Self-Configuring Applications for Heterogeneous Systems: Automating Programming Decisions Using Cognitive Techniques

Mary Hall, Yolanda Gil, and Robert Lucas, *Senior Member, IEEE*

Abstract—**This paper describes several challenges facing programmers of future edge computing systems, the complex and diverse multi- and many-core devices that will soon exemplify commodity mainstream systems. To call attention to programming challenges ahead, this paper focuses on the most complex of such architectures: integrated, power-conserving systems, inherently parallel and heterogeneous, with distributed address spaces. When programming such complex systems, several new concerns arise, such as computation partitioning across functional units, data movement and synchronization, managing a diversity of programming models for different devices, and reusing existing legacy and library software. We observe that many of these challenges are also faced in programming applications for large-scale, heterogeneous distributed computing environments, and solutions used in practice as well as future research directions in distributed computing can be adapted to edge computing environments. Further, optimization decisions are inherently complex due to large search spaces of possible solutions and the difficulty of predicting performance on increasingly complex architectures. Cognitive techniques are well-suited for managing systems of such complexity. We discuss how recent trends of using cognitive techniques for code mapping and optimization support this point. We describe how cognitive techniques could provide a fundamentally new programming paradigm for complex heterogeneous systems, where programmers design self-configuring applications and the system automates optimization decisions and manages the allocation of heterogeneous resources to codes.**

***Index Terms*—Optimizing compilers, Learning systems, Computer architectures, Distributed computing, Multi-core architectures, Self-configuring applications**

I. Introduction

Heterogeneous systems are comprised of a variety of special-purpose computing engines, complex memory hierarchies, and interconnects that link all of these resources together. Technology advances such as exponentially increasing chip densities have pushed hardware designers towards devices with multiple processing cores to better manage design costs and energy consumption. Heterogeneous devices can further exploit specialized functional units to increase performance and manage power for different phases of a computation. Currently, there are available a proliferation of systems with heterogeneous processing capability at various scales – from systems-on-a-chip FPGAs such as the Xilinx Virtex 4, standard PCs with graphics processors, heterogeneous chip architectures such as the IBM Cell, domain accelerators such as Clearspeed, high-end systems that incorporate co-processors such as the Cray XD1, and distributed systems comprised of clusters of diverse resources.

Heterogeneous platforms can accelerate many applications that mix compute-intensive and control-intensive phases of computation, which are best targeted to different processing elements. These applications include large-scale scientific computations, complex simulations of physical phenomena, and visualizations. One such example is entity-level simulations, wherein people and other entities are simulated by independent agents, are widely used for training and simulation. The US Joint Forces Command's (JFCOM) Urban Resolve experiment is an entity-level simulation that models military operations in urban terrain [Ceranowicz 05]. A million or more civilian entities are simulated to represent the complexity of a modern urban environment [Lucas 04]. These entities must determine if they are in the line-of-sight (LOS) of other entities, in particular models of sensor platforms. As the scale and complexity of military training and experimentation increases, the time to determine whether or not entities can see each other can quickly become the computational bottleneck. At their peak, LOS calculations can consume over half of the CPU time in urban scenarios such as JFCOM's Urban Resolve. To ensure that experiments progress in real time, Urban Resolve software engineers are forced to roughly halve the number of entities they could otherwise simulate. Recent results demonstrate that LOS computations are well suited for execution on graphics co-processing units (GPUs), providing an order of magnitude improvement in LOS performance [Salomon 04]. As a related example, traffic simulation of entire metropolitan areas has

Manuscript received March 17, 2007. This work was supported in part by the Department of Energy Scientific Discovery through Advanced Computation DE-FC02-06ER25765, National Science Foundation project CSR-0615412, and by a gift from Intel Corporation.

All authors are with the University of Southern California / Information Sciences Institute; 4676 Admiralty Way; Marina del Rey, CA 90292. Phone: 310-822-1511; fax: 310-823-6714; email: {mhall,gil,rflucas}@isi.edu.

been demonstrated using a mix of FPGA and conventional hardware [Tripp et al. 05].

While heterogeneous systems are promising from a performance and power perspective, additional programming challenges arise, including: (1) partitioning of the application across functional units; (2) managing data movement between functional units; (3) differences in programming models and tools across functional units; and, (4) managing reuse of code developed by others. Further, edge computing applications must be highly optimized for performance and power consumption. Just to put these challenges in perspective, (1), (2) and (4) are also challenges in porting across different *homogeneous* platforms. In the absence of tool support for this set of challenges, programmers of heterogeneous platforms must explicitly manage these details, which can dominate all other aspects of application programming for heterogeneous systems. The net result is that developing and debugging programs on such systems can be quite tedious, and is only approachable by highly-skilled individuals. ***A tremendous need exists for new approaches that can both increase the productivity of these highly-skilled developers and make these powerful systems more accessible to a broader group of users.***

Programmers of distributed applications face similar challenges. Here, the focus is on collections of codes that are submitted for execution on distributed, heterogeneous resources, where the available resources may not be known prior to execution. These applications are broken down into coarse-grain components, and each is assigned an aggregate resource (possibly a cluster) for execution. The flow of data across components, usually external files, must be managed appropriately, as well as any dependencies. Programmers of these distributed applications face a very diverse set of resources and requirements. They also must manage software components developed by others, much like programmers of heterogeneous architectures incorporate software libraries developed by others.

In this paper, we discuss programming challenges that, in our view, are common to heterogeneous platforms and to distributed heterogeneous systems. We introduce the approaches taken so far in these two communities, and argue that many of the challenges and approaches are generally applicable to both kinds of heterogeneous computing. This strategy is the result of our experiences in: (1) developing search-based compiler optimization for conventional memory hierarchies, FPGAs, and SIMD units in multimedia extensions and processing-in-memory devices; (2) cognitive techniques for managing large-scale distributed applications; and, (3) large-scale applications that will target edge computing systems.

Our key observation is that a systematic and principled approach to developing and executing heterogeneous applications, making use of cognitive techniques, can greatly increase programmer productivity and application performance for edge computing platforms. We introduce the concept of ***self-configuring applications***, a new paradigm for heterogeneous programming that combines model-based software development and empirically-driven optimization. The role of a programmer with such an approach is to design models for software components that declare explicitly parameters and other aspects that can be optimized and managed by the underlying system. The system, in turn, takes a proactive role in configuring the application by searching through the space of possible configurations guided by information learned from empirical data derived from prior executions.

TABLE I
PROGRAMMING EDGE COMPUTING VS. DISTRIBUTED COMPUTING

	Edge Computing	*Distributed Computing*
Programming Model Diversity	*Parallel languages:* pthreads, MPI, OpenMP, transactions; *GPU languages:* Ng, Cuda; *FPGA:* VHDL or Verilog; *Stream processing:* StreaMIT, Napa-C	Sequential, OpenMP, MPI, Global Address Space Languages, mixed languages (C, Fortran, C++, Java)
Libraries and Components	*Sources*: Device-specific libraries; domain-specific libraries; application libraries. *Interface.*	*Sources:* Target-specific code; domain-specific libraries; large-scale scientific simulation. *Interface.*
Computation Partitioning	*Stream processing:* DSP, GPU, Multimedia extension; *Control-intensive:* General-purpose; *Configurable:* FPGA	*Coarse-grain MPI code:* large number of nodes; *Data sharing:* SMP; *Sequential:* single node
Data Movement and Synchronization	*Communication and copying:* To/from shared memory; To/from buffers; Between functional units. *Impact on schedule.*	*Data Products in Files:* To/from different nodes or clusters; Data in catalogs; Intermediate data; *Impact on schedule.*

The remainder of the paper is organized as follows. Section II discusses the programming challenges of heterogeneous platforms in more detail, drawing similarities to distributed computing environments. Section III then proposes self-configuring applications as a programming paradigm for heterogeneous platforms that incorporates cognitive techniques to represent and manage complex mapping decisions. Section IV concludes the paper.

