

RCML: An Environment for Estimation Modeling of Reconfigurable Computing Systems

Casey Reardon, Brian Holland, Alan D. George, Greg Stitt, and Herman Lam
NSF Center for High-performance Reconfigurable Computing (CHREC)
University of Florida, Gainesville, FL

Reconfigurable computing (RC) is emerging as a promising area for embedded computing, where complex systems must balance performance, flexibility, cost, and power. The difficulty associated with RC development suggests improved strategic planning and analysis techniques can save significant development time and effort. This article presents a new abstract modeling language and environment, the RC Modeling Language (RCML), to facilitate efficient design-space exploration of RC systems at the estimation modeling level, i.e. before building a functional implementation. Two integrated analysis tools and case studies, one analytical and one simulative, are presented illustrating relatively accurate automated analysis of systems modeled in RCML.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Modeling Techniques; I.6.5 [**Model Development**]: Modeling Methodologies; C.0 [**General**]: System Architectures

General Terms: Modeling

Additional Key Words and Phrases: Reconfigurable computing, modeling, performance prediction

1. INTRODUCTION

Embedded systems continue to face ever-growing demands in terms of performance, power, size, flexibility, and cost. The demands for greater flexibility in embedded systems favor the use of programmable processing devices that can support multiple applications. At the same time, it is becoming more difficult to realize increased performance on traditional microprocessors through higher clock speeds and instruction-level parallelism alone, which typically come at the expense of increased power consumption. In the face of these challenges, modern embedded systems are increasingly designed with heterogeneous architectures that feature higher levels of component parallelism. Reconfigurable computing (RC) is poised to become an important and viable paradigm for embedded and high-performance computing under these circumstances, enabling designers to fully exploit the performance potential and parallelism of underlying hardware resources in reconfigurable-logic devices such as FPGAs in a highly adaptive manner. Heterogeneous systems including

This work was supported in part by the I/UCRC Program of the National Science Foundation under Grant No. EEC-0642422. The authors gratefully acknowledge vendor equipment and/or tools provided by Altera, MDesign Technologies, SRC, and Xilinx that helped make this work possible.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 20YY ACM 0000-0000/20YY/0000-0001 \$5.00

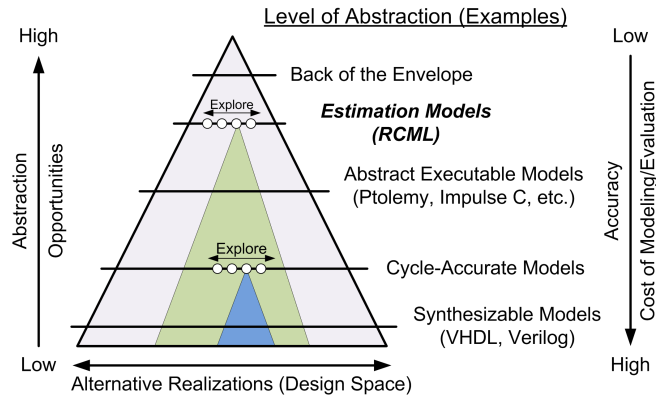


Fig. 1: Abstraction Pyramid Comparing Levels of Modeling for Hardware Applications (adapted from [Kienhuis et al. 2002])

microprocessors, application-specific integrated circuits (ASICs), and FPGAs can provide the benefits of performance and efficiency required during times where it is becoming increasingly critical to manage the size and power consumption of embedded systems. But despite the gains in performance and efficiency over traditional systems that RC can provide, RC application development and system design can be a prohibitively long and difficult process, hindering its further growth and usage.

The abstraction pyramid for system-level design, originally presented in Kienhuis et al. [2002] and illustrated in Fig. 1, summarizes the levels of abstraction designers can use to model their hardware systems, and the tradeoffs between each level. At the top of the abstraction pyramid, a back-of-the-envelope model is a set of simple mathematical relationships used to approximate basic performance metrics of the system. Estimation models are a more sophisticated and accurate alternative to back-of-the-envelope models, using more complex model inputs and calculations to predict system performance. While both of the previous two models are used to quickly predict and approximate key performance metrics, neither describe the full functional behavior or timing of a system. In contrast, an abstract executable model describes the functional behavior of the system, without information that describes the system behavior in terms of timing. A cycle-accurate model describes the functional behavior and timing of an architecture, as a multiple of clock cycles. Finally, a synthesizable model is one that contains enough behavior and timing details such that the model can be realized in silicon.

At each level of abstraction, designers can perform design-space exploration and analyze alternative designs. As more detailed models are used, more effort is needed to build a full model and a narrower region of the design-space can be explored, raising the risk of creating a model whose associated design space does not include a solution that meets all design constraints. In this case, considerable effort is wasted developing a design that will eventually be discarded. Early design-space exploration can help ensure that the proposed design will meet design constraints before intensive coding of a functional implementation has begun. Back-of-the-envelope models offer the highest level of abstraction, but often do not feature enough de-

tail to meaningfully represent candidate designs. Therefore, an abstract estimation model is needed at the beginning of the development process allowing systems to be modeled and analyzed quickly and meaningfully before moving towards a more detailed implementation. Designers can repeatedly analyze and alter the abstract model based on feedback from analysis tools, which can save considerable time and cost in overall development by reducing wasted effort in later stages of development.

This article introduces and presents a new language and environment tailored for estimation modeling of RC systems and applications, called the Reconfigurable Computing Modeling Language (RCML). RCML is designed to allow users to efficiently model RC systems in the early stages of RC development. The RCML framework enables users to separately model the algorithm and the execution platform architecture under study, providing specific constructs for defining parallelism, communication patterns, and other common aspects in RC applications. A complete system model is then created by mapping an application and platform model together. This process allows individual models to be reused for any number of mappings, supporting efficient iterative design-space exploration early in the development process.

RCML differs from most existing modeling environments because it is tailored to operate at the estimation-model level of abstraction. Existing modeling environments often used in embedded system design such as Ptolemy and Simulink, and high-level language compilers such as Impulse-C, typically use an abstract executable model as the user's entry point for specifying a design and moving towards a final implementation. Such tools require users to define and code a functional implementation in order to drive a full simulation or other analysis. Instead, RCML is intended to allow users to quickly model their systems before coding a functional implementation, using abstract constructs and attributes to define component behavior. Models built with RCML can then be used to bridge to functional specification environments such as Ptolemy and Impulse-C, complementing these tools in the overall development process. In other words, an environment such as RCML can be used to raise the initial level of abstraction that designers may use when beginning embedded system design. For these reasons, RCML is more similar to high-level modeling languages such as the Unified Modeling Language (UML) and the Architecture Analysis & Description Language (AADL), allowing models to be easily converted between these languages.

RCML is designed to feed any number of analysis tools for prediction and analysis to fully enable efficient design-space exploration. In addition to introducing the RCML language and environment, this article focuses on the integration of two analysis tools within the RCML environment for automated performance prediction: an analytic methodology called the RC Amenability Test (RAT) [Holland et al. 2009] and a script-based simulation framework for RC [Reardon et al. 2009]. Case studies presented in this article using a pair of image-processing applications show that, using RCML models that required only several minutes to create, the automated RAT tool was able to produce performance predictions within 0.6% to 6.2% of the actual execution time for a single-device application, while the simulation environment produced performance predictions within 2.3% to 7.4% of the actual execution time for a multi-device application. Future tools can also be de-

veloped to further extend the usefulness of RCML. For example, tools to perform code generation and automated implementation from an RCML model would reduce the effort needed to move from RCML to a high-level or cycle-accurate executable model of the design.

The remainder of this article is organized as follows. In Section 2, background and related research is presented. Next, an overview of RCML is provided in Section 3. Section 4 discusses two automated analysis tools integrated with RCML. In Section 5, a pair of case studies are presented to illustrate the effectiveness of design-space exploration through RCML using both analysis tools. Finally, conclusions and future work are summarized in Section 6.

