A Disciplined Approach to the Development of Platform Architectures

D. I. August[†], K. Keutzer[‡], S. Malik[†], A. R. Newton[‡]

{daugust, malik}@princeton.edu; {keutzer, rnewton}@eecs.berkeley.edu

ABSTRACT

Silicon capability has enabled the embedding of an entire system on a single silicon die. These devices are known as systems-on-a-chip (SOC). Currently, the design of these devices is undisciplined, expensive, and risky. One way of amortizing the cost and ameliorating this design risk is to make a single integrated circuit serve multiple applications, and the natural way of enabling this is through end-user programmability. The aim of the MESCAL project, which is the subject of this paper, is to introduce a disciplined approach to producing reusable architectural platforms that can be easily programmed to meet a variety of applications. (MESCAL stands for Modern Embedded Systems, Compilers, Architectures, and Languages.)

1. INTRODUCTION

While Moore's Law has enabled us to increase the complexity of integrated circuits at an exponential rate, this increase in complexity has provided tough challenges to the design and design automation communities. The design of individual devices and associated interconnect is becoming harder due to deep sub-micron effects resulting in increasing interconnect delay and coupling. At the same time there are an exponentially more number of devices to deal with. The situation is further exacerbated by the need to integrate heterogeneous elements – digital, analog and mixed signal, RF, and software – on the same piece of silicon. Finally, all of this comes with competitive pressures to further reduce the time to market. This results in a quadruple whammy for designers and has resulted in the well-publicized gap between manufacturing capability and design productivity.

In addition to the intellectual design challenges, there are significant economic challenges associated with non-recurring engineering costs associated with manufacturing. The International Technology Roadmap for Semiconductors predicts that while manufacturing complex System-on-Chip designs will be practical, at least down to 50nm minimum feature sizes, the production of practical masks and exposure systems will likely be a major bottleneck for the development of such chips. That is, the cost of masks will grow even more rapidly for these fine geometries, adding even more to the up-front NRE for a new design. Reports indicate a single mask set and probe card cost for a state-of-the-art chip is over \$0.5M for a complex part today, up from less than \$100K a decade ago (note: this does not include the design cost). At 0.15µm technology SEMATECH estimates we will be entering the regime of the "million dollar mask set."

One way of amortizing design cost and ameliorating design risk is to make a single integrated circuit serve multiple applications, and the natural way of enabling this is through end-user programmability. The aim of the MESCAL project, which is the subject of this paper, is to introduce a disciplined approach to producing reusable architectural platforms that can be easily programmed to meet a variety of applications. (MESCAL stands for Modern Embedded Systems, Compilers, Architectures, and Languages.) Achieving this goal requires coordinating research in applications, software environments, and application specific instruction processor (ASIP) design. We are currently focusing on networking applications. We aim to enable significant design-space exploration by means of a highly automated environment for generating compilers and simulators from a high-level architectural description. Our target architectures are multiprocessor networks consisting of specialized VLIW processing elements. Our approach puts equal emphasis on concurrency in the communication network and on the effectiveness of the individual processing element. We note that our target applications are highly concurrent and so are our target integrated circuits. The goal then is to easily capture the process-level, instruction-level, and bit-level concurrency of the application, and map it naturally onto the target device.

In this introduction, we have motivated the use of programmable architectural platforms. In Section 2, we further motivate the development of a disciplined approach to the design of these platforms by examining current approaches to platform-architecture development and by showing the deficiencies of these approaches in Section 3. Section 4 lists fundamental principles of platform architecture development, and the elaboration of these principles constitute the remainder of the paper.

2. CURRENT APPROACHES TO PLATFORM-ARCHITECTURE DEVELOPMENT

This section briefly surveys the architectural diversity of current programmable platform architectures and the approaches to their development.

2.1 Diversity of Current Architectures

Parallelism and concurrency. Current systems on a chip support parallelism at a variety of levels. It is the increasing trend that SOCs have multiple processing elements on-chip. Common combinations are to have a RISC microprocessor for control functions and a Digital Signal Processor (DSP) for communication functions. Enabled by Moore's Law, there is a trend toward even higher degrees of parallelism and already chips with 100's of individual processing elements have been fabricated. As each processing element may support multiple computational processes there is the potential for very high levels of *process-level parallelism* on each chip.

[†] Princeton University

[‡] University of California, Berkeley

Because of their superior power/energy efficiency relative to superscalar processors, very large instruction word (VLIW) processors are gaining in popularity for embedded applications. VLIW processors allow *instruction-level parallelism* (ILP) to be computed by the compiler rather than generated dynamically.

Some of the new architectural platforms also utilize reconfigurable computation fabrics, similar to Field-Programmable Gate Arrays (FPGA) on chip. These fabrics provide the opportunity for performing numerous bit-level operations in parallel. Although it comes with some cost in circuit area and performance, these reconfigurable fabrics offer *bit-level parallelism* in a user-programmable way.