II. PROGRAMMING CHALLENGES

This section discusses the previously mentioned programming challenges, first as they apply to heterogeneous edge computing platforms and second to distributed heterogeneous systems, with a summary in Table I. In the next section, we will discuss strategies for how to address these challenges. For this section, we assume an abstract architecture which includes a set of interconnected functional units of different capabilities, each with their own local memories, and with a global store. One of the devices may be a host processor which maintains control of the other devices. Examples of different devices include commodity general-purpose processors, SIMD functional units such as multimedia extensions, DSPs, FPGAs, GPUs, and domain accelerators such as Clearspeed.

A. Diversity of Programming Models

Perhaps the biggest challenge in targeting heterogeneous platforms is that each device may support a completely different programming model. An extreme example is a system that includes general-purpose processors, which are programmed with standard programming languages such as C and Fortran, coupled with FPGAs, which are traditionally programmed with hardware description languages such as VHDL and Verilog. While some C dialects exist to program FPGAs, for the most part these impose requirements on the C constructs that can be used and require hardware descriptions that are quite different from standard software programs. GPUs also have their own programming languages, such as, for example, Ng (for graphics programs) and Cuda (for application programs) by Nvidia. GPU-specific languages are needed because GPUs have a unique hierarchical structure of processing units and memories, and many constraints on the type and number of instructions that certain functional units can execute. Stream processing and data-parallel languages, which allow programmers to express repeated operations on data streams, can be used to target digital signal processing (DSP), multimedia extensions and other similar devices. In each case, the programming model facilitates expressing computation well-suited to the device while managing the constraints imposed by the architecture.

In distributed computing applications, codes are often written in different languages, and for different target architectures and classes of target architectures. Sequential portions of the code may be written in Java for a uniprocessor, while other portions are large-scale codes written in MPI. Still others might be developed for shared-memory multiprocessors and written in OpenMP, and hybrid MPI and OpenMP programs and use of modern global address-space languages such as UPC are becoming more common. Beyond the diversity of programming languages, distributed applications incorporate application binaries that have been already compiled for specific target architectures. Programmers need to select appropriate executables once the resources are selected. Further, beyond selecting resources based on the application properties, different types of resources are managed by different systems. Jobs may be submitted through different queues that have different policies regarding job length and priorities. Programmers also need to understand how much computation to fit within a job, by estimating the time that each of their many computations will take to execute and grouping them into jobs of a reasonable size. Creating adequate granularity for computations based on resource policies is an art, often learned through trial and error.

While it might seem beneficial if all of these resources could be programmed using the same universal programming model, this is an unrealistic goal, as it would: (1) compromise the strengths of the existing programming models for their own platforms; (2) make it impossible to incorporate legacy code into edge computing applications; and, (3) fail to be adopted by the community, as there will never be consensus on what is the right universal programming model. The appropriate strategy for managing a diversity of programming models must address these three considerations, as discussed in Section III.

B. Legacy Code and Highly-Tuned Libraries

Incorporating legacy code into edge computing platforms includes not only existing application code, but also highly-tuned libraries that have been designed for a particular device type. Vendors of commodity devices develop highly-tuned libraries to encourage adoption of their hardware. Prior to the availability of compilers and other programming tools for a new platform, such libraries provide a shortcut to obtaining high performance for application domains with common core computations that can be developed into libraries. Similarly, researchers develop highly-tuned domain-specific libraries designed to be portable across platforms, with notable examples including BLAS and ATLAS for linear algebra [Whaley 05] and FFTW [Frigo 05] and SPIRAL [Puschel 05] for signal processing.

In distributed computing, applications are often formed by composing codes developed independently. Many alternative encapsulations may be possible for already existing codes. Further, it is possible that various different implementations of the same component are available, each optimized for a specific architecture. Programmers then need to select the appropriate implementation for the resources assigned to the computation. This becomes a complex decision when many resources and many alternative implementations may be available and thus the space of choices becomes unmanageable and hard to optimize by hand.

C. Partitioning Across Functional Units

Edge computing systems include a diversity of functional units, each with unique capabilities for which they are best suited. If an application kernel is data-intensive, performing repeated operations on a stream of data, a DSP or GPU may be a good target functional unit. If it is instead control-intensive, a general-purpose processor or a special-purpose control processor would be a better choice. If specialized arithmetic modes are needed, these tend to be supported best by DSP or multimedia extension architectures. FPGAs are effective for computations that can be highly specialized to exploit parallelism, fine-grain data movement such as bit manipulation, and custom data types such as arithmetic on small objects. The decision as to which functional unit is best for a computation is called *computation partitioning*. This decision must consider not only which device can perform the computation most efficiently, but also whether the benefit of performing a computation on a particular device outweighs the cost of migrating the computation and associated data to/from the device, as discussed in Section D.

Similarly, complex distributed applications are often broken down into components that are designed for specific architectures depending on their characteristics. For example, a component that realizes a highly parallelizable code may be implemented as an MPI code optimized for a 128-node cluster, while another component may contain simpler computations and be implemented as a Java code that will work on any 64-bit Opteron. Parallel codes that have a lot of data sharing, and would perform poorly on clusters, may be better-suited for shared-memory multiprocessors. There are natural boundaries created by the requirements of the different aspects of the computation, and therefore the applications end up being modularized accordingly. Programmers, however, need to manage partitioning as well, by grouping together components that can be bundled for execution in a single resource so as to reduce queuing time.

D. Managing Data Movement and Synchronization

For an architecture that includes a set of independent, interconnected functional units, a key challenge in developing both correct and high-performing applications is managing the movement of data and synchronization among functional units. Data must be staged not only between functional units, but between global memories and local memories or buffers. This requires careful scheduling of data movement relative to computations, to ensure the correct data is available when it is needed.

In distributed computing, data must be staged in and out of catalogs where data collections are stored and managed. The results of intermediate computations that will be used in subsequent computations at the same location should remain available if possible. Programmers must be aware of the location of intermediate data products in making decisions about later computations, and must also be aware of memory requirements and availability. If memory availability at a location is insufficient, failures must be analyzed and repaired either through higher memory requests or by assigning the computation to alternate locations.

Related to computation partitioning, programmers of both types of systems also must consider cost of data movements among resources where the computations occur. When data movements take significant time compared to the time taken by the computations, selecting suboptimal resources for the computations themselves may be preferable if the data movements are reduced significantly.

E. Summary

This section described a large set of decisions that must be made by programmers of heterogeneous systems, representing a vast tradeoff space of possible solutions. Not only do these decisions affect performance of the result, but they also must be carefully managed to guarantee a correct program. ***The difference in complexity, in terms of development time, between a sequential application written for a homogeneous system and a parallel or distributed application for a heterogeneous platform can be several orders of magnitude, with the resulting code size also growing by more than an order of magnitude.*** Errors abound, as with any manually managed process, and users need to be able to understand error conditions and repair failures. Extending the application has cascading effects that have to be managed manually, making it impractical unless the additions were anticipated in advance. As we move to increasingly heterogeneous platforms of the future, having appropriate architecture-independent programming paradigms becomes of paramount importance.

