

# Developing Architectural Platforms: A Disciplined Approach

**Andrew Mihal, Chidamber Kulkarni,  
Matthew Moskewicz, Mel Tsai, Niraj Shah,  
Scott Weber, Yujia Jin, and Kurt Keutzer**  
University of California, Berkeley

**Christian Sauer**  
University of California, Berkeley,  
and Infineon Technologies

**Kees Vissers**  
University of California, Berkeley,  
and Chameleon Systems

**Sharad Malik**  
Princeton University

The Mescal project brings a formalized, disciplined methodology to the design of programmable platform-based systems, enabling the exploration of a wide array of architectures and a correct-by-construction path to implementation.

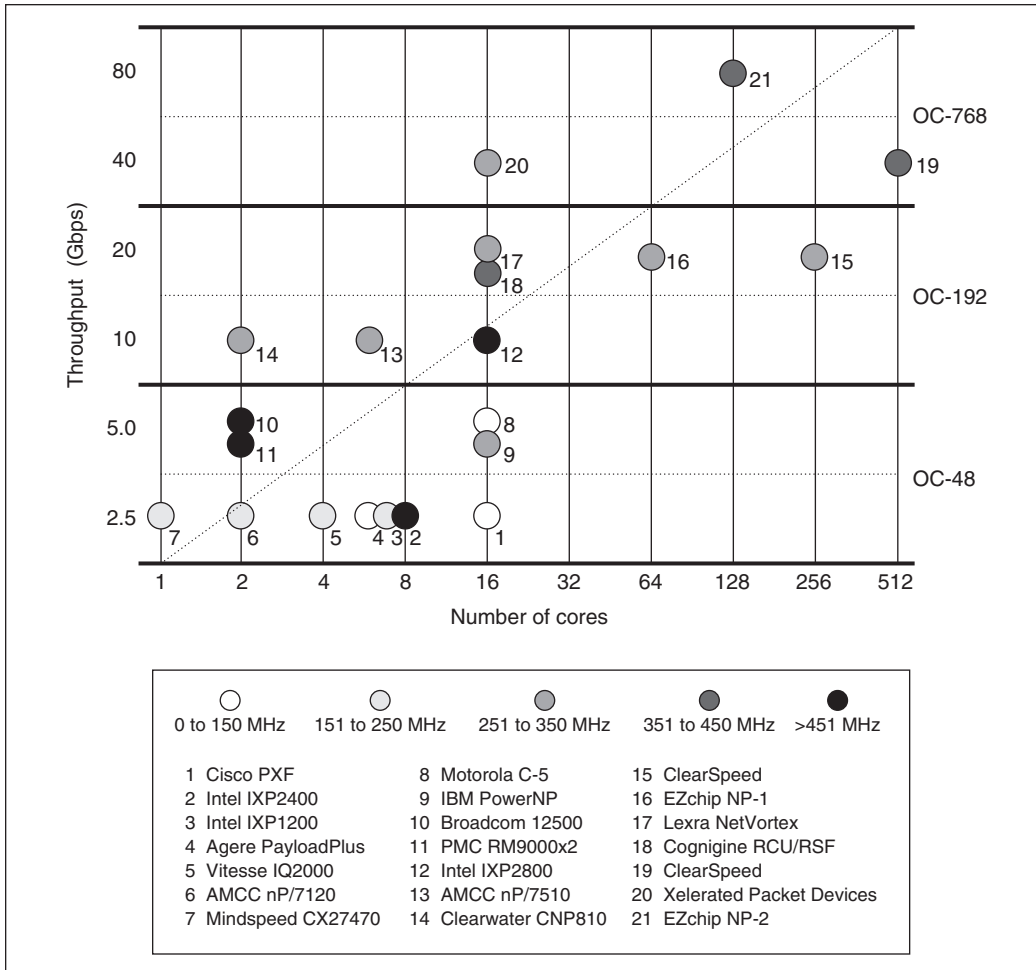
■ **THE EXPONENTIAL INCREASE** in IC complexity defined by Moore's law poses tough challenges for the design and design automation communities. First, the deep-submicron effects of this growing complexity—such as increasing interconnect delay and coupling—complicate the design of individual devices and associated interconnects. Second, the number of devices has increased exponentially. Third, IC designers must integrate heterogeneous elements—digital, analog and mixed signal, RF, and software—on the same piece of silicon. Fourth, competitive pressures require further reductions in time to market. These challenges have led to the well-publicized gap between manufacturing capability and design productivity.