2. BACKGROUND AND RELATED RESEARCH

Previous abstract modeling languages have been used in the computing field for years to help software and system designers plan their complex projects, and RCML shares many concepts with a pair of similar abstract modeling languages used in computing. The Architecture Analysis & Design Language (AADL) from the Society of Automotive Engineers (SAE) is a textual and graphical language designed for specifying software, hardware and system components of complex real-time and embedded systems [SAE 2004; Feiler et al. 2006]. AADL models are collections of pre-defined components which are further characterized through properties, interfaces, and networks of subcomponents. Most AADL components can be classified as either software, execution platform, or composite components. AADL supports language extensions through property sets and specific notations within core components that new analysis tools can interpret. While AADL incorporates many useful concepts that are leveraged within RCML, it lacks convenient constructs for parallelism and algorithm exploration that are important for early RC design-space exploration. Instead, AADL is typically used in areas such as avionics for representing a complex embedded system with a large number of interacting software processes and devices. As a result, existing analysis tools developed for AADL focus on analyzing the dependability [Rugina et al. 2006], memory [Singhoff et al. 2005], and real-time scheduling requirements [Sokolsky et al. 2006] of the system. AADL is also commonly used to automatically generate an implementation of the modeled system. RCML models can potentially be used for automated implementation as well, especially when the tasks in the algorithm model can be linked to cores in an existing core library. While such capabilities are critical in allowing users to transition to less abstract models of their RC system, automated implementation from RCML models will be the subject of future work and thus is beyond the scope of this article.

An even more popular modeling language within the computing field is the Unified Modeling Language (UML) [UML Revision Task Force 2001]. UML is widely used throughout the software-development industry as a language for planning and designing object-oriented, software-intensive projects. The UML specification defines 13 types of diagrams that each provide a partial view of the underlying meta-model, allowing users to interact with the diagram type(s) best suited for describing their problem. UML profiles such as SysML [Object Management Group 2008] and MARTE [Object Management Group 2009] have been created to support model-

ing complex and real-time embedded systems in UML, respectively. Like AADL, MARTE and SysML define extensive modeling specifications with software, hardware, and system constructs for modeling complex systems with a large number of interacting software processes and devices. Since these profiles are designed to support large modeling projects through many development stages, they may not be ideally suited for singular algorithm exploration. Furthermore, using a UML profile such as MARTE may require the user to learn and understand several different classes of UML models in addition to many fundamental UML concepts and profile-specific semantics, which may be discouraging for an application designer on a smaller system. In contrast, RCML is meant to provide the user a simpler and intuitive environment to abstractly model and analyze their proposed RC system.

Several modeling environments have been developed for building and evaluating functional models of heterogeneous and RC systems. Ptolemy is a well-known modeling framework for simulating and prototyping heterogeneous systems supporting multiple models of computation (MoC) [Buck et al. 1994]. Ptolemy studies the support of heterogeneous system design through the use of hierarchical heterogeneity [Eker et al. 2003]. Hierarchical heterogeneous models are divided into local submodels which share a common MoC. Each locally homogeneous submodel can then be treated as a single component in a model network with components featuring potentially several MoCs whose interfaces and interactions are automated through Ptolemy. As with most existing model-based development environments for embedded system design, Ptolemy provides users with an abstract executable model as an entry point in the development process. The RCML environment is intended to complement such tools by providing a framework to perform early design-space exploration at a higher level of abstraction, before specifying a functional model.

Unlike Ptolemy, Metropolis and Artemis are two examples of model-based frameworks that provide support for separately and independently modeling the functionality and architecture of an embedded system. Metropolis is a platform-based design environment based on a metamodel with formal execution semantics [Balarin et al. 2003]. The Metropolis metamodel can be used to specify the function (using existing or new MoCs), architecture, and mapping of the system. The Metropolis framework also includes tools for verification, simulation and synthesis of designs created with Metropolis, and has been extended for analysis and characterization of FPGA-based systems [Densmore et al. 2006].

The Artemis framework provides high-level modeling and simulation methods and tools for system-level performance evaluation and exploration of heterogeneous and reconfigurable embedded multimedia systems [Pimentel 2005; Pimentel et al. 2001]. Artemis describes a systematic approach to explore embedded system architectures at multiple levels of abstraction, following popular embedded design concepts such as "separation of concerns" [Keutzer et al. 2000] and the Y-chart design methodology [Kienhuis et al. 2002]. The Artemis workbench includes a system-level modeling and simulation environment known as Sesame [Pimentel et al. 2006]. In both Artemis and Sesame, applications are described as a Kahn Process Network (KPN) [Kahn 1974] at the highest level of abstraction, while architecture models are implemented from a library generic building blocks using either Pearl or SystemC. As the architecture is described in more detail, the abstract application model is

refined into a more detailed dataflow graph through trace transformations [Lieverse et al. 2001]. While Sesame and Metropolis have been shown to be effective approaches to heterogeneous embedded system design, they are not particularly suited for estimation modeling of RC systems. For example, a KPN assumes parallel processes communicate using unbounded FIFO channels, but effective use of limited buffering resources on RC devices is often critical to maximize performance. In addition, complex communication patterns and data parallelism are not conveniently expressed using their basic constructs, but could easily be generated and constructed from an existing RCML model which is intended to function at a higher level of abstraction than Artemis and Metropolis support.

A hierarchical model-based framework for FPGA development in RC is presented by Mohanty and Prasanna [2007] which includes support for evaluation of design alternatives early in the development process. The framework integrates a high-level performance estimator (HiPerE) and a design-space exploration tool (DESERT) for efficient evaluation of candidate mappings against user-specified performance requirements onto System-on-Chip (SoC) architectures described using the Generic Model (GenM) [Mohanty et al. 2002]. While the framework in Mohanty and Prasanna [2007] overlaps in some degree with this research, there are several key differences. First, the framework in Mohanty and Prasanna [2007] is intended to serve as a development environment, unlike RCML which is a modeling environment designed for estimation-level modeling of RC systems. Furthermore, RCML includes a larger number of pre-defined specialized constructs for modeling RC applications and platforms, intended to further improve productivity. Finally, HiPerE is only designed to evaluate architectures that can be described by GenM, which is designed for SoC architectures. RCML is designed to model and analyze a broader set of high-performance RC architectures.

RCML can be used to provide input to numerous complementary techniques for predicting and analyzing RC systems. A number of analytical [Smith and Peterson 2002; Steffen 2007] and simulative [Enzler et al. 2005; Fu and Compton 2006] techniques have been proposed to predict the performance of RC applications. Analytic and simulative approaches each have their own strengths and weaknesses when it comes to prediction and analysis of computing systems. Therefore, this article presents one analytical and one simulative technique that are each integrated within the RCML environment to enable automated performance prediction.

The RC Amenability Test (RAT) defines a set of analytic equations for predicting the potential speedup of a particular FPGA core design [Holland et al. 2009]. Simple and reasonably accurate throughput analysis is the primary focus of the RAT methodology, which uses a series of straightforward equations to predict application performance based upon known parameters and values estimated from the proposed design, though the framework considers numerical precision and resource utilization as well. Due to RAT's focus on overall performance modeling of applications running on an FPGA platform, an automated RAT-based analysis tool is integrated with RCML to enable nearly instantaneous performance prediction of RC systems modeled within the RCML environment.

The RC Simulation Environment (RCSE) is a trace-based simulation framework for performing fast and accurate simulations of RC systems [Reardon et al. 2009].

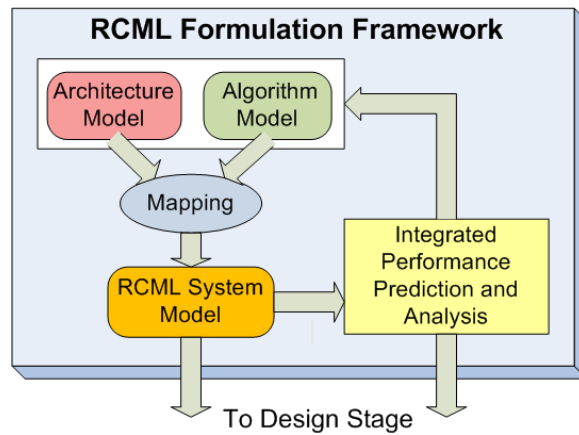


Fig. 2: Overview of RCML Methodology

RCSE is an RC-centric adaptation of the Fast and Accurate Simulation Environment (FASE) [Grobelyny et al. 2007] which seeks to balance speed and fidelity in simulating HPC systems. Applications in RCSE are characterized and abstracted into a script, which are processed by a discrete-event model of the platform during simulation. The scripts abstract away many computational details of the application, allowing simulations to execute more quickly than with traditional cycle-accurate simulations. An automated tool that targets RCSE has been developed in order to provide more detailed and accurate performance prediction results than analytic models typically provide, and still provide results in a relatively timely manner to support efficient early design-space exploration.

