# AIRA: A Framework for Flexible Compute Kernel Execution in Heterogeneous Platforms

Robert Lyerly, Alastair Murray, Antonio Barbalace, and Binoy Ravindran, *Member, IEEE*

**Abstract**—Heterogeneous-ISA computing platforms have become ubiquitous, and will be used for diverse workloads which render static mappings of computation to processors inadequate. Dynamic mappings which adjust an application's usage in consideration of platform workload can reduce application latency and increase throughput for heterogeneous platforms. We introduce AIRA, a compiler and runtime for flexible execution of applications in CPU-GPU platforms. Using AIRA, we demonstrate up to a 3.78x speedup in benchmarks from Rodinia and Parboil, run with various workloads on a server-class platform. Additionally, AIRA is able to extract up to an 87% increase in platform throughput over a static mapping.

**Index Terms**—Heterogeneous architectures, compilers, runtimes, programming models

## 1 Introduction

In recent years, diminishing returns in single-core processor performance due to the end of the "free lunch" has pushed hardware design towards increasing levels of parallelism and heterogeneity [1], [2]. Whether it be out-of-order latency-oriented multicore CPUs or massively parallel throughput-oriented GPGPUs, modern hardware is becoming increasingly diverse in order to continue performance scaling for a wide range of applications. It is clear that platforms will become increasingly heterogeneous [3], [4], [5], [6], [7], meaning that developers must embrace this new hardware diversity to achieve higher performance.

Programming frameworks such as OpenMP and OpenCL have emerged as industry standards for programming parallel and heterogeneous platforms. These frameworks are *functionally portable* in that they allow developers to parallelize applications to target different types of processors. In particular, they specify a write-once, run-anywhere interface that allows developers to write a single *compute kernel* (a computationally-intense portion of an application) that is runnable on many different architectures. However, the developer is responsible for orchestrating application execution on processors in the platform, a cumbersome and error-prone process. The developer must select an architecture by extensively profiling the compute kernel on all available processors (assuming exclusive access to the system), manually direct data transfers between disjoint memory spaces and initiate compute kernel execution on the pre-selected processors.

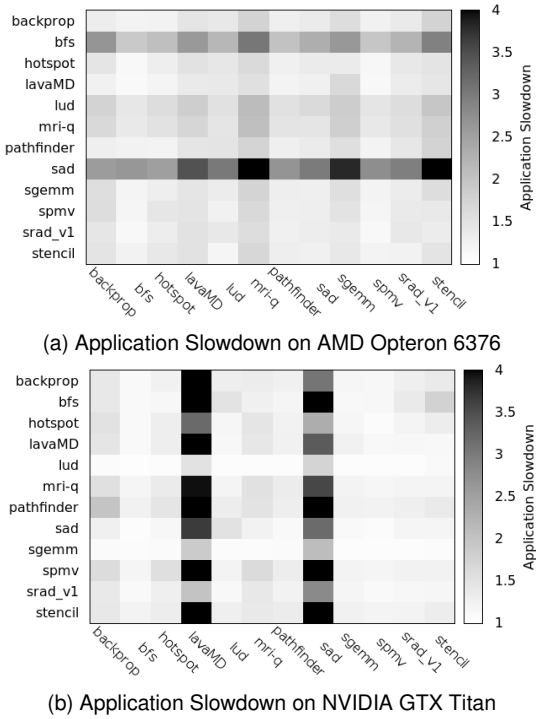
More importantly, these programming frameworks provide no mechanisms for flexible execution of compute kernels in platforms with dynamic and variable workloads because they require developers to hard-code processor selections at compile-time. Static decisions limit compute kernel execution efficiency and platform utilization in the face of diverse platform workloads. Dynamically selecting a set of processing resources on which to execute computation will become increasingly important as heterogeneous systems become ubiquitous in platforms with varying workloads. Previous works show that concurrently executing multiple compute kernels increases processor utilization for better system throughput [8], [9], [10] and energy efficiency [11]. Thus, the question that arises is: *how should applications be refactored and compiled so that their compute kernels can run across a dynamically-selected set of processors to reduce kernel execution latency and increase whole-platform throughput*?

Not taking into account platform workload can have disastrous effects on performance. Figure 1a shows the slowdowns experienced by pairs of applications from the Rodinia [12], [13] and Parboil [14] running concurrently on a 16-core CPU, where both applications fork one thread per core for a total of 32 threads. Figure 1b shows the slowdowns when the same pairs of applications are executed on a GPU. The squares indicate the slowdown of the benchmark listed on the y-axis when concurrently executed with the benchmark listed on the x-axis. Several applications cause severe disruption on the GPU, and applications on the CPU experience high inter-application conflict due to frequent context swaps [15]. These problems are exacerbated as more applications are co-run on the platform. Thus, it is likely there are performance benefits for being able to dynamically switch execution to another architecture.

However, naïvely switching between architectures can also lead to severe performance degradation. Table 1 shows the slowdowns experienced by applications executing compute kernels on the (application-specific) less-performant architecture in the same CPU/GPU platform. Some applications experience modest slowdowns on the non-ideal architecture, e.g. `pathfinder` and `spmv`, and could potentially benefit from execution on an alternate architecture when one architecture is highly loaded. Other applications, e.g. `mri-q` and `sad`, experience severe performance degradation when executed on an alternate architecture. These applications

- *R. Lyerly and B. Ravindran are with the Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA 24061. E-mail: {rlyerly, binoy}@vt.edu.*
- *A. Barbalace and A. Murray were with the Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, 24061. E-mail: antoniob@vt.edu, alastairmurray42@gmail.com.*

*Manuscript received June 12, 2016.*




(a) Application Slowdown on AMD Opteron 6376

(b) Application Slowdown on NVIDIA GTX Titan

Fig. 1: Application slowdown when co-executed with another application a 16-core AMD Opteron 6376 or NVIDIA GTX Titan. Squares indicate the slowdown experienced by the application on the y-axis when running concurrently with the application on the x-axis.

might instead benefit from cooperative sharing of processing resources, i.e. spatial partitioning of processor cores with other applications co-executing on the same architecture. However, no existing infrastructure provides the ability to investigate dynamic architecture selection and sharing.

Manually instrumenting applications for flexible execution in heterogeneous platforms is an intractable solution. Software costs will rise dramatically as developers try cope with the increasing number of devices by adding fragile management code that must simultaneously deal with architecture-specific execution models and architecture-agnostic scheduling decisions. We argue that developers need a compiler which automatically instruments applications and a runtime which drives resource allocation decisions for dynamic workloads in heterogeneous platforms. This would provide benefits in many contexts. Jobs in a datacenter could be scheduled to a wider variety of nodes due to relaxed resource requirements (e.g. removing the requirement for a node to contain a GPU). Datacenter operators that execute batch workloads (like Google [9]) could increases server throughput and utilization by concurrently executing multiple applications. Consolidating multiprogrammed workloads onto fewer machines could decrease acquisition and operating costs, e.g. the rCUDA framework [16] helps developers consolidate multiple workloads onto fewer GPUs to reduce energy consumption.

This work provides a framework for answering questions that arise when using functionally-portable programming frameworks in dynamic heterogeneous platforms:

1) **Architecture Suitability**: How performant is a compute kernel, or how quickly does it execute on each of the architectures in a heterogeneous platform? How do we rank performance on different architectures in order to automatically drive the mapping of compute kernels to architectures?
2) **Runtime Architecture Selection**: In a heterogeneous platform co-executing multiple applications, does dynamic architecture selection provide performance improvements (latency, throughput) versus a static architecture selection?
3) **Architecture Shareability**: For compute kernels co-executing on one architecture, does temporal or spatial partitioning of processing resources better minimize inter-application interference?

| Application | Execution Time Increase |
|---|---|
| backprop | 14% (GPU) |
| bfs | 7% (GPU) |
| hotspot | 1% (N/A) |
| lavaMD | 296% (GPU) |
| lud | 97% (x86) |
| mri-q | 688% (x86) |
| pathfinder | 1% (N/A) |
| sad | 711% (GPU) |
| sgemm | 236% (x86) |
| spmv | 12% (GPU) |
| srad_v1 | 335% (GPU) |
| stencil | 254% (x86) |

TABLE 1: Increase in application execution time when run on the sub-optimal architecture (in parentheses) in a system with an AMD Opteron 6376 and NVIDIA GTX Titan.

In this work, we present AIRA (**A**pplication **I**nstrumentation for **R**esource **A**djustment), a compiler and run-time system for automatically instrumenting and analyzing applications for flexible execution in heterogeneous platforms. AIRA is composed of several software components which automate selecting and sharing architectures at runtime, including a compute kernel analysis tool, a source-to-source compiler and a pluggable daemon for dynamic mapping of computation onto processor resources. AIRA targets co-executing OpenMP applications, and we envision AIRA being built into OpenMP 4.0, where AIRA would handle orchestrating execution onto compute resources and the OpenMP compiler would handle compute kernel code generation for various architectures. The contributions of this work are:

- We detail the implementation of AIRA's compiler and run-time components which provide the infrastructure for flexible execution in heterogeneous platforms;
- We describe a machine learning methodology for automatically training performance models from analyzed compute kernels, which better predicts the most performant architecture for the Rodinia and Parboil benchmark suites on a server-class CPU/GPU platform versus the state-of-the-art;
- We evaluate several policies that dynamically adjust architecture selection and sharing using AIRA's runtime daemon, demonstrating up to a 3.78x speedup (higher than load-balancing across homogeneous GPUs [17]) and an 87% increase in platform throughput over a static policy in a high-workload scenario.