Special Purpose Hardware. Modern platforms may have a variety of special purpose execution units tailored to their target application. For example, contemporary network processors may have built-in hardware for performing hash-table search. There may also be special purpose hardware for hiding memory latency.

On-chip Communication Networks. Current platforms must utilize a variety of different mechanisms to efficiently interconnect processing elements to memory, to pins, and to each other. The complexity of this is further complicated by the heterogeneity of the processing elements themselves.

2.2 DEVELOPMENT OF PLATFORM ARCHITECTURES

Simulation. Approaches to the development of platform architectures vary widely. We do not attempt to characterize best practices here. Instead, we simply report the most common practices. At the heart of architectural development is the development of the simulator. Although simulator-generator environments exist [1] the most common practice is to ``hand-craft'' the simulator from C-code for each instance of the architecture. This approach means that the development cost for exploring an architectural variant is high.

Compilation. There is limited use of automatically retargetable compiler components for platform architectures. As a result evaluation of benchmarks is principally limited to assembly coding. Benchmarks must be hand-coded in the assembly language of all new architectures. It can then be run on the hand-crafted simulator to evaluate performance. The cost in developing the benchmark, the assembler, and the simulator limits design space exploration.

2.3 WHY SO LITTLE TOOL SUPPORT?

Given the growing reliance on programmable solutions for embedded systems it is interesting to reflect as to why there is not more tool support for designing these platforms. The primary reason is that the number of new processor development projects significantly lags that of ASIC development projects. A company aiming to provide tools for integrated circuit development would have a larger market opportunity by providing tools for traditional IC design problems such as static timing analysis or equivalence checking.

3. DEFICIENCIES OF CURRENT APPROACHES TO THE DEVELOPMENT OF PLATFORM ARCHITECTURES

The *ad hoc* approaches currently employed in platform architecture design lead to a number of recurrent problems. The first problem is that because of the high cost of design-space exploration, resulting architectures may be over-tuned to a few application benchmarks, and poorly suited to others. The resulting architecture may be quite inefficient on the general application mix.

Because the development of the compiler and the software development environment lags that of the architecture, the resulting architecture may be poorly amenable to automated compilation techniques. The same computational idioms that made the architecture run efficiently on the small computation kernels coded in assembler may be impossible to capture in automated compilation.

While some current architectures have done better at both design space exploration [2] and supplying software development environments for their resulting processor (for example in Tensilica's Xtensa processor [3]), the deficiencies described here are very common.

4. KEY PRINCIPLES OF THE DISCIPLINE OF PLATFORM ARCHITECTURE DEVELOPMENT

Our goal is to develop and to further codify best practices in architectural platform development. In this section, we introduce a few of the key principles that guide our approach to architectural platform development.

- Modeling architectures using executable specifications that are formally analyzable.

The first description of a particular architecture may be as semantically meaningless as figures on a napkin. Our aim is first to give a flexible graphical framework for the description of a multiprocessor architecture such that as soon as the architectural specification is entered a simulation of that architecture is available. Secondly, we aim to not just support, but also require, that the concurrent operation of the architecture be described in a formal manner using models of computation in Ptolemy II [4]. Section 5 describes this process.

- Concurrent-development of hardware and software within an environment that supports multiple views (software development, simulation, architecture, hardware) of the architectural platform.

Architectural simulators used for performance analysis, analysis tools used for software development, and compilers require their own description of the architecture. Thus, there are two challenges: The first is to develop architectural descriptions to support each of the tools. The second challenge is to ensure that these views are consistent. In MESCAL, we obviate these problems by maintaining a single architectural database from which the views required by the various tools are automatically generated. We give details in Section 6.

- Enabling disciplined design-space exploration through descriptive benchmarks and automatically generated compilers and simulators.

There are a number of elements to exploring a design space associated with multiprocessors architectures in a disciplined manner. The first of these is the characterization of the design space itself. This entails elaborating the degrees of architectural freedom. The second is carefully identifying a set of computational kernels and benchmarks over which the architectures are to be evaluated. This is further elaborated in Section 5.

- Exportation of programmer's model to aid in platform exportation.

Efficient programming of an architecture platform currently requires understanding the salient features of the underlying architecture. In the MESCAL approach, the software compiler is automatically retargeted as the architecture is developed; however, this does not mean that the best approaches for programming the architecture will immediately be evident. Our aim is to augment the platform architecture and its software development environment, with a programming model that conveys to an application programmer how to get the best performance from the platform architecture. Section 8 motivates the need of a programmer's model.

5. ARCHITECTURE EXPLORATION

The goal of this part of the project is to provide an environment for the efficient exploration of concurrent architectures by means of a flexible interface provided with the heterogeneous simulation environment known as Ptolemy II [4]. Concurrency in our architecture is provided at multiple levels: at the bit level through specialized functional units, at the instruction level through VLIW processors, at the thread level through multi-threading and at the process level through multiple processors. The exploration environment enables the designer to specify a particular micro-architecture and architecture, and automatically export an interface to these that is used for the retargetable synthesis of the software environment (simulators, compilers and custom run-time systems). In addition to the specification of the processing elements (PEs), the environment provides for the specification of the communication between the PEs. The communication specification provides for flexibility in the physical network – topology as well as switching type (circuit or packet), as well as flexibility in the protocol for the network usage.