3. RCML OVERVIEW

RCML is an estimation-level language and environment for abstractly modeling the structure and behavior of RC systems before moving to implementation code. The framework of RCML is illustrated in Fig. 2. There are three different types of models within RCML’s framework, following the Y-chart methodology common in embedded design [Kienhuis et al. 2002]. First, an algorithm model in RCML is a platform-independent, RC-specialized task graph describing the algorithm’s task-level parallelism and communication. Second, an architecture model is a component diagram that describes the makeup and capabilities of the execution platform. Finally, a system model consists of an algorithm model mapped onto an architecture model, providing a complete representation of the system that can be used for analysis. By defining separate models for applications and platforms, these individual models can be developed independently and reused among any number of system mappings. The following subsections provide details pertaining to each type of model in RCML.

Additional information is added to an RCML model by defining custom *attributes* for a model or individual element. Each attribute consists of a name and value specified by the user, and any number of attributes can be defined for an RCML model or individual model element. Attributes allow users to freely embed behavioral in-

formation throughout their model, which can be used by external tools for analysis, code generation, and additional documentation. The following subsections provide a few examples of how attributes are used and their importance in RCML.








3.1 Algorithm Modeling

An algorithm model in RCML is a collection of blocks and RC-specialized constructs that easily allows a designer to represent the structure, data flow, and parallelism in an RC application prior to specifying implementation code. The algorithm model is implemented as a task graph, where nodes (i.e. blocks) correspond to parallel tasks, and edges represent communication and dependencies. A summary of the primary constructs in RCML algorithm models is presented in Table I. Algorithm constructs in RCML can be divided into two groups, *function blocks* and *data blocks*. Data blocks are used to represent the major entities of data that will be produced, consumed, and processed by the algorithm. Each data block is made up of a specified number of data elements. A *data element* can be as simple as a single primitive type, such as an integer or float, or a composition of primitive types. Two types of data blocks are currently defined in RCML, *data sets* (a single, finite group of data) and *data streams* (an infinite, continuous data entity). The data blocks are designed to make it easy for users to explore how system performance changes as the scale or size of their problem is varied. Since the transfer size over a link and the computation amount of a function block can be linked to the number of elements in a data block, changes to a data block can propagate throughout the model to facilitate quick and efficient design-space exploration.

An algorithm in RCML is modeled as a collection of tasks, in which each task is represented by a *function block*. There are currently two basic types of function blocks defined in RCML, *data-driven* and *control-driven*. Execution in data-driven blocks is triggered by the receipt of data on specified input data connections. The amount of data needed to trigger a block's execution can be defined by the user, for cases where multiple pieces of data are needed before a task can begin. Control-driven blocks are executed when the relevant control signals entering that block have been triggered. A *repetitive function block* is a task defined to iteratively execute a specified number of times upon each triggering, such as a basic loop. The number of repetitions for a function block can either be a static number or linked to the size of a data block (e.g. a function block may be defined to iterate once for each data element in a data set). Attributes can be used to further define the computational requirements of the function block, e.g. processing latencies, hardware resource requirements, etc.

Although an RC designer could potentially represent an algorithm as a pure task graph, it is important in RC to be able to concisely express common types of parallelism. For this reason, RCML includes two specialized parallel constructs. The first parallel construct is the *process line*, a special function block used to express deep functional parallelism in an algorithm. A process line consists of a sequence of separate tasks that data must pass through, each of which may execute concurrently on different pieces of data (typically realized as a pipeline in hardware circuits). RCML process lines are customized by defining the number of stages in the line along with function blocks to characterize each stage. The second parallel construct RCML provides are *replicated* function blocks, used to concisely

Table I. Algorithm Model Constructs

Name	Description	Icon
Data Block	A block representing a finite (data set) or continuous (data stream) group of data processed by the algorithm	
Basic Function Block	Algorithm task triggered either by incoming data (data-driven) or control signals (control-driven)	
Repetitive Function Block	Algorithm task whose execution is repeated a specified number of times	
Process Line Function Block	Task sequence that can execute concurrently to support deep parallelism	
Replicated Function Block	A task that is physically replicated to support wide parallelism	
Buffer	A location where data elements can be stored in a queue	
Conditional Branch	A decision block that chooses one of multiple outputs to trigger	

express wide data parallelism in an algorithm. Replicated function blocks represent tasks that can be physically replicated, each able to concurrently perform the same task on different data. Wide functional parallelism, where different tasks operate concurrently on the same (or different) data, is naturally expressed as parallel paths in the task graph. These parallel constructs facilitate design-space exploration by allowing users to easily change the amount of parallelism the system features, in order to efficiently study the effects of varying levels and types of parallelism on system performance.

The communication and dependencies between tasks are defined by *connections*. A connection can carry an arbitrary size of undefined data, elements of a data block, or a control signal, all defined using built-in parameters for each connection. Each connection also has a priority, which defines the ordering (if any) between multiple outgoing connections leaving the same function block. Additionally, a communication pattern must be specified for any connection involving replicated function blocks. Each pattern defined in RCML mimics a standard function in the Message Passing Interface (MPI) [Walker 1994], even though the connection may or may not result in inter-node communication. Some of the supported patterns include *broadcast* and *scatter* for one-to-many connections, *gather* for many-to-one connections, and *all-to-all* for many-to-many connections. Due to the discrete-event dataflow semantics of data-driven function blocks, synchronization constructs are often unnecessary for building an algorithm model (i.e. all connections between blocks represent dependencies in a dataflow model, therefore tasks naturally will only execute after all dependencies are satisfied). For select cases where explicit synchronization is needed or desired, a *barrier* is also available to force multiple tasks to synchronize before continuing. A barrier is placed on multiple connections, and the barrier can be set to be enforced either immediately before or after a set of transfers occur on all of the connections.

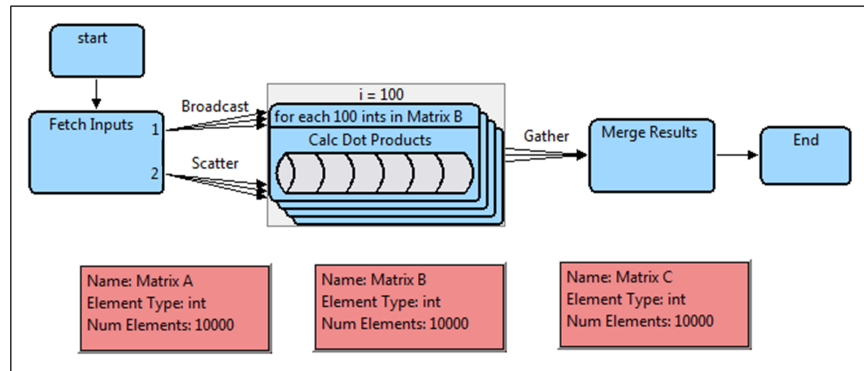


Fig. 3: Example RCML Algorithm Model of a Matrix Multiply

In addition to the basic constructs for defining data and tasks in an algorithm, a pair of miscellaneous algorithm constructs are currently supported in RCML. The first and most important miscellaneous construct is the *buffer*. Buffers are used to model locations in the algorithm where data elements may be stored in a queue. Buffers are a critical component of many embedded and RC applications. Each buffer in the algorithm must be mapped to a memory instance in the system mapping stage. Buffers do not affect the functionality of the algorithm and thus are not necessary to build a representative algorithm model, but are used to provide finer implementation details used during analysis and code generation after system mapping. A *conditional branch* construct is also provided, which models a segment of the algorithm that will follow one of multiple outgoing paths depending upon the evaluation of the condition. The conditional branch can also provide an alternative way to model loops in addition to an repetitive function block.