III. Self-Configuring Applications: A New Programming Paradigm for Heterogeneous Computing

In the previous section, we discussed a set of programming challenges currently faced by programmers of edge computing systems. In this section, we propose an approach, based on partial solutions from the literature, which would provide assistance or automate many of these decisions currently left to the programmer. Some of this prior art comes from existing distributed computing systems, while the rest reflects new strategies for code optimization. ***We introduce the concept of self-configuring applications, whereby the programmer expresses an application as a high-level workflow comprised of tunable software components that are abstractions of implemented codes.*** The high-level workflow is instantiated and optimized for the edge computing platform, in the presence of training data that is representative of real execution environments. The optimization process relies on empirical search to execute and evaluate portions of a collection of equivalent alternative implementations of the workflow for the most suitable implementation. Machine learning, a rich knowledge representation, and an experience base aid in pruning and navigating the search space. Thus, through a systematic and principled strategy for formulating application optimization for heterogeneous platforms, the programmer's partial specification of a high-level workflow is realized as an edge computing application, as will be discussed in this section.

A. Self-Tuning and Selectable Components

Workflows represent complex applications as components and their associated data flow. Before describing how workflows are expressed, we first focus on properties of components that could be used to instantiate and optimize workflows for edge computing. A component is a code segment packaged with the interfaces needed for someone other than the component developer to correctly invoke and use the component in an application. Component technology has been commonly used for over a decade as a strategy for facilitating code reuse and interoperability. As one notable example, the Common Object Request Broker Architecture (CORBA) facilitates interoperability between two programs, potentially written in different languages and executing on different vendor's platforms. Despite the existence of CORBA and other component standards, where high performance or efficiency is required, such as in very high-end systems and embedded systems with hard timing and resource constraints, component technology is usually dismissed by application developers, who perceive component packaging as introducing too much overhead to be practical. Due to the growing complexity of application codes in these high performance or high efficiency

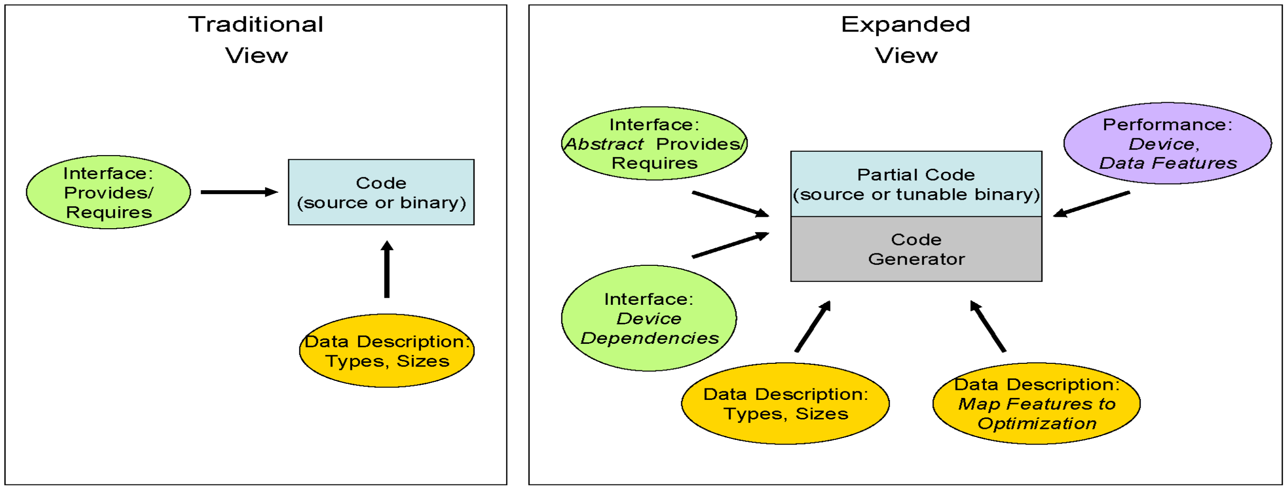
Traditional View
Interface: Provides/ Requires
Code (source or binary)
Data Description: Types, Sizes
Expanded View
Interface: *Abstract* Provides/ Requires
Partial Code (source or tunable binary)
Code Generator
Performance: *Device, Data Features*
Interface: *Device Dependencies*
Data Description: Types, Sizes
Data Description: *Map Features to Optimization*

Fig. 1. A comparison of traditional and proposed components.

regimes, recent efforts have introduced component technology that has lower overheads and/or richer interfaces, e.g., Common Component Architecture (CCA) for high-end computing [Allan et al. 06], and DoD's Software Communications Architecture (SCA) for embedded software radio applications.

Why components are needed?

The use of components in edge computing addresses several of the challenges in Section II. From the workflow perspective, it provides a sufficiently coarse-grain boundary for reasoning about application composition and optimization. It provides a unified mechanism for applications that mix legacy code, highly tuned libraries, and new self-configuring components either written by an application developer or generated automatically by a compiler. At the component level, the same programming model is used, regardless of functional unit and data movement protocol. Details of invoking a particular functional unit are managed by the system, replacing abstract data movement specifications with device-specific libraries. In the extreme example of programming model diversity, combining general-purpose with FPGA devices, computations to be performed on the FPGA appear no different than other software components. This overall strategy is consistent with Sun's Fortress language for High Productivity Computing, which is designed around the concept of building applications through composing libraries.

Requirements for components of self-configuring applications

A component for self-configuring applications must support automatic code selection and achieve high performance or high efficiency, and therefore goes beyond the traditional concept of facilitating software reuse and interoperability. The requirements are as follows:

- *Interchangeable code variants:* the interface describes features of the implementation and their relation to input data characteristics. This information can be used by the system to select the appropriate component for a given functional unit.
- *Self-tuning:* the interface exposes optimization parameters used to adjust performance based on data features.
- *Lightweight:* the executable interface must have low execution time and storage overheads.

The remainder of this subsection focuses on details of these properties and how they are used in self-configuring applications.

Components as part of workflows.

To manage the workflow at the application level and at the resource level, the component models used by workflow systems include three distinct aspects: a functional model, an invocation model and a performance model. The functional model includes the type and order of input data, arguments and parameter information, and output data. The functional model is used to communicate with the programmer at the application level. It also allows the workflow system to reason about the interactions among components in the workflow, and support programmers in workflow creation and validation. The invocation model provides the information needed by the system to map components for execution: a calling sequence, a pointer to the source or binary code, the device for which it has been developed, and any other requirements. In distributed computing, the invocation model is used by the system to generate mappings to execution sites that comply with the component requirements. The invocation model is also used to manage and optimize data movements. The performance model will capture all of the aspects of the component used in optimization, including expected execution time of the code on a particular device based on training data, and parameters of optimization that can be adjusted off-line or dynamically, such as loop unroll factors, a mapping between data set features and values of these parameters. These performance models could be improved over time through empirical data of actual performance of the component.