5.1 Defining the Space for Architectural Exploration

Before an architectural design space can be explored, it must first be defined. If the initial definition of the search space is incomplete then *all* resulting architectures will be inadequate or incomplete because they will have failed to consider some architectural possibilities. In MESCAL, we have begun with a thorough categorization of existing network processor architectures, functional units, and co-processors [5].

5.2 Identifying Representative Benchmarks

A benchmark is simply a standard by which something can be measured or judged. As the set of candidate architectures will be measured over and over relative to their performance on the set of candidate benchmarks, the choice of candidate benchmarks is very important. In MESCAL, our target platform architectures are network processors and our primary goal in developing network processor benchmarks is to compare network processors in a quantitative way. The secondary goals of these benchmarks are to provide insight into the network processor's expected real-world application performance and to highlight the salient architectural features that are most useful for network processing applications. Our efforts on benchmarks and techniques for network processor comparison are currently being written up in [6].

5.3 Single Processing Element Environment

A designer creates a processing element in the MESCAL environment by creating a back-of-the-envelope schematic of the structure of the micro-architecture. A graphical interface based on the tools Diva and Swing allows the user to drag and drop visual components and connect ports of the components with edges. The semantics of the schematic are created through the use of Ptolemy II and Vergil [4]. Specifically, processing elements designed in the MESCAL environment utilize a new Ptolemy II domain that implements multiplexed cycle based static dataflow semantics with cycle precision stalls.

Spatial control is defined by creating instructions and their associated resource usage sets. After constructing a schematic of the model in the architecture view, the designer enters the *instruction-set architecture* (ISA) view of the model. In the ISA view, the designer names an instruction and clicks on the resources that the instruction uses. Automatic inference of the resource usage provides syntax directed editing of the spatial control. This greatly improves the productivity of the designer. Cycle based static dataflow semantics allow reservation tables to then be extracted by the use of static timing analysis based on the resource usage sets and structure of the model. The designer does not have to insert retiming and pipeline registers; they are implicit. Consistency is automatically maintained from the architecture view to the ISA view so that structural and timing changes are reflected in the reservation tables. The model is simulated using multiplexed cycle based static dataflow semantics by simulating the resources in a topological order on the cycles specified in the reservation table for a given instruction. On each cycle, a new instruction is fetched and the appropriate schedule is queued; this allows for an efficient simulation of a pipelined machine. A compiler uses the extracted reservation tables to create a valid schedule.

Temporal control is defined by choosing the appropriate hierarchical splits in the model. Splits occur on boundaries where control signals that affect the temporal flow of data cross. A processing element model has two domain input ports, an instruction and a stall signal. The instruction signal's purpose determines the spatial control. The stall signal determines the temporal control of a given spatial configuration. Stall signals cause the pipeline to interlock on the specified pipe stage. Designers need not model the pipeline interlocking since it is implicit to the domain. This allows models to more easily incorporate interrupts, exceptions, and dynamic control. The hierarchical splits usually occur on boundaries that separate the control plane from the data plane. The data plane is described using multiplexed cycle based static dataflow semantics. A number of different semantics may be used to describe the control plane as long as the appropriate instruction and stall signals are generated.

5.4 On-chip Communication Architectures for Multiprocessors

As mentioned above, silicon resources available today easily permit us to consider integrated circuits consisting of multiple processors on a single die. As applications traditionally have significant concurrency in them, now with significant concurrency available in the platforms, we can more efficiently map the application concurrency onto the architectural concurrency of the platform.

A key element of the methodology to enable this is to consider the on-chip communication architecture as a first class element of the architectural exploration environment, rather than as an afterthought. There is a wide diversity of possible communication architectures available, and selection of an appropriate architecture can impact all metrics such as performance, power, etc. just as much, if not more, as the computational architecture.

In the space of computational architectures, much effort has gone into classifying and modelling of different forms of architectures (RISC/CISC, VLIW/Superscalar, SIMD/MIMD) and elements of microarchitectures (pipelines, memory hierarchies, accelerators such as branch predictors etc.). However, while individual communication architectures have been modelled, there is little systematic classification of on-chip communication architectures or of the micro-architectural primitives that go into constructing such architectures. As part of the MESCAL effort to build an architectural exploration environment, we are developing a general modelling infrastructure for on-chip communication architectures. As part of this infrastructure, we have developed a class hierarchy of on-chip communication architectures as shown in Figure 1. The key idea here is that the entire family of possible communication architectures can be organized as a tree with specific instances at the leaves. A particular node in the tree inherits all the properties/characteristics of its ancestors and can add some of its own that are passed on to its descendents. The properties/characteristics are captured in a parametric executable behavioural model as part of the Ptolemy II [4] environment. As shown in the figure, this hierarchy is quite diverse, covering the space from conventional buses and circuit switching to more recently emerging packet switching architectures.