The rest of the paper is organized as follows – Section 2 describes related work. Section 3 discusses the design and

implementation of AIRA and its core components. Section 4 describes the methodology for automatically generating performance prediction models and the policies evaluated in the results section. Section 5 analyzes the results when using those policies. Finally, Section 6 concludes the work.

## 2 Related Work

**Application characterization.** Lee *et al.* [18] and Ïpek *et al.* [19] present statistical methodologies for predicting application performance and power usage for CPUs while varying microachitecture, e.g. functional units, cache sizes, etc. Thoman *et al.* present a suite of microbenchmarks that characterizes the microarchitecture of heterogeneous processors [20]. Baldini *et al.* use machine learning to predict GPU performance based on CPU executions [21]. All of these works quantify arithmetic throughput, the memory subsystem, branching penalties and runtime overheads. Inspired by these methodologies, AIRA predicts compute kernel performance based on microarchitecture features to drive architecture selection in heterogeneous platforms.

**Frameworks for managing execution.** SnuCL [22] is a framework for executing compute kernels in heterogeneous clusters, utilizing OpenCL semantics in a distributed context. VirtCL [17] is a framework for load balancing compute kernels in systems with multiple identical GPUs. Both frameworks require the developer to write low-level OpenCL, including manual data movement between memory spaces. Merge [23] is a language, compiler and runtime that allows users to provide architecture-specific implementations of common functions to be deployed at runtime. Merge requires the developer to use a map-reduce model, whereas AIRA uses general-purpose OpenMP. Additionally, none of [17], [22], [23] consider the suitability of a compute kernel for an architecture, but instead either require the developer to determine suitability [23] or perform naïve load balancing [17], [22]. Liquid Metal [24] is a language and runtime for executing compute kernels across architectures in heterogeneous systems. Similarly, PTask [15] and Dandelion [25] are a runtime system and programming framework for managing compute kernel execution on GPUs. Both Liquid Metal and PTask automatically manage memory movement, but require developers to re-write their application using a task/data-flow model. OmpSs is a framework for managing multicore CPUs and GPUs [26] by composing compute kernels into a task graph, which is executed asynchronously. However, OmpSs requires developers to statically map compute kernels to architectures and requires the developer specify data movement via pragmas. None of Liquid Metal, PTask and OmpSs consider architecture suitability for compute kernels, and all assume the application being executed has exclusive access to the platform. However, new platforms will not be restricted to executing a single application. Instead, AIRA allows developers to use standard OpenMP, and automatically manages multiple concurrently executing applications using performance predictions and cooperative resource allocations.

Panneerselvam *et al.* [27] propose (but do not provide an implementation for) a framework similar to AIRA for runtime architecture selection. However, they do not consider any methodology for deciding resource allocations at runtime. AIRA provides both an implementation to automatically instrument applications for dynamic resource allocations and studies policies for making resource allocation decisions.

**Mapping, scheduling and load balancing.** Emani *et al.* present a methodology that utilizes machine learning models to dynamically determine the optimal number of threads for an OpenMP parallel section in the presence of external workload [28]. Callisto [29] is a framework that avoids inter-application interference of co-running parallel applications on multicore CPUs by mitigating synchronization and scheduler overheads. HASS [30] is a framework for asymmetric homogeneous-ISA cores that utilizes architecture signatures (based on memory usage) to map single-threaded applications to cores that differ in clock frequency. None of these works consider heterogeneous systems. StarPU [31] is a system for scheduling numeric compute kernels on heterogeneous multicores, but requires developers refactor applications into a new task programming model, encode data access characteristics and provide implementations for each architecture in the system. Grewe *et al.* present a compiler that generates OpenCL kernels from OpenMP code and a methodology for mapping the generated kernels to a CPU or GPU based on models trained using machine learning [32]. Their compiler does not refactor the application to support dynamic resource allocation (including automatically managing data transfers) and only considers mapping a single executing application. Their tool, however, could be used in conjunction with AIRA to generate device code. Wen *et al.* [33] present a scheduling policy which prioritizes application execution based on predicted speedup when executing on a GPU (by predicting the speedup to be either high or low) and input data size. However, this scheduling policy strictly time multiplexes the devices in the system, whereas AIRA also supports partitioning processing resources (e.g. cores in a CPU) between co-executing applications. Additionally, AIRA's design advocates a regression-based performance prediction model (instead of a classifier) to support systems in the future that are highly heterogeneous.

## 3 Design & Implementation

AIRA consists of three software components that provide the infrastructure for dynamic selection and sharing of processing resources between compute kernels in heterogeneous platforms. The first of AIRA's components is a *feature extractor* (Section 3.1) that analyzes compute kernels and extracts execution characteristics offline. The features extracted from this tool are used to build predictive performance models, which are used as the basis for architecture selection and resource allocation policies. The second component is a source-to-source compiler, named the *partitioner* (Section 3.2), which instruments applications for coordination and execution on resource allocations. The final component is the *load balancer* (Section 3.3), a daemon that has a pluggable interface for resource allocation policies. We describe implementation details of AIRA in Section 3.4.

Figure 2 shows how applications are analyzed and transformed by AIRA's offline components. An application flows through the framework starting with the feature extractor, which characterizes its compute kernels utilizing profiling

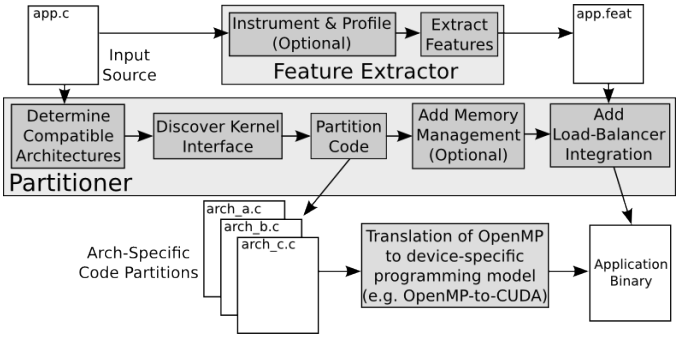



Fig. 2: Offline analysis and refactoring process. Application source is characterized using the **Feature Extractor**, which collects a set of compute kernel features. The source and extracted features are fed to the **Partitioner** which refactors the application for dynamic compute kernel execution.

information when available or compiler heuristics when not. Next, the partitioner refactors the application so that it coordinates with the load balancer to get a resource allocation at runtime and executes on that resource allocation. Figure 3 shows where AIRA sits in the runtime software stack. At runtime, the load balancer is started as a user-space daemon and applications communicate with the load balancer to get resource allocations through inter-processor communication before launching compute kernels on processing resources.

### 3.1 Feature Extractor

The feature extractor is a compiler pass used to accumulate a set of features from compute kernel source code that can be used to generate performance prediction models. The features extracted from a compute kernel are the means by which the compute kernel is characterized – features describe how the compute kernel executes and what types of operations it performs. With an incomplete or inaccurate set of features, the generated models see an incomplete picture of the compute kernel's execution and cannot make accurate predictions. The feature extractor is a GCC compiler pass that is inserted into the compilation process after optimization so that the extracted features reflect the generated machine code. The feature extractor iterates over GCC's internal intermediate representation, generating a feature file for each function in the application.

#### *3.1.1 Extracted Features*

Similarly to previous work [28], [32], [34], we chose features that expose underlying architectural details in order to decide the suitability of a compute kernel for an architecture. We extracted several categories of features that highlight different aspects of compute kernel execution:

- **General Program Features.** Counters for several types of instructions executed by the compute kernel.
- **Control Flow.** Control flow instructions and estimations of divergence.
- **Available Parallelism.** Amount of independent work available for a given compute kernel invocation.
- **Device Communication.** Cost of transferring the compute kernel's data to and from an architecture.

Table 2 lists the compute kernel features used for characterization. Features 1-8 are collected per basic block and

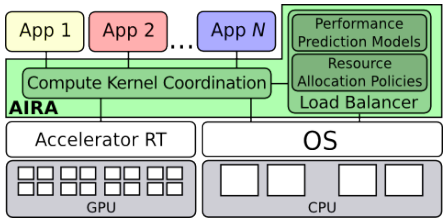



Fig. 3: The runtime software stack. Applications communicate with AIRA's runtime, which predicts compute kernel performance and selects the resources on which the applications' compute kernels should run.

| # | Feature | Description |
|---|---|---|
| 1 | *num_inst* | # of instructions |
| 2 | *int_ops* | # of integer math operations |
| 3 | *float_ops* | # of floating-point math operations |
| 4 | *logic_ops* | # of bitwise and boolean operations |
| 5 | *load_ops* | # of memory loads |
| 6 | *store_ops* | # of memory stores |
| 7 | *func_calls* | # of function calls |
| 8 | *cond_branches* | # of conditional branches |
| 9 | *cycl_comp* | Cyclomatic complexity |
| 10 | *work_items* | # of work items |
| 11 | *memory_tx* | # of bytes transferred to device |
| 12 | *memory_rx* | # of bytes transferred from device |

TABLE 2: Kernel features collected by the feature extractor.

scaled according to the number of times the basic block is executed (obtained through profiling or compiler heuristics). Feature 9 (cyclomatic complexity) is generated once per function, and features 10-12 are generated at runtime. The feature set captures the information about a compute kernel which influences the compute kernel's execution time.

