Reusable Software Components for Accelerator-based Clusters

M. Mustafa Rafique*, Ali R. Butt, Eli Tilevich

Department of Computer Science, Virginia Tech
2202 Kraft Drive
Blacksburg, VA 24061
Phone: +1-540-231-0489
Fax: +1-540-231-9218

Abstract

The emerging accelerator-based heterogeneous clusters, comprising specialized processors such as the IBM Cell and GPUs, have exhibited excellent price to performance ratio as well as high energy-efficiency. However, developing and maintaining software for such systems is fraught with challenges, especially for modern high-performance computing (HPC) applications that can benefit the most from leveraging accelerators. If accelerator-based clusters are to deliver on their initial promise to provide a viable and cost-effective HPC solution to researchers and practitioners, one must find a software solution to lower the barrier to entry for the average user. In this paper, we investigate how a software component based approach can be used to provide a reusable and adaptable architecture for executing HPC tasks on accelerator-based clusters. In our implementation, we leverage the lessons from the software engineering research for component-based layered architectures. Our results indicate that the complexity of developing and maintaining accelerator-based cluster software can be as effectively tamed by solid software engineering approaches as that of software in more traditional domains. Specifically, we were able to reuse 83.6% of our implementation code across different architectures and resource configurations, while achieving the overall execution performance only 1.5% off that of an optimally hand-tuned, albeit non-reusable version.

Keywords: Accelerator-based systems, heterogeneous clusters, reusable software components, mixin-layers, high-performance computing

*Corresponding author

Email addresses: `mustafa@cs.vt.edu` (M. Mustafa Rafique), `butta@cs.vt.edu` (Ali R. Butt), `Tilevich@cs.vt.edu` (Eli Tilevich)

Reusable Software Components for Accelerator-based Clusters

M. Mustafa Rafique*, Ali R. Butt, Eli Tilevich

Department of Computer Science, Virginia Tech
2202 Kraft Drive
Blacksburg, VA 24061
Phone: +1-540-231-0489
Fax: +1-540-231-9218

1. Introduction

Heterogeneous accelerator-based clusters (ABCs) are no longer a research curiosity. The impressive performance advantages provided by such clusters make them desirable computing facilities for researchers and enterprises alike [1, 2]. The heterogeneity of the hardware units of ABCs, the complexity of their API, and the ad-hoc nature of their communication interfaces confine the use of these powerful and power-efficient computing facilities to all but advanced computer users. Researchers in other fields — whose simulations and computer-models for investigating a myriad of fields, e.g., medicine, high-speed physics, etc., can benefit from such resources — are simply left out. The current state-of-the-art is such that one literally needs to be a seasoned computer scientist to be simply able to set up and use an ABC, let alone optimize and derive peak performance from one.

The combination of low-cost and performance acceleration comparable to that of a small supercomputer, but without the associated extra high maintenance and power costs [3], makes an ABC a viable and economical solution for a large class of performance intensive tasks [4]. The main obstacle to using an ABC is the complexity of programming and maintaining its software.

In this paper, we present an approach to lowering the barrier to entry for users and organizations that need to take advantage of the computing power and energy efficiency offered by ABCs. Our solution leverages the advances in component-based software engineering, in which complex software systems are constructed from reusable and adaptable components. In particular, we explore *how layered software architectures can alleviate many of the difficulties of building and maintaining component-based systems on ABCs*.

Layered architectures are a proven approach for expressing the logic of complex computer systems [5, 6, 7, 8], with several implementation techniques. Some of these techniques require specialized languages or language extensions. We have chosen an approach called *mixin-layers* [9], which provides all the benefits of layered architectures within the confines of standard C++ [10]. This design choice is influenced by the unique requirements of our target environment.

A typical ABC requires coordination of the execution of multiple heterogeneous devices connected to each other through a high-speed interconnect. Due to the ubiquity of C++, one can find a standards-compliant C++ compiler almost on any computing device and operating system. One of the advantages of C++ is its natural interoperability with C. Thus, even if a C++ compiler is not available on some esoteric device, it is always possible to write a module in C and link it with the encompassing C++ component.

We aim at demonstrating how solid software engineering principles can help address some of the biggest challenges of advanced system construction, without sacrificing high performance. To this end, we started with an optimized implementation described in prior research projects concerned with optimizing ABCs [11, 12, 13]. As is commonly the case, our initial point was an hand-coded implementation for particular deployment environment, with the resulting code not easily reusable or adaptable. We report on the results of rearchitecting this initial code into a mixin-layer-based implementation that has subsequently undergone three modifications. These modifications involved both changes to the hardware units used and their corresponding software environments.

*Corresponding author
Email addresses: `mustafa@cs.vt.edu` (M. Mustafa Rafique), `butta@cs.vt.edu` (Ali R. Butt), `Tilevich@cs.vt.edu` (Eli Tilevich)

Our results show that the layered architecture helped reduce the amount of code that had to be changed to support each modification: the majority of components (83.6% lines of code on average) could be reused as is. We were able to effectively encapsulate both reusable and custom feature implementations within separate components. Custom implementations—both heterogeneous and platform-specific—were introduced as new components, which were then plugged-into our component architecture to take advantage of the available resources of a given ABC configuration.

Based on the results of our evaluation, this work streamlines the construction and maintenance of ABCs by making the following contributions:

- A reusable and adaptable software component framework for setting up accelerator-based heterogeneous systems; our framework provides both ease-of-use and extensibility advantages.
- A prototype implementation of the framework that we have evaluated using three realistic setups with different constituent hardware devices.
- An empirical evaluation that includes both performance and software engineering metrics to confirm that our solution can harmoniously combine high performance and solid software engineering.

The rest of this paper is organized as follows. Section 2 provides the background and related work of this research. Section 3 describes the design of our framework. Section 4 illustrates an example of how the framework can be utilized. Section 5 presents our evaluations. Section 6 discusses the applicability of the developed framework. Finally, Section 7 concludes the paper.

2. Background

In this section, we discuss key enablers of our study, namely: commodity accelerators that serve as the computational platform for our research; and techniques for designing layered software architectures.

2.1. Commodity Accelerators

It is a common practice in large-scale systems, e.g., Google's infrastructure [14], Amazon's EC2 [2], etc., to leverage off-the-shelf hardware components for obtaining cheap and energy-efficient computational power. Traditionally, mostly general-purpose processors were used. However, recent commoditization of asymmetric accelerator-type processors — for example in Cell-based Sony PlayStation 3 (PS3) [15, 16] and NVIDIA GPU-based graphics engines [17, 18] — is popularizing the use of accelerators in mainstream enterprise-scale setups as well. Thus, we focus on using such accelerators in this investigation.

The major draw of accelerator engines is that they provide a much higher performance-to-cost ratio than conventional processors. Because accelerator engines are special purpose processors, their designs trade versatility for performance. As a result, an asymmetric cluster of accelerators can provide the required performance at a fraction of the setup and maintenance cost of a traditional symmetric cluster.