The usage of this class hierarchy in architectural exploration is as follows. If the specific communication architecture to be explored is already present as a leaf, then its model is already available and this can be parameterised and used in architectural evaluation for simulation. If a new communication architecture is being developed, then its nearest ancestor is located in the hierarchy and customized by adding on specific characteristics. This is then added back to the hierarchy for potential reuse in the future. This methodology provides for an environment that provides rapid development of reusable customisable executable models, as well as a classification of the space of available architectures. Using this hierarchy, we were able to develop a detailed executable model for the AMBA [7] bus with relatively little additional effort once we had a model for the CoreConnect Bus [8] in place. With these specific well known instances of bus architectures now available as part of the architectural environment, any time a bus architecture is desired, either of these instances can be rapidly evaluated, or possibly a new custom bus architecture rapidly designed using the existing models as a base, and then evaluated.

We see this hierarchy as being a key element of exploration of the combined computation and communication architectural space.

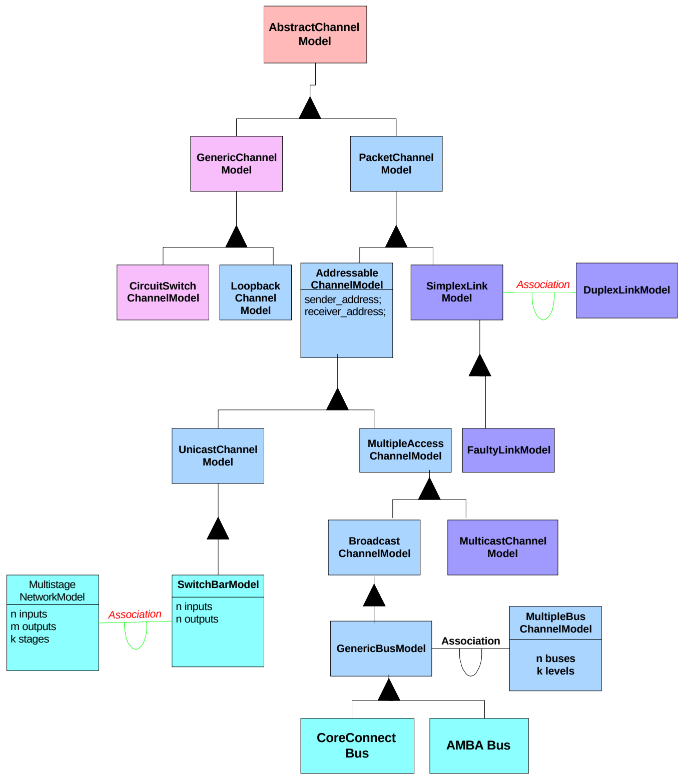

Figure 1: Class Hierarchy of On-chip Communication Architectures

6. ARCHITECTURAL PLATFORM VIEWS

6.1 The Need for Multiple Views

The architectural exploration environment is geared towards enabling the architect to rapidly assembly a diverse range of instruction set architectures and microarchitectures for the processing elements, as well as the communication architectures. Once a particular architecture has been assembled, it is likely to be subject to evaluation and potential use in the future. This requires that the architectural details be exported outside of this environment. The exact information that needs to be exported is a function of the intended use. For example, an instruction set simulator generator will need to know all the details of the microarchitecture, including the execution semantics of each microarchitectural component, as well as the instruction set architecture, along with a precise description of exactly how each instruction is executed. On the other hand, a compiler generator needs to know the semantics of each instruction, along with “well summarized” microarchitectural information regarding the usage, including constraints, of microarchitectural elements during instruction execution. By “well summarized” we mean the information is in directly usable form, such as a reservation table for VLIW processors, and does not require very complex analysis to extract.

This diversity of types of exported information makes it clear that there is no single uniform architectural model that can be used for all the intended functions. Instead, we must support multiple views, each of which is geared towards supporting a specific function. Currently MESCAL supports an ISA view, which exports the instructions and their semantics, a simulator view for the simulator generator, a compiler view for the compiler generator and a memory view, which describes the memory hierarchy. The views are possibly overlapping in as much as the same information may be available in the same or possibly different form in more than one view. A key requirement on the generation of these views is that they guarantee consistency among the different views. The above set of views is not intended to be complete; instead the environment is expected to be flexible enough to permit the generation of additional views that may be required in the future.

6.2 MESCAL Architectural Description (MAD)

The MESCAL Architecture Description is a compiler and simulator view that is generated from the architectural environment and serves as a target description for a retargetable simulator and compiler. The architecture description describes both the computation as well as the communication architecture. The description itself is in XML, which makes it flexible, and enables leverage of standard XML tools.