Features 1-7 are general program features – they describe the number and type of operations in a compute kernel. Some architectures may be better suited for certain types of operations (e.g. the AMD Opteron 6376 contains more integer arithmetic units than floating-point units [35]). Features 8 and 9 are control flow features used to capture the amount of divergence in a compute kernel; conditional branches quantify the number of conditionals encountered, while cyclomatic complexity is defined as the number of linearly independent paths through a function's control flow graph. These features influence suitability by distinguishing architectures that perform branch prediction and speculative execution (e.g. CPUs) from those that suffer large performance losses from divergence (e.g. GPUs). Feature 10 quantifies the amount of available parallelism in a compute kernel. This feature is equal to the number of loop iterations in a loop parallelized using an OpenMP `for` pragma, and helps map a compute kernel to an architecture with suitable parallel computational resources. Features 11 and 12 describe the communication costs of transferring a compute kernel's data to a given device[1].

As shown in Figure 2, the feature extractor generates a feature file for each of the analyzed compute kernels. These files provide the feature vectors for the training corpus used to generate performance prediction models. Additionally,

1. Compute kernel launch times on GPUs are implicitly factored into the kernel's execution time.

these files are ingested by the partitioner, which inserts the collected features into a data structure in the application source which is passed to the load balancer. The load balancer feeds the features to the generated models at runtime to make a performance prediction for the compute kernel.

### 3.2 Partitioner

The role of the partitioner is to refactor an application so that it executes compute kernels on a variable set of processing resources obtained dynamically from the load balancer. This is achieved by inserting wrapper functions around compute kernels that handle all communication with the load balancer and dispatch execution to the appropriate resources. The partitioner is so named because it creates *source code partitions* by duplicating and refactoring each OpenMP work-sharing construct for each of the available architectures in the system. During execution, the wrapper dispatches execution to the appropriate resources by calling the corresponding source code partition. The partitioner was built using the ROSE source-to-source compiler infrastructure [36]. It is composed of several passes, each of which analyzes and transforms the code [37].

The partitioner must refactor an application so that its compute kernels can execute on a dynamic allocation of processing resources. The partitioner automatically inserts all architecture-specific boilerplate code necessary to launch a compute kernel on a particular architecture, transfer data to and from that architecture and even refactor the compute kernel into an architecture-specific programming model. For example, Listings 1 and 2 demonstrate how the partitioner would refactor a vector sum calculation. The partitioner requires four passes (and one optional pass) through the abstract syntax tree (AST) of the application for analysis and refactoring. Each pass builds upon results from previous passes, but each one can be run separately – information is stored between passes by inserting pragmas into the source code so that developers can see how the tool refactors the code. This also allows developers to tune the results to help with the conservative nature of static analysis. Figure 2 shows the passes of the partitioner:

**Determine Compatible Architectures.** The partitioner traverses the compute kernel AST and looks for incompatible or illegal functions for each architecture. For example, calls to the I/O API of the C standard library are not allowed on a GPU. Note that built-in or library math functions, e.g. `sin(x)`, are compatible for most architectures.

**Discover the Kernel Interface.** The partitioner traverses the compute kernel AST to determine what data and definitions are necessary to compile and launch the compute kernel on an architecture. It searches for inputs and outputs that must be transferred to and from the architecture (function arguments, global variables, function return values, inputs with side-effects), functions called by the compute kernel, and abstract data types used by the compute kernel (e.g. `struct` types). In Listing 1, the partitioner discovers inputs `A`, `B` and `size` and output `C` as the kernel interface.

**Partition the Code.** This pass performs the bulk of the refactoring work – it examines information collected from previous passes to create the source code partitions. It performs the following steps:

1) The partitioner transforms the OpenMP compute kernel into a stub, named the *compute kernel wrapper*, from which architecture-specific code partitions are called after coordinating with the load balancer (lines 1-17 in Listing 2). The original compute kernel is moved to a separate function, which becomes the CPU partition (lines 18-27).
2) Copies of the kernel code, supporting functions and ADT definitions are placed into each code partition for compilation. The `vecSum` kernel does not use any supporting functions or user-defined structures, so no action is needed in this step.
3) A per-kernel handshake for each device is inserted, which coordinates launching a the kernel on that device. This includes all data transfers and launching execution on the appropriate resources. Lines 38-41 in Listing 2 implement the GPU handshake for `vecSum` (memory management using CUDA APIs has been omitted).
4) The OpenMP compute kernel is refactored into device-specific code. We use OpenMPC [38], [39] to perform OpenMP-to-CUDA translation to execute the compute kernel on NVIDIA GPUs. OpenMPC generates `vecSumKern` on line 42 in Listing 2.

**Add Memory Management (Optional).** AIRA provides a library which tracks application memory allocation in order to automate data transfers to and from devices. The library must maintain locations and sizes of both statically and dynamically allocated memory in order to automatically handle data transfers. First, the library wraps the standard C dynamic memory functions (e.g. `malloc`, `free`) using the linker in order to observe dynamic memory management. Second, the library provides an API that allows the application to notify the library of statically allocated data such as global memory and stack variables. The partitioner inserts calls to this API to index statically allocated non-scalar variables for later retrieval (scalar variables have a known size). The library stores data sizes by address using a red-black tree, which is queried when a compute kernel is launched on a device that requires data movement. For example, `vecSum_gpu` queries the library when allocating GPU memory (lines 37-39) to find the sizes of vectors allocated via `malloc`. Note that if the memory management pass is not used the developer must specify data sizes for copying data in and out of compute kernels using AIRA's pragmas, similarly to OpenMP 4.0 `data` clauses.

**Add Load Balancer Integration.** The partitioner inserts code to coordinate with the load balancer. The partitioner embeds the compute kernel features from the feature extractor into the wrapper function, e.g. `vecSum_feat` on lines 4-9. Then, the partitioner inserts calls to a library to communicate with the load balancer, e.g. the call to `aira_get_alloc` on line 10. Before executing the compute kernel, the application sends the features to the load balancer to be used in the prediction models and the resource allocation policies. The load balancer returns a resource allocation to the application, which is used by the wrapper to launch the compute kernel on the specified resources (lines 11-15). After the compute kernel has finished execution, the application notifies the load balancer so that the load balancer can keep its internal workload accounting information accurate (line 16).

The partitioner generates several files (as shown in

```
1 void vecSum(const double *A, const double *B,
2             double *C, size_t size) {
3   size_t i;
4 #pragma omp parallel for
5   for(i = 0; i < size; i++)
6     C[i] = A[i] + B[i];
7 }
```

Listing 1: OpenMP vector sum before refactoring.

Figure 2) including the refactored source of the original application and a file for each partition. These files compiled together generate a binary that is ready for deployment.

### 3.3 Runtime Load Balancer

AIRA's final component is a runtime daemon which applications query at runtime to obtain resource allocations for executing compute kernels. Applications communicate using an inter-process communication (IPC) library, implemented using Unix sockets, to talk to the daemon. Applications send extracted compute kernel features to the load balancer and receive a resource allocation in return. The application blocks until receiving the resource allocation, allowing the load balancer to control when compute kernels are launched in the system. The load balancer provides a pluggable interface for resource allocation policies to allocate resources to applications. Resource allocation policies provide a means to select an architecture, and for architectures that allow fine-grained control of processing resources, the number of processor cores to use for executing the compute kernel (e.g. the number of CPU cores to use).

Figure 4 shows the interaction between the application and the load balancer at runtime. When an application enters the wrapper function inserted by the partitioner, it sends the compute kernel's features to the load balancer. The load balancer feeds the features to the performance model, and the outputs from the performance model are fed to the resource allocation policy. The policy generates a resource allocation which is returned to the application. The wrapper then launches the compute kernel on the specified resources. After finishing executing the compute kernel, the application notifies the load balancer of its completion and continues normal execution.

The daemon keeps track of which applications are executing by maintaining a per-architecture *running-list* using check-in/check-out communication. Lists maintain an entry (including the application's PID and resource allocation) for each currently executing compute kernel. Resource allocation policies utilize this information with performance predictions to make allocations. Running-lists influence resource allocations in an intuitive way – if a given architecture is experiencing high load (i.e. it has a long running-list), the policy should adjust resource allocations to account for this load (e.g. by allocating fewer cores or switching execution to a different architecture).

Although the information tracked in the running-lists is simple and may not model complex cross-application interference, we used this design for several reasons. First, we wanted the runtime model evaluation to be as lightweight as possible. Several of the applications have very short running times, meaning excessive overheads due to evaluating complex policies could cause non-trivial performance

```
1 /* Compute Kernel Wrapper */
2 void vecSum(const double *A, const double *B,
3             double *C, size_t size) {
4   aira_kernel_features vecSum_feat = {
5     .num_inst = 500000000,
6     .int_ops = 100000000,
7     .float_ops = 100000000,
8     ⟨ Other features from Table 2 ⟩
9   };
10  aira_alloc alloc = aira_get_alloc(&vecSum_feat);
11  switch(alloc.device) {
12  case CPU: vecSum_cpu(A, B, C, size, alloc); break;
13  case GPU: vecSum_gpu(A, B, C, size, alloc); break;
14  ...
15  }
16  aira_notify(&alloc);
17 }
18 /* CPU Source Code Partition */
19 void vecSum_cpu(const double *A, const double *B,
20                 double *C, size_t size,
21                 aira_alloc alloc) {
22  size_t i;
23  omp_set_num_threads(alloc.num_procs);
24 #pragma omp parallel for
25  for(i = 0; i < size; i++)
26    C[i] = A[i] + B[i];
27 }
28 /* GPU Source Code Partition */
29 void vecSum_gpu(const double *A, const double *B,
30                 double *C, size_t size,
31                 aira_alloc alloc) {
32  dim3 grid, blocks;
33  double *A_d, *B_d, *C_d;
34  size_t A_size, B_size, C_size;
35  A_size = aira_get_size(A);
36  B_size = aira_get_size(B);
37  C_size = aira_get_size(C);
38  ⟨ Allocate & transfer A, B & C to GPU ⟩
39  ⟨ Calculate launch dimensions based on size ⟩
40  vecSumKern<<<grid, blocks>>>(A_d, B_d, C_d, size);
41  ⟨ Copy C back from GPU ⟩
42 }
```

Listing 2: Vector sum after partitioner refactors the code. Allocations and data movement using standard CUDA APIs (lines 38-39 and line 41) have been omitted.