Unfortunately, accelerator engines are notoriously hard to program due to the complexity of their resource management [19, 20]. In particular, building an accelerator-based system requires coordinating multiple Instruction Set Architectures (ISAs), which constitute different compilation targets. Mixing accelerator engines with conventional processors is also problematic: accelerators typically have higher compute density and raw performance than conventional processors, which can lead to a performance imbalance in mixed setups. Being special-purpose processors, accelerators typically have limited on-board storage and limited – if any – support for system services such as I/O. As a result, the programmer needs to carefully orchestrate data transfer and processing between heterogeneous hardware architectures.

Other Software Engineering innovations share our goals to streamline parallel processing with computational accelerators by providing abstractions for programming heterogeneous platforms. One such example is OpenCL [21] that extends the C language with a domain-specific extension for writing compute kernels on attached devices. OpenCL facilitates the implementation of low-level accelerator functionality by providing ready-to-use subroutines that the programmer can incorporate. The goals of OpenCL are nevertheless orthogonal to ours. While our approach aims at increasing reusability by imposing a component-based methodology, OpenCL focuses on raising the abstraction level for programming accelerators. In fact, OpenCL can be used to implement the functionality of some of our reusable components.

In terms of the specific differences between OpenCL and our approach, OpenCL extends a subset of ISO C99, which does not provide any support for object-oriented programming or generic programming with templates. By contrast, our approach heavily relies on the object-oriented and generic features of C++. It is these features that enable the high levels of reusability and adaptability in our approach. Furthermore, OpenCL focuses on managing the intra-node heterogeneity by providing a unified API for multiple accelerators within the same node. Our approach focuses more on managing the inter-node heterogeneity and coordination. Specifically, we provide components to execute computational logic on heterogeneous nodes and also to coordinate their execution across the cluster.

The specific accelerator engines that we use in this study are the Cell processor [22, 23, 24] on commodity Sony PS3 nodes, and the Compute Unified Device Architecture (CUDA) [25] enabled NVIDIA GeForece 9600M GT Graphical Processing Unit (GPU) [26]. These devices are readily available at a low-cost and are representative of common industry trends, thus making the results of our investigations broadly applicable.

The Cell is a heterogeneous multi-processor chip with one general-purpose PowerPC SMT core (PPE) that serves as a front-end manager, and eight vector processors (SPEs) that are specialized for data-parallel computations, all connected via a very fast interconnect. In contrast, the GPUs do not have an internal front-end manager and the systems main processor performs that job.

The common practice for programming accelerators, especially GPUs, is to use CUDA. To facilitate programming, CUDA exposes three special language abstractions: a hierarchy of thread groups, shared memories, and barrier synchronization. The use of these abstractions is conducive to the programmer dividing her program into coarse-grain sub-problems that can be executed independently in parallel. As an additional advantage, individual sub-problems are amenable to be further divided into finer-grain slices, which can also be solved cooperatively in parallel. This arrangement leverages one of the key benefits of threads: enabling them to cooperate with each other while solving individual sub-problems. Finally, the assignment of slices to the physical processors is done by the underlying runtime, enabling flexible parallel designs in which the exact number of physical processors need not to be known until the runtime.

Despite all the above useful features of the CUDA programming abstractions, they lack the degree of modularity and encapsulation that would enable their seamless reuse in multiple programming scenarios across different applications. The goal of our work is to explore how these low-level programming abstractions can be utilized to create higher-level abstractions, which could be exposed as reusable components. Equipped with such reusable components, a programmer would be able to create accelerator-based solutions at a fraction of the current development time and costs.

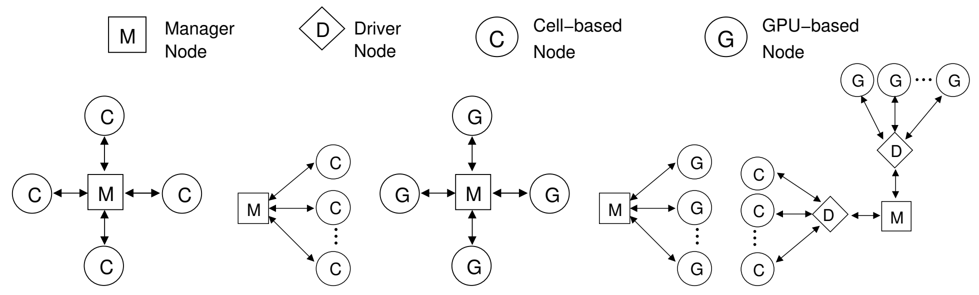
2.2. Feature-Oriented Programming and Mixin-Layers

Feature-oriented programming (FOP) is a software development methodology in which features are first-class citizens in the software architecture [27, 28]. FOP decomposes applications into a set of features that together provide the requisite functionality. Composing multiple objects from a single set of features separates the core functionality of an object from its refinements making the resulting software more reusable and robust.

In addition, FOP allows easy mix-and-match composition of features in a modular fashion, as an application is built using step-wise refinement. A common implementation strategy in FOP is to use a layered architecture, in which layers correspond to features. The resulting “feature stack” is composed of many layers with each layer (1) providing a single feature, and (2) refining existing features in the stack [29].

To incrementally refine and flexibly compose features of an ABCs, we use mixin-layers, a novel layered architectures’ implementation that uses advanced C++ programming techniques. Mixin-layers model different collaborating roles within each layer by means of inner classes [30]. Inner classes mirror the inheritance relationships of their enclosing classes in the layer above. With mixin-layers, the programmer can add functionality flexibly but systematically: each added layer supplies those inner classes that provide the required functionality.

Mixin-layers fits the design goals of this work: it implements features as collaborations of smaller, encapsulated units, each of which can be refined step-wise. As our experimental infrastructure matures, new research issues may warrant switching to another implementation strategy that better satisfies the newly-discovered set of requirements. For example, domain-specific languages have been shown to be effective for flexibly composing features [31, 32, 33]. Following a different implementation strategy, however, would still leverage the key conceptual contributions of our approach: demonstrating how a reusable and adaptable software component framework can streamline the process of composing heterogeneous ABCs.



(a) *Conf I:* Cell-based cluster. (b) *Conf II:* Cell-based n-node cluster. (c) *Conf III:* GPU-based cluster. (d) *Conf IV:* GPU-based n-node cluster. (e) *Conf V:* Multi-level heterogeneous cluster.

Figure 1: Resource configurations for Cell and GPU based heterogeneous clusters.

3. Design

To the best of our knowledge, this work is the first to explore how the functionality of an ABC can be effectively broken into reusable components, so that the required custom functionality can be clearly identified. In this section, we first present several realistic hardware accelerator configurations, and then explain how mixin-layers can effectively encapsulate reusable and custom features. Strong encapsulation promotes reuse and increases separation of concerns, both of which improve programmer productivity.

3.1. Resource Configurations

Figure 1 shows different resource configurations of heterogeneous clusters that we have considered in this work. Although not exhaustive, we believe these configurations cover most of the cases encountered in ABC design. We have used accelerators of various capabilities as the computing devices (or compute nodes) in different configurations.