An example of a simple RCML algorithm model for performing a matrix multiplication is presented in Fig. 3. There are three data sets listed for this algorithm along the bottom of the diagram, each one representing a matrix with 100×100 integers taking part in the multiplication. Even though no visible connections are depicted between the data sets and the rest of the algorithm model, the data sets are linked to various elements of the task graph, which will be discussed later in this paragraph. The task graph begins with a task (labeled *Fetch Inputs*) which retrieves the two input matrices. The retrieval of the two matrices is assumed during the execution of the Fetch Inputs task since the output connections from Fetch Inputs are the first to transfer elements of those data sets. The following task (labeled *Calc Dot Products*) then performs the multiplications and additions to compute the elements of the resulting product matrix in a pipelined process. In this matrix-multiply algorithm, the Broadcast connection from Fetch Inputs to Calc Dot Products first sends the entire data set Matrix A to each instance of Calc Dot Products. The number and type of data elements transmitted across each connection are not shown, but are specified as attributes inside each connection. For example, selecting the Broadcast connection would show attributes that specify that all of Matrix A is being transmitted over that connection. After the broadcast is complete, the columns of Matrix B are scattered between each kernel via the










Scatter connection between the same two blocks. The ordering of the two transfers between Fetch Inputs and Calc Dot Products is depicted by the numeric labels next to the beginning of each connection leaving the Fetch Inputs task (the priority labels are omitted when a connection is the only one leaving a function block and thus no ordering needs to be defined). The Calc Dot Products task will then perform its task once for every column of elements (i.e. 100 integers) of Matrix B that it receives, as specified along the top of the Calc Dot Products function block. Since each of the 100 columns of matrix B may be processed in parallel, 100 instances of the Calc Dot Products task are specified (depicted by the $i = 100$ label just above the task). Like any RCML task, the Calc Dot Products task may be hierarchical and contain submodels used to describe its computational structure in more detail, as more parallelism can be exploited within each execution of Calc Dot Products. After each instance of Calc Dot Products completes all of its iterations, the corresponding number of data elements for Matrix C, which represents the result of the matrix multiplication, are sent to the Merge Results task via the Gather connection.

As previously stated, an RCML algorithm model is designed to represent the structure, data flow, and parallelism in an RC application for performing early design-space exploration, and thus only an abstract description of the algorithm behavior is needed. For example, the Fetch Inputs function block in the matrix-multiply algorithm model does not specify how the algorithm splits up the Matrix B data set along columns to be divided among kernels in Calc Dot Products. Instead, the model only specifies that the Fetch Inputs task is splitting the Matrix B data set and distributing the data among instances of Calc Dot Products. Similarly, the Merge Results task does not specify how the final product Matrix C is assembled and stored, only that the task will not execute until all of the results have been gathered from the Calc Dot Product kernels before forming and sending out the Matrix C data set that represents the product of the multiplication. This is a key advantage of using an estimation modeling environment such as RCML, as more detailed behavior is often unnecessary to accurately estimate application performance and perform design-space exploration. Nevertheless, in some cases users will want to model parts of their algorithms with more detail, specifically with cores designed to be implemented on reconfigurable devices. Future work will investigate incorporating alternative modeling frameworks, such as the CMD framework proposed by Wang et al. [2009], into the RCML environment to support estimation modeling of reconfigurable cores at lower levels of abstraction.

3.2 Architecture Modeling

Independent of algorithm models, architecture models attempt to capture the structure and capabilities of the execution platform under study. Currently, RCML provides several generic component types used to construct architecture models. A designer may add any number of attributes for each component to create a more detailed architecture model, whose information can be used by external tools for analysis. The classes and graphical representations of architecture components closely follow the set of architecture component classes defined in AADL, with the notable addition of the reconfigurable processor. The reconfigurable processor class can be used to represent any RC device, from an FPGA to a coarse-grained recon-

Table II. Architecture Model Constructs

Name	Description	Icon
Fixed Processor	A fixed processing element, such as an ASIC or microprocessor	
Reconfigurable Processor	A processing element that supports reconfiguration, such as an FPGA	
Memory Block	An instance of memory or storage in the system	
Cache Block	A specialized memory block for multi-level caches	
Interconnect Bus	A system or network bus for data transfers	
Interconnect Switch	A network switch for data transfers	
Middleware	A middleware package used by platform component(s)	
Source/Sink Device	Generic devices, such as a sensor (source) or display (sink)	
Node Container	A container that may be replicated to model scalable architectures	

figurable device. Reconfigurable processors in RCML are assumed to be capable of concurrently executing all tasks presently configured on the device independent of the processing load of other tasks, assuming enough hardware resources are available for all tasks. Furthermore, an explicit reconfiguration of the reconfigurable processor is needed to execute a task that is not presently configured on the device. While a fixed processor (where hardware reconfiguration is not supported, such as a microprocessor) is often capable of concurrent execution of multiple tasks via threads, the performance for each task is dependent upon the load from other concurrent threads and the number of cores in the processor. The network switch, memory cache, and middleware classes are additional architectural components in RCML that do not have a direct equivalent in AADL, but are typically modeled by manipulating an instance of a related AADL class. A summary of the RCML architecture constructs is presented in Table II. Architecture blocks can also be grouped together to form a *node* using the node container, which can then be replicated in order to easily model scalable platforms.

Each of the generic architecture constructs can be further refined to represent a specific instance of that class by defining custom attributes. For example, the attributes for a network switch can be used to define the speed, width, protocol, and the type of switching backplane of the switch. With this approach, the network switch class can represent a particular instance of a HyperTransport, Ethernet, or other interconnect switch. Memory blocks could be refined to represent memory types from SRAM to external disks by defining attributes for their size, speeds, and protocol. Memory blocks may also be embedded within a processor or device block to represent an internal memory for the host block. Not only does this approach support a very flexible and robust modeling environment capable of modeling many types of architectures, users can efficiently perform design-space exploration by

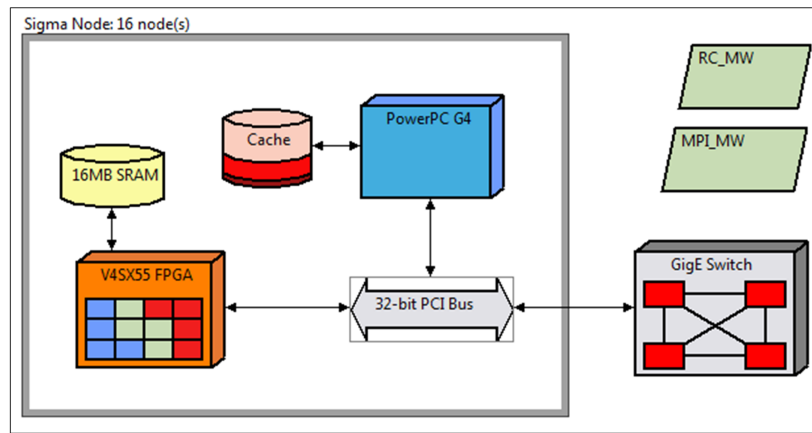


Fig. 4: Example RCML Architecture Model of FPGA Cluster

changing architectural attributes of interest. For example, studying the effect of using a future generation of FPGAs can be approximated by increasing the number of resources defined for a reconfigurable processor. Architectural scalability studies can often be easily performed by simply changing the number of nodes for a node container.