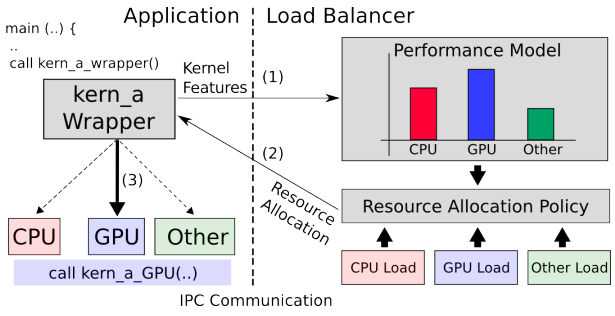



Fig. 4: Runtime check-in/check-out communication with the load balancer to get a resource allocation for compute kernel execution.

degradation (Section 5 quantifies allocation overheads). Second, in the context of this work applications come and go rapidly, meaning that by the time the load balancer had evaluated cross-application interference the interfering application may have finished execution. Hence, simple running-list information sufficed for our needs.

We implemented communication using sockets because their client/server semantics are a natural fit for AIRA's communication pattern, and because they can be easily

ported to other OSs. However, AIRA could be adapted to use any mechanism that implements send/receive functionality, e.g. AIRA could be extended to clusters using RDMA verbs [40]. In future work, we plan to integrate the load balancer into the operating system scheduler to give the load balancer a more complete view of the system and to reduce IPC overheads.

### 3.4 Implementation

AIRA's components were developed using a combination of C and C++, and AIRA currently supports C applications. Features were extracted from applications using profiling information – applications were compiled using GCC's `-fprofile-generate` switch to collect basic block execution statistics, which were fed back into the feature extractor to scale the collected features. OpenCV's machine learning module was used as the basis for model generation and evaluation, described in Section 4. Communication with the load balancer was implemented using Unix sockets, which enabled easy synchronization and queuing of requests. Overall, the framework required 13,900 lines of code. Although AIRA is implemented assuming compute kernels were written using OpenMP, its design principles could be applied to other functionally-portable programming framework (e.g. OpenCL, OpenACC, etc.). The feature extractor was 700 lines of C++, the partitioner was 8,200 lines of C++, the memory management library was 700 lines of C, the load balancer was 2,120 lines of C/C++ and the machine learning tools were 2,250 lines of C++.

## 4 Performance Prediction and Resource Allocation Policies

In order to drive resource allocation policies, a method is needed to predict an application' suitability for each of the available architectures. We chose to use statistical machine learning methods due to established success in previous works [18], [32], [34]. Utilizing the extracted compute kernel features, performance prediction models were generated using machine learning (Section 4.1) and used as the basis for the resource allocation policies (Section 4.2). Finally, we extended these policies to show how AIRA can be leveraged to prioritize a compute kernel's execution (Section 4.3).

### 4.1 Model Generation via Machine Learning

Artificial neural networks (ANN) [41] were used for the performance prediction models because of their flexibility and because they are regressors. ANNs predict how performant a compute kernel is on a given architecture (a continuous value); classifiers (as used in [32]) predict the most performant architecture by selecting a class, e.g., GPUs are the best architecture. Regressors are better suited for heuristic-based approaches for runtime resource adjustment. OpenCV was leveraged to provide an ANN implementation, and the back-propagation [42] algorithm was used to train the models. Each model was trained using a corpus of data consisting of the extracted compute kernel features and compute kernel runtimes from sample benchmark runs. The models took compute kernel features as inputs and generated performance predictions for each architecture in the platform as outputs. Each benchmark was run 80 times on each architecture in order to generate the corpus, requiring a total of three hours of profiling. This training time is a one-off cost per platform – the models can be utilized for previously unseen applications that provide features from the feature extractor. After collecting training data, leave-one-out cross validation was used (as described in Section 5) to generate models, requiring 13.6 seconds per model.

Evaluating the models is computationally lightweight, as values only propagate through three layers of neurons. Standard pre-processing was applied to the inputs – all features were scaled to have similar ranges, and principal component analysis (PCA) [43] was applied to reduce the dimensionality of the input data. PCA is a method for combining inputs into meta-features that extract the most important feature information. This also improves the model evaluation time by reducing the number of values that must propagate through the network. Scaling and PCA reduction were applied during the training process and at runtime by the load balancer before feeding inputs to the trained models. The neural networks were configured by empirically determining the best PCA and middle-layer dimensions. The models were configured with a PCA projection down to seven dimensions (which retained 99% of the variance) and a hidden layer consisting of nine neurons.

| Policy | Core Sharing | Arch Selection |
|---|---|---|
| *Static* (baseline) | ✗ | ✗ |
| *Share* | ✓ | ✗ |
| *Select* | ✗ | ✓ |
| *Share+Select* | ✓ | ✓ |

TABLE 3: Characteristics of resource allocation policies.

### 4.2 Resource Allocation Policies

Three resource allocation policies, named *Share*, *Select* and *Share+Select*, were developed to evaluate resource selection and sharing in our evaluation. All resource allocation policies utilized the prediction models to determine the suitability of each architecture for each compute kernel and then applied a combination of two heuristics[2] to select processing resources. Table 3 lists each of the evaluated resource allocation policies and the heuristics used by each.

The *core sharing* heuristic was used in the *Share* and *Share+Select* policies. This heuristic, designed to exploit the increasing parallelism of multicore processors, spatially partitioned an architecture's available cores between applications co-executing on that architecture. Note that this heuristic was applied only on the multicore CPU, as current GPUs do not allow fine-grained management of GPU cores. The *Share* and *Share+Select* policies first selected an architecture based on the model predictions (and in the case of *Share+Select*, the *architecture selection* heuristic described below). If the CPU was determined to be the ideal architecture, the application was given a proportional share of the CPU cores based on the running-list length:

$$Cores_{allocated} = \lceil \frac{Cores_{CPU}}{RunningList_{CPU}} \rceil,$$

2. The heuristics are simple in nature and were developed to show the untapped potential of cooperative selection and sharing – future work will investigate ways to improve them.

where $Cores_{allocated}$ is the number of cores allocated to the application, $Cores_{CPU}$ is the total number of available cores and $RunningList_{CPU}$ is the number of compute kernels currently running on the CPU (maintained by the load balancer's running-lists). For example, consider a workload with three arriving applications, A, B and C, all of which are selected to run on a 16-core CPU. When A arrives, AIRA instructs it to use all 16 cores. When B arrives (while A is still running), AIRA instructs it to use 8 cores, and likewise when C arrives AIRA instructs it to use 4 cores. Although the CPU is currently oversaturated (28 threads), when A finishes the system becomes under-saturated (12 threads). On average, this policy matches the number of threads running in the system to the number of available CPU cores[3].

The *architecture selection* heuristic was used in the *Select* and *Share+Select* policies. This heuristic let the policies switch execution from the model-selected best architecture to a less ideal architecture, depending on the difference in performance when executing on the alternate architecture and the load on the ideal architecture. This heuristic achieved this goal by adjusting the performance predictions of a compute kernel on each architecture based on the number of applications currently executing on that architecture:

$$Perf_{adjusted} = Perf_{arch} \times RunningList_{arch},$$

where $Perf_{adjusted}$ is the adjusted performance prediction, $Perf_{arch}$ is the original performance prediction from the model and $RunningList_{arch}$ is the number of applications currently executing on the architecture. This heuristic approximated performance degradation proportionally to the number of applications currently executing on each architecture, i.e. for an architecture currently executing $N$ compute kernels, the arriving compute kernel is expected to take $N$ times as long to complete. These adjusted performance predictions were then used to select an architecture, e.g. `bfs`'s compute kernels could be executed on the GPU instead of the CPU if the CPU was overloaded.

These three policies were compared against the *Static* baseline, which modeled the behavior of a developer assuming exclusive access to the entire platform. Compute kernels were executed on the most performant architecture for that compute kernel (as determined from profiling information) and allocated all processing cores available on that architecture. Architecture selection was never dynamically adjusted.

Currently, none of the policies control when kernels execute (i.e., no temporal execution control) but rather only adjust on which resources the kernels execute. Applications make requests to the load balancer, which returns allocations immediately. Although the load balancer does implement the ability to block application execution by forcing them to wait for an allocation, we leave policies which exploit this capability as future work.