Given this set of models, workflows may include abstract descriptions of components. That is, instead of specifying a particular component implementation, the workflow can refer

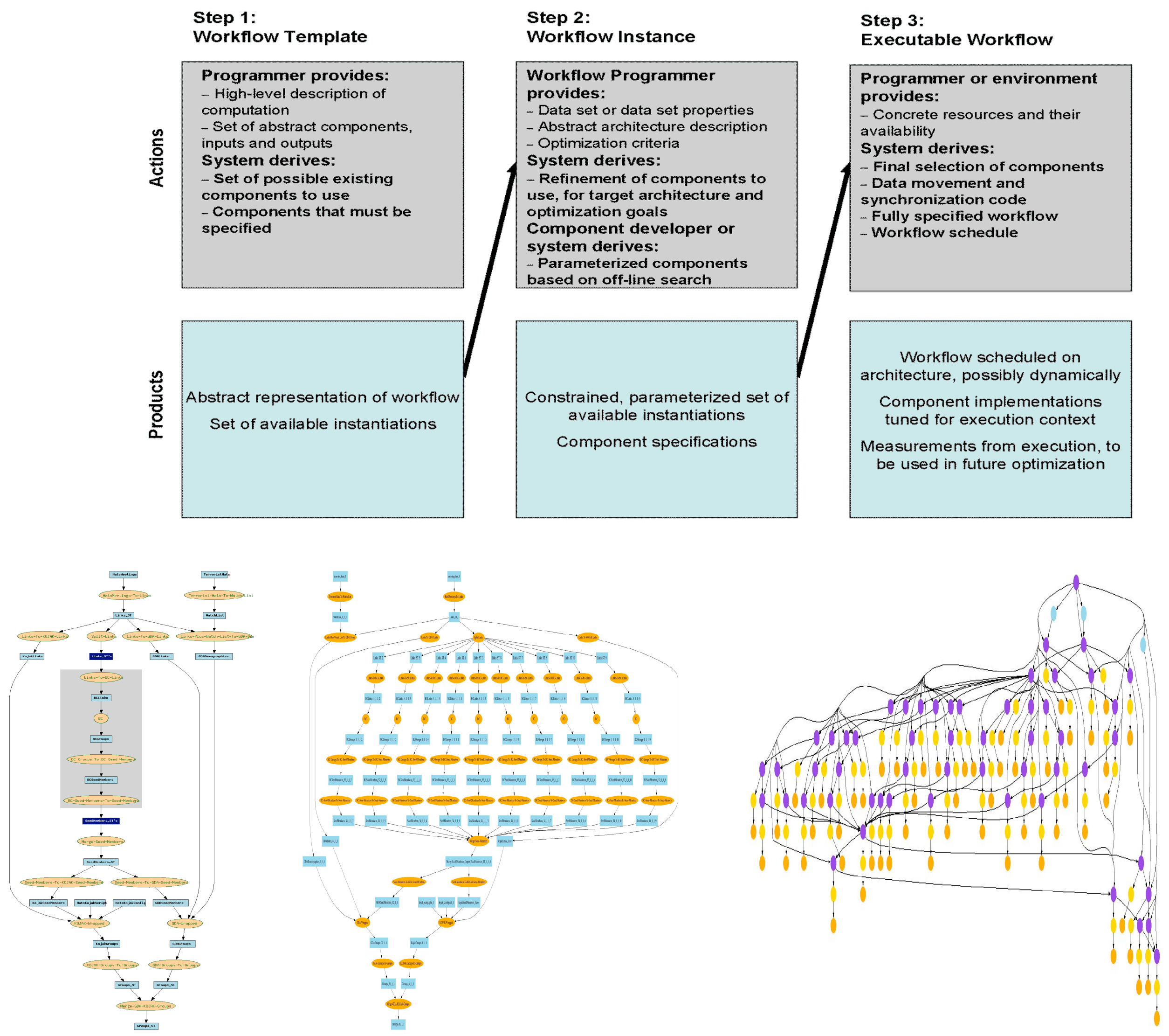
Step 1:
Workflow Template
Step 2:
Workflow Instance
Step 3:
Executable Workflow
Actions
Products
Programmer provides:
– High-level description of computation
– Set of abstract components, inputs and outputs
System derives:
– Set of possible existing components to use
– Components that must be specified
Workflow Programmer provides:
– Data set or data set properties
– Abstract architecture description
– Optimization criteria
System derives:
– Refinement of components to use, for target architecture and optimization goals
Component developer or system derives:
– Parameterized components based on off-line search
Programmer or environment provides:
– Concrete resources and their availability
System derives:
– Final selection of components
– Data movement and synchronization code
– Fully specified workflow
– Workflow schedule
Abstract representation of workflow
Set of available instantiations
Constrained, parameterized set of available instantiations
Component specifications
Workflow scheduled on architecture, possibly dynamically
Component implementations tuned for execution context
Measurements from execution, to be used in future optimization

Fig. 2. Creating a self-configuring application as a workflow. The top portion describes the steps, and the bottom portion shows an example workflow.

to a category or type of component. For example, a workflow could simply indicate the functional characteristics required to process data at that particular point in the application. These functional characteristics may then be realized by multiple components. This gives the system tremendous flexibility in terms of choosing which instantiation of the component to select based on the input data and the resources available for a given execution.

Figure 1 shows how components appear in the system to support this process. In a traditional view, component descriptions include code (source or binary) as well as an interface that describes the functional behavior of the component (what functions are provided externally and required from the external environment), and a description of the data that will be provided as input to the component. These notions are extended in the proposed framework. In terms of interface, for the functional model, it must match an abstract interface that describes a set of alternative implementations to be selected during workflow instantiation. To support the execution model, any device dependences are described – for binaries targeting a specific device, the device itself is reported, while for source code, the programming model and any compiler dependences are described, in effect an assertion of what tools are likely to support the component. In addition, rather than including code, partial code is provided (either parameterized source or tunable binary), with a code generation mechanism to finalize code generation in the context of the provided data. Included with the data description is a mapping of data features to code optimization strategy, to guide code generation. Finally, to support the performance model, expected or measured performance is reported for specific devices and specific data features. Many of these concepts have been demonstrated in the Standard Template Adaptive Parallel Library (STAPL) framework, which provides tuning parameters, data features and instantiated learners as part of the library components to enable dynamic algorithm selection [Yu et al. 2004][Thomas et al 2005].

Can we make components lightweight enough?
The minimal component interface must verify the correctness of the invocation model, and manage the movement of data between devices. More complexity may be added if optimization decisions including code selection and parameter identification are performed dynamically. While this additional abstraction will indeed add overhead, components ought to be somewhat coarse-grained – as we switch to a new functional unit, even a hand-tuned application also involves data copying. Thus, the overhead should be small relative to the key computations. The system can optimize workflows by merging small components for the same device, or exploit data-parallelism by splitting coarse-grained components, similar to the notions of combining and splitting streams in StreaMIT [Gordon et al. 02]. Recent results on component technology show a performance improvement in the use of components due to the benefits of breaking code into manageable chunks that are more effectively optimized by a compiler, as long as the per-processor computaition granularity of the component is sufficiently large [Yoon et al 07]. The SmartApps system employs customization of run-time services as part of the application code to reduce component overhead [Rauchwerger et al. 2000].

B. Composing and Optimizing Workflows

Representing workflows at multiple levels of abstraction is critical to facilitate code reuse and automate many implementation aspects that are not expressed directly in the application. We consider three stages of creation of workflows, each stage corresponding to a different type of information being added to the workflow:

1. Defining *workflow templates* that are data- and execution-independent specifications of computations. Workflow templates identify the types of components to be invoked and the data flow among them. The nature of the components may constrain the input data that the workflow is designed to process, but the specific data set to be used is not described in the template. This level of specification supports analysis of potential instantiations of the workflow, but there is not enough information to perform optimization. A workflow template should be shared and reused when performing the same type of computations.
2. Creating *workflow instances* that are execution-independent. Workflow instances specify the input data needed for an application. A workflow instance can be created by selecting a workflow template that describes the desired application and binding its data descriptions to specific data to be used (or representative data). The concrete optimization criteria are also specified here, such as whether to focus on end-to-end performance, throughput, power, or a combination, and if needed, specific requirements for each of these. While a workflow instance logically identifies the full application, it does not include execution details such as what hardware resources should be used or where data should be placed. That is, the same workflow instance can be mapped into different executable workflows that generate exactly the same results but use different resources available in alternative execution environments. A workflow instance can be optimized off line to derive a collection of possible components that can incorporated into the application, parameterized by anticipated features of the execution environment.
3. Creating *executable workflows*. Executable workflows are created by taking workflow instances and assigning actual resources that exist in the execution environment and reassigning them dynamically as the execution unfolds. Executable workflows fully specify the resources available in the execution environment (*e.g.,* hardware and memory resources) that should be used for execution. The system can also automatically insert requisite data movement and data staging steps. This mapping process ideally is incremental and dynamic. Only the initial workflow steps might be assigned to resources, while later steps can wait until the execution of the initial steps is finalized, responding to application behavior as it unfolds. Any decisions made during this mapping must be on line and therefore instantaneous. The complex tradeoffs must be evaluated for workflow instances and then used here.