Fig. 4 shows an example of an RCML architecture model used to represent a cluster of FPGA-enhanced embedded compute nodes. Each node contains a PowerPC microprocessor with multi-level cache attached to it, and an FPGA card containing a Xilinx Virtex-4 SX55 FPGA and SRAM memory block. The microprocessor and FPGA are connected by a PCI bus. The node elements all reside within a node container, and 16 instances of the node are specified to realize the full cluster. Each node is connected to a Gigabit Ethernet switch to model full connectivity between nodes. Finally, two middleware components are defined in this model. Other architecture components can link to middleware instances by declaring middleware associations in their attributes. Since it may be common for a middleware package to reside on a large number of component instances, no visual connection is depicted in the figure. Instead, the information is accessed by viewing the attributes of an individual components, which show all of the middleware instances that the component is associated with. In the example in Fig. 4, the *MPI_MW* middleware component is linked to the PowerPC microprocessor in each node (via attributes not depicted in the figure) and is used to characterize additional delays introduced by the MPI communication layer and protocols used for inter-node transfers. The *RC_MW* component is linked to the PowerPC and the Virtex-4 FPGA in each node, and is used to characterize additional communication delays introduced by the middleware associated with the FPGA board. Parameters defining the sizes and speeds of the various devices are embedded as attributes in the individual components, and are accessible through interfaces that appear along the bottom of the RCML tool screenshot in Fig. 5 (it should be noted that the RCML model depicted in Fig. 5 is an example of an algorithm model, but the interfaces for defining and

editing attributes are generally the same within the tool when editing any type of RCML model).

3.3 System Modeling and Mapping

An RCML system model combines an algorithm model and architecture model, representing a complete view of the RC system under study. System models are supplemented with additional information that defines how the algorithm and architecture are mapped together. RCML supports multiple levels of mappings, so the user can choose whether to quickly define a basic mapping of the algorithm onto the architecture, or whether to specify a more complete and detailed mapping using additional optional mapping procedures.

The basic mapping in RCML requires that each function block in the algorithm model is mapped to a processing element in the architecture model. Each non-hierarchical function block can only be mapped to a single processing element, unless the function block is replicated in which case each instance may be mapped individually. Each stage of a process line may also be independently mapped to different processing elements. In addition to the spatial mapping that is required for all tasks, tasks executing on an RC device must be mapped temporally as well, since dynamic reconfiguration can be used during runtime to change the tasks supported by the device. Thus, a reconfigurable core manager is also provided to allow users to explicitly define groups of tasks that will reside concurrently on an RC device at any point in time. The information from the core manager is used by analysis tools to determine the necessary reconfiguration events and their associated impact on system performance.

In many cases, users will want to define additional information regarding how their application is mapped onto the execution platform. For these cases, optional mapping procedures are supported to define additional mapping information. For example, users can map data blocks in the algorithm to memory blocks in the architecture. For each data block, a starting and final memory location can be defined. If a data block is not mapped to a memory block for its starting (final) location, it is assumed that the data is created (consumed) at the location of the first (last) function block that communicates elements in the data block. Also, each input data port on a function block may be mapped to a memory location, allowing intermediate memory locations for an application's data during execution to be defined. This mapping can be used, for example, to define whether data sent to an FPGA is initially written to onboard SRAM or BRAM upon receipt. In the absence of this mapping information, analysis tools are forced to make assumptions regarding where data resides during intermediate portions of the application, which can lead to less accurate prediction results.

3.4 RCML Editor

A graphical tool for creating and editing RCML models has been developed in Eclipse. The RCML editor in Eclipse allows users to drag and drop RCML components onto the model's canvas and directly edit selected components through graphical interfaces and dialogs. A screenshot of the matrix-multiply algorithm model example from Section 3.1 in the Eclipse-based RCML Editor is shown in

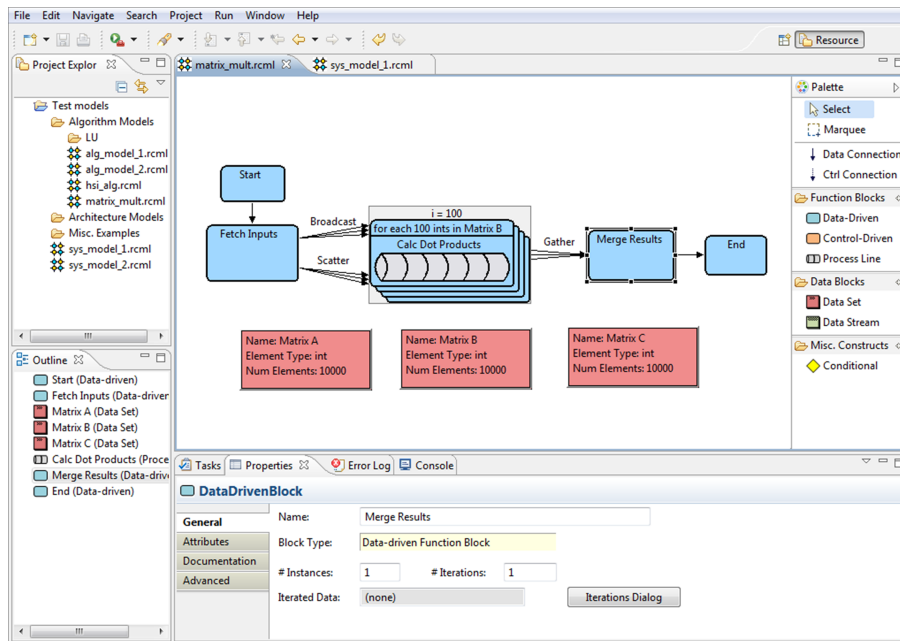


Fig. 5: Screenshot of RCML Model Editor in Eclipse

Fig. 5. Additional examples of RCML models built using the Eclipse RCML tool are presented in Section 5.

4. INTEGRATED ANALYSIS TOOLS

In order for RCML to support early design-space exploration, methods to predict and analyze the behavior and quality of the modeled design are required. Without such techniques, it will be difficult for designers to judge whether their initial system design is expected to meet project specifications and thus avoid wasting time on unnecessary design iterations. Existing prediction and analysis methods can be integrated within the RCML environment to provide tools for automated analysis of RCML models. In this section, two such tools are presented based on analysis methodologies introduced in Section 2. The first tool performs an analytical performance prediction based on the RAT methodology. The second tool generates application scripts and parameter files to automate simulations in the trace-driven RCSE simulation framework. Both prediction tools use a common model parser which traverses the RCML algorithm model within a system model, constructing a list of partial paths through the task graph that are used by the analysis tools.

4.1 Automated RAT Analysis Tool

RAT [Holland et al. 2009] is a methodology for predicting the performance of a specific algorithm on a specific FPGA platform prior to detailed implementation. Key algorithm and architectural features are parameterized and used to compute the communication and computation times and ultimately the system performance. The RAT automated analysis tool provides an infrastructure for analyzing and gath-

ering the necessary parameters from the RCML models to compute the analytical performance predictions.

Before RCML, RAT parameters were manually gathered from a “pencil and paper” plan for the algorithm. Users can now explore and plan their intended algorithm within RCML and affix the performance characteristics as attributes of the model blocks. Using RCML, constructing algorithm and architecture models for use with RAT becomes intuitive but still must follow the basic structural and behavioral conventions of RCML. Information must be provided in a format similar to the existing manual RAT analysis. For example, computational throughput, number of elements, and number of computation per element must be provided within the corresponding function blocks.

Automated RAT analysis via RCML begins with the model parser for the algorithm model. Based on the path information provided by the parser, the RAT tool can estimate the performance of each function block (i.e. task). If a block receives data from another block mapped to a different device, the communication time is considered. The overall system performance is an agglomeration of the individual communication and computation times adjusted based on the algorithm path.

One key advantage of automated RAT analysis versus manual calculation is the speed and efficiency of prediction, especially as algorithms increase in complexity and are quickly revised by the designer. More importantly, RCML enables rapid exploration of different algorithms, which is useful when RAT predictions for one particular algorithm do not meet design constraints.

4.2 Automated RCSE Analysis Tool

In some cases, analytic prediction techniques are not sufficient for the designer’s needs, and a more accurate analysis technique is desired. In such cases, simulation tools are used extensively to provide more detailed and reliable analyses. As discussed in Section 2, the RCSE framework [Reardon et al. 2009] uses discrete-event models of RC systems stimulated by application scripts which define the behavior of the application in terms of key events (e.g. communication transfers, blocks of computation, etc.). The individual components that make up the discrete-event platform models are highly generic and use parameter files to describe their performance. The tool presented here automatically generates application scripts for the RCSE framework from an RCML system model. Additionally, this tool also creates parameter files for each component in the RCML architecture model that is used by the equivalent component within the simulation environment.

