynamic scheduler. Thus, the compiler need only generate the aforementioned code to set up the scheduler for execution.

In the case of static scheduling, we additionally set up filter layout parameters which specify on which SPE a filter should be run. The initialization and steady state schedules are explicit in the code: filters are loaded and run according to the schedule, and callbacks are used to set up parameters for and to run the next filter in the schedule.

## 7. Performance

We evaluate the performance of the MSL library and StreamIt compiler backend on a set of four StreamIt applications and using different scheduling methodologies. The applications are described in Table 1. For statically-scheduled benchmarks, the scheduler executes a sufficiently coarsened steady state to reduce library overhead, and imposes an explicit synchronization barrier between steady state iterations. The dynamic scheduler automatically coarsens work functions as necessary.

| | |
|---|---|
| BitonicSort | 8-element bitonic sort |
| DCT | 16x16 IEEE reference DCT |
| FFT | 256-element FFT |
| MPEG | MPEG-2 block and motion vector decoding (subset of full MPEG-2 decoder) |

**Table 1.** Benchmark applications.

The benchmarks were run on PlayStation 3 hardware, which provides only six usable SPEs. All benchmarks were scheduled to use six SPEs except for the two versions of the MPEG benchmark, which use five SPEs. Benchmarks were run for a large number of steady state iterations to reduce the effect of one-time execution startup overhead. Performance results are given in Table 2.

The *Util* column gives the average SPE utilization of the benchmark. This is the percentage of total execution time spent inside filter work functions, averaged over all SPEs. The remainder is overhead, which we divide into two categories: *library overhead* and *scheduling overhead*. *Library overhead* is time during which an SPE has active filter_run commands but is not running a work function. This type of overhead is added entirely by library code when it is either dispatching or executing other commands. *Scheduling overhead* is time during which an SPE has no active filter_run commands. During this time, the SPE has no useful work to do. It may be either *i*) waiting for filters to be scheduled or *ii*) waiting for sufficient input or output data to be transferred to allow a scheduled filter to run (i.e., due to inadequate double-buffering). When large, scheduling overhead can be viewed as resulting from the scheduling algorithm (or the nature of the program). However, a component of scheduling overhead is also due to the latency/efficiency with which the library executes commands.

The *Lib* and *Sched* columns give the average library and scheduling overhead as a percentage of total execution time, averaged over all SPEs. The *Min Sched* and *Max Sched* columns give the scheduling overhead (as a percentage of total execution time) on the SPEs with the minimum and maximum scheduling overhead; wider ranges indicate larger work imbalance resulting from the scheduling algorithm. For BitonicSort_static, the *Throughput* column gives billions of compare operations per second. For the MPEG benchmarks, the *Throughput* column gives the number of 352x240 frames processed per second. For all other benchmarks, the *Throughput* column gives GFLOPs.

The BitonicSort_static, DCT_static, and FFT_static benchmarks are statically scheduled and generated by the StreamIt compiler. These applications are fully data-parallel, and the compiler fuses the stream graph into a single filter, which is then duplicated to the number of cores. As expected from fully data-parallel programs, average utilization remains nearly the same as the number

| | Util (%) | Lib (%) | Sched (%) | Min Sched (%) | Max Sched (%) | Throughput |
|---|---|---|---|---|---|---|
| BitonicSort_static | 97.1 | 0.1 | 2.9 | 2.9 | 2.9 | 0.5 GOPs |
| DCT_static | 97.7 | 1.0 | 1.3 | 1.3 | 1.3 | 3.2 GFLOPs |
| FFT_c | 99.1 | — | — | — | — | 2.5 GFLOPs |
| FFT_static | 98.6 | 0.6 | 0.9 | 0.9 | 0.9 | 1.9 GFLOPs |
| FFT_dynamic | 88.8 | 9.2 | 2.0 | 1.2 | 2.9 | 2.2 GFLOPs |
| FFT_pipeline | 78.0 | 3.6 | 18.5 | 4.2 | 33.9 | 1.9 GFLOPs |
| MPEG_static | 95.1 | 1.0 | 3.9 | 1.5 | 8.1 | 46.9 fps |
| MPEG_dynamic | 97.8 | 1.7 | 0.5 | 0.3 | 0.8 | 48.2 fps |

**Table 2.** Benchmark performance.

of SPEs is varied from one to six, and scheduling overhead is essentially identical on all SPEs, demonstrating nearly perfectly linear speedup. The total overhead in each benchmark is less than 3%. Scheduling overhead appears to dominate total overhead, largely due to the synchronization barrier after each steady state iteration that creates repeated additional costs as the program executes.

For programs consisting of a single fused data-parallel filter, the dynamic scheduler produces identical performance as static scheduling. Results for the dynamic scheduler on these applications are not separately given.