The evolution of a workflow through these three stages is summarized in the upper portion of Figure 2 and illustrated in the lower portion with an example. Ovals indicate computations, and rectangles indicate data. The workflow template has a portion that will process data in parallel (highlighted by the dark grey rectangle). The workflow instance includes specific computations for each slice of the specific dataset to be processed. Finally, the executable workflow (showing here only computations and not data) includes all details required to execute the computations in different resources as well as data movement steps. More details of this process are described in [Gil 06b; Kim et al 07; Gil et al 07; Deelman et al 05].

In summary, self-configuring applications can be managed automatically when the system can access to: (1) expressive models of workflow components, (2) workflow representations at different levels of abstraction, and (3) alternatives for completing user-provided partial workflow descriptions. These enable increased automation of workflow creation and execution management tasks to relieve the programmer from the complexities of the heterogeneous execution environment.

C. Compiler Technologies and Generating Components

The preceding discussions of components and workflows did not address how the various facets of the components, beyond application code, would be derived. System-level and device-specific compilers must play a critical role in assembling this information, in conjunction with the execution environment which captures execution history.

How to generate self-configuring components.

Compiler optimization typically performs a static analysis of application code and generates a single binary of the code. Due to the complexity of today's architectures, such a static, purely model-based approach generates code that falls far short of hand-tuned levels of performance. Recent work has dealt with the complexity of modern architectures through empirical techniques, where optimization decisions are guided by executing code variants directly to measure and compare performance. This research began by developing domain-specific libraries such as ATLAS for linear algebra [Whaley et al. 05], FFTW [Frigo et al. 05] and SPIRAL [Puschel et al. 05] for signal processing, OSKI for sparse linear algebra [Vuduc et al. 2005] and sorting libraries [Li et al. 2004][Li et al. 2005] as notable examples.

For all of these domain-specific libraries, manual optimization strategies have been developed over many years, and as a consequence, much domain knowledge exists regarding how to yield high performance given different architectural features. This domain knowledge is incorporated into the optimization strategies. Empirical techniques simply select among alternative optimization strategies and identify values of optimization parameters.

While such libraries are without question very useful to obtaining high performance on edge computing platforms, an application developer cannot always count on the availability of such libraries to implement all the performance-critical portions of their application. Therefore, compiler technology that can incorporate empirical techniques to tune application code, if able to achieve close to hand-tuned levels of performance, would be highly desirable to dramatically improve programmer productivity.

This goal has inspired several compiler efforts that employ empirical optimization to evaluate alternative compiler optimization strategies, leading to much higher performance than the standard compiler approach that relies on static models. Examples of these include work on iterative compilation [Knijnenberg et al. 04], and recent work using polyhedral transformation frameworks [Cohen et al. 07].

Models have been shown to achieve results close to hand-tuned performance [Yotov et al. 05b], and sophisticated approaches using training and mutual information maximization can quickly generate fairly accurate models [Cavazos et al., 2006], but due to the growing complexity of predicting interactions of different architectural features, there is nevertheless a performance gap between optimizations based purely on models and those using empirical techniques.

A hybrid approach called *model-guided empirical optimization* combines the complementary strengths of compiler models and heuristics with empirical techniques [Chen et al 05a]. The models and heuristics limit the search to a small number of candidate implementations, and the empirical results provide the most accurate information to the compiler to select among candidates and tune optimization parameter values. Others have also shown success with hybrid techniques combining models and empirical search [Yotov et al. 05]. Using such an approach, compilers can achieve results comparable to hand-tuned for dense-array computations, as compared against hand-tuned libraries [Chen et al 05b, Yotov et al. 05].

Compiler as experiential engine.

Using model-guided empirical optimization, the compiler generates a set of experiments to perform a search among possible implementations and optimization parameter values. In the previous discussion, the experiments were the result of compiler algorithms. However, the compiler can also serve as an experiments engine for the application programmer, assisting with evaluating application-level parameters. For example, a programmer who wants to direct the compiler to evaluate a set of alternative unroll factors for a loop nest can be supported by an experiments engine that generates the alternative code segments, executes them, and determines the most appropriate for a class of data sets [Lee et al. 05].

Compilers that can be conveniently retargeted.

If as previously described we construct compilers that can systematically search a space of alternative optimization strategies, then by construction, the compiler design ought to also be systematic. We envision a compiler that is constructed in a principled way, so that optimizations are well-defined and easily composed. Today's compilers are monolithic and very large -- in the hundreds of thousands or even millions of lines of code. Optimization strategies are complex, sometimes fragile, and often obscured and buried within the compiler modules. Optimizations are typically composed by applying each one independently, rather than combining them to achieve the best effect.

We propose a compiler based on formal descriptions of optimization behavior that can easily be composed. While not possible for every optimization, several prior works describe such an approach for loop transformations, commonly used for array-based computations that arise in scientific computing, and multimedia applications [Kelly 96, Cohen et al. 05, Chen 07]. A similar idea has been described for more standard scalar optimizations [Soffa et al. 05]. Formalizing the optimization framework and making optimizations composable facilitates generating decision procedures for optimization that are easily understood and can be constructed or modified rapidly. This approach makes it possible to in turn rapidly retarget such decision procedures from one device architecture to another. In conjunction with searching, discussed in the next section, such an approach creates a powerful approach to compiler design that can assist in targeting heterogeneous resources.

D. Searching for the Best Solution

We have discussed so far how many different types of decisions could be made automatically to reduce the complexity faced by a programmer in edge computing platforms. Such decisions can take into account application-level features, such as the density of a graph or the connectivity within a partitioned data set, or target hardware architectures, such as data storage locations, execution resources, data movements, etc. Today, these decisions and the associated search are hard-coded in the compilers or the workflows. When a new hardware platform appears, costly resources must be devoted to writing new compiler systems that manage those decisions in a manner that is customized to the particular features of the platform. Today, we already find that it is expensive to experiment and explore the best compiler design for a new platform. For the new edge computing architectures of tomorrow, this mode of operation will be absolutely impractical.

Cognitive techniques provide a systematic and principled paradigm to manage these decisions and to automate as many of them as possible. We envision two major benefits to come from these techniques. First, we anticipate significant productivity gains for programmers. Because programmers today must manually experiment and learn how to optimize codes for each type of application and for each type of target architecture, programming is largely manual and has a costly learning curve. The second anticipated benefit is major performance improvements for applications. Because programmers today are not necessarily knowledgeable of the best optimization strategies for a given architecture, we believe that automating many of these decisions will lead to significant performance improvements across the spectrum of applications.