In order to generate a complete application script, each function block must contain attributes to describe the computational demands of the task. Attribute sets that enable a RAT analysis for the system are typically sufficient for the simulation tool. Alternatively, users may define attributes for the number of clock cycles required by tasks mapped to RC devices, or the number of instructions and memory/cache accesses performed by tasks mapped to microprocessors.

A separate application script is needed for each participating processor in the system. Therefore, the tool creates a script for each processor with one or more function blocks mapped to it. Each function block mapped to a fixed processor generates a computation entry in the mapped processor’s script, while each RC core defined in the system mapping creates a function call to the RC device preceded

by a core configuration command. Linked function blocks mapped onto different processing elements will generate communication events in the scripts. A script command for intra-node communication is inserted into the corresponding script(s) when two processing elements in the same node are involved. Otherwise, an MPI function call which matches the connection's communication pattern is placed into the script of each participating processor.

Alternatively, automatic generation of parameter files is straightforward. For each architecture block, the block's attributes are compared against a list of predefined attributes for that component class, and the values for matching attributes are recorded. The parameter files and application scripts represent all inputs needed to drive a simulative analysis of the system in RCSE.

5. CASE STUDIES

By integrating the automated analysis tools introduced in Section 4, RCML now provides an environment for abstract modeling and evaluation of preliminary designs of RC systems. In this section, case studies are conducted to demonstrate and validate the effectiveness of the RCML environment and the two integrated automated prediction tools in enabling early design-space exploration. First, the automated RAT tool is used in Section 5.1 to provide an analytic performance prediction for an image filtering application. Next, Section 5.2 presents a case study in which the multi-node performance of a hyperspectral imaging application is predicted through simulations in RCSE driven by inputs automatically generated from an RCML model. Both of the case studies presented in this section illustrate how estimation-level models of RC systems can be effectively built and analyzed using the RCML environment, which can provide substantial gains in overall productivity.

5.1 Automated RAT Case Study

For this case study, the target application is an image filter consisting of a 3×3 discrete 2-D convolution of an image segment (i.e. a pixel and its 8 neighbors) with a user-specified filter. The RCML algorithm diagram for the image filter, constructed in the Eclipse RCML editor, is shown in Fig. 6. The primary kernel in this algorithm is the pipelined computation for the image filter which uses the input stream of pixel values and user-provided filter values to produce a stream of filtered pixels. The input image, defined by the data set labeled *Input Image*, is preprocessed and segmented into three streams at the *Image Segmentation* task prior to the image filter computation. The Input Image data set is transferred over the connection between the Preprocessing and Image Segmentation tasks (this is not depicted in the model's figure). Although nine data elements are required per pixel computation, the data is buffered using shift registers so that only three new data values are required per pipeline input. Only three new pixels are required for each new pipeline iteration because (except for pixels on the border of the image) a horizontally or vertically adjacent pixel will share six pixels used in the 2-D convolution with the current pixel. Each of the three input streams, defined by the data stream labeled Input Stream, is mapped to one of the input connections feeding the *Image Filter* pipeline task (not depicted). The Image Filter pipeline is then set to execute every time a new pixel is available on each of the three input connections (not depicted). An input data set of 65,536 pixels is illustrated in

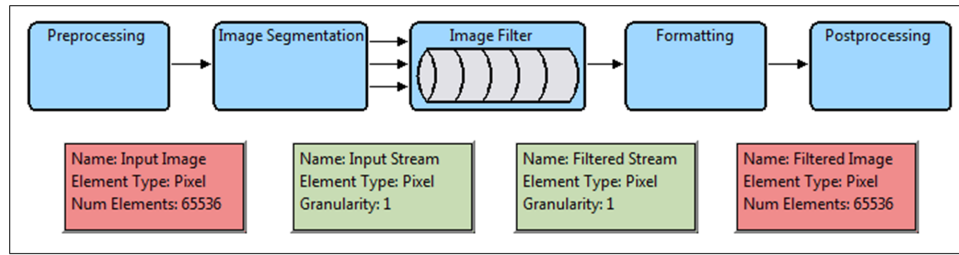


Fig. 6: RCML Algorithm Model of Image Filtering Application used in RAT Case Study

the *Input Image* data set in Fig. 6, representing a 256×256 image arranged and replicated for the image filtering computation. However, several image sizes are used to validate the model and are specified by simply changing the number of data elements in the Input Image and Filtered Image data sets. The Formatting and Postprocessing tasks, represented by function blocks with the same names, assemble the filtered pixel stream into a contiguous data block that represents the final filtered image. The assembly of the Filtered Image is specified to occur during the Formatting task since the output connection for that task is the first to transfer elements from the Filtered Image data set.

The experimental platform used in this case study is an SRC-7 system featuring one MAP-H FPGA node connected to a 3.0GHz dual-core Xeon processor via the proprietary SNAP interconnect (Fig. 7). Each MAP-H node consists of two Altera Stratix EP2S180 FPGAs, one primary and one secondary, with eight dual-ported SRAM banks. The secondary FPGA is unused for this application. Although the target MAP-H system used for this case study is installed in a ground-based system, the models (and subsequent implementation code) also apply to embedded-system versions produced by SRC. The algorithm and architecture models are annotated with RAT parameters as defined in Holland et al. [2009], but are not depicted in Fig. 7. These parameters include the data set sizes, the computational requirements and throughput of the image filter algorithm, the SNAP interconnect throughput, and the FPGA clock frequency. The algorithm and architecture models are mapped together to form the system model for the application. The image segmentation, filtering, and formatting tasks are performed on the FPGA while the preprocessing and postprocessing are mapped on the Xeon microprocessor.

The results of the RAT performance prediction as compared to the subsequent hardware implementation for the end-to-end runtime of the application are summarized in Table III. The validity of the analytical inputs and models is confirmed by the low error which further decreases with increasing image size. The largest source of error for this application (and other algorithms with a similar mapping on the SRC-7 platform) is the latency of computation and communication. The application implementation is automatically pipelined using the Carte compiler and the exact pipeline depth is not known a priori. This unaccounted latency is most significant when the problem size is relatively small. Similarly, the communication latency will impact the performance of short data transfers. More detailed algorithm layout and communication microbenchmarking can further reduce prediction errors at the expense of greater design effort during RCML-level abstract modeling.

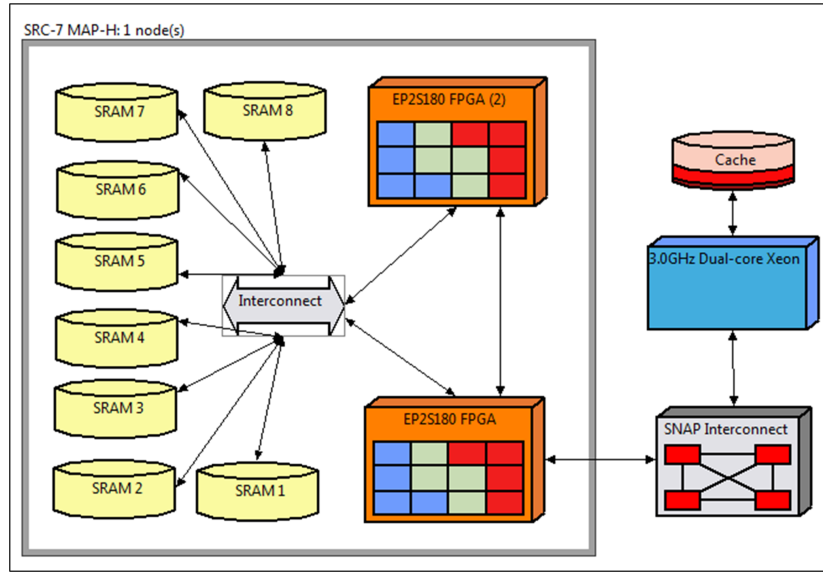


Fig. 7: RCML Architecture Model of SRC Platform used in RAT Case Study

Table III. Analytical Validation Results

Image Size	Analytical Prediction	Experimental Runtime	Error
256x256	5.24E-03s	5.58E-03s	6.11%
512x512	2.10E-02s	2.14E-02s	2.15%
1024x1024	8.39E-02s	8.47E-02s	0.92%
2048x2048	3.36E-01s	3.38E-01s	0.62%