### 4.3 Prioritized Allocation Policies

In addition to the previously described policies, three more policies were developed to prioritize a single application, i.e. to maximize performance for a single application with minimal impact on system throughput. These policies are useful in instances where certain jobs should have execution priority over others (e.g. a latency-sensitive job serving a search query versus maps batch processing in Google's systems [9]). These policies, named *Naïve*, *Priority* and *Priority+Share*, utilized the OS's capabilities to augment AIRA's dynamic resource allocations.

The *Naïve* policy only utilized the OS scheduler's capabilities for prioritization. For applications executing without high priority (dubbed *regular* applications), the *Static* baseline described in Section 4.2 was used for resource allocation. For *high-priority* applications, the *Naïve* policy instructed the application to set its threads to the highest scheduling priority[4]. This ensured that the application's threads were scheduled first, when either executing the compute kernel on the CPU or when managing compute kernel execution on the GPU (i.e. enqueuing data transfers or kernel launches). Note that because the GPU driver is close-source the threads cannot adjust the GPU task queue, meaning kernels enqueued by the high-priority application cannot jump ahead of previously enqueued kernels in the GPU driver's task queue.

The *Priority* policy utilized AIRA's capabilities in addition to setting scheduling priorities. Regular applications were again allocated resources according to the *Static* baseline. When a high-priority application requested a resource allocation from the load balancer, the load balancer locked the architecture on which the high-priority application was mapped. This meant that later-arriving regular applications were automatically mapped to a non-locked architecture, e.g. if `srad_v1` was the high-priority application, when it requested a resource allocation it was mapped to the GPU and subsequently-arriving requests were mapped to the CPU. After architecture selection, high-priority applications mapped to the CPU were again told to set their threads to the highest scheduling priority. When the high-priority application had finished compute kernel execution, the device was unlocked and available for use by regular applications. Device locking provided a coarse-grained method for temporal reservations, and in particular helped to compensate for the non-preemptive nature of GPUs.

Finally, the *Priority+Share* policy operated similarly to the *Priority* policy but used the *Share* policy instead of the *Static* baseline to allocate resources for regular applications. This meant that CPU cores were spatially partitioned among co-executing regular applications. High-priority applications mapped to the CPU were, however, allocated all available CPU cores, regardless of what regular applications were concurrently executing on the CPU. The load balancer performed device locking, and the high-priority application set its threads to the highest scheduling priority.

## 5 Results

We utilized AIRA to quantify application performance and platform throughput using the resource allocation policies described in Section 4.2. We evaluated these policies on a multicore CPU/GPU platform using compute benchmarks. In order to perform a thorough evaluation, we first tested the accuracy of the models to predict the most suitable architecture for a compute kernel versus the state-of-the-art [32].

3. Current OpenMP runtimes do not allow changing the number of threads while executing a work-sharing region.

4. The highest priority of interactive user applications is -20 in Linux (SCHED_OTHER) [44].

| | AMD Opteron 6376 | NVIDIA GTX Titan |
|---|---|---|
| # of Cores | 16 | 14 (2688 CUDA cores) |
| Frequency | 2.3 GHz | 837 MHz |
| Core Design | Superscalar/OoO | SIMT |
| Compiler | GCC 4.8 | NVCC 6.5 |
| OS | Ubuntu 12.04 LTS w/ Linux 3.13 | |

TABLE 4: Architectures in the platform used for evaluation. The GPU executes using a single-instruction/multiple thread model and is connected via PCIe 2.0, x16.

| Feature | Description |
|---|---|
| transfer ÷ (compute + memory) | communication / computation ratio |
| coalesced/memory | % of memory accesses that are coalesced |
| (local_mem ÷ memory) × average # work items | local/global access ratio × compute kernel threads |
| compute/memory | computation / memory ratio |

TABLE 5: Features used to train competitor's decision tree models, compared to AIRA's extracted features in Table 2.

After we had verified our models were able to accurately predict suitability, we tested the resource allocation policies with varying levels of system load.

For our experimental evaluation, we ran experiments on a platform containing an AMD Opteron 6376 CPU and an NVIDIA GeForce GTX Titan GPU (shown in Table 4). For our experiments, we extracted features, partitioned benchmarks and trained models using 12 benchmarks from the Rodinia [12] and Parboil [14] benchmark suites. Some benchmarks were omitted because AIRA currently cannot marshal data types with arbitrarily nested pointer types (e.g., linked list nodes in non-contiguous memory) and because OpenMPC was not able to correctly handle some more advanced OpenMP kernels, especially ones with function calls. Only the OpenMP versions of benchmarks were used, which were analyzed and refactored using AIRA's feature extractor and partitioner. These were then executed in conjunction with AIRA's load balancer using the resource allocation policies described in Section 4.2 and Section 4.3.

### 5.1 Overhead Analysis

We first sought to verify that resource allocation overheads (i.e. inter-process communication costs with the load balancer) were acceptable. Figure 5 shows the mean overhead, in $\mu s$, for coordinating with the load balancer with varying numbers of co-executing applications. These numbers are per compute kernel invocation, and are composed of the time to obtain a resource allocation and the time to cleanup after compute kernel execution. Resource allocation times include IPC, model prediction and policy evaluation costs. Cleanup times include the IPC costs for notifying the load balancer after execution has completed and any running-list accounting. As evident in the graphs, these overheads are minimal compared to application runtime, with total costs rising to no more than 250 $\mu s$. This is several orders of magnitude smaller than the shortest benchmark – `backprop` executes in an average of 351 ms on the CPU.

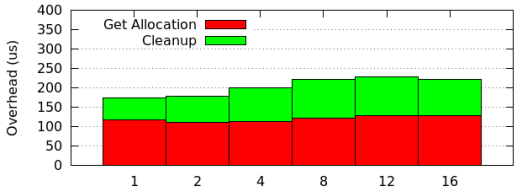



Fig. 5: Mean per-compute kernel coordination overheads with varying numbers of concurrently executing applications, in $\mu s$.

Thus, the overheads introduced by AIRA are acceptable for the tested applications. AIRA may not be suitable for applications which cannot amortize these overheads, such as applications that launch many short-lived compute kernels.

### 5.2 Static Architecture Selection

In order to test the ability of our trained models to make accurate performance predictions for unseen benchmarks on the architectures in the platform, we used *leave-one-out cross-validation* (a standard machine-learning technique) to test the generalizability of the produced models. During this process, a single benchmark was designated as the *testing* benchmark, and the rest of the benchmarks were designated as the *training* benchmarks. A model was generated using training data from only the training benchmarks. Once the model was trained, the model was evaluated using the testing benchmark to determine if the model was able to make accurate performance predictions. This process was then repeated in turn for each of the benchmarks.

We tested the ability of the models to predict the best architecture for compute kernels running without any external workload. We trained the models using the methodology described in Section 4.1. We compared the accuracy of our models to the methodology described by Grewe *et al.* [32] – their approach collects a set of meta-features (shown in Table 5) formed from raw program features which were used to train decision trees [45]. Both AIRA's and Grewe *et al.*'s models were trained using the same raw program features extracted from benchmarks (except for coalesced memory accesses, which were gathered using NVIDIA's profiling tools), and trained using leave-one-out cross validation. The only differences between the approaches were whether or not the methodology utilized PCA or hand-crafted meta-features, and the underlying model (ANN vs. decision tree). Figure 6 shows the results of the comparison. The bars in the graph show the runtime (in seconds) of the architecture selected using different methodologies. Note that Table 1 lists the less-performance architecture for each of the benchmarks. In our setup, `backprop` is the shortest-running benchmark at $351ms$, while `stencil` is the longest at $14.1s$ (or $28.5s$ for the competitor).

Our models are able to accurately predict the best architecture on which to execute different compute kernels. We match the oracle for every benchmark except `pathfinder`, which has minimal performance loss on the incorrect architecture. This is because it is both memory throughput-bound (better for GPUs) and branch-heavy (better for CPUs). However the competitor does not do as well – the trained decision trees fail to predict the correct architecture for `bfs`,

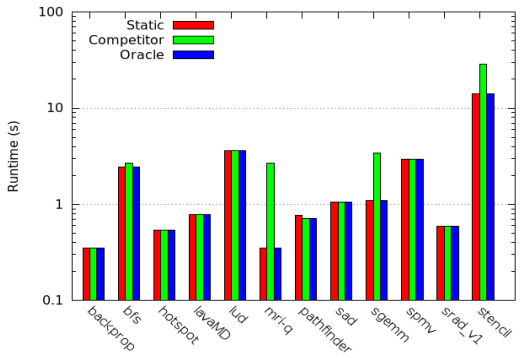



Fig. 6: Runtimes (in seconds) of benchmarks from Rodinia and Parboil on architectures selected using different methodologies.

`mri-q`, `sgemm` and `stencil`, resulting in significant lost performance. This is due to the hand-crafted meta-features versus our use of PCA to statistically combine features. Additionally, our feature set is slightly more comprehensive, including details such as individual types of operations and divergence costs. In general, we believe the user should provide the machine learning model with a set of detailed raw features and let machine learning processes determine how features should be combined.