The computational architecture is specified by providing the ISA, and the microarchitectural description required by the simulator and compiler. It combines a behavioral and a structural model, as well as the mapping of the former onto the latter, for this purpose.

The *behavioral model* provides details of the instruction set architecture, and describes the operations, opcodes, encodings, operands, various categories of operations, and the semantics of the operations. The *structural model* provides details of the microarchitectural components such as decoders, pipeline stages as well as their interconnection. The interface between the behavioral and the structural model is key in describing the physical execution of instructions. This describes the mapping of operands to functional units, as well as operations to pipeline stages. A key feature of MAD is system support capability. This includes the support for privileged instructions, the ability to model interrupts, and the ability to support specialized functional units such as communication assists - a critical element of this methodology in dealing with communication between distributed computation.

The communication architecture again has a behavioral model and a structural model associated with it. The structural model is a netlist of the components of the communication architecture. The behavioral model captures the semantics of the exact communication occurring between these. This is accomplished by using the same parametric description of communication architectures as shown in Figure 1. An instance of a communication architecture is described by specifying a leaf in the tree, with specific parameters for each of the nodes on path from the root of the tree of the leaf. The semantics of each of the nodes on this path describe the semantics of the communication architecture. This can be used by the simulator and compiler generators. The parameters quantify various metrics such as bandwidth, throughput, latency and power of basic operations, buffer sizes etc.

There are several competing architectural specification languages available today such as nML [9], ISDL [10], Maril [11], HMDES [12], LISA [1], PRMDL [13], RADL [14] and Expression [15]. We have done a detailed comparison of the capabilities of MAD with these forms of specification. None of them provide either system support or the ability to specify communication architectures as MAD does. In terms of support for describing basic processor elements, LISA and Expression come the closest to MAD. In comparison with LISA, MAD has a higher level of abstraction, and thus more compact specification. It also directly provides support for VLIW compilers in terms of clear description of instruction level parallelism. MAD and Expression are similar in their goals of providing support for simulators and compilers for VLIW processors. However, MAD has less redundancy in the description and thus fewer issues of consistency in description.

6.3 Liberty Simulator Specification

There are two steps to generating a simulator. First, a view generator converts a MAD Description into a Liberty Simulator Specification (LSS). Then, using this specification, the Liberty Simulation Environment (LSE) constructs a simulator by instantiating microarchitectural *modules* and connecting them with instantiated *channels*. A *module* is a microarchitectural component template, which interacts with other parts of the simulator via input and output *ports*. A *channel* is a communication template, which connects one or more module ports. In general, modules describe significant functionality such as caches, branch predictors, and functional units while channels describe timing interactions using notions such as wire, queue, and filter. The LSS directs the instantiation and connection of the module and channel templates.

The Liberty Simulator Specification serves two purposes. First, LSS is a Liberty simulator view of the architecture - LSS represents the microarchitecture in a manner conducive to simulator generation. Second, LSS allows the designer to directly manipulate the microarchitecture to be modeled. This access is necessary for modeling irregularities introduced by VLSI constraints or other details not describable with MAD. Unlike MAD, LSS approximates the computer architect's view of the microarchitecture - modules and channels generally have a physical counterpart in the hardware. The relative ease of manipulating the LSS makes customizing a simulator to match a candidate microarchitecture an efficient process for the microarchitect. Like MAD, LSS is stored as an XML ASCII description, but typically a graphical visualizer is the preferred user interface.

A complete Liberty Simulator Specification consists of four parts: module instantiations to create architectural functions, channel instantiations to create connections, control points to specify complex or non-local control not implied by channel connections, and event instantiations to provide the user or compiler with performance feedback.

Module Instantiation. To create an architectural component in the simulator, a module instantiation is indicated in LSS using the instantiation directive, `INST`. In Figure 2, an instruction fetch unit `ifetch1` is created from the `ifetch` template. Each instantiated module can have its behavior customized via parameters. Module instantiation tools use these parameters at simulator construction time to create code appropriate for the specific case. All module types define a default set of parameters to use if a parameter is not specified in LSS. Here the default `ifetch` parameter `decode_latency` is overridden in `ifetch1` to have a value of 1 cycle.

Channel Instantiation and Connection. Channels are instantiated much like modules - they have parameters and are derived from a channel type template. In Figure 2, a queue named `pred_chan` is created between the instruction fetch (`fetch1`) `lookup` port and the branch predictor (`bpred1`) `predict` port to initiate branch predictions at the appropriate time. In this example, the channel uses the default type parameters. Data passing through channels has the type specified by `datatype`. Here, the channel only relays a trigger signal without associated data.

```
<INST name="ifetch1" type="ifetch">
  <PARAMETERS>
    <PARM name="decode_latency" value="1"/>
  </PARAMETERS>
  <CONTROLFUNC name="decide_to_fetch">
    if(QUERY_CALL(rename1, free_regs) < 8)
      return SIM_decision_no;
    return SIM_decision_yes;
  </CONTROLFUNC>
</INST>
<CHANNEL name="pred_chan" datatype="none"
         type="queue" model="q">
  <CONNECT inst="ifetch1" name="lookup"/>
  <CONNECT inst="bpred1" name="predict"/>
</CHANNEL>
```

Figure 2: Portion of a Liberty Simulator Specification.