5.2 Automated RCSE Case Study

The application used in this case study is a parameterizable benchmark which performs target detection and classification on a hyperspectral image (HSI) [Chang et al. 2004]. The RCML algorithm diagram for HSI, built using the Eclipse RCML editor, is shown in Fig. 8. The HSI algorithm used in this article is the same as the implementation described in Jacobs et al. [2008], which can be divided into three primary stages: calculation of the auto-correlation sample matrix (ACSM), weight vector calculation (WVC), and target classification (TC). The input data processed by this application is a 3-dimensional image, represented by the *ACSM Input* data set in Fig. 8, which is used to help determine the total amount of computation performed by the ACSM Calculation and Target Classification tasks. The ACSM Input data set is transferred over the Scatter connection between the PreProcessing and ACSM Calculation tasks (not depicted), thus each instance of ACSM Calculation receives a fraction of the total number of elements of ACSM Input. In the ACSM stage, which consists of the *PreProcessing*, *ACSM Calculation*, and *Average ACSM Sums* tasks in Fig. 8, the cross-product of each pixel vector is calculated and the results are summed together. The resulting sum, first produced by the ACSM

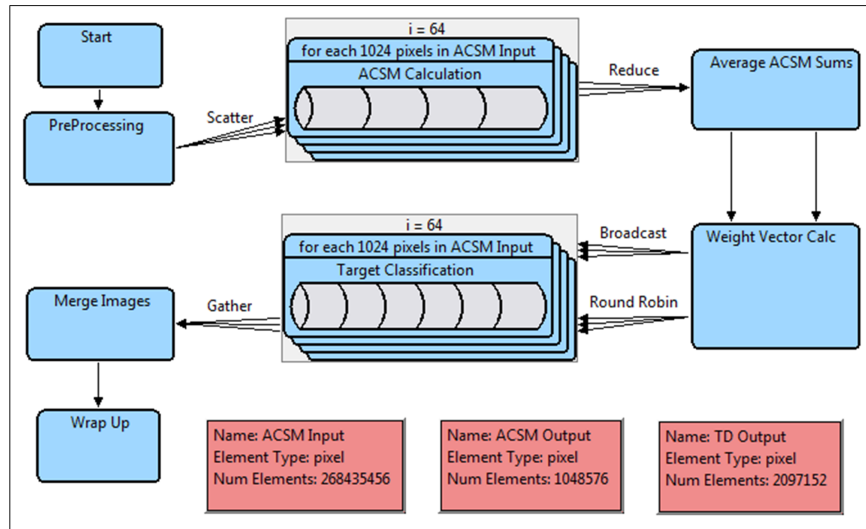


Fig. 8: RCML Algorithm Model of HSI Application used in RCSE Case Study

Calculation task and then transferred to the next two tasks (not depicted) is represented by the *ACSM Output* data set. The pipelined cross-product calculation in ACSM is represented as an RCML process line (labeled *ACSM calculation*), which can be duplicated and performed in parallel as many times as hardware resources will support since each pixel vector may be processed independently. Following the serial WVC stage, the TC stage involves comparing each pixel vector against the spectral signature of each target class. Each pixel vector can again be processed concurrently and the comparisons against each target may be pipelined in hardware, as reflected in the duplicated process line labeled *Target Classification*. While two of the three stages can be processed on the FPGA (ACSM and TC), the FPGA is used to accelerate the *ACSM Calculation* task only in this case study, while the remaining stages are mapped to microprocessors. The input image size (defined by the data set labeled *ACSM Input*) is assumed to be 512×512 pixels with 1024 spectral bands and 8 target classifications, unless otherwise noted. The image size is edited by simply changing the number of data elements in each data set of the RCML algorithm model.

The experimental platform used in this case study is a 4-node cluster of Linux servers. Each node is equipped with dual 1.42GHz PowerPC G4 processors, 1GB of PC2700 system memory, and an Alpha Data ADM-XRC-4 FPGA board which consists of a Xilinx Virtex4-SX55 FPGA along with 16MB of SRAM split across four banks. A 32-bit PCI bus running at 33MHz connects the FPGA board to the remainder of the node. All of the nodes are connected via a Gigabit Ethernet switch. The RCML architecture diagram for this system was originally shown in Fig. 4 and described in Section 3.2. The RCML model for this platform is scaled up to 16 nodes to demonstrate analysis of hypothetical systems that are not physically available. The RCML algorithm and architecture models are annotated with the performance parameters presented in Jacobs et al. [2008], adjusted as necessary for

Table IV. Simulative Validation Results

System Size	# Bands	Simulative Prediction	Experimental Runtime	Error
2 Nodes	256	8.26s	7.98s	3.4%
2 Nodes	1024	84.72s	78.47s	7.4%
4 Nodes	256	6.49s	6.71s	3.3%
4 Nodes	1024	60.26s	58.86s	2.3%

the 512×512 image data. The two RCML models presented in this section are mapped together to create the system model analyzed in the following case study. The *only* changes made to the RCML model between simulative experiments is the basic mapping information based on the number of nodes used, and the data set sizes to reflect changes to the size of the input image.

The first simulative experiments are used to validate the accuracy of the simulation inputs and models. Table IV summarizes results from experiments comparing simulative projections of HSI performance against experimental results presented in Jacobs et al. [2008]. Simulative results from three of the four validation experiments yield predictions that are within 4% of the experimental runtime. The remaining validation experiment yields an error of 7.4%, still within a range that can provide useful insight to designers during early design-space exploration. These experiments demonstrate that accurate simulation inputs are generated by the tool. Furthermore, the inputs are generated much more efficiently and reliably through an automated tool, otherwise users would be required to reproduce scripts manually for each simulative experiment. Thus, having an environment and tool to automate these tasks greatly increases the productivity of simulative analysis.

The second simulative experiment analyzes the scalability of HSI on larger systems by scaling the platform in the RCML model up to 16 nodes to demonstrate analysis of hypothetical systems that are not physically available to the designer. Having validated the accuracy of the simulations for 2- and 4-node systems, we can use the RCML model and simulation tools to project the performance of HSI on larger systems. Fig. 9 shows the simulative projected performance of HSI as the system size is scaled from two to 16 nodes for a 512×512 image with 256 and 1024 spectral bands. When scaling up to eight nodes with 1024 spectral bands, significant reductions in runtime are projected due to parallelism that can be exploited during the ACSM and TC stages. But Fig. 9 also shows that the runtime is minimally reduced when scaling beyond 12 nodes. The penalties for communication and the serialization of the weight computation stage limit the parallel efficiency of HSI on larger system sizes. Furthermore, smaller relative gains in performance are observed with 256 spectral bands, whose runs have a smaller computation-to-communication ratio, thus gains from parallel execution are offset by communication overhead.

6. CONCLUSIONS

While RC devices such as FPGAs are becoming an important option for realizing highly efficient and flexible systems for high-performance and/or embedded computing, the time and difficulty associated with developing applications for RC systems are often prohibitive, making it difficult to exploit the potential gains in

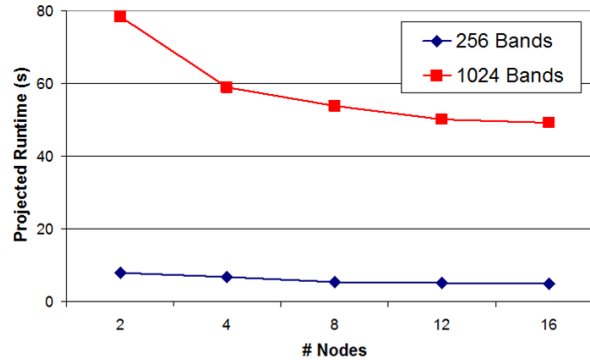


Fig. 9: Predicted Runtime vs. Number of Nodes

performance and power savings that RC can provide. To improve RC productivity, better concepts and tools are needed which allow designers to plan and analyze their designs coding a specific (and possibly fruitless) implementation.