A rich area for cognitive search techniques concerns global decisions regarding the ordering of optimizations to apply to the code. This issue is complicated by the fact that there may be complex interactions among the decisions involved in the various optimization steps that affect the performance of the resulting code, and the fact that the right decision depends on the target architecture [Kisuki et al 00]. The work by [Almagor et al 04] exemplifies the benefits of representing and analyzing the search space to improve performance. Three distinct search algorithms generated orderings with 15% to 25% better performance than the human-designed fixed sequence originally used in the compiler. This work provides strong evidence that search techniques could effectively improve the performance of optimizing compilers.

In distributed environments, specialized cognitive search algorithms such as planning and scheduling have been used to select workflow components and to select execution resources [Gil et al 04; Blythe et al 03; Blythe et al 05]. Optimization techniques that include workflow restructuring have been explored in [Deelman et al 05]. Expressive representations of workflows include rich metadata propagation rules and semantic hierarchies of component and data types [Gil et al 07; Kim et al 04]. Defining layers of abstractions over the decision space is an effective way to structure the search space and explore different decisions with appropriate search algorithms for each. [Gil et al 07] describe a layered approach to workflow creation that started with a high-level description of an earthquake simulation workflow. This workflow instance was automatically assigned resources and data movements to result in an executable workflow of 24,135 jobs and that executed for a total of 1.9 CPU years.

Relevant cognitive search techniques include systematic search, approximate search, and constraint-based search. Systematic search algorithms are designed to consider decisions in a principled manner, ensuring that all possible partial combinations of decisions are represented in the search space and visited at least once during the search [Nilsson 80]. Systematic algorithms can be made more efficient by incorporating heuristics to prioritize and eliminate portions of the search space [Pearl 85]. Properties of heuristics such as admissibility can be defined that guarantee certain properties of the search process. Approximate search algorithms explore the space of possible solutions not by exploring partial combinations of decisions and appending new ones but by navigating complete solutions and their variants [Goldberg 02; Kirkpatrick et al 83]. Examples include genetic algorithms and simulated annealing. Finally, constraint-based search algorithms explore the space of interrelated constraints among variable values associated with the decisions to be made [Dechter 03].

To exploit cognitive search techniques for edge computing, several areas of research much be addressed. The first step will be to articulate and represent declaratively the kinds of decisions that can affect the quality of the programming solution, and how these decisions are related to application-level and hardware architecture features. Such an approach for loop transformations for memory hierarchy is described in [Chen et al. 05b]. By representing declaratively all these decisions, the search for a solution and better yet the search for an optimal (or quasi-optimal) solution can be conducted by using a variety of well-known algorithms and optimization techniques. In addition, declaratively representing these decisions will enable a better understanding of their nature and their interdependencies. This determines the complexity of the search and more importantly it determines what the best algorithm and search strategy is.

Search techniques for compiler optimization have the additional requirement of on-line response. That is, faster response is crucial and sub-optimal decisions in that time frame are far better than slower but optimal response. This is desirable for off-line optimization, but an essential requirement when code is optimized at run time. There is a clear tradeoff between search time and solution quality, where more time allows the algorithms to search a larger portion of the solution space and therefore have a better chance to find optimal or closer to optimal solutions. Fortunately, the tradeoff between search time and solution quality has been present in Artificial Intelligence research since its inception, as humans are notorious for managing this tradeoff and approach complex problems as sub-optimal decision makers [Simon 69]. In addition, there is a wealth of research on a special type of search algorithm called "real-time" or "anytime" search algorithms [Korf 90; Zilberstein & Russell 95; Hansen & Zhou 07]. These algorithms are designed to have several important properties: (1) an initial and likely sub-optimal solutions is found quickly; (2) at any time, a solution can be returned by the algorithm; and, (3) the quality of the solution returned improves as the algorithm is given more time to search the solution space.

Search techniques for distributed workflow environments have an additional and important requirement. The dynamic availability of distributed resources, where network connections may fail or resources may be temporarily withdrawn, requires search techniques that can handle uncertainty and replanning. In addition, optimization techniques must take into account other workflows and applications running on the same hardware. Many cognitive techniques are applicable in this context, including uncertainty reasoning, abstraction search, and metareasoning [Gil 06a]. Programming in edge computing environments may benefit from addressing these issues as well.

In summary, programming for edge computing architectures presents a complex optimization problem. We have advocated for the automation of the process with cognitive techniques, so that the best algorithms can be identified based on their computational properties to address the vast search space and highly interdependent optimization criteria.

E. *Learning from Experience*

Machine (automated) learning is another cognitive technique that will be advantageous to support programming for unforeseen heterogeneous computing architectures and applications. While learning may not be a strict requirement for architectures and applications where human expertise exists or can be developed, it will be absolutely necessary to learn to search and optimize the space of decisions when completely new architectures and applications appear with the envisioned complexity, heterogeneity, and uniqueness.

Learning techniques are used routinely to develop intelligent agents that make decisions while learning not to repeat the same mistakes and to improve their performance in tasks they perform routinely. Learning ranges from simply collecting performance metrics and deriving statistical predictive models, to more complex learning where efficient search heuristics are derived by reasoning about the properties of a problem domain. Learning can also improve performance by recognizing common failure conditions and designing mechanisms to anticipate and avoid them. Effective learning techniques relevant here include reinforcement learning from reward feedback, learning Markov decision processes to improve overall policies, and symbolic compilation of behavior-triggering rules [Kaelbling et al 95; Dietterich 00; Hengst 00; Anderson et al 04].

A rich area for cognitive learning techniques is automatically learning local decisions concerning individual optimizations. Typically, the problem is cast as parameter selection for a given optimization, such as determining whether a loop should be unrolled and the unroll factor. Recent work demonstrates that machine learning techniques could effectively be used to automate the construction of compiler optimization heuristics [Stephenson et al 05]. The compiler developer hand-selected code features believed to be relevant to making decisions for loop unrolling heuristics. Loops in a suite of benchmarks were annotated regarding the value of each of these features and which unroll factor yields the best performance. This represents a kind of machine learning problem called classification: given a set of feature values, which decision (unroll factor value) is most appropriate. Two machine learning algorithms were trained, Nearest Neighbor (NN) and Support Vector Machine (SVM). Both algorithms were trained off-line based on the labeled benchmark data, and the results were tested against new codes and compared with the existing loop unrolling in the compiler, achieving up to a 9% overall performance improvement. Given that the features used by these classifiers can be automatically extracted from code, and given that the performance of the learned heuristics is as good or better than manually developed heuristics, this work provides strong evidence that machine learning techniques could effectively automate an otherwise costly and non-portable compiler development effort.

Another kind of decision is whether or not a given optimization should be done at all given what other optimizations are being planned. The work by [Cavazos and Moss 04] exemplifies the benefits of this line of research, and investigates machine learning techniques to decide whether to schedule a block of instructions. The decision is also framed as a classification problem for machine learning. A dozen features were identified as relevant, and a number of blocks in a benchmark suite were annotated regarding whether or not scheduling improved the block. Using a well-known rule-learning algorithm, they obtained over 90% of the improvement of block scheduling with less than 25% of the effort. In related work, [Moss et al 97] explored learning techniques to decide which instruction to schedule next. These results demonstrate the effectiveness of learning techniques to select transformations in adaptive compilers.

In the context of resource selection in distributed workflow systems, learning techniques have also been explored but in more limited ways than they have in adaptive compilers [Galstyan et al 05]. These techniques would be beneficial in edge computing in selecting resources for blocks of code.

In summary, a fertile area of future research is the rich representation of the decisions and the relevant features involved in adaptive compilers for edge computing. An important additional research area demands a more extensive exploration of available cognitive learning techniques that are best suited to each aspect of the optimization process.