In addition to the intellectual design chal-

lenges, the nonrecurring engineering (NRE) costs associated with manufacturing pose significant economic challenges. The *International Technology Roadmap for Semiconductors* predicts that although manufacturing complex system-on-a-chip designs with feature sizes as small as 50 nm will be feasible between 2005 and 2007, the production of masks and exposure systems will likely impede the development of such chips. That is, the cost of masks will grow even more rapidly for these fine geometries, compounding the up-front NRE costs for a new design. At 0.15-micron technology and below, designers are in the realm of the “million-dollar mask set.”

One way to amortize design cost and ameliorate design risk is to make a single IC that serves multiple applications via user programmability. Programmability comes with an associated power and delay overhead, however, which designers can mitigate through application-specific hardware resources in the programmable platform. Although these platforms already exist in some application domains, their design process is largely ad hoc. Furthermore, despite high development costs, such platforms tend to be difficult to program, and very little software support is available.

Our research project and methodology



**Figure 1. Diversity in the network processor space. Processors are ordered according to their claimed performance classes—that is, numbers 1 through 7 are in the lowest performance class; numbers 19 through 21 are in the highest.**

attempt to fill these gaps. The Modern Embedded Systems, Compilers, Architectures, and Languages (Mescal) project introduces a disciplined approach to the production of reusable architectural platforms that can be easily programmed to handle various applications within a domain. In particular, we focus on thorough design-space exploration for network-processing applications. The success of this project requires coordination of research efforts in several areas, including applications, software environments, and application-specific instruction processor (ASIP) design.

### Need for a disciplined approach

The reduced-instruction-set computing (RISC) processor architecture follows a disci-

plined approach that uses benchmarks to quantify the effectiveness of architectures in the design space.<sup>1</sup> Since the invention of this architecture, researchers have explored architectural design spaces in a disciplined fashion for simple instruction set processors, such as very long instruction word (VLIW) processors<sup>2</sup> and some systolic architectures.<sup>3</sup> The architectural diversity of modern ASIPs requires a technique that addresses heterogeneity in both computation and communication.

A particularly good example of ASIP architectural diversity is in the network processor segment. As Figure 1 shows, researchers have proposed very dissimilar architectures for similar network processing problems.<sup>4</sup> The number of processing elements varies widely for the

same performance class, and, alternatively, the same number of processing elements targets different performance classes. Thus, the architectural design space for ASIPs—and network processors, in particular—is very large and poorly suited to exploration through human intuition alone.

Our methodology, supported by tools, defines and explores the design space in a disciplined fashion. Computation<sup>5</sup> and architecture models, combined with a disciplined mapping methodology, are our guiding principles. The key elements of the methodology are

- developing representative benchmarks,
- defining the architectural design space,
- exploring the design space using a disciplined approach based on a common integrated tool framework,
- formally verifying the consistency between various facets of an architecture, and
- mapping applications onto architectures in a disciplined fashion.

### Disciplined benchmarking

Benchmarking lets designers draw quantitative performance conclusions regarding a computing system. In particular, a designer must be able to identify and quantify the performance bottlenecks inherent in the application domain.

The three main characteristics of a good benchmarking methodology for design space exploration are representativeness, precise specification, and workload characterization. Internet protocol forwarding, for example, is a representative benchmark for the network core and edge equipment.<sup>6</sup> The precise specification separates the concerns of purely functional aspects (the number of route table entries, for example) and environmental aspects (such as architectural interfaces) of the benchmark. The Internet Engineering Task Force benchmarking working group bases their workload characterization (packet mix) on a study of Internet traffic.

### Defining the design space

We define the architectural space of ASIPs using several axes: degree of parallel process-

ing, special-purpose hardware used, on-chip communication mechanisms, memory architectures, and peripheral integration. Although we attempt to describe these axes separately and practically, all choices are interrelated. We illustrate them with examples from network processing.

#### Parallel processing

In network processing, we observe moderate instruction-level parallelism, and hence network processors tend to have a larger number of processing elements (or threads) as compared to a larger issue rate. To keep up with embedded systems' increasing computational requirements, ASIP architects must exploit application-level parallelism at three levels:

- processing-element level (packet processing, for example), in which the main concerns are the number of processing elements and target functionality;
- instruction level (signal processing, for example), which is based on multiple issues with either hardware or software control; and
- bit level (such as encryption algorithms like the Triple Digital Encryption Standard).

#### Special-purpose hardware

An application domain's computational kernels are prime candidates for hardware acceleration on an ASIP. The key parameters in designing special-purpose hardware blocks are functionality, interface, and size—that is, whether the hardware implementation fits within the pipeline. The two most common approaches to providing specialized hardware are

- coprocessors, which can store state and have direct access to memories and buses; and
- special functional units, which exist within a pipeline stage, are usually stateless, and write back only to registers.