Our first configuration, *Conf I* (Figure 1(a)), consists of four Cell-based accelerator nodes connected directly with the manager node via high-speed interconnect network (1 Gbps Ethernet in our case). Our second configuration, *Conf II* (Figure 1(b)), is a generalization of *Conf I* to *n* Cell-based accelerators. In these configurations, any workload assigned to the manager is dynamically divided among the attached accelerator nodes by the manager node.

Our third and fourth configurations, i.e. *Conf III* (Figure 1(c)) and *Conf IV* (Figure 1(d)), are similar to *Conf 1* and *Conf II*, respectively, but instead of Cell-based accelerators, use GPU-based computational accelerators.

The fifth configuration, *Conf V* (Figure 1(e)), employs a mix of Cell-based and GPU-based computational accelerators in a hierarchical settings. Both the Cell-based and GPU-based compute nodes are connected with the manager node through a driver node, which acts as a 'local manager' for the attached accelerator nodes. Here, any workload assigned to the manager is dynamically divided between the attached driver nodes that further divide the assigned tasks to the attached computational accelerators based on the accelerators' capabilities. In this configuration, the manager node has to interact with only two driver nodes instead of all the accelerator-based compute nodes of the cluster, thus the driver nodes alleviate from the manager node the pressure of fine-grained computational resource management and scheduling.

3.2. Design Overview

A major challenge in designing a component-based system is to decide what functionality should be included into which software components. Among the major criteria to be considered is the intuitiveness of the client interfaces and ease-of-reuse. Other criteria tend to be more domain-specific, defined by the constraints of a given computing platform. In this case, our major objective is to encapsulate reusable functionality that can serve as convenient building blocks for constructing ABCs. We aim at hiding much of the low-level implementation details from those programmers who simply want to use these software components to quickly put together a cluster of accelerator engines. In the following discussion, we outline the main software components that we chose to make available as part of our infrastructure. We support our decision to expose this particular set of functionality as software components by describing their functionality and client interfaces.

The software components are divided between manager and compute node roles. Figure 2 shows different manager and compute nodes software components that provide the execution logic of our design, and their corresponding interactions with each other. In the following we explain the functionality of these components.

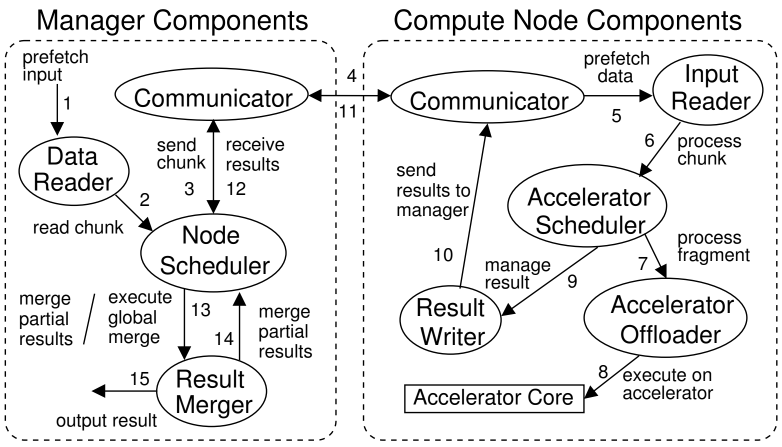


Figure 2: Manager and compute node components and their interactions.

3.3. *Manager Components*

We now describe the main components of our manager layer.

3.3.1. *Communicator*

The *Communicator* component encapsulates the methods required to communicate between different heterogeneous resources of the cluster. This component also implements communication and computation overlapping opportunities by implementing techniques such as double-buffering for each end of the communicating device. The *Communicator* is an extensible module, which can also be used to introduce hardware-component-specific optimizations such as hiding communication latency and improving bandwidth utilization.

3.3.2. *Data Reader*

This component prefetches the user specified input data and stores it in the framework according to the application requirement. The data is read in large chunks from the storage device — to amortize access costs such as disk seek times — and further divided and arranged based on application-desired layout. The newly arranged data is then passed to the *Node Scheduler* for streaming to available compute nodes. The *Data Reader* provides interfaces to manipulate the prefetched data and to change the layout as desired.

3.3.3. *Node Scheduler*

The *Node Scheduler* component manages a compute node. For each compute node available to the cluster, the manager initiates one instance of the *Node Scheduler*. Each *Node Scheduler* receives a chunk of unprocessed data from the *Data Reader*, and further optimally streams it to the corresponding accelerator-based compute node. The streaming is attuned to the compute node's data handling capabilities, and can be specified by the user in this component. The streaming process is continued until the entire input data is consumed and processed by the compute node.

The *Node Scheduler* also retrieves the results from the compute nodes, and sends the partial-results to the compute nodes for final merging, as explained next.

3.3.4. *Result Merger*

The *Result Merger* exposes the interfaces to the programmer for including application-specific mechanisms for merging partial results from individual compute nodes into a combined result set, and the global merging criteria for producing the final results at the manager node. The *Node Scheduler* retrieves and passes on the partial results from compute nodes to the *Result Merger* as the results become available. The *Result Merger* combines the partial results based on the application-specific criteria, e.g., in-order sort, and perform the specified global-merge function to produce the final results. Note that, if needed, the *Result Merger* may also use the *Node Scheduler* again for offloading the combining and merging operations to the accelerator-based compute nodes to improve performance.

3.4. Compute Node Components

We now describe the main components of the compute nodes layer as shown in Figure 2, and how data is manipulated between the different components at the compute nodes.

3.4.1. Input Reader

The *Input Reader* at the compute node prefetches blocks of data from the manager, and passes it to the *Accelerator Scheduler* component. We have used multiple buffers for reading the data from the manager to overlap the communication and computational latency, as well as provided support for specifying more optimizations as necessary. If an empty buffer is available, the *Input Reader* initiates a request for another block of data. If no empty buffers are available for the input data, *Input Reader* simply waits until some buffers become free.

3.4.2. Accelerator Scheduler

The *Accelerator Scheduler* component schedules the data read from the *Input Reader* for execution on the attached computational accelerator of the node. This component is specific to the kind of attached accelerator of the node, and provides the opportunity to implement any device specific optimization. The input buffer from the *Input Reader* is further divided into small slices suitable to fit into the available memory of the corresponding accelerator. These small slices of data are then scheduled to be streamed to the *Accelerator Offloader* component for execution on the accelerator.

3.4.3. Accelerator Offloader

The *Accelerator Offloader* component provides an interface to execute the device specific offload routines, such as application specific data processing and merging functions. This component provides abstractions for the methods required to integrate device-specific programming languages, such as C for CUDA, and C for PS3, with a general-purpose language such as C++. The application and accelerator specific compute routines can be specified in separate files to maintain the modularity of the framework.

