Design Tools for Application Specific Embedded Processors

Wei Qin[1], Subramanian Rajagopalan[1], Manish Vachharajani[1], Hangsheng Wang[1], Xinping Zhu[1], David August[1], Kurt Keutzer[2], Sharad Malik[1], and Li-Shiuan Peh[1]

[1] Princeton University, Princeton NJ 08544, USA
[2] UC Berkeley, Berkeley, CA 94720, USA

Abstract. A variety of factors make it increasingly difficult and expensive to design and manufacture traditional Application Specific Integrated Circuits (ASICs). Consequently, programmable alternatives are more attractive than ever. The flexibility provided by programmability comes with a performance and power overhead. This can be significantly mitigated by using application specific platforms, also referred to as Application Specific Embedded Processors, or Application Specific Instruction Set Processors (ASIPs).
ASIPs and the embedded software applications running on them, require specialized design tools - both during architectural evaluation to provide feedback on the suitability of the architecture for the application; as well as during system implementation to ensure efficient mapping and validation of design constraints. These functions result in requirements different from those of traditional software development environments. The first requirement is retargetability, especially during the early architectural evaluation stage where a rapid examination of design alternatives is essential. The second requirement is for additional metrics such as power consumption, real-time constraints and code size.
This paper describes a set of design tools and associated methodology designed to meet the challenges posed by architectural evaluation and software synthesis. This work is part of the MESCAL (Modern Embedded Systems, Compilers, Architectures, and Languages) project [3].

1 Introduction

Designing an ASIC in today's deep sub-micron geometries is harder than ever, and the problems continue to worsen with shrinking geometries. Design tools are finding it difficult to handle the complexity and electrical design challenges posed by each new technology generation. The net consequence is increasingly lowered design productivity despite increasingly expensive design tools. ASIC manufacturing costs are also rising - multi-million dollar mask sets are projected for sub-100nm designs. These high non-recurring design and manufacturing costs

[3] MESCAL is part of the Gigascale Silicon Research Center (GSRC), funded by DARPA and MARCO.

imply either larger break even volumes at fixed per-unit costs, or prohibitive per-unit costs at fixed volumes. An alternative implementation style to ASICs that is rapidly emerging is the use of programmable solutions - alternatively referred to as programmable platforms, Application Specific Embedded Processors, or Application Specific Instruction Set Processors (ASIPs). For the hardware developer the programmability of these devices enables a larger volume, as multiple related applications, as well as different generations of an application can be mapped onto the same ASIP. For the application developer, a programmable solution provides a much lower risk as well as a predictable and shorter time-to-market solution - writing and debugging software is cheaper than designing, debugging and manufacturing working hardware. However, for the class of applications of interest here, the power/delay overhead of general purpose programmable solutions is unacceptable. ASIPs attempt to match application characteristics with hardware support to minimize the power and performance overhead of programmable solutions to the point where they are an attractive alternative to ASICs. There are a number of application domains where this class of highly specialized embedded processors is catching on as a replacement for ASICs - notably network and communication processing. In fact, there are signs of a revolution afoot, with an increasing trend of engineers from hardware application groups going off and rapidly deploying the application in software on available domain specialized processors [1].

Historically, designers adopting manageable alternatives have heralded a significant change in design methodology, typically much before the change in design methodology has stabilized and acceptable tool flows become widely available. The move from schematic capture to logic synthesis and simulation in the mid-80s was led by designers unwilling to deal with increasing complexity in a non-scalable methodology. Home-grown rudimentary simulation and synthesis tools were enough to deliver enough increased productivity for them to abandon the old tools and also some design optimality. It did not take mature stable tools for them to make the change - those tools followed to convert the trend to accepted design practice on a larger scale. We believe that we are at a similar watershed in design implementation practice today. The individual ASIC designers that today are abandoning hardware design for the productivity benefit of software solutions on an ASIP, even at some loss of design quality (measured in area, delay, power), portend the acceptable design practice of tomorrow. This paper describes tools that will get us rapidly to that tomorrow by targeting the development and deployment of these ASIPs.