Most network processors use both techniques extensively for lookup acceleration, queue management, pattern matching, check-

sum and cyclic redundancy checks (CRC) computation, and encryption and authentication.

### Memory architectures

Design decisions in this area significantly affect ASIP performance. The memory architecture's basic parameters include size, type (for example, SRAM or SDRAM), location (on- or off-chip), and memory connectivity. Approaches for hiding memory access latency can be hardware based (cache memory, queues, and content addressable memory) or software based (scratch-pad memory, prefetching, and multithreading).

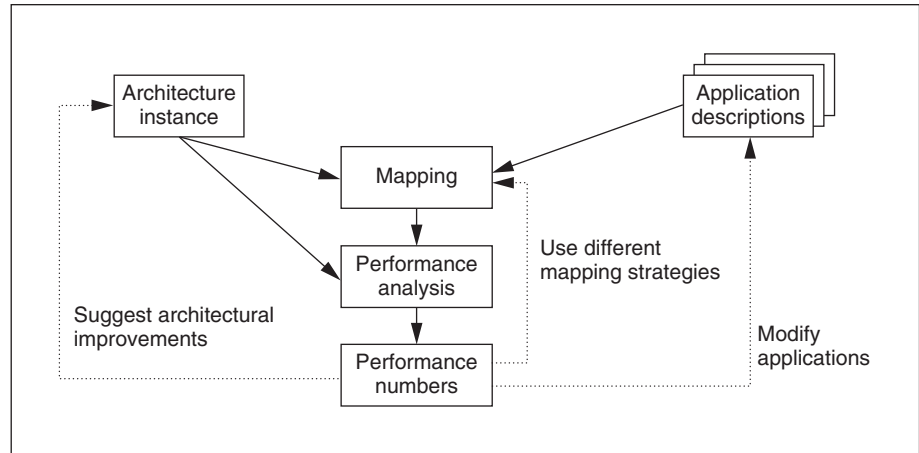
### On-chip communication mechanisms

As the number of components in ASIP architectures grows, the type of on-chip connection and communication schemes for processing elements, coprocessors, memories, and peripherals gains importance. On-chip communication schemes have four major parameters:

- topology—flat or hierarchical;
- connectivity—point-to-point or shared;
- link type—including bit width, throughput, and sharing mechanisms; and
- programmability—hardwired, configurable, or programmable.

### Integration of peripherals

Because all ASIPs are embedded in larger systems, they must interact with peripherals. A network processor must communicate with a media access control layer device to send and receive packets, for example. Depending on the application, an ASIP architect might want to integrate certain peripherals on chip. For ASIPs to be deployed in a variety of environments, programmable or configurable interfaces might be desirable. For example, Motorola's C-5 digital convergence platform (DCP) contains microcoded "Serial Data Processors" that designers can program to perform lower-layer functions.<sup>7</sup>

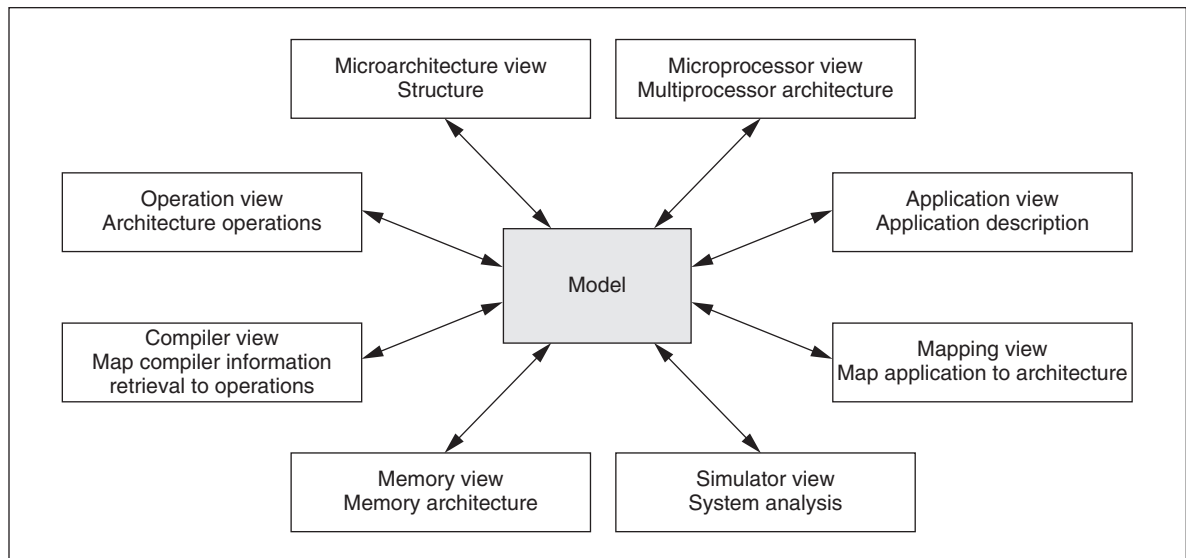


**Figure 2. Y-chart scheme for design space exploration. The Y-chart is an iterative design methodology that allows concurrent development of hardware and software.**