The *Accelerator Offloader* reads the input data from the *Accelerator Scheduler* and processes it on the attached *Accelerator Core*. The *Accelerator Core* represents the specific accelerator device, such as Cell or GPU, which executes application specific task on the given data and produces the result. The results are then returned back to the *Accelerator Scheduler* that passes them to the *Result Writer* component.

3.4.4. Result Writer

The *Result Writer* component receives the results from the *Accelerator Scheduler*, and sends them to the manager. Once a block of result data is sent to the manager, its associated buffer is marked as free and can be reused by the *Input Reader*. Note that the result from the *Accelerator Offloader* can not be sent directly to the *Result Writer* component without involving the Accelerator Scheduler component. The *Accelerator Scheduler* component should be notified when the processing of a particular data slice is completed at the *Accelerator Offloader* component so that the next data slice can be scheduled. If results are sent directly to the *Result Writer* component without involving the *Accelerator Scheduler* component, then some signaling mechanism needs to be implemented to notify the *Accelerator Scheduler* component that the processing of a particular data slice has been completed, which would increase the synchronization overhead between components.

3.4.5. Summary

In this section, we have shown how the functionality of ABC can be decomposed into distinct software components. The goal is to design a component for each of the manager and client functions, such that the resulting components are easy-to-use and modify, thus yielding a modular and flexible design. These software components are reusable and can be maintained with minimal effort. Furthermore, the components can be used to design heterogeneous ABCs comprising different types of accelerators.

4. Illustrative Example

In this section, we present an illustrative example of a real cluster to describe how different components of our design operate and interact with each other. By describing each component's functionality, this example illustrates how our framework orchestrates pre-existing software components to complete a concrete computational task. In

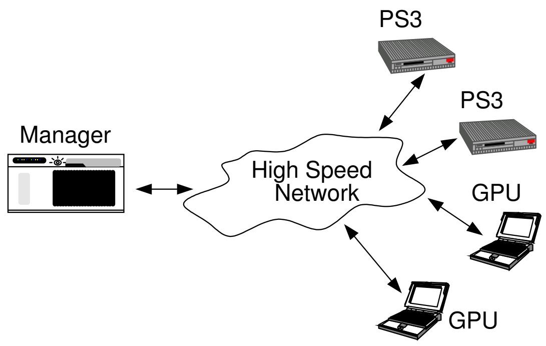


Figure 3: An example cluster comprising of PS3 and GPU nodes.

particular, we discuss a *Word Count* application, and show how the computationally-intensive problem of counting word frequencies in text files can be naturally decomposed for efficient execution on an ABC.

Figure 3 shows our heterogeneous cluster used for this example, which consists of two PS3 compute nodes, two GPU-based compute nodes, and a manager, all connected via a high-speed network.

4.1. Synthesizing the Application

First, we discuss the software components needed for the *Word Count* application. Programming-wise, the entire functionality of each cluster node is encapsulated in a template instantiation of the required components for the node. For our example application, the template instantiation for the manager node is as follows:

```
typedef Manager<WordCount> manager;

#define NUM_PS3          2
#define NUM_GPU          2

manager::DataReader               datReader;
manager::NodeScheduler<PS3>       PS3Schedule[NUM_PS3];
manager::NodeScheduler<GPU>       GPUSchedule[NUM_GPU];
manager::ResultMerger             resMerger;

void main () {
  datReader.startReadingThread();
  for (int i = 0; i < NUM_PS3; ++i) {
    PS3Schedule[i].startSchedulingThread();
  }
  for (int j = 0; j < NUM_GPU; ++j) {
    GPUSchedule[j].startSchedulingThread();
  }
  resMerger.doMerging();
}
```

The corresponding template instantiation for a PS3 compute node is as follows:

```
typedef PS3<ComputeNode<WordCount>> cNode;

cNode::InputReader                inpReader;
cNode::AcceleratorScheduler       accSchedule;
cNode::AcceleratorOffloader       accOffloader;
cNode::ResultWriter               resWriter;

void main () {
  inpReader.startReadingThread();
  accSchedule.startSchedulingThread();
  accOffloader.startOffloadingThread();
  resWriter.doWriting();
}
```

Finally, a GPU-based compute node's template instantiation is as follows:

```
typedef GPU<ComputeNode<WordCount>> cNode;

cNode::InputReader              inpReader;
cNode::AcceleratorScheduler     accSchedule;
cNode::AcceleratorOffloader     accOffloader;
cNode::ResultWriter             resWriter;

void main () {
  inpReader.startReadingThread();
  accSchedule.startSchedulingThread();
  accOffloader.startOffloadingThread();
  resWriter.doWriting();
}
```

Note the functionality of each component as described in Section 3, e.g., *Input/Data Reader*, is expressed as an inner class in each of the above template instantiations.

Figure 4 illustrates how different software components are represented as mixin-layers, both for the manager node as well as for the Cell and GPU-based compute nodes. Each layer represents a component, defined as a unit of functionality with multiple roles. For example, the `ComputeNode` component defines the `InputReader`, `Scheduler`, `Offloader`, and `ResultWriter` roles. These roles define the distinct operations that are used during the execution of a generic `ComputeNode`. Each layer is added to the composition to either refine or extend the existing components. For example, the `GPU` or `PS3` components add the functionality in their roles that are specific to their respective architectures.

Implementation-wise, each layer is implemented as a template C++ class, whose inner template classes comprise the layer's roles. Both the main component classes and their roles participate in the inheritance relationship with the corresponding classes in the layer above. Thus, to reuse a component, with all its roles, the programmer only has to include that component into a template instantiation. As long as the component has the needed roles (which can be ensured by following careful design practices), its functionality becomes immediately available for constructing any application.

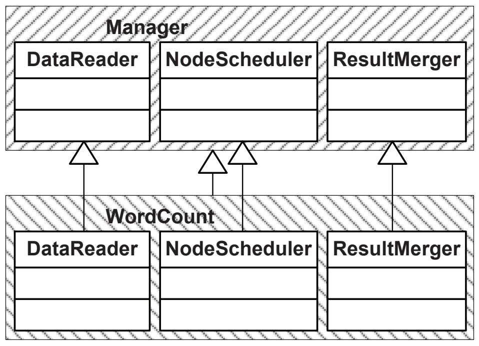
For the *Word Count* application whose code listings appear above, two out of three components for the compute nodes can be reused out-of-the-box. In the figure, the reused components are colored (shaded) identically. Even though the reusable components will need to be recompiled for different hardware architectures, their functionality will remain the same. This small but realistic example demonstrates how a layered software architecture can be leveraged to provide easy-to-use-and-reuse software components, which can be both architecture independent or device specific. This observation leads us to believe that following this software construction paradigm has the potential to alleviate many implementation complexities for the average programmer.