Control Points and Exported State. Channel connections imply local and regular control information. For example, instruction fetch initiates branch predictions with the `pred_chan` channel. Unfortunately, channels cannot express all control possible in real machines, so a control point mechanism exists to express this non-local or irregular control logic. In Figure 2, the control function `decide_to_fetch` stops instruction fetch whenever the rename logic has fewer than 8 free registers. To accomplish this, the control function uses the `free_regs` exported state function from the `rename1` instantiated module in making a fetch decision.

Event Points. Since the type of information necessary from the simulator is application specific, LSS allows the inclusion of code at various event points. The code at an event point executes whenever a particular event has occurred. The user specified code at these event points can perform a variety of tasks ranging from collecting simple statistics to driving a visualizer or debugger.

7. PLATFORM ARCHITECTURE EVALUATION ENVIRONMENT

Both a compiler and a simulator are necessary to evaluate a candidate platform. However, to influence the design process in a meaningful manner the time and resources put into their design and construction must be moved off the processor design critical path. Therefore, the compiler and simulator must be automatically generated.

7.1 Compiler Development

Platform architectures are only truly programmable when they include an optimizing compiler. The performance of a platform architecture is a combination of the quality of the code generated by the compiler and the efficiency of the hardware while executing that code. The only meaningful measure of the fitness of a particular platform architecture is one which includes the entire system, compiler and hardware. Naturally, the most advanced architectural mechanisms are of little worth if compiler technology is not able to exploit them. Disciplined development of platform architectures requires the consideration of many candidate compiler/hardware pairs. Therefore, an *automatically* retargeted compiler is a requirement.

While many compilers in the past have been retargetable [16, 17, 18], they often do not have automatic retargeting of machine specific optimizations, beyond instruction selection, scheduling and register allocation. To make things worse, most processors (DSPs, network processors, etc.) require machine specific optimizations [19, 20]. In fact, programmers often code for some of these devices in assembly language since compilers, when they exist, do not regularly generate performance-boosting code [21]. This is not an acceptable solution for programmable platforms.

The MESCAL compiler and MAD together provide a solution to the problem of creating an automatically retargetable compiler. The MESCAL compiler uses the MAD description to guide instruction selection, optimization, scheduling, and register allocation. Since the MESCAL compiler makes no assumptions about the target machine not explicitly described in MAD or implied by MAD's domain, automatic retargetability becomes tractable. The key to success is to extend current compiler technology rather than restrict MAD's domain.

Consider, for example, the conflicting requirements that instruction set architects and compiler writers often have for the design of the instruction set. The architects are driven by micro-architecture complexity and code size considerations; while the compiler writers prefer clean orthogonal instruction sets amenable to regular compiler algorithms. For a variety of reasons, special purpose processors often have very irregular architectures that are very hard to compile for and that need specialized optimization techniques. Instead of avoiding these irregular architectures or writing a customized optimizer for each one, the MESCAL compiler and MAD generalizes these architectures to create retargetable compiler support for them.

Traditional compilers work directly with physical resources to determine what can and cannot be scheduled in parallel. It is desirable to leverage this large body of work for the MESCAL compiler. Using a MAD view, irregularities in the ISA can be matched with the regular resource-based requirements of classic compilers by generating a set of artificial resources from the ISA specification. The artificial resources are generated by solving a combinatorial graph labeling problem, which is generated from the MAD ISA specification. These resources may not correspond to any real physical resources, but are equivalent in as much as they specify the instructions which can be executed in simultaneously in a manner equivalent to the real resource specification [22].

The MESCAL compiler contains other generalizations which enhance the domain of automatically retargetable compilers. By designing the compiler in this way, compiler writers can create compilers for architectures *before* they are specified, moving their efforts off the critical path, and enabling meaningful design of platform architectures.

7.2 Simulator Development

A requirement for disciplined development of platform architectures is a retargetable, fast, and precise simulation tool. Most simulators in the past have not been retargetable, have supported only a limited class of microarchitectures, or have abstracted away important design details [23, 24]. Many have not accurately modeled control effects due to this abstraction. These inaccuracies can lead designers to make incorrect design decisions [24]. The Liberty Simulation Environment (LSE) (`http://liberty.princeton.edu`) addresses these requirements.

Automatic Retargetability. A Liberty simulator is automatically created by the simulator builder from a Liberty Simulator Specification (LSS), modules from a module library, and channels from a channel library. Of these, only the LSS creation must be on the design critical path. Simulator writers should spend their time creating module and channel libraries *prior* to the start of platform architecture exploration. The modules and channels they create should be flexible enough to be used in a wide class of architectures.