### Mescal's architecture development framework

A successful design space exploration methodology for programmable systems requires a clear distinction between applications and architectures. This distinction enables a systematic exploration of an architecture's suitability for an application domain. Figure 2 illustrates one such method, based on the Y-chart.<sup>8</sup> To implement a system with this methodology, a designer maps an application onto a specific architecture using an explicit mapping step. Performance analysis of this implementation might lead the designer to alter the architecture, adapt the application description, or choose a different mapping to achieve the desired performance. These feedback paths make the Y-chart an iterative design methodology that enables the coevolution of hardware and software. The Mescal architecture development framework (called Teepee) implements a design space exploration methodology based on the Y-chart.

Teepee requires a retargetable compiler and simulator capable of supporting a sufficiently rich set of programming models and architectures that can express parallelism at all levels (process, instruction, and bit), but still allow the automatic generation of an efficient compiler, simulator, and hardware realization. To meet these diverse goals, we use the multiple-views methodology.



**Figure 3. Multiple-views methodology. Designers combine the architecture's structural description and the instruction set semantics, and restrict the architecture's flexibility to manage tool generation complexity.**

### Multiple views

In the past, architecture description languages (Mimola, for example)<sup>9</sup> focused on modeling the architecture's structure or its instruction set semantics (such as the Instruction Set Description Language and nML). A current trend in ADLs is to combine the structural description and the instruction set semantics (as in Expression and Lisa). Designers then restrict the architecture's flexibility to manage tool generation complexity. Verifying that the architecture, compiler, and simulator are consistent after adding new features is a complex problem that has not been well defined. The existing solution is to use a single syntax to describe the architecture's restricted space at all points in the design flow. Relaxing the architectural restrictions makes the consistency and tool-generation problems intractable. The main difficulty with this approach is that architects, compiler designers, and simulator writers are concerned with dissimilar and disjoint facets of the architecture.

We propose a single, simple, formal description of the architecture with an extremely minimal set of design space restrictions. This lets us describe a broad class of architectures. A view of this model is an abstraction encapsulating the information required by a facet of the architecture. We implement each view with a syn-

onymous tool in the Teepee framework designed to interact with the architecture using the corresponding abstraction.

Figure 3 shows how we categorize the views. A compiler view, for example, encapsulates the architectural properties that a compiler needs to know to generate executable code. Because the base description is generic, we cannot expect a particular view to properly or efficiently handle an arbitrary architecture model. In this case, we need to either modify the design to meet the current view's restrictions or use a variant of the view. Views are implemented only with semantics present in the model, and thus are consistent with the model by construction.

Two views that must be consistent, for example, are views of the microarchitecture and of the operations that the microarchitecture implements. To ensure consistency in this area, we can synthesize the microarchitecture from a traditional description of an instruction set, perhaps with some microarchitectural hints (such as the pipeline layout). Another option is to treat the instruction set as a specification and the microarchitecture as an implementation. We can then formally verify the consistency of the implementation and specification.<sup>10</sup> Both of these options can be intractable in practice unless we significantly restrict the instruction set and processor

space to be explored, however. We consider this an imbalance between the architectural model and the compiler and hardware realization views. The Teepee environment approaches this problem using a bottom-up design methodology that unifies the machine operations and the microarchitecture description. The compiler view can easily extract the model's primitive instructions and generate a correct compiler. Similarly, the microarchitecture view can extract the implementations of these instructions to generate the hardware realization.

#### Processing element modeling

Traditional hardware description languages (HDL) describe a processor at the bit level with little formal distinction between the control logic and the data path. Designers must somehow externally verify the desired instruction set against this processor description. In contrast, the Teepee framework describes a processor as a graph of primitive computational components (*actors*) and rules describing acceptable actor use. An actor can have multiple valid uses. Commonly, an actor, in addition to its "normal" behavior, might sometimes do nothing. Actors communicate via signals that are colored by the values and types of data they carry. For instance, a simple binary adder might specify that it can be properly used only if both of its input signals are colored as valid integers. This means its output must be a valid integer as well.

Teepee's rules limit the use of the processor as a whole to configurations that perform useful computation. We call a useful configuration an *operation*. The operations view can automatically generate the set of all operations, which constitutes the processor's instruction set.