### 5.3 Dynamic Selection and Sharing

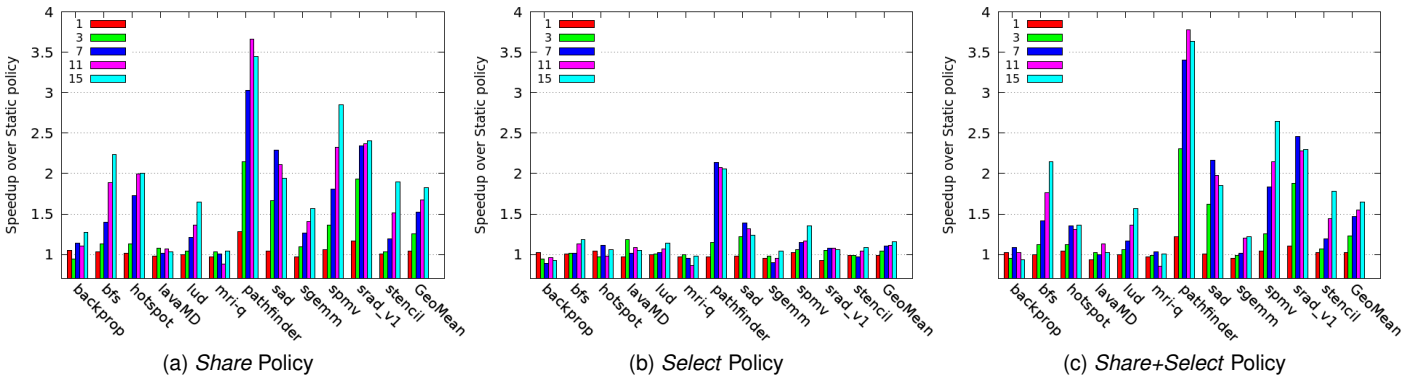
After analyzing the models offline, we evaluated application performance with system load using AIRA's load balancer. We developed and integrated the resource allocation policies from Section 4.2 into AIRA's load balancer. In order to generate system load for the experiments, we executed *workload launchers*. Each workload launcher generated a randomized list of the training benchmarks for the current testing benchmark and looped infinitely through the list, launching the benchmarks as external workload in the system while running the testing benchmark. All co-executing benchmarks, including the testing and training benchmarks run by workload launchers, communicated with and received resource allocations from AIRA's load balancer at runtime. Note that because the workload launchers executed randomized external workload, the system ran a random mixture of benchmarks, modeling a live production server handling compute job requests [9].

Figure 7 shows the speedups achieved when running each benchmark with varying numbers of workload launchers for each policy. Speedups indicate the reduction in application runtime when using each of the policies versus runtime using the *Static* policy at the same workload level. Alternatively, Figure 8 shows the slowdown for each policy versus running the application on a system with no external workload. Each benchmark was executed 100 times in each of the workload scenarios in order to mitigate noise. As shown in Figure 7, the policies demonstrate sizable improvements for the testing benchmarks versus the static policy when co-executed with as few as three workload launchers. Speedups continue to rise with increasing numbers of co-executing applications. As expected, Figure 8 shows latency increasing compared to running in an unloaded system as more co-executing applications compete for resources. Clearly, however, a cooperative resource allocation can provide significantly better performance than a static resource allocation for multiprogrammed systems.

Many of the applications experience significant and increasing speedups as the amount of external workload increases, e.g. `bfs`, `pathfinder`, `sad`, `spmv`, `srad_v1` and `stencil`. The `pathfinder` benchmark shows the largest benefit from AIRA's load balancer, due to the fact that it has comparable performance on both architectures and also because it still achieves good performance with smaller core allocations on the Opteron. Other benchmarks experience small or non-existent speedups versus the *Static* policy, e.g. `backprop`, `lavaMD` and `mri-q`. As mentioned previously, `backprop` is the shortest running benchmark, meaning there is little inter-application interference and thus less room for improvement. `lavaMD` and `mri-q` are highly compute bound (versus other benchmarks that are better balanced between compute and memory), meaning the performance loss from reduced compute resources offsets any gains from reduced inter-application interference. However, for the majority of the benchmarks, the policies are able to provide increased performance with increased workload.

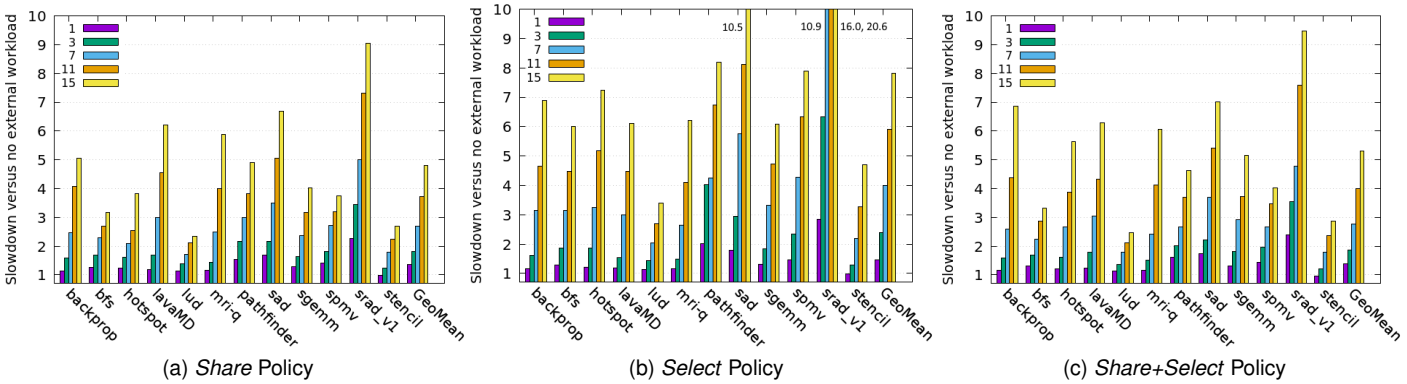
The *Share* policy shows the best performance improvements out of any of the policies, with mean speedups of 25%, 52%, 67% and 83% for 3, 7, 11 and 15 workload launchers, respectively. This is due to the high number of cores in the Opteron – the compute kernels that run on the CPU still receive a large partition of CPU cores even when several applications use the Opteron simultaneously. Note that applications running on the GPU with the *Share* policy must time-share the processor due to hardware limitations, although we expect this restriction to be lifted as GPUs become more general-purpose. The *Select* policy is also able to obtain speedups by switching execution between the two architectures (4%, 10%, 11% and 16%), albeit at a reduced level compared to *Share*. *Share+Select* maintains most of the benefits of *Share* (23%, 47%, 55% and 64%), but does not successfully exploit switching architectures to gain extra performance on top of partitioning CPU resources. This is due to the simple nature of the heuristic – it sometimes switches architectures (most often from the CPU to the GPU) when the compute kernels would still execute more quickly on a partitioned set of CPU cores. However, `pathfinder` (which benefits from both dynamic architecture selection and sharing) experiences a 3.78x speedup with 11 workload launchers, the highest speedup of any tested benchmark. Figure 8 shows similar trends – the *Share* policy has the smallest slowdown compared to an unloaded system, with the *Share+Select* and *Select* policies following suit (the *Select* policy is slightly better but comparable to the *Static* policy).

Figure 9 shows the average system throughput using each of the policies for each workload scenario, scaled to show the number of benchmarks executed per minute. The trends are consistent with results from the graphs in Figure 7. The *Static* policy demonstrates the worst system throughput, where throughput saturates with only 3 workload launchers. The other policies, however, show better scalability. The *Share* policy is able to extract the highest system throughput, demonstrating increasing scalability with more system load (an 87% improvement in system throughput with an external workload of 15). *Share+Select* achieves




(a) *Share* Policy (b) *Select* Policy (c) *Share+Select* Policy

Fig. 7: Speedup over the *Static* policy with varying numbers of external workload launchers. Both the specified policy and the *Static* policy were run using the same number of workload launchers.




(a) *Share* Policy (b) *Select* Policy (c) *Share+Select* Policy

Fig. 8: Slowdown of different policies versus running in a system with no external workload.

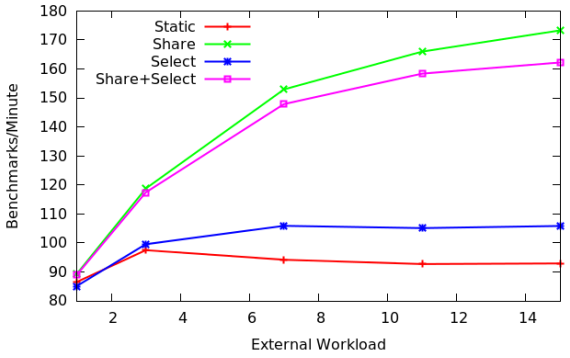



Fig. 9: System throughput for resource allocation policies with varying numbers of workload launchers. Indicates the number of benchmarks launched per minute, including both training and testing applications.

slightly worse performance, but still shows comparable scalability through the highest workload (75% improvement). *Select* demonstrates better throughputs than the static policy, but throughputs saturate at 7 external workload launchers; it only achieves up to a 14% increase in throughput. By comparing Figure 8 and Figure 9, it is easy to see how the system can trade off single-application latency for throughput; for example, by co-executing 8 applications using the *Share* policy the system could increase application latency by 2.5x in order to increase system throughput by 66%.

### 5.4 Prioritization

Finally, we integrated and evaluated the prioritized allocation policies presented in Section 4.3. We evaluated the ability of these policies to prioritize applications using the previously described performance prediction methodologies and policies with an experimental setup similar to Section 5.3. We launched testing benchmarks with increasing numbers of workload launchers, where the testing benchmark was designated as the high-priority benchmark in all experiments (i.e. there was at maximum one high-priority application running at a time).