This paper is organized to highlight key components of the MESCAL Design Environment shown in Figure 1. First, in Section 2, we describe the Liberty Simulation Environment (LSE), a simulator construction infrastructure used for all system simulation. In addition to a simulator construction engine, LSE provides a library for processing elements (PEs) and specialized hardware. To model on-chip communication architectures (OCAs), Section 3 presents a modeling methodology that distinguishes functional and architectural views. Section 4 describes the methodologies used to model the power consumed by both PEs and OCAs.

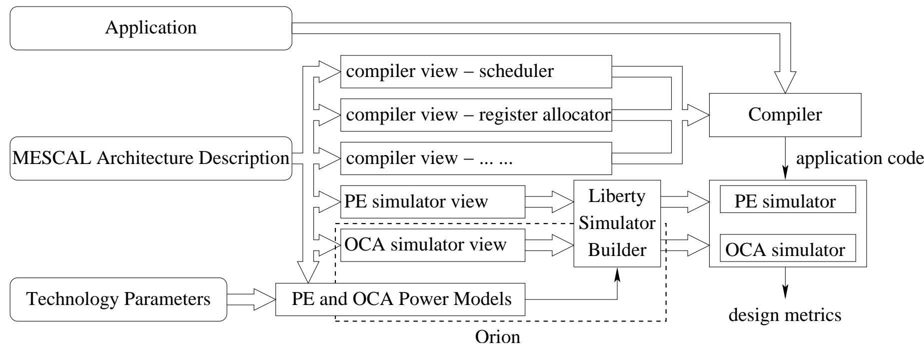

Fig. 1. The MESCAL Design Environment

Section 5 discusses the use of these methodologies in building Orion, a power-performance simulator for networks including OCAs. The MESCAL Architectural Description, described in Section 6, serves as the common specification used to generate appropriate views for use in modeling and compilation. The compiler is responsible for efficient mapping of applications onto these highly specialized architectures. Section 7 describes the philosophy and nature of algorithms of this compiler. We conclude with some final thoughts in Section 8.

2 The Liberty Simulation Environment

Traditionally, architects evaluate microarchitectures by running applications on a simulator written in a sequential programming language such as C or C++. Unfortunately, this and other simulator construction methods are not well suited for design-space exploration for ASIPs. To approach ASIC performance, ASIPs often contain elements that operate independently of the main execution pipeline, thus the timing is difficult to model in conventional sequential simulators. Furthermore, traditional techniques result in that simulators offer little to assure the architect of the accuracy of the simulation and do little to facilitate an understanding of the model from the simulator description. To make matters worse, many ASIPs contain multiple processing elements, that also have complex timing and memory interactions that are also difficult to model and lead to simulation inaccuracies [2].

These difficulties arise because the architect must map the microarchitecture, which is inherently structural and concurrent, to a sequential programming language. Correctly modeling small modifications to the structure of the microarchitecture can require large changes to the sequential simulator code, especially if the changes affect the relative timing of loosely coupled concurrently executing hardware elements. To avoid the laborious and error prone task of re-mapping for every candidate microarchitecture, designers may be tempted to make small changes to the simulator that approximate the changes in the microarchitecture. Unfortunately, these small changes in the simulator will often correspond to

large, unanticipated, changes in the microarchitecture leading to an inaccurate evaluation of the change.

The Liberty Simulation Environment [3, 4] (LSE) is a deliberate effort to address the mapping problem. LSE is a tool that can automatically map a concurrent and structural microarchitecture specification to an efficient simulator. Since LSE automatically maps microarchitectural specifications to a sequential program, an LSE microarchitecture specification resembles the hardware it models, thus only small changes in the specification are necessary to model small changes in a microarchitecture. Furthermore, LSE has been designed to allow modeling of specialized hardware common in ASIPs, but less frequently used in general purpose processors. The benefits of LSE allow architects to devote their full effort to exploring microarchitectures, instead of having them commit time, energy, and patience to manage an inherently complex and opaque sequential simulator.

In LSE, a user specifies the microarchitecture by describing the instantiation of architectural components, called *modules*, and their port to port interconnections. Each module roughly corresponds to a hardware block and has concurrent execution semantics, like actual hardware blocks, making specification easier and more accurate. The LSE specification allows a user to customize and extend modules using a *module extension* mechanism. Users can utilize pre-existing modules from the LSE library or create their own.