In the microarchitecture view shown in Figure 4a (next page), designers connect microarchitecture actors in a schematic editor to create the architecture. Designers then use a theorem prover to extract the state-to-state operations supported by the architecture. Experiments show that extracting instructions from a DLX machine takes less than 1 second. The operation view, shown in Figure 4b, displays extracted operations in terms of the individual ports and connections each operation uses.

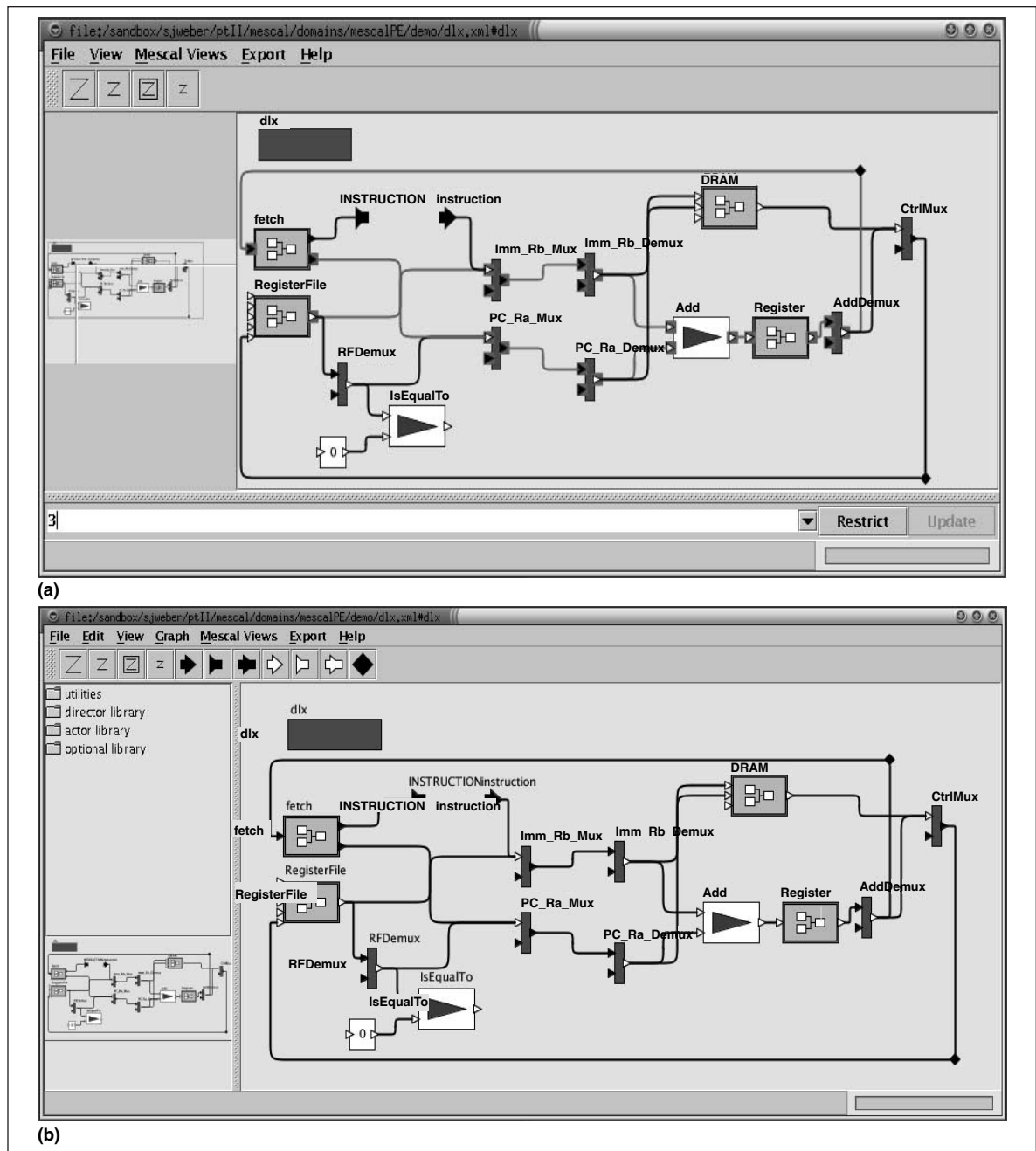
To analyze the architecture, the Teepee sim-

ulator view generates an interpretative simulator. The simulator assumes that all state elements use one global clock and that at each clock cycle it executes a set of conflict-free operations. Future versions will support multiple clocks and multiple control sources. We are also working on a compiler that maps application instructions in an intermediate representation to the architecture's state-to-state operations (compiler view). We will also use the compiler technology to develop a cycle-accurate compiled code simulator (simulator view). We built these two views, along with the other views described in this article, on top of Ptolemy II and Vergil.<sup>5</sup>

#### Memory modeling

The memory view aims to model a large memory design space at a high level of abstraction. The memory design space ranges from simple memory elements, such as register files, to complex memory systems, such as memory hierarchies. We model memory architectures in the memory view by assembling and configuring components from an extendable library. The memory view offers two main benefits. First, the parameterized library elements enable automated exploration during the initial design stage. Second, its flexibility allows an expert memory architect to explore novel designs without starting from scratch.