IV. CONCLUSION

We have argued that distributed systems and compiler systems research have investigated very relevant issues to the future of programming for edge computing. The issues and approaches explored are complementary, and there is much to be gained from synergies and more aligned research agendas for heterogeneous edge computing architectures.

Significant results to date have been obtained from the combination of cognitive techniques and systems research. However, the spectrum of approaches explored is very limited, both from the cognitive and systems perspective. From the cognitive perspective, a small amount of search techniques from the wide range of available algorithms have been applied, and a small number of learning techniques from the many that exist. From the systems perspective, also a small range of decisions have been explored and for each only a small number of cognitive techniques have been used. For example, while [Almagor et al 04] focuses on low-level compiler optimization sequences (*e.g.,* iteration peeling and algebraic reassociation), [Stephenson et al 05] addresses higher-level optimization decisions (e.g., loop unrolling). Exploring the spectrum of techniques from the cognitive side as they are applied to the spectrum of decisions in the system side is a crucial area of future research.

Pursuing these research areas would bring us closer to a new generation of systems for what we call ***strategic optimization***. Strategic decision-making approaches would produce global optimizations for an entire application. In contrast, we think of the approaches pursued to date as tactical decision-making approaches, where the decisions are isolated and often taken when little leeway is possible. In tactical optimization, we are left making small decisions about loop unrolling factors that lead to significant but very local performance gains. In strategic optimization, we envision that upstream decisions of which component implementation to select would be taken in light of the available execution architecture and taking into account the overall application optimization choices. We believe that strategic optimization, which optimizes application components in their execution context, would result in unprecedented gains in programmer productivity while achieving a high level of performance.

Combining human expertise and automation is also a crucial area for future research. [Cavazos and Moss 04] prefer to use machine learning algorithms that produce rules that are expressive, compact, and more human readable than other learning approaches such as neural networks. Looking at the automatically learned rules, human experts could suggest new features to the learning algorithm or add more examples not covered under the current training set. [Cooper et al 05] discuss how compiler settings can be selected by the programmer in collaboration with the system. [Stephenson 06] explores collaborative compilation where a community of programmers can contribute training data for learning algorithms in a compiler, envisioning optimizing compilers that are dynamically customized to a community of users and their particular type of applications. There is a wealth of research on cognitive techniques for human-machine collaboration for complex problem solving for many other domains that could be relevant for this area of research. For example, planning and constraint checking techniques have been demonstrated to assist users in workflow creation [Kim et al 04]. One could imagine programming environments for edge computing where intelligent assistance is used to extract from the programmer crucial application features relevant to performance, combined with automation of exploration and learning of optimization strategies at the compiler level. Programmers will remain at the application level, while the system will take care of execution details and of learning strategies to customize its behavior to the current architecture.

ACKNOWLEDGMENTS

We gratefully acknowledge funding from the National Science Foundation under grants CSR-0615412 and CCF-0725332 and from an internal grant by USC/ISI. We also thank our many collaborators who have inspired some of the ideas in this paper, and the referees for their helpful remarks.

REFERENCES

[Ceranowicz 05] Ceranowicz, A. & Torpey, M., (2005), Adapting to Urban Warfare, *The Journal of Defense Modeling and Simulation* Vol 2 No 1, January 2005, San Diego, California.

[Lucas 03] Lucas, R., & Davis, D., (2003),"Joint Experimentation on Scalable Parallel Processors," in the *Proceedings of the Interservice/Industry Simulation, Training and Education Conference*, 2003.

[Salomon 04] Salomon, B., Govindaraju, N. K., Sud, A., Gayle, R., Lin, M. C., & Manocha, D., "Accelerating Line of Sight Computation Using Graphics Processing Units", *Proc. of Army Science Conference*, 2004.

[Tripp et al. 05] Tripp, J., Mortveit, H. S., Hansson, A., and Gokhale, M. Metropolitan Road Traffic Simulation on FPGAs, In Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05), 2005.

[Whaley 05] Whaley, R. C. and A. Petitet, A. Minimizing Development and Maintenance Costs in Supporting Persistently Optimized BLAS, *Software: Practice and Experience,* 35(2): 101-121, Feb., 2005.

[Frigo 05] Frigo, M. and Johnson, S. G. The Design and Implementation of FFTW3, *Proceedings of the IEEE,* Special Issue on Program Generation, Optimization, and Platform Adaptation, 93(2):216-231, Feb. 2005.

[Puschel et al. 05] Puschel, M., Moura, J.M.F., Johnson, J.R., Padua, D., Veloso, M. M., Singer, B. W., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R. W. and Rizzolo, N. SPIRAL: Code Generation for DSP Transforms, *Proceedings of the IEEE,* Special Issue on Program Generation, Optimization, and Platform Adaptation, 93(2):216-231, Feb. 2005.

[Allan et al. 06] Allan BA *et al.* A component architecture for high-performance scientific computing. *International Journal of High-Performance Computing Applications* 2006; 20:163.

[Yu et al. 2005] H. Yu, D. Zhang and L. Rauchwerger. "An Adaptive Algorithm Selection Framework," Proceedings of the Parallel Architectures and Compilation Techniques, September, 2004.

[Thomas et al. 05] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. Amato, and L. Rauchwerger. "A Framework for Adaptive Algorithm Selection in STAPL," Proceedings of the Conference on Principles and Practice of Parallel Programming, June, 2005.

[Gordon et al. 02] Gordon, M. and Thies, W. and Karczmarek, M., Lin, J., Meli, A.S., Leger, C., Lamb, A.A., Wong, J., Hoffman, H., Maze, D.Z. and Amarasinghe, S. A Stream Compiler for Communication-Exposed Architectures, ASPLOS, October 2002.

[Yoon et al. 07] Yoon, Y., Browne, J.C., Crocker, M., Jain, S., and Mahmood, N. Productivity and Performance Through Components: The ASCI Sweep3D Application, *Concurrency and Computation: Practice and Experience (CCPE*), 19(5): pp. 721-742, April 2007.

[Rauchwerger et al. 2000] L. Rauchwerger, N. Amato, J. Torrellas."SmartApps: An Application Centric Approach to High Performance Computing," Proceedings of the Workshop on Languages and Compilers for Parallel Computing, Aug. 2000.

[Vuduc et al. 2005] R. Vuduc, J. W Demmel, and K. A Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," Journal of Physics: Conference Series **16** (2005) 521–530.

[Gil 06b] Gil, Y. "Workflow Composition." In Workflows for e-Science, D. Gannon, E. Deelman, M. Shields, I. Taylor (Eds), Springer Verlag, 2006.

[Kim et al 07] Kim, J., Deelman, E., Gil, Y., Mehta, G., and Ratnakar, V. "Provenance Trails in the Wings/Pegasus Workflow System." To appear in Concurrency and Computation: Practice and Experience, Special Issue on the First Provenance Challenge, 2007.

[Li et al. 2004] X. Li, M. J. Garzaran, and D. Padua. "A Dynamically Tuned Sorting Library," Proceedings of the International Symposium on Code Generation and Optimization, March 2004.

[Li et al. 2005] X. Li, M. J. Garzaran, and D. Padua. "Optimizing Sorting with Genetic Algorithms," Proceedings of the International Symposium on Code Generation and Optimization, March 2005.

[Knijnenberg 04] P.M.W. Knijnenburg, T.Kisuki, K.Gallivan, and M.F.P. O'Boyle, "The effect of cache models on iterative compilation for combined tiling and unrolling. Concurrency and Computation: Practice and Experience, 16(2--3):247--270, 2004.

[Cohen et al. 07] L.-N. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. "Iterative optimization in the polyhedral model: Part {I}, one-dimensional time," Proceedings of the International Conference on Code Generation and Optimization, March 2007.