4.2. Runtime Interactions

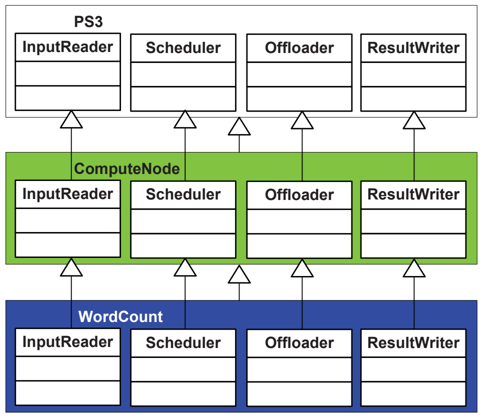
Next, we describe how the components we have described above are used at runtime. The execution control flow steps in this discussion are illustrated in Figure 2 as numbers along the arrows.

The cluster receives a request to start computing the frequencies of each word in a set of disk files. This causes the *Data Reader* component, located at the manager node, to be invoked (Step 1). The *Data Reader* prefetches and divides the input text file in small chunks. As chunks are read into memory, a separate *Node Scheduler* component for each compute node is instantiated at the manager node, a total of four in this example. Each *Node Scheduler* component reads the next available chunk from the *Data Reader* (2), and divides it into smaller blocks: 4 MB blocks for PS3 nodes, 12 MB blocks for our GPU nodes. Then, the *Communicator* transmits the scheduled data blocks to their target compute nodes (3, 4). This process is repeated until the computation is complete.

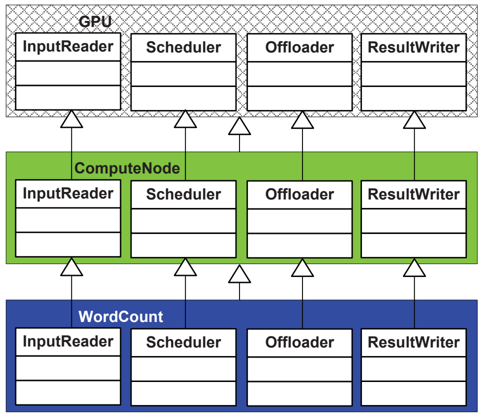
Once a compute node is done with counting different word frequencies in its assigned block, the result is sent back (11, 12) to the manager via the node's associated *Result Writer*. At the manager node, the *Result Merger* combines all the received word lists by sorting them as required for Word Count (13). When the *Result Merger* is done with sorting, the combined word lists must be processed for combining repeated word counts to determine final word frequencies. Since the counting of repetitions is computationally intensive, the work must be once again distributed among the compute nodes (14). As before, this distribution task is accomplished by the *Node Scheduler*. The final consolidated result is then computed by the *Result Merger* component after all the compute nodes have finished their computations (13).



(a) Mixin-layers for manager node.



(b) Mixin-layers for Cell-based compute node.



(c) Mixin-layers for GPU-based compute node.

Figure 4: Manager and compute nodes mixin-layers and the defined roles.

On the compute node, the *Input Reader* is responsible for retrieving the assigned blocks from the manager (5). The received blocks are then passed to the *Accelerator Scheduler* (6) in a loop. The *Accelerator Scheduler* is unique to each accelerator engine. Specifically, these components encode the logic required to divide the assigned blocks into slices that can be processed by the underlying architecture — 32 KB for PS3 and 256 KB for our GPUs. Each slice is then passed to the *Accelerator Offloader* component (7), which then either counts the different word frequencies in the assigned slice initially, or counts repeated words in a list for merging repetitions later (8).

Once the *Accelerator Scheduler* has received all the results computed for each data slice, they are passed to the *Result Writer* (9), which in turn sends them back to the manager node by means of the *Communicator* (10). The results are then reported to the user (15), completing the application run.

Finally, if the hardware configuration is changed, the programmer can easily reuse many of the software components, thus saving time and reducing time-to-solution.

5. Evaluation

In this section, we first briefly describe our prototype implementation, followed by the description of our experimentation testbed and the benchmarks that we have used to evaluate our designed components for reusability and performance, and then give the results.

5.1. Implementation

We have implemented a prototype of our design as lightweight libraries for each of the platforms, i.e., x86 on the manager and driver, PowerPC and SPE on the Cell-based PS3 compute nodes, and GPU-based x86 compute nodes using only about 1650 lines of C/C++ and CUDA code. The libraries provide the application programmers with necessary constructs for using different components of the framework.

In our implementation for *Conf V*, we leverage the reusability of the components to build a hierarchical ABC. For instance, the driver node is primarily composed of components reused from the manager and the compute nodes, i.e., the driver instantiates the same `InputReader` as that used for the compute node in the remaining configurations for reading the data from the manager. Moreover, the `NodeScheduler` and `ResultMerger` on the driver are instances of code designed for the manager in the other configurations and is used to manage the attached computational accelerators and merge the partial results. Finally, the `ResultWriter` is similar to that of the compute nodes, and is used to return the results back to the manager.

5.2. Experimental Setup

Our testbed consists of several Sony PS3s and GPU enabled Toshiba Qosmio laptop computers, a manager node, and an 2-node standard multi-core cluster to serve as drivers. All components are connected via 1 Gbps Ethernet. The manager has two quad-core Intel Xeon 3 GHz processors, 16 GB main memory, 650 GB hard disk, and runs Linux Fedora Core 8. The driver nodes are identical to the manager except that they have 8 GB of main memory. The PS3 is a hypervisor-controlled platform, and has 256 MB of main memory (of which only about 200 MB is available for applications), and a 60 GB hard disk. Of the 8 SPEs of the Cell, only 6 SPEs are visible to the programmer [34, 35] in the PS3. Moreover, each PS3 node has a swap space of 512 MB, and runs Linux Fedora Core 7. Each GPU enabled Toshiba Qosmio laptop computers has Intel Dual-Core 2 GHz processor, and 4 GB of main memory installed in it. Moreover, each of the laptops has one GeForce 9600M GT CUDA enabled GPU device, with 32 cores and 512 MB of memory, and uses CUDA toolkit 2.2.

For our experiments, we distribute the resources as described in section 3.1. For *Conf II* and *Conf IV* we set *n=10*. In *Conf V* the two drivers have four PS3s and four GPUs connected to them, respectively, and are connected with the manager node. Our goal is to determine the effect of different heterogeneous environments in our prototype implementation, specifically, varying the number, type, and hierarchy of the accelerators.

5.3. Methodology

We conducted the experiments using our prototype implementation that uses the mixin-layers for building high-performance ABCs. The focus is on evaluating our design decisions and to investigate how well we can reuse the mixin-based components in different benchmarks applications, and how well it performs as compared to a hand-tuned implementation of the benchmark applications.

We have used a number of well-known parallel applications in this study. Each of these applications has different computation to communication densities. These applications are commonly used in scientific computing environments, including image segmentation, epidemiology, statistical analysis, and environmental science [36, 37, 38] and have characteristics that are representative of many parallel applications. Specifically, these applications include the following:

- *Linear Regression:* This application takes as input a large set of 2D points, and determines a line of best fit for them.
- *Word Count:* This application counts the frequency of each unique word in a given input file. The output is a list of unique words found in the input along with their corresponding occurrence counts.
- *Histogram:* This application takes as input a bitmap image, and produces the frequency count of each color composition in the image.
- *K-Means:* This application takes a set of points in an N-dimensional space and groups them into a set number of clusters with approximately an equal number of points in each cluster.