Memory architectures described in the memory view are formal, functional models. We build all memory elements and systems bottom-up by configuring and connecting three basic actors: table, access, and arbitrate. The table actor is a storage element. The access actor accesses table actors for storing and retrieving data. The arbitrate actor orders multiple accesses by different access actors into the same table actor. The memory view's underlying formalism guarantees precise memory system descriptions and enables a correct-by-construction synthesis path toward an implementation. To ensure consistency with other views, an interface connects the memory view to the microarchitecture view. This interface lets us use a memory system described in the memory view within a processing element model as a processing element actor, while preserving the functional consistency between the two views.



**Figure 4. Microarchitecture (a) and operation (b) views in Teepee. In the microarchitecture view, designers connect microarchitecture actors to a schematic editor. The operations view displays operations according to the ports and connections used.**

#### On-chip communication modeling

Communication architecture design in Teepee is based on the network-on-a-chip paradigm.<sup>11</sup> Teepee's multiprocessor view portrays the system architecture at the coarsest granularity level. In this view, the basic blocks are processing elements and interconnect hard-

ware components such as buses and switches. Architects construct network topologies using an extendable library of network-on-a-chip components.

As with the microarchitecture and memory views, we based the multiprocessor view on a formal computation model that gives the model

functional semantics. This enables a correct-by-construction synthesis path from the complete multiprocessor architecture model to an implementation, such as a synthesizable HDL model or a compiled-code cycle-accurate simulator. In addition, the computation model makes the communication architecture's capabilities explicit. Teepee can use this knowledge to export the correct programmer's model to the application designer. We use this information to carefully map applications onto architectures.

The multiprocessor view offers many opportunities for design space exploration. As stated previously, communication architecture programmability is an important design choice. Communications flexibility distinguishes systems that are highly tuned for one application domain from those that are useful for a range of application domains. For example, an architect might replace hardwired bus interface units with programmable communication-assist coprocessors.<sup>12</sup> This affects the model that the programmer exports to the application designer and broadens the class of applications that the system can implement. The multiprocessor view helps architects explore the tradeoffs associated with this sort of design choice.

#### Application environment

Application design for embedded systems is a special challenge because embedded systems do not simply perform computations; they also interact with their environment. Thus the systems must interface with and react to real processes. To make this happen, designers must juggle real-time constraints, concurrency, and heterogeneity. Sequential programming languages such as C are inadequate. Designers need more powerful tools and abstractions to manage such complex issues.

Formal computation models provide the necessary abstractions. A computation model provides a mathematical framework for concurrency and heterogeneity, thus capturing the complex semantics that are all but impossible to correctly describe in conventional programming languages. Formal models make semantic complexity implicit in the application model.

Ptolemy II is a framework for modeling applications at a high abstraction level using

heterogeneous computation models. We have already described how Teepee extends the Ptolemy II's modeling capability to model architectures. In the Application view, Teepee adds new computation models to Ptolemy II that capture the semantics of the applications driving the Mescal research project.

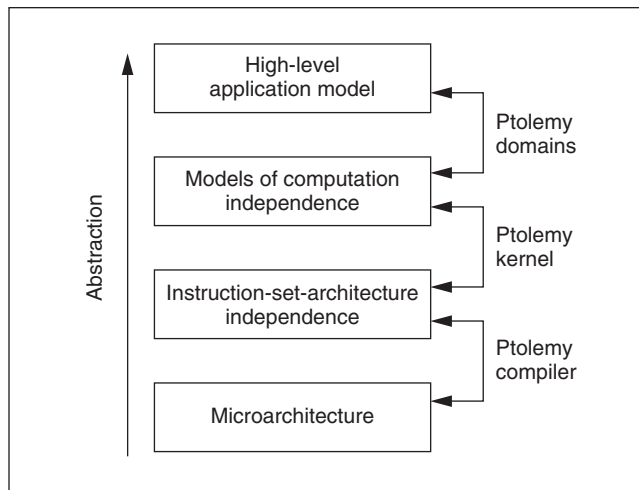
#### Mapping environment

The process of mapping applications onto architectures is the keystone of the Y-chart design methodology. In this step, the designer implements the high-level application model that runs on the target architecture. This is known as *code generation*.