In this article, a new abstract, hierarchical modeling language and environment for representing RC systems was introduced, called RCML. The RCML framework includes three types of models: algorithm models which represent applications as RC-specialized task graphs, architecture models which represent execution platforms as a collection of components, and system models which map together an algorithm and architecture model. Constructs in RCML are tailored to allow RC designers to quickly and concisely define their parallel algorithm and execution platform at varying levels of abstraction. RCML also provides a foundation for automated performance prediction during initial development stages. Two such tools for automated performance prediction of RCML models were introduced. Case studies involving automated analysis of applications using both integrated analysis tools illustrate how RCML enables efficient early design-space exploration of RC systems.

Future work includes a number of expansions to the RCML environment and tool, and the development of algorithms to automate task mapping and partitioning of RCML system models. Methods for code generation from RCML models are also planned, to facilitate the transition from estimation-level RCML models to more detailed, functional hardware models. Automated implementation presents a challenge, given the large number of different FPGA languages and platforms available for users to target. Nevertheless, existing core libraries could easily be linked to RCML blocks to enable automated implementation of circuits using existing IP cores. Finally, a framework for lowering the level of modeling abstraction is being studied and incorporated with RCML in order to model and accurately predict the clock frequency, latency, and resource utilization of reconfigurable hardware circuit designs [Wang et al. 2009].

REFERENCES

- BALARIN, F., WATANABE, Y., HSIEH, H., LAVAGNO, L., PASSERONE, C., AND SANGIOVANNI-VINCENTELLI, A. 2003. Metropolis: an integrated electronic system design environment. *Com-ACM Journal Name*, Vol. V, No. N, Month 20YY.

- puter* 36, 4 (April), 45–52.
- BUCK, J., HA, S., LEE, E. A., AND MESSERSCHMITT, D. G. 1994. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation* 4, 152–184.
- CHANG, C.-I., REN, H., AND CHIANG, S.-S. 2004. Real-time processing algorithm for target detection and classification in hyperspectral imagery. *IEEE Trans. on Geoscience and Remote Sensing* 39, 4 (April), 760–768.
- DENSMORE, D., DONLIN, A., AND SANGIOVANNI-VINCENTELLI, A. 2006. FPGA architecture characterization for system level performance analysis. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*. European Design and Automation Association, 764–739.
- EKER, J., JANNECK, J. W., LEE, E. A., LIU, J., LIU, X., LUDVIG, J., NEUENDORFFER, S., SACHS, S., AND XIONG, Y. 2003. Taming hereogeneity - the ptolemy approach. *Proc. of the IEEE* 91, 1 (January), 127–144.
- ENZLER, R., PLESSL, C., AND PLATZNER, M. 2005. System-level performance evaluation of reconfigurable processors. *Microprocessors and Microsystems* 29, 2-3 (April), 63–75. Special Issue on FPGA Tools and Techniques.
- FEILER, P., GULCH, D., AND HUDAK, J. 2006. The architecture analysis and design language (AADL): An introduction. Tech. Rep. (CMU/SEI-2006-TN-011, ADA455842), Software Engineering Institute, Carnegie Mellon University.
- FU, W. AND COMPTON, K. 2006. A simulation platform for reconfigurable computing research. In *International Conference on Field Programmable Logic and Applications, 2006. FPL '06*. 1–7.
- GROBELNY, E., BUENO, D., TROXEL, I., GEORGE, A., AND VETTER, J. 2007. FASE: A framework for scalable performance prediction of HPC systems and applications. *Simulation: Trans. of Society for Modeling and Simulation International* 83, 10 (October), 721–745.
- HOLLAND, B., NAGARAJAN, K., AND GEORGE, A. 2009. RAT: RC amenability test for rapid performance prediction. *ACM Trans. on Reconfigurable Technology and Systems (TRETS)* 1, 4 (January), 22:1–22:31.
- JACOBS, A., CONGER, C., AND GEORGE, A. 2008. Multiparadigm space processing for hyperspectral imaging. In *Proc. of IEEE Aerospace Conference*. Big Sky, MT.
- KAHN, G. 1974. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress*. Number 74. North-Holland Publishing Co.
- KEUTZER, K., MALIK, S., NEWTON, A. R., RABAAY, J. M., AND SANGIOVANNI-VINCENTELLI, A. 2000. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 19, 12 (December), 1523–1541.
- KIENHUIS, B., DEPRETTERE, E. F., VAN DER WOLF, P., AND VISSERS, K. 2002. *Embedded Processor Design Challenges*. Springer, Chapter A Methodology to Design Programmable Embedded Systems: The Y-Chart Approach, 18–37.
- LIEVERSE, P., VAN DER WOLF, P., AND DEPRETTERE, E. 2001. A trace transformation technique for communication refinement. In *Proc. of the International Symposium on Hardware/Software Codesign*. 134–139.
- MOHANTY, S. AND PRASANNA, V. K. 2007. A model-based extensible framework for efficient application design using FPGA. *ACM Trans. on Design Automation of Electronic Systems* 12, 2 (April), 13.
- MOHANTY, S., PRASANNA, V. K., NEEMA, S., AND DAVIS, J. 2002. Rapid design-space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation. In *Proc. of the joint conference on Languages, compilers and tools for embedded systems (LCTES/SCOPES)*. ACM, New York, NY, USA, 18–27.
- OBJECT MANAGEMENT GROUP. 2008. *OMG Systems Modeling Language (OMG SysML), v1.1*, formal/2008-11-01 ed. Object Management Group.
- OBJECT MANAGEMENT GROUP. 2009. *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, v1.0*, formal/2009-11-02 ed. Object Management Group.

- PIMENTEL, A. D. 2005. The artemis workbench for system-level performance evaluation of embedded systems. *Intl. Journal of Embedded Systems* 3, 3, 181–196.
- PIMENTEL, A. D., ERBAS, C., AND POLSTRA, S. 2006. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. on Computers* 55, 2 (February), 99–112.
- PIMENTEL, A. D., HERTZBETGER, L. O., LIEVERSE, P., VAN DER WOLF, P., AND DEPRETTERE, E. F. 2001. Exploring embedded-systems architectures with artemis. *Computer* 34, 11 (November), 57–63.
- REARDON, C., GROBELNY, E., GEORGE, A., AND WANG, G. 2009. A simulation framework for rapid analysis of reconfigurable computing systems. *ACM Trans. on Reconfigurable Technologies and Systems (TRETTS)*, to appear.
- RUGINA, A., KANOUN, K., AND KAANICHE, M. 2006. An architecture-based dependability modeling framework using AADL. In *Proc. 10th IASTED Intl Conference on Software Engineering and Applications (SEA'2006)*. Dallas, TX, USA.
- SAE. 2004. *SAE Standards: AS5506, Architecture Analysis & Design Language (AADL)*. Society of Automotive Engineers.
- SINGHOFF, F., LEGRAND, J., NANA, L., AND MARCÉ, L. 2005. Scheduling and memory requirements analysis with AADL. In *Proceedings of the 2005 Annual ACM SIGAda International Conference on Ada*. ACM, New York, NY, USA, 1–10.
- SMITH, M. C. AND PETERSON, G. D. 2002. Analytical modeling for high-performance reconfigurable computers. In *Proc. of SCS International Symposium on Performance Evaluation of Computer and Telecommunications Systems (SPECTS)*. San Diego, CA.
- SOKOLSKY, O., LEE, I., AND CLARKE, D. 2006. Schedulability analysis of AADL models. In *Proc. 20th IEEE International Parallel & Distributed Processing Symposium*. Rhodes Island, Greece.
- STEFFEN, C. P. 2007. Parameterization of algorithms and FPGA accelerators to predict performance. In *Proc. of Reconfigurable System Summer Institute (RSSI)*. Urbana, IL, 17–20.
- UML REVISION TASK FORCE. 2001. *OMG Unified Modeling Language Specification, v1.4*. Object Management Group.
- WALKER, D. W. 1994. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing* 20, 4, 657–673.
- WANG, G., STITT, G., LAM, H., AND GEORGE, A. 2009. A framework for core-level modeling and design of reconfigurable computing algorithms. In *Proc. of High-Performance Reconfigurable Computing Technology and Applications Workshop (HPRTCA)*. ACM, 29–38.

Received Month Year; revised Month Year; accepted Month Year