5.3.1. Mixin-Layer Components Reusability

We evaluate the effectiveness of our framework in reducing the amount of software-engineering effort for designing applications for the targeted asymmetric hardware resources. One potentially confusing issue is the meaning of the term *component*. In a mixin-layer composition, a component is a template class whose functionality is defined by its inner classes. What is more important for this evaluation is our unit of reusability. Even though the unit of reusability is an entire mixin-layer component, any instantiation can use only the needed roles by simply creating objects of

Component (Class)	Linear Regression LOC	Code reuse (in %)		
		Word Count	Histogram	K-Means
`Communicator`	360	100	100	100
`DataReader`	42	14.2	11.9	42.8
`NodeScheduler`	423	100	100	100
`ResultMerger`	38	38.9	40.1	27.1

Table 1: Manager classes and corresponding reusable LOC (wrt. *Linear Regression*) for benchmark applications in *Conf I*.

Component (Class)	Linear Regression LOC	Code reuse (in %)		
		Word Count	Histogram	K-Means
`InputReader`	92	100	100	100
`AcceleratorScheduler`	78	100	100	100
`AcceleratorOffloader`	148	15.4	19.5	35.8
`ResultWriter`	102	100	100	100

Table 2: Compute node classes and corresponding reusable LOC (wrt. *Linear Regression*) for benchmark applications in *Conf I*.

the appropriate inner classes. Therefore, each role can be reused independently of the other roles in the same layer. Therefore, while our implementation is component-based, our measurements are specific to the software engineering metrics (e.g., lines of code) of the inner classes.

Code reusability for different applications. In this set of experiments, we evaluated how well we can reuse the mixin-layer components across our benchmark applications while keeping the cluster hardware configurations and resources fixed. Table 1 shows the total number of lines of codes (LOC) of the major classes at the manager node for *Linear Regression* along with the percentage of LOC that we were able to reuse across different applications, all in *Conf I*. Note that except for the `DataReader` and `ResultMerger` classes that are unique to different applications, all the other classes, i.e., `NodeScheduler` and `Communicator` are reused by the benchmarks without any modifications. Nonetheless, since we have used modular programming in our implementation, we were able to use some parts of our code in the application-specific user-defined classes as well. Overall 90.7% of the code of the classes at the manager node is reused between different applications in *Conf I* of our implementation.

Table 2 shows the total number of LOC of the major classes at the compute nodes for *Linear Regression* in *Conf I*. Here, except for the `AcceleratorOffloader` class, which contains the application specific routines executed on the computational accelerators of the compute nodes, all the other major classes, i.e., `InputReader`, `AcceleratorScheduler`, and `ResultWriter` are reused without any modifications. Our evaluation shows that overall 64.7% of the code of the classes at compute nodes can be used across different applications. However, the `AcceleratorOffloader` class can not be reused as a whole since it is unique for each application, even so by using the mixin-layers abstractions we are able to use 23.6% on average across all benchmark applications.

Code reusability for different configurations. In this set of experiments, we fix the application (*Linear Regression*) and vary the hardware configuration to determine how well the mixin-layer components can be reused across different hardware. Table 3 shows the total number of LOC of the major classes at the manager node in *Conf I* along with the percentage of the LOC that we were able to reuse for *Conf II*, *Conf III*, *Conf IV*, and *Conf V*, respectively. As observed from the table, we were able to use all of the classes of the manager except the `NodeScheduler` class. This is because `NodeScheduler` interacts with the attached accelerator-based compute nodes that change across the configurations. However, we were still able to reuse 52.2% of our code because of our component-based development approach. Note that in *Conf V*, we have reused 99.3% of the overall code since the driver node is composed of some of the roles of the manager and the compute nodes. The node scheduler component in *Conf V* is exactly the same as the manager node, since the driver node manages the attached nodes in the same fashion as the manager node manages the compute node. The difference in *Conf V* comes in the `DataReader` component, where the driver node reads the data either from the manager, or from a distributed storage. In the case of reading the input from a distributed storage, the driver reads only the part of input that is assigned to it by the manager node. Another difference in the driver comes in `ResultMerger` component, where in addition to merging the results, it sends the merged results to the manager node.

Table 4 shows the total number of LOC of the major classes at the compute nodes in *Conf I* for *Linear Regression* benchmark, and the LOC that we were able to reuse across different hardware configurations. Here, we have reused

Component (Class)	*Conf I* LOC	Code reuse (in %)			
		Conf II	*Conf III*	*Conf IV*	*Conf V* wrt. *I* and *III*
`Communicator`	360	100	100	100	100
`DataReader`	42	100	100	100	90.4
`NodeScheduler`	423	100	52.2	52.2	100
`ResultMerger`	38	100	100	100	94.7

Table 3: Manager components and corresponding reusable LOC for *Linear Regression* benchmark in different configurations.

Component (Class)	*Conf I* LOC	Code reuse (in %)			
		Conf II	*Conf III*	*Conf IV*	*Conf V* wrt. *I* and *III*
`InputReader`	92	100	62.5	62.5	98.1
`AcceleratorScheduler`	78	100	71.5	71.5	99.1
`AcceleratorOffloader`	148	100	20.5	20.5	100
`ResultWriter`	102	100	85.2	85.2	98.6

Table 4: Compute node components and corresponding reusable LOC for *Linear Regression* benchmark in different configurations.

100% of our code while moving from *Conf I* to *Conf II*, and also from *Conf III* to *Conf IV*, as the only difference here is the number of compute nodes. Interestingly, we reused 54.9% of our overall compute node code while transforming our code from *Conf I* to *Conf III*. This is because different type of accelerators offer different optimization opportunities for the corresponding type of attached accelerator, and we have to modify our accelerator code base for the new accelerator to exploit the specific optimization opportunities available. Note that most of the source code that has to be modified for supporting a new accelerator is in the platform specific `AcceleratorOffloader` class, which as discussed earlier, contains the application-specific functions that are executed on the accelerator of the compute node. Also, note that while reporting the code reusability for *Conf V*, we compare it with the *Conf I* and *Conf III* since the compute nodes in *Conf V* are the combination of the two configurations.

5.3.2. Benchmark Performance

In this set of experiments, we focus on the performance of the mixin-layers based code generated for our studied ABC configurations. First, we compare our code with available hand-tuned implementations [12, 13] of the benchmarks. We have extended our hand-tuned implementations [12, 13] from Cell-based compute nodes to GPU-based compute nodes. We have implemented several optimizations in our hand-tuned implementations. These optimizations include implementing locality-aware data distribution between compute nodes, implementing optimal offloading workload size for Cell and GPU based nodes, and implementing double buffering at each communication layer (manager-to-driver and driver-to-compute node) to overlap computation with communication.