The ASIP design process is often incremental. Key parts of the system are initially modeled, and additional parts and details of the system are added as the application domain and design goals require. Furthermore, describing the complete microarchitecture for simulation is often a daunting task and development of novel microarchitectures for an application domain often involves simulating pieces of a microarchitecture without paying attention to unimportant details of the machine. LSE provides for this type of partial specification by assigning default semantics to unconnected ports; modules are required to behave in reasonable fashion if some ports are left unconnected. In this way, complete architecture descriptions can be developed incrementally, one piece at a time.

For rapid specification of ASIP design variants for architectural exploration, LSE provides a rich parameter system that allows the user to specify not only simple parameters such as sizes of hardware arrays, but also parameters that are algorithms. As a result, the user can override or augment the functionality of a module to incorporate new computation and form new hardware blocks from pre-existing ones. LSE also has special algorithmic parameters called *control points*. At each module port there is a control point in which the user may specify how data flows through the datapath, how signals are arbitrated, and how hardware blocks are activated. Since it would be quite tedious to have to write a control specification for each port on each module, or specify every parameter value, parameters have default values with reasonable defaults defined by the module author. Typically, for control points, this default control corresponds to back pressure control in a pipeline. Since computation in a microprocessor component often requires state from other portions of the machine, algorithmic

parameters may reference another module's explicitly exported state to perform their computation.

During different parts of the design cycle, designers require different information from the simulator. Furthermore, different members of the design team will want different information. Hardware designers may be interested in monitoring hardware bottlenecks, and software developers may be interested in collecting profile data for the compiler. As a result, LSE has *data collectors* and *events* to facilitate data collection that is orthogonal to the simulator specification. Each time something "interesting" occurs in a module, the module will emit an event. The event notification will be tagged with the module that produced it, the time it was produced, and any associated data that the module wishes to emit. Data collectors, which are specified independently of the described architecture, get notified when events occur and aggregate the data contained in the event. Since certain collectors may be interested in only certain events, a mechanism is provided for collectors to filter the events they receive. These events and data collectors are similar to *aspects* in aspect-oriented programming [5].

Other systems can be used to specify ASIP designs, however, they are all less then ideal. HDLs, for example, typically require the user to specify every last detail of a complete machine. By employing default semantics, LSE allows the user to produce a working simulator from a partial machine description. LSE also allows for module communication through abstract data types, so that the user need not manage wires and bus widths and data encoding manually. Other concurrent languages, such as SystemC [6], partially address shortcomings of sequential languages and HDLs, but the user must still resolve, by hand, all issues related to partial specification and interoperability of the specified components.

3 On-Chip Communication Architecture Modeling

3.1 Motivation

The distributed computation architecture of the ASIPs being considered here can be generally decomposed into two inter-related parts: The Processing Elements (PEs) and the On-Chip Communication Architecture (OCA). The PEs are responsible for the computation of the desired functions and the OCA provides the communication mechanisms. Just as the computational capabilities of the PEs must provide a match for the computational requirements of the application domain, the communication capabilities of the OCA must be well matched to the communication requirements of the concurrent computation. This match significantly impacts the timing as well as power characteristics of the implementation.

Technology advances have provided designers greater freedom in selecting from different types of communication schemes. The traditional way of interconnecting on-chip modules is via on-chip buses, such as the IBM CoreConnect Bus Architecture [7] and the ARM AMBA bus system [8]. An emerging option for integrating a large number of processors is to use on-chip networks [9–11].

A design environment that can potentially select any one of these OCAs must be able to model and support these choices with their variants. This decision process must be guided through a design space exploration framework where these different OCA (and PE) design choices can be tried, simulated and evaluated within an execution-driven virtual prototyping environment in a "plug and play" fashion.

3.2 Methodology

The discipline of computer architecture clearly distinguishes between the Instruction Set Architecture (ISA), which is the programming/functional view of the processor, and the micro-architecture, which comprises implementation details such as the number of pipeline stages, size and organization of cache memories, the number and type of function units (FUs) etc. OCAs can also be viewed in the same fashion. On the functional side, there exist different ways of sending and receiving data from the OCA, e.g. read/write data through shared memory, send/receive data through a message-passing network. On the implementation side, the OCA contains various details such as input/output controllers, buffers, arbiters, crossbars, etc. While the elements of ISAs and a micro-architectures for PEs are well-understood and defined - even across a broad range of PEs, this is not the case for OCAs. This work attempts to fill this gap.