Figure 10 shows the speedups achieved with prioritization policies over a static resource allocation, while Figure 11 shows the slowdowns versus an unloaded system. The *Naïve* policy is able to achieve speedups for most applications, and provides comparable performance to *Share* from Section 5.3 – it achieves a 100% average improvement versus the *Static* baseline with 15 workload launchers. By prioritizing through the OS scheduler, applications are able to get significant improvements in performance. The *Priority* policy provides similar benefits, but is able to extract extra performance improvements through device locking. Kernels of high-priority applications see reduced interference, benefiting applications on both the CPU and GPU. `lud` greatly benefits from this capability because it performs many small GPU compute kernel launches in succession; by reserving the GPU, these small launches are not interleaved with the execution of other compute kernels. The *Priority+Share*

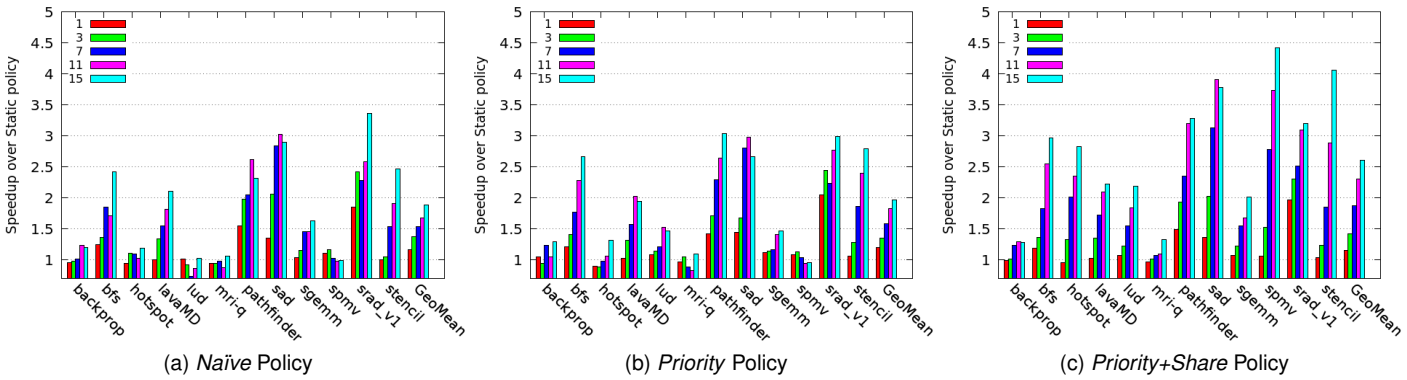



Fig. 10: Prioritization policy speedups over the non-prioritized *Static* baseline.

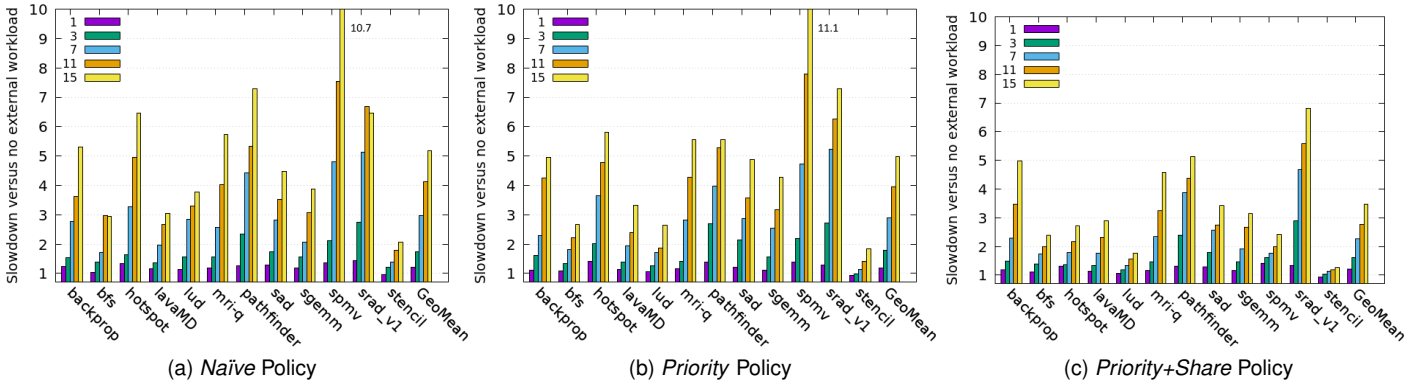



Fig. 11: Slowdown of different prioritization policies versus running in a system with no external workload.

policy provides the most benefit, with applications experiencing up to an average of 2.7 times speedup over the static resource allocation, and provides much higher speedups than any of the policies mentioned in Section 5.3. It is able to reap whole-platform benefits by augmenting the benefits from partitioning the CPU resources with those from higher scheduling priorities and device locking. While the *Naïve* and *Priority* policies demonstrate poor slowdowns versus an unloaded system, the *Priority+Share* policy shows much smaller slowdowns, even as the amount of external workload increases. Several benchmarks (`backprop`, `mri-q`, `srad_v1`) experience larger slowdowns because they are short-running and are highly sensitive to interference.

Table 6 shows the reduction in platform throughput for each of the policies versus the *Static* baseline. All prioritization policies demonstrate reductions in throughput versus the baseline; however, this is to be expected given the design goal of maximizing performance. The *Priority+Share* policy never saw more than a 19% reduction in throughput versus the baseline, but simultaneously managed to achieve significant application speedups. This demonstrates that using AIRA, applications can be prioritized with small tradeoffs in throughput on heterogeneous platforms.

### 5.5 Discussion

These results show the benefits obtainable by enabling dynamic resource adjustment. In particular, we see that cooperative resource allocations allow applications to better utilize platforms for increased performance throughput. We can draw several key insights:

| Workload | 1 | 3 | 7 | 11 | 15 |
|---|---|---|---|---|---|
| *Naïve* | 9% | 7% | 17% | 14% | 20% |
| *Priority* | 31% | 29% | 25% | 31% | 32% |
| *Priority+Share* | 14% | 19% | 12% | 13% | 15% |

TABLE 6: Reduction in throughput versus the *Static* policy due to prioritization.

**Automatic architecture selection is crucial for performance.** The ability of the trained models to accurately predict performance is crucial for achieving good performance. Our methodology correctly predicted the most performant architecture for 11 out of 12 benchmarks (while the mispredicted application experienced minimal performance loss). Evaluating these models is lightweight, making them suitable for use in production systems at runtime. Additionally, dynamic architecture selection provided additional performance improvements. In particular, `pathfinder` experienced the best performance improvement when leveraging dynamic architecture selection.

**Dynamic architecture selection increases platform flexibility.** When using dynamic architecture selection as shown in Figure 7b, applications can achieve speedups over a static architecture selection. AIRA is able to gain performance benefits from switching execution between architectures dynamically, and `pathfinder` experiences the largest latency reduction of any benchmark when utilizing this ability.

**Sharing compute resources provides whole-platform benefits.** Supervising allocation of CPU cores between applications provided significant performance improvements,

even for applications executing compute kernels on the GPU. Whole-platform performance was improved by reducing the number of threads competing for CPU time (consistent with [15]). New mechanisms that allow software-controlled allocation of GPU resources such as accelOS [46] would provide additional performance improvements.

**Flexible compute kernel execution has a variety of use cases.** It was trivial to add a prioritized resource allocation policy to AIRA's load balancer. Using this policy in conjunction with OS scheduling for platform-wide resource allocations provides significantly better speedups through CPU core partitioning and device locking. The flexibility afforded by AIRA has many applications for workloads in platforms ranging from SoCs to the datacenter.

## 6 Conclusion

As heterogeneous platforms become ubiquitous and increasingly multiprogrammed, it becomes more important that systems software provides execution management. We introduced AIRA, a framework to automatically enable flexible execution of compute kernels in heterogeneous platforms from multiple applications. AIRA provides offline tools to automatically refactor and analyze applications, relieving the developer from having to manually instrument application code. AIRA uses compute kernel features gathered by the feature extractor in conjunction with current system load to make resource allocation decisions using several policies. By leveraging AIRA, we demonstrated that there are significant benefits obtained by dynamic architecture selection and spatial partitioning of processing resources versus a static resource allocation that relies on time-multiplexing resources among concurrently executing applications. On a server-class CPU/GPU platform, AIRA predicts the most suitable architecture for a compute kernel. This was crucial for good performance – applications experienced a 2.7x slowdown on average when executed on the wrong architecture. This architecture selection can be adjusted at runtime to obtain up to a maximum of 3.78x speedup, with an average of 16% speedup. Moreover, applications experienced up to a mean of 83% speedup and the platform experienced up to a mean of 87% throughput improvement when cooperatively sharing the high core count CPU. This leads us to conclude that in heterogeneous systems, both dynamic architecture selection and resource sharing can increase application performance and system throughput. Additionally, cooperative resource allocation decisions are advantageous for heterogeneous-ISA platforms. For future work, there are many new platforms and hardware features which can be utilized by AIRA. Inter-application interference can be reduced by new hardware capabilities, e.g. cache partitioning [51]. Additionally as new shared-memory heterogeneous platforms emerge [4], [52] AIRA can be extended to better coordinate memory placement within heterogeneous memory hierarchies. Other emerging platforms [5] will enable higher execution flexibility, allowing more fine-grained execution management.