Table 5 shows the comparison of the execution time for both implementations using *Conf V*. Note that the hand-tuned implementation is unique for each benchmark application, and is not reusable across different applications. The result shows that the performance of our prototype implementation is comparable to the performance of the hand-tuned implementation. Overall, our mixin-layer based implementation has an average overhead of 1.5% across all the benchmark applications as compared to the hand-tuned optimized implementation, which is a reasonable overhead considering the ease-of-reuse and adaptation advantages provided by our approach.

Next, we study how our implementation performs across our studied configurations. Table 6 shows the execution time of our benchmark applications under different resource configurations. We have used the input size of 512 MB for our benchmark applications for each of the resource configurations. The result of executing different applications of various computational densities shows that our implementation scales well with increasing the same kind of accelerator-based compute nodes, as well as using them in a highly heterogeneous environment. Note that between *Conf 1* and *II* (and *Conf III* and *IV*), only the number of accelerator-based compute nodes are increased. In both these scenarios, we increase the number of compute nodes by a factor of $(10/4 =)2.5$, and observe the average speedup by a factor of 2.2 in our prototype implementation. We also observe an increase in the manager workload in distributing the given input and merging the received results from the compute nodes, when we increase the number of compute nodes in *Conf II* and *IV* as compared to *Conf I* and *III*, respectively. Similarly, we observe a linear increase in execution time for the benchmark applications in *Conf V* as compared to *Conf I* and *III*.

In summary, our evaluation reveals that our mixin-layer based implementation of the software components can be reused with minimal modifications across all the studied heterogeneous ABCs and benchmark applications. Furthermore, the performance of the component-based implementation is reasonable, as it has little overhead (1.5% on average) compared to the hand-tuned and optimized implementations, which are nontrivial to achieve and maintain.

Application	Execution Time (sec.)		Overhead (%)
	Hand-Tuned	Mixin-Layer	
Linear Regression	10.1	10.4	2.0
Word Count	65.3	66.4	1.5
Histogram	22.1	22.9	1.8
K-Means	93.4	94.2	0.8

Table 5: Execution time and overhead for hand-tuned and mixin-layer implementations for benchmark applications using *Conf V*.

Application	Execution Time (sec.)				
	Conf I	*Conf II*	*Conf III*	*Conf IV*	*Conf V*
Linear Regression	10.4	4.7	39.8	16.5	18.0
Word Count	66.4	28.2	238.8	97.2	110.3
Histogram	22.9	11.0	86.8	36.5	39.8
K-Means	94.2	50.5	329.4	144.5	159.4

Table 6: Benchmark execution time with different resource configurations.

6. Applicability of the Approach

Our approach effectively addresses the challenges of constructing ABC-based applications in an important HPC domain. Next we outline the characteristics of this domain and the types of applications that can benefit from our approach. We also describe the types of applications that are unlikely to derive any reusability benefits if they choose to adopt our component methodology.

Our approach puts forward a framework that can be followed to represent the functionality of an HPC application as reusable and customizable components. A key characteristic for an HPC algorithm that determines whether it is amenable to our approach is whether the algorithm can be naturally divided into self-encapsulated units of functionality. These units correspond to the distinct algorithm's phases that map naturally to software layers. The strong encapsulation provided by our approach is conducive to representing the phases as plug-in replaceable components. That is, if a superior implementation of any component becomes available, it can be easily integrated into the existing components stack.

An example domain that can benefit from our approach are task-farm setups. In such a setup, the manager accepts input tasks and places them into a global FIFO queue. The distributed clients then retrieve the tasks, execute them locally, and send the results back to the manager. Task-farm setups are widely used for computationally intensive applications including video transcoding, encryption/decryption, and image processing. Another example domain are applications that tend to follow strict Master Worker communication patterns. Specifically, Workers communicate and interact with each other infrequently, relegating all the coordination and merging tasks to the Master. Such setups have been widely used for distributed searching and sorting.

Obviously our approach is not a solution to all the challenges of ABC-based computing. There are some HPC algorithms that are unlikely to achieve significant levels of reusability if they follow our component methodology. They are likely still to enjoy other Software Engineering benefits of component-based construction such as a clear separation between interfaces and implementations, but the resulting components would have to be constructed for each ABC-based application.

In particular, if an HPC algorithm requires that distributed nodes extensively communicate and synchronize their execution, they are unlikely to follow well-defined computational phases, and as such would not be amenable to be modeled as standalone components. HPC algorithms whose implementation tend to follow these patterns include matrix multiplication and prime factorization.

These applicability preconditions can be easily discerned by examining only a high-level description of an HPC algorithm. From such a description, the programmer can immediately ascertain whether following a component-based implementation imposed by our framework will yield the expected Software Engineering benefits. This clarity is likely to further reduce the implementation effort by precluding the programmer from having to explore the effectiveness of our methodology through trial and error.

7. Conclusion

Nowadays, when choosing an HPC platform for the majority of computations, one must take into consideration multiple factors, including not only the platform's price-to-performance ratio and energy efficiency, but also the complexity of its implementation process. The salient metrics for choosing an HPC platform is its time-to-solution, the total time it takes from posing a problem to arriving at a solution. The ability to systematically reuse and adapt existing code rather than writing it from scratch can significantly lower time-to-discovery.

This paper presents a layered approach to building software for the emerging accelerator-based heterogeneous clusters. Our approach adopts the well-established layered software architecture implemented as mixin-layers. Applying this approach not only achieves the required high performance, but as our evaluation shows, makes it possible to reuse the majority of the code, when types, number, or hierarchy of accelerators is changed. These results indicate that our approach enables effective reuse at the software component granularity, which can reduce the time-to-solution metrics for creating ABCs. Currently, leveraging these resources for HPC requires the effort and expertise commensurate to that of a seasoned computer scientist. However, the ability to build software for these resources from reusable components has the potential to lower the barrier-to-entry for ABCs, making them accessible to researchers from a myriad of scientific and engineering fields.

Acknowledgment

This research is supported in part by the NSF (CAREER CCF-0746832, CNS-1016793) and by IBM (Virginia Tech Foundation grant VTF-874574). In addition, M. Mustafa Rafique is supported by a Scholarship from the Fulbright Foreign Student Program.

References

[1] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, J. C. Sancho, Entering the petaflop era: The architecture and performance of Roadrunner, in: Proc. Supercomputing, 2008.

[2] Amazon, Amazon Elastic Compute Cloud (Amazon EC2), `http://www.amazon.com/b?ie=UTF8&node=201590011`.

[3] K. R., K. Farkas, N. Jouppi, P. Ranganathan, D. M. Tullsen, Processor Power Reduction via Single-ISA Heterogeneous Multi-core Architectures, Computer Architecture Letters 2.

[4] AMD, The Industry-Changing Impact of Accelerated Computing (2008).

[5] L. Laibinis, E. Troubitsyna, Fault tolerance in a layered architecture: A general specification pattern in b, Software Engineering and Formal Methods, International Conference on 0 (2004) 346–355. doi:http://doi.ieeecomputersociety.org/10.1109/SEFM.2004.10035.