The FFT_dynamic and FFT_pipeline benchmarks are different manual implementations of the FFT application. The FFT stream graph consists of a single pipeline with 15 filters. FFT_dynamic schedules this pipeline using the dynamic scheduler.

In FFT_pipeline, the stream graph is first manually fused into a pipeline of six filters, each of which is statically placed on a different SPE. All communication except for input and output is done directly between SPE local stores and remains entirely on-chip. GFLOPs numbers for FFT_dynamic and FFT_pipeline can be compared with each other but not directly to FFT_static. The former two have manual work function implementations that are slightly more efficient than the compiler-generated code.

For FFT_dynamic, scheduling overhead is low and very similar across SPEs. This indicates that the dynamic scheduler has no difficulties keeping all SPEs supplied with work. Moreover, the results show that if there is sufficient memory bandwidth, as is the case with the Cell architecture, it is practical to perform all buffering to memory, avoiding core-to-core communication entirely. Average utilization remains nearly identical as the number of SPEs is varied from one to six, indicating almost perfectly linear speedup.

However, the library overhead on this benchmark is high, approaching 10%, resulting in somewhat low average utilization. This overhead has two main sources: *i*) individual filters in the pipeline have much lower communication–computation ratios than the single fused filter in the FFT_static benchmark; and *ii*) the dynamic scheduler continually issues additional commands to switch filters between SPEs, which are not needed in a static schedule.

For FFT_pipeline, a single filter/SPE in the middle of the pipeline is the bottleneck. Although average utilization is low, it is not due to library overhead, which is also low. Average scheduling overhead is high and varies widely between SPEs: the bottleneck SPE had 92% utilization, while the least-utilized SPE had only 63% utilization. In general, this illustrates the difficulty of performing direct static SPE–SPE pipelining. Although SPE–SPE pipelining keeps communication on-chip, where extremely high bandwidth is available and there is no danger of exhausting comparatively limited memory bandwidth, work imbalances between filters make it difficult to fully utilize all SPEs.

For comparison, FFT_c is a hand-tuned implementation of FFT_static that does not use the MSL library. The same work function code is used as FFT_dynamic and FFT_pipeline (however, this is different from FFT_static, which is generated by the StreamIt compiler), and data transfers are fully double-buffered. Compared to FFT_c, FFT_dynamic is approximately 12% slower, most of which is due to library overhead. FFT_pipeline is significantly slower, but this is entirely due to the major work imbalance it exhibits.

MPEG has a small amount of state and cannot be fused into a single filter. The compiler fuses the stream graph down to a single stateful filter and a single stateless filter as the branches of a two-way splitjoin. When mapping this to the MSL library, both statically- and dynamically-scheduled benchmarks explicitly treat the scattering and gathering operations as separate stateless filters, resulting in four total filters to schedule.

The MPEG_static benchmark statically software-pipelines the four filters. We generated the static schedule for five SPEs by manually profiling, duplicating, and partitioning filters, and manually generating control code to issue the resulting schedule. The mapping of filters to SPEs in the partition that we obtained is given in Table 3.

| SPE | Filters |
|---|---|
| 0 | $n$ splitter, $n$ stateful |
| 1 | $6n$ joiner |
| 2 | $2n$ stateless |
| 3 | $2n$ stateless |
| 4 | $2n$ stateless |

**Table 3.** Partition for MPEG_static benchmark.

This partition exhibits a slight work imbalance: SPE 0 has slightly less work than SPE 1, which has slightly less work than the remaining SPEs. As a result, there is a small variation in scheduling overhead, resulting in somewhat lower utilization than could be achieved. The bottleneck SPEs were 98% utilized, while SPE 0 was 90% utilized. Aside from the work imbalance, which can be reduced with better partitioning, this benchmark shows that static software pipelining using the MSL library can be done with very low overhead.

The MPEG_dynamic benchmark uses the dynamic scheduler to schedule the four filters. As with FFT_dynamic, average utilization is very similar across all SPEs, and there is still nearly perfectly linear speedup as the number of SPEs is varied. Compared to the static schedule, the dynamic scheduler is not limited to executing repetitions of a steady state and does not impose any synchronization barriers. However, in order to dynamically switch filters on cores, it must issue more commands. This translates into slightly lower scheduling overhead but slightly higher library overhead. Overall, the dynamic scheduler is able to distribute work more efficiently across all available cores, resulting in higher average utilization and 2.8% increased throughput compared to MPEG_static.