## Acknowledgments

This work is supported by the US Office of Naval Research under Contract N00014-12-1-0880.

## References

[1] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," March 2005, http://www.gotw.ca/publications/concurrency-ddj.htm.

[2] ——, "Welcome to the jungle," August 2012, http://herbsutter.com/welcome-to-the-jungle.

[3] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE, 2014.

[4] D. Bouvier, B. Cohen, W. Fry, S. Godey, and M. Mantor, "Kabini: An AMD accelerated processing unit system on a chip," *Micro, IEEE*, vol. 34, no. 2, pp. 22–33, 2014.

[5] HSA Foundation, "HSA foundation members preview plans for heterogeneous platforms," October 2015, http://www.hsafoundation.com/hsa-foundation-members-preview-plans-heterogeneous-platforms/.

[6] J. Stuecheli, B. Blaner, C. Johns, and M. Siegel, "CAPI: A coherent accelerator processor interface," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 7–1, 2015.

[7] PCWorld, "Intel's first processor with performance-boosting FPGA to ship early next year," November 2015, http://www.pcworld.com/article/3006601/components-processors/intels-first-server-chip-with-performance-boosting-fpga-to-ship-early-next-year.html.

[8] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU concurrency with elastic kernels," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 407–418. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451160

[9] J. Mars and L. Tang, "Whare-map: heterogeneity in homogeneous warehouse-scale computers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ACM, 2013, pp. 619–630.

[10] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte, "The case for GPGPU spatial multitasking," in *IEEE International Symposium on High-Performance Comp Architecture*, Feb 2012, pp. 1–12.

[11] Q. Jiao, M. Lu, H. P. Huynh, and T. Mitra, "Improving GPGPU energy-efficiency through concurrent kernel execution and DVFS," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 1–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2738600.2738602

[12] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*. IEEE, 2009, pp. 44–54.

[13] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads," in *Proceedings of the International Symposium on Workload Characterization (IISWC)*. IEEE, 2010, pp. 1–11.

[14] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[15] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: operating system abstractions to manage GPUs as compute devices," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 233–248.

[16] J. Duato, A. J. Pea, F. Silla, R. Mayo, and E. S. Quintana-Ort, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," in *2010 International Conference on High Performance Computing Simulation*, June 2010, pp. 224–231.

[17] Y.-P. You, H.-J. Wu, Y.-N. Tsai, and Y.-T. Chao, "VirtCL: a framework for OpenCL device abstraction and management," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2015, pp. 161–172.

[18] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *ACM SIGPLAN Notices*, vol. 41, no. 11. ACM, 2006, pp. 185–194.

[19] E. Ïpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, *Efficiently exploring architectural design spaces via predictive modeling*. ACM, 2006, vol. 40, no. 5.

[20] P. Thoman, K. Kofler, H. Studt, J. Thomson, and T. Fahringer, "Automatic OpenCL device characterization: guiding optimized kernel design," in *Proceedings of the Euro-Par Parallel Processing Conference*. Springer, 2011, pp. 438–452.

[21] I. Baldini, S. J. Fink, and E. Altman, "Predicting GPU performance from CPU runs using machine learning," in *2014 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 2014, pp. 254–261.

[22] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "SnuCL: an OpenCL framework for heterogeneous CPU/GPU clusters," in *Proceedings of the 26th ACM International Conference on Supercomputing (ICS)*. ACM, 2012, pp. 341–352.

[23] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: A Programming Model for Heterogeneous Multi-core Systems," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems (ASPLOS XIII)*, vol. 36, no. 1. Seattle, WA, USA: ACM, Mar. 2008, pp. 287–296.

[24] J. Auerbach, D. F. Bacon, I. Burcea, P. Cheng, S. J. Fink, R. Rabbah, and S. Shukla, "A compiler and runtime for heterogeneous computing," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 271–276.

[25] C. J. Rossbach, Y. Yu, J. Currey, J.-P. Martin, and D. Fetterly, "Dandelion: a compiler and runtime for heterogeneous systems," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 49–68.

[26] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: a proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[27] S. Panneerselvam and M. M. Swift, "Operating systems should manage accelerators," in *Proc. of the Second USENIX Workshop on Hot Topics in Parallelism, Berkeley, CA*, 2012.

[28] M. K. Emani, Z. Wang, and M. F. O'Boyle, "Smart, adaptive mapping of parallelism in the presence of external workload," in *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2013, pp. 1–10.

[29] T. Harris, M. Maas, and V. J. Marathe, "Callisto: co-scheduling parallel runtime systems," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 24.

[30] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar, "HASS: a scheduler for heterogeneous multicore systems," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 66–75, 2009.

[31] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[32] D. Grewe, Z. Wang, and M. F. O'Boyle, "Portable mapping of data parallel programs to OpenCL for heterogeneous systems," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2013, pp. 1–10.

[33] Y. Wen, Z. Wang, and M. F. O'Boyle, "Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms," in *High Performance Computing (HiPC), 2014 21st International Conference on*. IEEE, 2014, pp. 1–10.

[34] K. Kofler, I. Grasso, B. Cosenza, and T. Fahringer, "An automatic input-sensitive approach for heterogeneous task partitioning," in *Proceedings of the 27th international ACM conference on International conference on supercomputing (ICS)*, 2013, pp. 149–160.

[35] AMD, "Software optimization guide for AMD family 15h processors," AMD, Tech. Rep., January 2012, http://developer.amd.com/wordpress/media/2012/03/47414_15h_sw_opt_guide.pdf.

[36] D. Quinlan, C. Liao, J. Too, R. Matzke, M. Schordan, and P.-H. Lin, "ROSE compiler infrastructure," November 2013, http://rosecompiler.org/.

[37] R. F. Lyerly, "Automatic scheduling of compute kernels across heterogeneous architectures," Master's thesis, Virginia Polytechnic Institute and State University, 2014.

[38] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP programming and tuning for GPUs," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–11.

[39] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A source-to-source compiler infrastructure for multicores," *Computer*, vol. 42, no. 12, pp. 36–42, 2009.

[40] J. Hilland, P. Culley, J. Pinkerton, and R. Recio, "RDMA protocol verbs specification," *RDMAC Consortium Draft Specification draft-hilland-iwarp-verbsv1. 0-RDMAC*, 2003.

[41] B. Yegnanarayana, *Artificial neural networks*. PHI Learning Pvt. Ltd., 2009.

[42] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," DTIC Document, Tech. Rep., 1985.

[43] I. Jolliffe, *Principal component analysis*. Wiley Online Library, 2005.

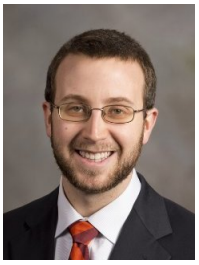
[44] I. Molnar, "Modular scheduler core and completely fair scheduler [CFS]," September 2013, http://lwn.net/Articles/230501/.

[45] J. R. Quinlan, "Simplifying decision trees," *International journal of man-machine studies*, vol. 27, no. 3, pp. 221–234, 1987.

[46] C. Margiolas and M. F. P. O'Boyle, "Portable and transparent software managed scheduling on accelerators for fair resource sharing," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO '16. New York, NY, USA: ACM, 2016, pp. 82–93. [Online]. Available: http://doi.acm.org/10.1145/2854038.2854040

[47] Intel, "Improving real-time performance by utilizing cache allocation technology," April 2015, http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf.

[48] S. Gupta, "IBM POWER8 CPU and NVIDIA Pascal GPU Speed Ahead with NVLINK," April 2016, https://www.ibm.com/blogs/systems/ibm-power8-cpu-and-nvidia-pascal-gpu-speed-ahead-with-nvlink/.



**Robert Lyerly** received a BS degree in Computer Science, and BS/MS degrees in Computer Engineering from Virginia Tech, USA. He is currently working towards his PhD with the Systems Software Research Group at Virginia Tech, led by Professor Binoy Ravindran. His current research interests include compilers, run-times, operating systems, and heterogeneous architectures.

**Alastair Murray** Alastair Murray received BSc Hons. and PhD degrees in Computer Science from the University of Edinburgh, Edinburgh, U.K. in 2006 and 2012 respectively. In 2012 he started a post-doc with the System Software Research Group in the ECE Department at Virginia Tech, Blacksburg, U.S.A. Since 2014 he is currently a Senior Software Engineer in Compilers at Codeplay Software Ltd, Edinburgh, U.K. His research interests cover language-design, language-implementation and compilation techniques for parallel and heterogeneous devices.

**Antonio Barbalace** received BS/MS degrees in Computer Engineering, and a PhD in Industrial Engineering from the University of Padova, Padua, Italy. He is a Research Assistant Professor in the ECE Department at Virginia Tech. His main areas of research are operating systems, virtualization environments, runtime libraries, and compilers for parallel and distributed computer architectures. He is broadly interested in performance optimization and energy consumption minimization.

**Binoy Ravindran** is a Professor of Electrical and Computer Engineering at Virginia Tech, where he leads the Systems Software Research Group, which conducts research on operating systems, run-times, middleware, compilers, distributed systems, concurrency, and real-time systems. Ravindran and his students have published more than 230 papers in these spaces. His group's papers have won the best paper award at 2013 ACM ASP-DAC, 2012 USENIX SYSTOR, and selected as one of The Most Influential Papers of 10 Years of ACM DATE (2008) conferences. Dr. Ravindran is an ACM Distinguished Scientist.