As our first step, we define the following atomic constructs as the functional primitives of the OCA in the shared memory and message passing models (each PE is denoted by its address $i \in N$):

OCA read $(x,\ u)$ moves data x in the shared memory into local variable u
OCA write $(y,\ v)$ writes value of local variable y into shared variable v
OCA send $(x,\ i)$ sends the value of x to PE i asynchronously
OCA receive $(y,\ j)$ receives the value of x from PE j synchronously

This generic OCA "ISA" provides a basis for the communication primitives needed. For a specific OCA, the actual functional primitives may vary, but they are likely to be variants of the above (e.g. blocking reads and writes).

The above separation of the functional primitives and their actual implementation is useful in both system simulation and compilation. Functional simulation needs to understand only the semantics of the primitives and thus can be fast. Detailed timing simulation will naturally require the micro-architectural implementation models. Compilation can use these primitives, with additional latency information provided for them, for the distributed mapping of the application. They can also be mapped to lower level operations using library routine calls.

On the structural side, after detailed object-oriented analysis (OOA) [12], we have derived a set of abstract object classes which are sufficient to compose a wide range of OCAs. As shown in Figure 2, a relatively small set of object classes is sufficient here. The key observation here is that the OCA micro-architectural primitives belong to one of the following small set: links, buffers, resource schedulers, and interfaces. Within each element of this set, there are variations which over time get added to the class hierarchy.

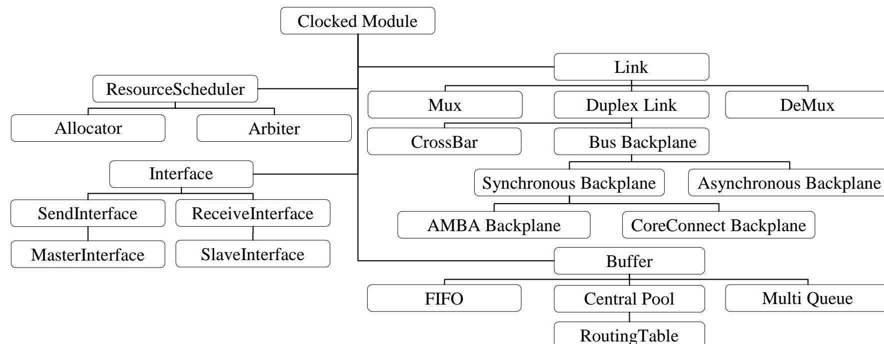

Fig. 2. Class Inheritance Hierarchy of OCAs

3.3 Use of Methodology

We have used this methodology to model OCAs as part of Orion in LSE (see Section 5), as well as Ptolemy II [13], an object-oriented, heterogeneous design and modeling framework. Both environments support construction of executable models in a modular fashion. For a specific OCA model, the designer needs to examine and implement individual building blocks by either instantiating or extending available modules in the class hierarchy. Thus, OCA design is simplified as a process of integration of these “plug and play” modules. Within these two environments, we implemented cycle-accurate models of two on-chip bus systems, AMBA [8] and CoreConnect [7], and an on-chip packet-switching network, the RAW [11] network [14]. Our experience finds that adopting the reusable hierarchical class diagram greatly reduces development time.

4 Power Modeling

As highlighted in Section 1, power has become a design metric that is as important, if not more important than timing. Thus, power modeling and simulation is an essential part of the design environment. In this section, we detail how power models are derived for the different hardware modules of both PEs and OCAs.

To facilitate retargetable simulation and design space exploration, several requirements are imposed on power models: flexibility, re-usability, and fidelity. These requirements are met by the following modeling hierarchy that enables easy model composition.

4.1 Model Hierarchy

The model hierarchy consists of 4 layers: atomic layer, structure layer, prototype layer and physical layer, from bottom to top as illustrated in Figure 3. Each layer

plays a different role and layers are relatively independent of each other so that they can be modified without affecting other layers. The layers cooperate in computing the switching energy $E = \frac{1}{2}\alpha C V_{dd}^2$.

- The prototype and physical layers collect information needed by the lower layers and assemble results reported by lower layers.
- The structure layer computes switching activity factor α, and the atomic layer computes switching capacitance C.