It must be noted that the throughput obtained in these benchmarks is low compared to the maximum performance attainable on Cell and the performance of other implementations of the same benchmarks. No SIMD vectorization (either automatic or manual) was performed within filter work functions for any of the benchmarks; as a result, they did not take advantage of the SIMD capability of Cell SPEs. In addition, filter code could have been considerably optimized. For instance, FFT_static performs more integer operations to maintain loop counters than actual FLOPs. However, the high utilization demonstrated in the benchmarks should extend to real-world, optimized applications as long as individual filter work functions perform a comparable amount of work.

## 8. Related Work

Streaming languages provide an attractive alternative to sequential languages for many applications. In a streaming language, the programmer defines actors that operate on data streams and composes programs by connecting actors. Many common high-performance applications, especially those involving signal, audio, image, and video processing, have explicit block diagram structure and are most easily conceptualized in that manner, making streaming languages a natural choice for these applications. Because programs written in streaming languages are explicitly structured as communicating actors, they typically expose a high degree of parallelism. This allows streaming language compilers to easily analyze stream programs and efficiently parallelize them. Compared to sequential languages, streaming languages free programmers from the burden of having to explicitly write parallelized code for specific architectures, allowing them to focus on expressing the high-level semantics of the algorithm in a natural way. Recent developments in streaming languages include Brook [4], Cg [14], StreamC/KernelC [11], and StreamIt [1]. The MSL is not tied to any particular language choice and simply relies on the expression of computation as a dataflow graph.

MPI and OpenMP are arguably the most well-known and widely used parallelization frameworks for computing clusters and traditional SMP architectures. These two programming APIs represent opposite ends of a spectrum: MPI defines a language-independent, low-level network communication API, while OpenMP defines a set of high-level annotations that programs can use to direct compatible compilers to generate multi-threaded code. The multicore streaming layer and library are intended to provide the same kind of low-level functionality as MPI, with two major differences: *i*) they are more naturally suited for multicores rather that clusters, and *ii*) they are specific to streaming applications and hence provide more targeted control functionality.

The work presented here also complements parallelization frameworks that have been designed specifically for multicores or Cell; see [5] for a review. RapidMind [16] and MPI Microtask [17] follow a "task-based" approach by providing runtime systems that help schedule program-defined tasks, or kernels. Mercury's MultiCore Framework [3] adopts a similar approach. However, it is primarily designed for exploiting data parallelism in large matrix computations instead of streaming. CellSs [2] automatically generates tasks from annotations to linear code. Sequoia [6] is a language that exposes an abstract hierarchical memory model and allows the programmer to define communication and local computation; its compiler and runtime system can efficiently execute Sequoia programs on Cell. The MSL allows for the exploration of various scheduling methodologies for executing programs on multicores.

The Stream Virtual Machine (SVM) also aims to simplify compilation to diverse multicore architectures [15, 13]. As a specification, the SVM is more general than the MSL, encompassing a broader class of architectures and applications while potentially incurring higher runtime overhead. The SVM supports a rich parameterized machine model encompassing FIFO interconnects, specialized DMA processors, and hierarchical control processors, while the MSL targets regular distributed-memory machines with a single control processor and implicit DMA between processors. A complete implementation of the SVM must support automatic flow control, end-of-stream markers, and suspension/termination of filters; the MSL elides these functions because they complicate the interface and are difficult to implement efficiently. The MSL API simplifies the tracking of filter dependences by assigning a unique identifier to each *command* instance, rather than tracking the underlying filters and sometimes depending on the state of those filters (as in the SVM). Finally, the MSL was designed from the beginning to be implemented as a standalone library; while a library implementation is also possible with the SVM, the specification was influenced by an intention to employ a special-purpose "low-level compiler" to process the client code.

Gummaraju and Rosenblum use the SVM to map programs written in a streaming fashion to a general purpose multi-threaded processor [9]. Their results demonstrate that using a streaming programming model can benefit general purpose cache-based architectures. They use a scheduling model where dependencies between computation and communication are determined at compile time and scheduling is performed at runtime. However, they assume that there is a shared cache level between thread contexts, and hence do not have to deal with the general communication challenges that arise in a distributed memory multicore. The MSL and library presented in this paper offer a more general solution that can also deal with future large scale multicores, especially ones patterned after a distributed memory model.

## 9. Conclusions and Future Work

Streaming languages provide an excellent way to target new multicore architectures while placing minimal parallelization burden on the programmer. Multicore architectures such as Cell that are designed to offer high peak performance are well suited for streaming applications. This paper described a runtime framework for streaming applications on multicores consisting of *i*) a common Multicore Streaming Layer (MSL) that provides high-level primitives for schedulers, *ii*) an implementation of the MSL for an existing processor, namely Cell, *iii*) a lightweight dynamic scheduler for stream graphs, and *iv*) a static scheduler for stream graphs. The framework offers automatic management and optimization of communication, and greatly simplifies the task of a streaming language compiler or scheduler.