Since the same module or channel may be reused in many vastly different contexts, it is unlikely that an efficient and generic module could be written once in a high-level language. Doing so would forgo the opportunity to optimize the module for each particular parameter set. Consequently, LSE uses a statically instantiated object system to create specialized components. The process is akin to object instantiation in JAVA, except that it is done statically to allow host compiler optimization, and it is done with simulation specific knowledge to allow algorithmic-level optimizations not achievable any other way.

Fast Simulation. To explore many points of the design space, the simulator must be fast while remaining feature rich. Static instantiation of the modules and channels allows the code to be host compiler optimized for each particular case. The simulator construction environment is built so that one only pays for the features that are used. For example, if an architect is not yet interested in modeling the effects of various types of speculative update of branch predictor state, the branch predictor does not waste time updating its state speculatively. Also, constructed simulators need not be complete. For example, if one is only interested in the pipeline, the remainder of the machine need not be instantiated. Semantics for unconnected module ports ensure proper simulator operation.

LSE also takes a different approach to instruction decode. Microprocessor simulators often perform instruction decode dynamically, an unnecessary task. This is not to say that one does not need to know which instruction is being executed, but that all information associated with an instruction can be statically pre-decoded. Liberty implements this technique by statically decoding instructions and generating code for evaluating them (this is known as compiled-code simulation). Note, however, that Liberty's novel approach allows it to simulate, in a cycle accurate fashion, *dynamic* effects such as mis-speculation despite the use of compiled-code simulation. Using compiled-code simulation precludes the use of self-modifying code, but gives huge performance gains - generally two orders of magnitude when compared to an equivalent dynamically decoded functional simulator. A dynamic decoder is available to handle self-modifying codes when they are encountered. Using compiled-code simulation, functional simulation with a Liberty simulator is typically only one to four times slower than native hardware.

Precise Simulation. With the LSE, we can precisely model non-local and irregular complex control, including those stemming from VLSI considerations without a need to understand or directly modify simulator code. Module writers must insert place holders for certain control decisions in their code. As shown earlier, a user can specify the control logic in the LSS using control functions. These control functions can create their own state or query state exported by other module instances.

7.3 Other tools

While performance simulators and compilers are critical elements in the evaluation loop of architectural platforms, there are other tools that can also be significant.

Increasingly, power is emerging as possibly more critical than performance as a design metric. Thus, we need to provide support for power evaluation and optimization at the same level as performance. We have done this in the Liberty simulator by building a power simulator in parallel with the performance simulator. It shares the same design as the performance simulator, differing only by using power models instead of performance models. As with the performance simulator, this is completely retargetable.

Several platform applications have real-time deadlines. Guaranteeing hard real-time deadlines is not possible using simulation. We have integrated a retargetable static timing analysis engine as part of the MESCAL framework for this purpose [25]. The analysis engine uses our previously developed algorithms based on integer linear programming for implicit path analysis, and static cache modeling techniques.

Debuggers and performance visualization tools are very important for enhancing designer productivity; however, thus far, we have been unable to devote any resources to this area.

8. EXPORTING THE ARCHITECTURAL PLATFORM THROUGH A PROGRAMMER'S MODEL

MESCAL is aimed at platform architectures for embedded-system applications. These architectures are certain to have high-level process concurrency, operator/instruction level parallelism, bit-level parallelism, and multiple application-specific execution units. Our goal in MESCAL is to create easy to understand programmer's models of the target architectures that enable application developers to get as close to assembly-language programming quality as possible.

We use as inspiration the development of the C-language as a programmer's model for minicomputers and workstations. Although the C-compiler technology for the C-language was relatively primitive, the C-language became *the* standard for high performance programming of minicomputers and workstations. We believe that this was due to a judicious choice of the key assembly language features (pointer arithmetic, bit-manipulation, register keywords) to make programmer visible in the C-language. Similarly, our aim is to determine the 20% of the architectural and microarchitectural features that allow for 80% of the architecture's performance. Initially we are focusing on network processors and looking for constructs to make easily visible their features such as zero-overhead context switching, queue management, and built-in primitives such as hash-table lookup.

9. SUMMARY, CONCLUSIONS

Current approaches to IC design are becoming unmanageable. The development of programmable platforms provides an attractive way to minimize design risk and cost. There is currently little discipline to the development of programmable platforms. This is the gap that the MESCAL project aims to fill. We also have considerable effort focused on mapping applications onto programmable platforms, but space limitations in this paper have only allowed us to present the broadest details of our current research efforts on platform architecture development. More details and reports of progress on MESCAL can be learned from our website at www.gigascale.org/mescal.