Producing a code generation tool, such as a compiler, for a heterogeneous multiprocessor system requires considerable effort. In an iterative design methodology like the Y-chart, manual implementation of code generation tools is simply not acceptable. The complexity of the mapping step quickly dominates the design process. Every time the architecture changes even slightly, or whenever the application domain varies subtly, designers must rewrite the previous code generator.

We can automatically generate C compilers based on architectural descriptions with instruction extensions. This is not sufficient, however. Languages like C fail to capture the most critical design requirements such as concurrency and heterogeneity. Mescal lets application designers apply high-level computation models directly to system implementation.

Rather than relying on monolithic code generation tools that are tied to single high-level application languages and single architectures, the Mescal mapping methodology builds on existing application design abstractions to provide a code generation framework. Figure 5 shows the spectrum of abstractions. Tools provide paths between adjacent abstractions. For example, a traditional C compiler allows designers to program in a language that is instruction set architecture (ISA) independent, given an existing microarchitecture. Atop this abstraction, toolkits like the Ptolemy II kernel enable the design of heterogeneous applications using general computation models. Ptolemy domains then implement specific



**Figure 5. Spectrum of mapping abstractions. Tools, such as the Ptolemy kernel, link adjacent abstractions.**

computation models atop the Ptolemy kernel. Combining concepts from these tools, the Mescal mapping methodology opens an implementation path from high-level application models to architectures.

Teepee's mapping view implements the Mescal methodology. It presents the application as a set of components that rely on a particular computation model to function, and the architecture as a virtual machine that implements the semantics of a particular computation model. Our methodology's formal description of the communication architecture reduces the mapping problem to matching the communication and control semantics that the application components require with the communication and control semantics that the architecture provides. The mapping view portrays this problem as a communications networking problem, formulating the necessary semantics as an application-layer protocol, and the architecture's capability as a physical-layer protocol.

We build a bridge between application and architecture semantics by creating protocol stacks using the open systems interconnection paradigm. Teepee provides an extendable library of protocol components that are useful for common computation models. Design space exploration in the mapping view is inherent in the selection and implementation of these protocols. Like the application and archi-

tecture views, the mapping view uses a formal computation model to ensure correctness and consistency of the system model.

The mapping view describes the interaction between application and architecture. From this description, Teepee automatically generates the software for each of the architecture's processing elements in an ISA-independent language. This includes application processes as well as operating system kernels and device drivers. We integrate the mapping view with the compiler view to produce executable code for each processing element.

**THE Mescal MAPPING METHODOLOGY** lets us map high-level application models onto architectures while preserving the application model semantics. The cost of this approach is reduced performance when compared to hand coding. Designing a custom monolithic code generation tool with in-depth knowledge of a particular application domain and architecture enables various optimizations. Because the Mescal approach enforces several abstraction layers, we lose the capability to perform cross-layer optimizations. Without a flexible and formal mapping strategy, however, designing systems using the Y-chart paradigm is not feasible. A flexible and formal mapping strategy will let us explore a much larger design space using the Y-chart paradigm than would a custom monolithic code generation tool. We believe that the benefits of using the Y-chart design methodology will outweigh the performance loss in the mapping step for a smaller class of architectures. ■

## References

1. D. Patterson and J. Hennessey, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Francisco, 1996.
2. K. Vissers, "The Trimedia VLIW Media Processor," *Proc. Embedded Processor Forum*, In-Stat/MDR, Scottsdale, Ariz., 2001.
3. B. Rau and M. Schlansker, "Embedded Computer Architecture and Automation," *Computer*, vol. 34, no. 4, Apr. 2001, pp. 75-83.
4. N. Shah, "Understanding Network Processors," master's thesis, Dept. of Electrical Eng. and Com-

puter Sciences, Univ. of Calif., Berkeley, 2001.