[6] H. Davulcu, J. Freire, M. Kifer, I. V. Ramakrishnan, A layered architecture for querying dynamic web content, SIGMOD Rec. 28 (2) (1999) 491–502. doi:http://doi.acm.org/10.1145/304181.304225.

[7] I. F. Cruz, Y. F. Huang, A layered architecture for the exploration of heterogeneous information using coordinated views, in: VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing, IEEE Computer Society, Washington, DC, USA, 2004, pp. 11–18. doi:http://dx.doi.org/10.1109/VLHCC.2004.2.

[8] J. Salz, A. Snoeren, H. Balakrishnan, TESLA: A Transparent, Extensible Session-Layer Architecture for End-to-End Network Services, in: 4th Usenix Symposium on Internet Technologies and Systems, Seattle, WA, 2003.

[9] Y. Smaragdakis, D. Batory, Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs, ACM Trans. Softw. Eng. Methodol. 11 (2) (2002) 215–255. doi:http://doi.acm.org/10.1145/505145.505148.

[10] Y. Smaragdakis, D. S. Batory, Mixin-based programming in c++, in: GCSE '00: Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering-Revised Papers, Springer-Verlag, London, UK, 2001, pp. 163–177.

[11] M. M. Rafique, B. Rose, A. R. Butt, D. S. Nikolopoulos, Supporting mapreduce on large-scale asymmetric multi-core clusters, ACM Operating Systems Review 43 (2).

[12] M. M. Rafique, B. Rose, A. R. Butt, D. S. Nikolopoulos, Cellmr: A framework for supporting mapreduce on asymmetric cell-based clusters, in: Proc. IEEE IPDPS, Rome, Italy, 2009.

[13] M. M. Rafique, A. R. Butt, D. S. Nikolopoulos, Designing accelerator-based distributed systems for high performance, in: Proceedings of the 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, IEEE/ACM, New York, NY, USA, 2010.

[14] L. A. Barroso, J. Dean, U. Hlzle, Web Search for a Planet: The Google Cluster Architecture, IEEE Micro 23 (2) (2003) 22–28. doi:http://doi.ieeecomputersociety.org/10.1109/MM.2003.1196112.

[15] Astrophysicist Replaces Supercomputer with Eight PlayStation 3s, `http://www.wired.com/techbiz/it/news/2007/10/ps3_supercomputer`.

[16] Mueller, NC State Engineer Creates First Academic Playstation 3 Computing Cluster, `http://moss.csc.ncsu.edu/~mueller/cluster/ps3/coe.html`.

[17] GraphStream, Inc., GraphStream scalable computing platform (SCP), 2006, `http://www.graphstream.com`.

[18] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. H. M. Buijssen, M. Grajewski, S. Turek, Exploring weak scalability for FEM calculations on a GPU-enhanced cluster, Parallel Computing. 33 (10-11) (2007) 685–699. doi:http://dx.doi.org/10.1016/j.parco.2007.09.002.

[19] D. Vianney, G. Haber, A. Heilper, M. Zalmanovici, Performance analysis and visualization tools for cell/b.e. multicore environment, in: IFMT '08: Proceedings of the 1st international forum on Next-generation multicore/manycore technologies, ACM, New York, NY, USA, 2008, pp. 1–12. doi:http://doi.acm.org/10.1145/1463768.1463777.

[20] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, W.-M. W. Hwu, Cuda-lite: Reducing gpu programming complexity (2008) 1–15.

[21] K. G. Std", The OpenCL Specification, Version 1.0, Online. Available: `http://www.khronos.org/registry/cl/specs/opencl-1.0.33.pdf` (April 2009).

[22] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, Cell Broadband Engine Architecture and its first implementation - A performance view, IBM Journal of Research and Development 51 (5) (2007) 559–572. doi:10.1147/rd.515.0559.

[23] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, D. Shippy, Introduction to the Cell multiprocessor, IBM Journal of Research and Development 49 (4/5) (2005) 589–604.

[24] IBM Corp., Cell Broadband Engine Architecture (Version 1.02).

[25] NVIDIA Corporation, NVIDIA CUDA Programming Guide (Nov. 2007).

[26] NVIDIA Corporation, GeForce 9600M GT, `http://www.nvidia.com/object/product_geforce_9600m_gt_us.html` (2008).

[27] C. Prehofer, Feature-Oriented Programming: A Fresh Look at Objects, in: ECOOP, 1997.

[28] D. Batory, J. N. Sarvela, A. Rauschmayer, Scaling step-wise refinement, IEEE Transactions on Software Engineering 30 (6) (2004) 355–371. doi:http://dx.doi.org/10.1109/TSE.2004.23.

[29] S. Apel, T. Leich, G. Saake, Aspectual Mixin Layers: Aspects and Features in Concert, in: ICSE, 2006.

[30] Y. Smaragdakis, D. Batory, Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs, ACM Transactions on Software Engineering and Methodologies (TOSEM) 11 (2) (2002) 215–255.

[31] S. Apel, T. Leich, M. Rosenmüller, G. Saake, FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming, in: GPCE, 2005, pp. 125–140.

[32] T. Mens, P. Van Gorp, D. Varró, G. Karsai, Applying a model transformation taxonomy to graph transformation technology, Electronic Notes in Theoretical Computer Science (ENTCS) 152 (2006) 143–159. doi:http://dx.doi.org/10.1016/j.entcs.2005.10.022.

[33] K. Czarnecki, S. Helsen, Feature-based survey of model transformation approaches, IBM Systems Journal 45 (3) (2006) 621–645. doi:http://dx.doi.org/10.1147/sj.453.0621.

[34] M. M. Rafique, A. R. Butt, D. S. Nikolopoulos, DMA-based Prefetching for I/O-intensive Workloads on the Cell Architecture, in: CF '08: Proceedings of the 2008 conference on Computing frontiers, ACM, New York, NY, USA, 2008, pp. 23–32. doi:http://doi.acm.org/10.1145/1366230.1366236.

[35] J. Kurzak, A. Buttari, P. Luszczek, J. Dongarra, The PlayStation 3 for High-Performance Scientific Computing, Computing in Science and Engineering 10 (3) (2008) 84–87. doi:http://doi.ieeecomputersociety.org/10.1109/MCSE.2008.85.

[36] P. Burton, L. Gurrin, P. Sly, Extending the simple linear regression model to account for correlated responses: An introduction to generalized estimating equations and multi-level mixed modelling, Statistics in Medicine 17 (11) (1998) 1261–1291.

[37] A. Guisan, T. C. Edwards, T. Hastie, Generalized linear and generalized additive models in studies of species distributions: setting the scene, Ecological Modelling 157 (2-3) (2002) 89 – 100.

[38] T. N. Pappas, N. S. Jayant, An adaptive clustering algorithm for image segmentation, IEEE Transactions on Signal Processing 40 (4) (1992) 901–914.