10. ACKNOWLEDGMENTS

In addition to the faculty-authors the MESCAL project consists of a number of graduate students, post-doctoral fellows, and industrial visitors. Industrial visitors are Christian Sauer of Infineon and Kees Vissers of Trimedia. Chidamber Kulkarni, formerly of IMEC, is our post-doctoral fellow. Berkeley graduate students include Yujia Jin, Andrew Mihal, Matthew Moskewicz, Niraj Shah, Melvin Tsai, and Scott Weber (author of section 5.3). Princeton graduate students include Zhining Huang, David Penry, Wei Qin, Subbu Rajagopalan, Spyros Triantafyllis, Manish Vachharajani, Shaojie Wang and Hangsheng Wang. Discussions with Jan Rabaey, Alberto Sangiovanni-Vincentelli, and Kees Vissers have helped shape our platform design vision.

11. REFERENCES

[1] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, Heinrich Meyr, "LISA — Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures," In *Proceedings of Design Automation Conference*, June 1999, New Orleans.

[2] G.J. Hekstra, D.D. La Hei, P. Bingley, F.W. Sijstermans, "TriMedia design space exploration," In *Proceedings of ICCD 1999*, Austin, Texas, pp 599-606.

[3] Tensilica, "The Xtensa Processor Generator", http://www.tensilica.com/technology.html.

[4] E. A. Lee, "Overview of the Ptolemy Project," *Technical Memorandum UCB/ERL M01/11,* University of California, Berkeley, March 6, 2001.

[5] N. Shah, "Understanding Network Processors," M. S. Thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, August 2001.

[6] M. Tsai, "Methodologies and Techniques for Network Processor Comparison," M. S. Thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, (in preparation).

[7] AMBA On-Chip Bus Rev 2.0 Specification, ARM Ltd, 2000.

[8] CoreConnect Bus Architecture, White Paper, IBM Corp, 1999.

[9] A. Fauth, J. Van Praet, and M. Freericks, "Describing instructions set processors using nML," In *Proceedings of European Design and Test Conference*, Paris (France), March 1995, pp. 503--507.

[10] G. Hadjiyiannis, S. Hanono, and S. Devadas, "ISDL: An instruction set description language for retargetability," In *Proceedings of Design Automation Conference*, May 1997, Anaheim, CA.

[11] David G. Bradlee, Robert R. Henry and Susan J. Eggers. "The Marion System for Retargetable Instruction Scheduling," In *Proceedings of the Conference on Programming Language Design and Implementation*, June, 1991, Toronto Canada.

[12] J. C. Gyllenhaal, B. R. Rau, and W. W. Hwu, "HMDES Version 2.0 Specification," Technical Report IMPACT-96-3, The IMPACT Research Group, University of Illinois, Urbana, IL, 1996.

[13] A.S. Terechko, E.J.D. Pol and J.T.J. van Eijndhoven, "PRMDL: A Machine Description Language for Clustered VLIW Architectures," In *Proceedings of European Design and Test Conference*, March, 2001, Munich, Germany.

[14] Chuck Siska, "A Processor Description Language Supporting Retargetable Multi-Pipeline DSP program Development Tools," In *Proceedings of the 11th International Symposium on System Synthesis*, December, 1998, Taiwan China.

[15] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, A. Nicolau, "Expression: A Language for Architecture Exploration through Compiler/Simulator Retargetability," In *Proceedings of Design Automation and Test in Europe*, 1999, Munich, Germany.

[16] S. Hanono and S. Devadas, "Instruction selection, resource allocation, and scheduling in the AVIV retargetable code generator," In *Proceedings of the 35th Design Automation Conference*, June 1998.

[17] S. Rajagopalan, S. P. Rajan, G. Araujo, S. Rigo, and S. Malik, "Using the IMPACT VLIW compiler framework to implement a compiler for a fixed point DSP," In *Proceedings of the 5th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, March 2001.

[18] D. Lanneer, J. V. Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens, "CHESS: Retargetable code generation for embedded DSP processors," In, *Code generation for embedded processors*, pp. 85-102, Boston, MA: Kluwer Academic Publishers, 1995.

[19] S. Liao, *Code Generation and Optimization for Embedded Digital Signal Processors*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 1996.

[20] Intel Corporation, *IA-64 Application Developer's Architecture Guide*, May 1999.

[21] E. Stotzer, B. Huber, R. Tatge, and A. Ward, "Programming a VLIW DSP in assembly language," In *Proceedings of the 2nd International Workshop on Compiler and Architecture Support for Embedded Systems,* October 1999.

[22] S. Rajagopalan, M. Vachharajani and S. Malik, "Handling Irregular ILP Within Conventional VLIW Schedulers Using Artificial Resource Constraints," In Proceedings of CASES 2000, November 2000.

[23] B. Black and J. P. Shen, "Calibration of microprocessor performance models," *IEEE Computer*, vol. 31, pp. 59-65, May 1998.

[24] R. Desikan, D. Burger, and S. W. Keckler, "Measuring exprimental error in microprocessor simulation," *Proceedings of the 28th International Symposium on Computer Architecture*, July 2001.

[25] K. Chen, S. Malik and D. August, "Retargetable Static Timing Analysis for Embedded Software", *In Proceedings of the International Symposium on System Sciences*, (ISSS) 2001.