[Yotov et al. 2005b] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, P. Stodghill. Is Search Really Necessary to Generate High-Performance BLAS? Special issue on "Program Generation, Optimization, and Platform Adaptation," Proceedings of the IEEE, Vol. 93, No. 2, pp. 358-386. February 2005.

[Cavazos et al. 2006] J. Cavazos, C. Dubach, F. Agakov, Edwin Bonilla, M. O'Boyle, Grigori Fursin, and Olivier Temam. Automatic Performance odel Construction for the Fast Software Exploration of New Hardware esigns. In International Conference on Compilers, Architecture, And Synthesis For Embedded Systems (CASES 2006), October 2006.

[Yotov et al. 05a] Yotov, K., Pingali, K., and Stodghill, P., Think Globally, Search Locally, Proceedings of the International Conference on Supercomputing, June, 2005.

[Chen et al. 05a] Chen, C., Chame, J., and Hall, M.W., Combining Models and Guided Empirical Search to Optimize for Multiple Levels of the Memory Hierarchy, Proceedings of the Code Generation and Optimization Conference, March, 2005.

[Kelly 96] Kelly, W.A. Optimization within a Unified Transformation Framework, PhD Dissertation, Dept. of Computer Science, University of Maryland, Dec, 1996.

[Cohen et al. 05] Cohen, A., Girbalm S, Parello, D., Sigler, M., Temam, O. and Vasilache, N. Facilitating the Search for Compositions of Program Transformations, Proceedings of the International Conference on Supercomputing, June, 2005.

[Chen 07] Chen, C. Model-guided empirical optimization on arbitrary loop nests for memory hierarchy, PhD dissertation, Dept. of Computer Science, University of Southern California, May, 2007.

[Soffa et al. 05] Zhao, M., Childers, B.R., and Soffa, M.L. A Model-Based Framework: An Approach for Profit-Driven Optimization, Proceedings of the Code Generation and Optimization Conference, March, 2005.

[Lee et al. 05] Lee, Y., Diniz, P., Hall, M. and Lucas, R. Empirical Optimization for a Sparse Linear Solver: A Case Study, *International Journal of Parallel Programming*, vol. 33, 2005.

[Kisuki et al 00]. Kisuki, T, Knijnenburg, P.M.W., and O'Boyle, M.F.P. Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation, PACT 2000.

[Almagor et al 04] Almagor, L., Cooper, K.D., Grosul, A., Harvey, T.J., Reeves, S.W., Subramanian, D., Torczon, L., Waterman, T. Finding effective compilation sequences Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems (LCTES'04), 2004.

[Gil et al 04] Gil, Y., Deelman, E., Blythe, J., Kesselman, C., and Tangmurarunkit, H. Artificial Intelligence and Grids: Workflow Planning and Beyond, *IEEE Intelligent Systems*, January 2004.

[Blythe et al 03] Blythe, J., Deelman, E., Gil, Y., Kesselman, C., Agarwal, A., Mehta, G., Vahi, K. The Role of Planning in Grid Computing, Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS), June, 2003.

[Blythe et al 05] Blythe, J., Jain, S., Deelman, E., Gil, Y., Vahi, K., Mandal, A. and Kennedy. K. Task Scheduling Strategies for Workflow-Based Applications in Grids, Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGRID), May 2005.

[Deelman et al 05] Deelman, E., G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, J. Kim, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, D. S. Katz. Pegasus: A Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming*, Vol. 13, No. 3, 2005.

[Gil et al 07] Gil, Yolanda, Varun Ratnakar, Ewa Deelman, Gaurang Mehta, and Jihie Kim. Wings for Pegasus: Creating Large-Scale Scientific Applications Using Semantic Representations of Computational Workflows, Proceedings of the 19th Annual Conference on Innovative Applications of Artificial Intelligence (IAAI), July, 2007.

[Kim et al 04] Kim, Jihie, Marc Spraragen, and Yolanda Gil. "An Intelligent Assistant for Interactive Workflow Composition", Proceedings of the 2004 International Conference on Intelligent User Interfaces (IUI), Jan. 2004.

[Nilsson 80] Nilsson, N. Principles of Artificial Intelligence, San Francisco: Morgan Kaufmann, 1980.

[Pearl 85] Pearl, Judea. "Heuristics: Intelligent Search Strategies for Computer Problem Solving", Addison-Wesley, 1985.

[Goldberg 02] Goldberg, David E. The Design of Innovation: Lessons from and for Competent Genetic Algorithms, Addison-Wesley, Reading, MA, 2002.

[Kirkpatrick et al 83] Kirkpatrick, S. and C. D. Gelatt and M. P. Vecchi, Optimization by Simulated Annealing, Science, Vol 220, Number 4598, pages 671-680, 1983.

[Dechter 03] Rina Dechter. "Constraint Processing". Morgan Kaufmann, 2003.

[Chen et al. 05b] Chen, C., Chame, J., Hall, M., and Lerman, K. A Systematic Approach to Model-Guided Empirical Search for Memory Hierarchy Optimization, Proceedings of the Workshop on Languages and Compilers for Parallel Computing, October, 2005.

[Simon 69] H. A. Simon, "The Sciences of the Artificial". MIT Press, Cambridge, MA, 1969.

[Korf 90] Korf R. E., Real-time heuristic search. Artificial Intelligence, 42, 189-211, 1990.

[Zilberstein and Russell 95] Zilberstein, S., and Russell, S. Approximate reasoning using anytime algorithms. In Imprecise and Approximate Computation. Kluwer Academic Publishers, 1995.

[Hansen and Zhou 07] Hansen, E. and R. Zhou. Anytime Heuristic Search. Journal of Artificial Intelligence Research, 28: 267-297, 2007.

[Gil 06a] Gil, Yolanda. On Agents and Grids: Creating the Fabric of a New Generation of Distributed Intelligent Systems, Journal of Web Semantics, Volume 4, Issue 2, June 2006.

[Kaelbling et al 96] Kaelbling, L. P., Littman, M., L., and Moore, A. W. Reinforcement Learning: A Survey, Journal of Artificial Intelligence Research, Vol 4, pp. 237-285, 1996.

[Dietterich 00] Dietterich, T. "Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition", JAIR 13: pp. 227-303, 2000.

[Hengst 00] Hengst, B. "Generating Hierarchical Structure in Reinforcement Learning from State Variables", Lecture Notes in Artificial Intelligence, 2000.

[Anderson et al 04] Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., and Qin, Y. "An Integrated Theory of the Mind", Psychological Review 111, (4). 1036-1060, 2004.

[Stephenson 05] Stephenson, M. and Amarasinghe, S. Predicting Unroll Factors Using Supervised Classification, Proceedings of the Code Generation and Optimization Conference, March, 2005.

[Cavazos and Moss 04] Inducing Heuristics to Decide When to Schedule, Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation, June, 2004.

[Moss et al 97] Moss, E., Utgoff, P., Cavazos, J., Precup, D., Stefanovid, D., Brodley C. and Scheeff, D. Learning to Schedule Straight-Line Code, Neural Information Processing Systems, 1997.

[Galstyan et al 05] Aram Galstyan, Karl Czajkowski, and Kristina Lerman, (2005), Resource Allocation in the Grid with Learning Agents, *Journal of Grid Computing*, 3, pp. 91-100, 2005.

[Stephenson 06] Stephenson, Mark. "Automating the Construction of Compiler Heuristics Using Machine Learning". PhD thesis. Computer Science and Artificial Intelligence Laboratory. Massachusetts Institute of Technology. May 2006.