The real benefit provided by the framework, in particular the MSL runtime library, is that it allows a scheduler to think directly in terms of filters and how they are scheduled instead of lower-level architecture-specific details. We found that it required far less code to implement scheduling patterns on top of the library than at a lower level on Cell. The MSL library also allows for far more complex patterns to be implemented than is directly feasible at a lower level. The library running the data-parallel fused FFT benchmark produces a reasonably small amount of overhead (1.4%), and the dynamic scheduler running the pipelined version of the benchmark produces an acceptable amount of overhead (11.2%).

The MSL library currently provides two orthogonal branches that can be further developed. First, it is important to reduce the 12% overhead observed in the pipelined FFT tests involving the dynamic scheduler. This overhead is entirely due to the cost of switching between commands when many are active, and it can probably be significantly reduced by optimizing library code.

In addition, the implementation currently lacks real support for filters with dynamic rates (i.e., I/O rates that change over time and across executions). The library simply leaves the responsibility of tracking rates to the scheduler entirely. Feedback from the library

on the amount of data produced and consumed by individual filters would be very useful for schedulers; ultimately, the library should have some way of running filters with unbounded dynamic rates. The latter requires a general mechanism to suspend dynamic rate filters in the middle of executing their work functions.

The dynamic scheduler can be extended in many directions. The simplest additions involve adjusting the metric used for selecting filters to test and improve the performance of the dynamic scheduler as work becomes more and more imbalanced between filters. In addition, an important advantage of dynamic scheduling in general is the ability to react to dynamic rate filters and the runtime distribution of work in the stream graph; implementing robust support for dynamic rate filters in the stream graph would drastically increase its usefulness.

## Acknowledgments

We thank the reviewers for their help in improving the paper. We are indebted to members of the StreamIt team, both past and present, and especially Allyn Dimock, Michael Gordon, and William Thies, for their contributions to StreamIt and this work. This work is supported in part by DARPA grants PCA-F29601-03-2-0065 and HPCA/PERCS-W0133890, and NSF awards CNS-0305453 and EIA-0071841.

## References

[1] StreamIt homepage. http://cag.csail.mit.edu/streamit/.

[2] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. CellSs: a Programming Model for the Cell BE Architecture. In *Proceedings of the ACM/IEEE conference on Supercomputing*, 2006.

[3] Brian Bouzas, Robert Cooper, Jon Greene, Michael Pepe, and Myra Jean Prelle. MultiCore Framework: An API for Programming Heterogeneous Multicore Processors. Technical report, Mercury Computer Systems, Inc., 2006.

[4] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *Proceedings of the 31st International Conference on Computer Graphics and Interactive Techniques*, 2004.

[5] Alfredo Buttari, Piotr Luszczek, Jakub Kurzak, and Jack Dongarra. SCOP3: A Rough Guide to Scientific Computing On the PlayStation 3. Technical report, University of Tennessee Knoxville, 2007.

[6] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2006.

[7] Michael Gordon, William Thies, and Saman Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[8] Michael Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Christopher Leger, Andrew A. Lamb, Jeremy Wong, Henry Hoffman, David Z. Maze, and Saman Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[9] Jayanth Gummaraju and Mendel Rosenblum. Stream Programming on General-Purpose Processors. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, 2005.

[10] H. Peter Hofstee. Power Efficient Processor Architecture and the Cell Processor. *Proceedings of the 11th International Symposium on High Performance Computer Architecture*, 2005.

[11] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable Stream Processors. *Computer*, 2003.

[12] Theodoros Konstantakopoulos. *Energy Scalability of On-Chip Interconnection Network*. Ph.D. Thesis, Massachusetts Institute of Technology, 2007.

[13] Francois Labonte, Peter Mattson, Ian Buck, Christos Kozyrakis, and Mark Horowitz. The Stream Virtual Machine. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[14] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A System for Programming Graphics Hardware in a C-like Language. In *Proceedings of the 30th International Conference on Computer Graphics and Interactive Techniques*, 2003.

[15] Peter Mattson, William Thies, Lance Hammond, and Michael Vahey. Streaming Virtual Machine Specification Version 1.0. Technical report, 2004. http://www.morphware.org.

[16] Michael D. McCool. Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform. In *GSPx Multi-core Applications Conference*, 2006.

[17] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. MPI microtask for programming the Cell Broadband Engine processor. *IBM Systems Journal*, 2006.

[18] Robert Stephens. A Survey of Stream Processing. *Acta Informatica*, 1997.

[19] David Zhang. A Streaming Computation Framework for the Cell Processor. M.Eng. Thesis, Massachusetts Institute of Technology, 2007.