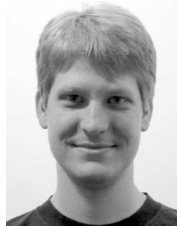
5. J. Davis et al., "Ptolemy II: Heterogeneous Concurrent Modeling and Design in Java," tech. report UCB/ERL M99/40, Dept. of Electrical Eng. and Computer Sciences, Univ. of Calif., Berkeley, 1999.
6. M. Tsai et al., "A Benchmarking Methodology for Network Processors," *Proc. 8th Int'l Symp. High-Performance Computer Architectures (HPCA 02)*, IEEE CS Press, Los Alamitos, Calif., 2002, pp. 75-85.
7. *C-5 Network Processor Architecture Guide*, Motorola, Schaumburg, Ill., 2001, <http://www.motorola.com>.
8. B. Kienhuis et al., "An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures," *Proc. Int'l Conf. Application-Specific Systems, Architectures, and Processors (ASAP 97)*, IEEE CS Press, Los Alamitos, Calif., 1997, pp. 338-349.
9. W. Qin and S. Malik, "Architecture Description Languages for Retargetable Compilation," *The Compiler Design Handbook: Optimizations and Machine Code Generation*, Y.N. Srikant and P. Shankar, eds., CRC Press, Boca Raton, Fla., 2002.
10. J.R. Burch and D.L. Dill, "Automatic Verification of Pipelined Microprocessor Control," *Proc. 6th Int'l Conf. Computer Aided Verification (CAV 94), Lecture Notes in Computer Science*, vol. 818, Springer Verlag, Berlin, 1994, pp. 68-80.
11. M. Sgroi et al., "Addressing the System-on-a-Chip Interconnect Woes through Communication-Based Design," *Proc. Design Automation Conf. (DAC 01)*, IEEE CS Press, Los Alamitos, Calif., 2001, pp. 667-672.
12. M. Horowitz and K. Keutzer, "Hardware-Software Co-design," *Proc. Synthesis and System Meeting and Int'l Interchange (SASIMI 93)*, 1993, pp. 5-14.



**Andrew Mihal** is a PhD student at the University of California, Berkeley. His research interests include the implementation of models of computation on heterogeneous multiprocessor architectures. Mihal has a BS in electrical and computer engineering from Carnegie Mellon University.



**Chidamber Kulkarni** is a postdoctoral fellow at the University of California, Berkeley. His research interests include tools and methods for cost-efficient embedded-system implementations. Kulkarni has a PhD in electrical engineering from Katholieke Universiteit Leuven, Belgium.



**Matthew Moskewicz** is pursuing a PhD in electrical engineering at the University of California, Berkeley. His research interests include satisfiability, formal modeling and languages for system design, and constraint-based programming. Moskewicz has a BSE in electrical engineering from Princeton University.



**Mel Tsai** is pursuing a PhD in electrical engineering at the University of California, Berkeley. His research interests include computer architecture, network processors, and performance evaluation. Tsai has a BS in electrical engineering from Michigan State University.

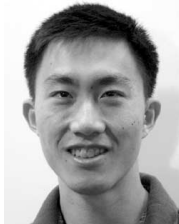


**Niraj Shah** is a PhD candidate in the electrical engineering and computer science department at the University of California, Berkeley. His research interests include tools and methodologies for ASIP design. Shah has an MS in electrical engineering from the University of California, Berkeley.



**Scott Weber** is pursuing a PhD in electrical engineering at the University of California, Berkeley. His research interests include tools and methods for architecture design.

and implementation. Weber has a BS in electrical and computer engineering and a BS in computer science, both from Carnegie Mellon University.



**Yujia Jin** is pursuing a PhD in electrical engineering at the University of California, Berkeley. His research interests include computer architecture with emphasis on memory architecture. Jin has a BS in electrical engineering and computer science from the University of California, Berkeley.



**Kurt Keutzer** is a professor of electrical engineering and computer science at the University of California, Berkeley, where he also serves as associate director of the Marco-funded Gigascale Silicon Research Center. His research interests range widely over computer-aided design of ICs, but today are principally focused on the development of architectures and programming environments for embedded applications, and the development of low-power and high-performance design techniques for ICs. Keutzer has a PhD in computer science from Indiana University.



**Christian Sauer** is a research engineer at Infineon Technologies Corporate Research in Munich, and is a visiting industrial fellow in the Electrical Engineering and Computer Science Department at the University

of California, Berkeley. His research interests include tools and methodologies for application-specific systems and related applications, and ASIP architectures. Sauer has a master's degree in electrical engineering from Technical University of Dresden, Germany.



**Kees Vissers** is the chief technology officer at Chameleon Systems and a part-time research fellow at the University of California, Berkeley. His research interests include reconfigurable and programmable systems, and the systematic design of systems using the Y-chart methodology. Vissers has a master's degree in electrical engineering from Delft University, the Netherlands.



**Sharad Malik** is a professor in the Department of Electrical Engineering at Princeton University. His research interests include design tools for embedded computer systems, and synthesis and verification of digital systems. Malik has a PhD in computer science from the University of California, Berkeley.

■ Direct questions and comments about this article to Chidamber Kulkarni, 545s Cory Hall #1770, Univ. of California, Berkeley, CA 94720; kulkarni@ic.eecs.berkeley.edu.

**For further information on this or any other computing topic, visit our Digital Library at <http://computer.org/publications/dlib>.**