Atomic Layer This is the lowest layer. An atomic component consists of several capacitance elements which always switch simultaneously. For example, if gate A drives one input of gate B, then the output capacitance of gate A, the connection wire capacitance, and the input capacitance of gate B are in the same atomic component.

$$E_{atomic} = \frac{1}{2} V_{dd}^2 C_{atomic} \tag{1}$$

Structure Layer This layer corresponds to circuit building blocks that perform some basic functions, e.g. decoder, comparator, etc. One structure layer component consists of several atomic components.

$$E_{structure} = \sum_i (\alpha_i E_{atomic}^i) \tag{2}$$

where α_i is the switching activity factor of the ith atomic component.

Prototype Layer This layer models "virtual" function units, i.e. abstract function units with complete structure, but no specified functionality. A prototype model is a collection of structure models, with parameter definitions to specify structure model types, properties and connections. For example, the most widely used prototype model is the uniform array model, which essentially models generic SRAM array. The model has a complete list of structure components and parameters specifying whether these components are present and their model types.

$$E_{prototype} = \sum_i E_{structure}^i (input_i, state_i) \tag{3}$$

where $input_i$ and $state_i$ are information needed by structure models to compute switching activities.

Physical Layer This layer models real function units. A physical layer model is just a prototype model with concrete parameters specifying functionality of the function unit. For example, the aforementioned uniform array model can model a data cache if configured with tag array, tag comparators and certain hit/miss policies, or a shared central buffer [15] if configured without tag array but with pipelined banks.

$$E_{physical} = E_{prototype} \tag{4}$$

Advantages of Model Hierarchy

1. Separation of micro-architecture dependency and technology dependency: Only atomic layer power models depend on fabrication technologies, while other layers are technology independent. Technology dependency is resolved by maintaining a minimal set of low level capacitance constants and scaling them according to technologies.
2. Fine-grained modeling granularity: This enables the tracking of dynamic switching activity at a level low enough to reflect physical reality. This scheme can achieve higher accuracy than using average switching activity or operation activity, which is unable to capture the effects of some low power techniques aiming at reducing switching activities.
3. Reusable prototype layer power models (templates) and structure layer power models (building blocks) to ease developing new models: For instance, when adding the crossbar power model, we re-use the structure layer tri-state output buffer model which is a component of cache model. In Table 1 and Figure 3, we show how the uniform array model can be re-used to model different function units through parameter mapping.

Table 1. Parameter mapping between prototype layer and physical layer models

Prototype layer parameters of uniform array model	Physical layer parameters of data cache	Physical layer parameters of shared central buffer
rows	number of sets	number of chunks
cols	cache line size	pipeline depth × flit width
number of comparators	associativity	0
data width	integer width	flit width
...	...	...

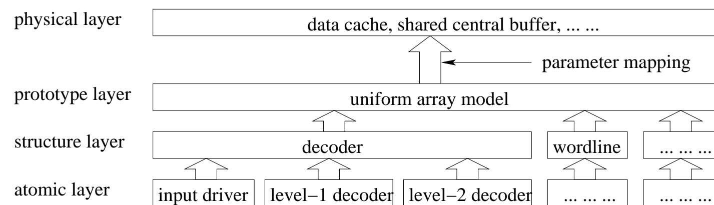

Fig. 3. Model hierarchy of data cache and central buffer power models

4. Easy maintenance: Modifying lower layer models or adding new model types will not affect higher layers.

4.2 Use of Methodology

Our methodology guided our development of power models for PEs as well as OCAs. Our PE power models have been integrated into LSE (see Section 2) using LSE's module extension mechanism and our network power models into Orion (see Section 5). For PEs, we have built a variety of power models: cache, branch predictors, register files, ALUs, and even some highly specialized function units such as the OMFLIP unit [16]. For networks, we have modeled input/output buffers, shared central buffers, crossbars, arbiters and links. Our models have been validated through comparisons with other simulators and low-level power estimates. Existing work focuses on further validation of the models, development of new models and support for static power in the modeling.

5 Orion – OCA Power Modeling and Simulation

As the use of multiple interconnected PEs becomes increasingly prevalent in application-specific embedded systems, the need to consider both power and performance of networks becomes pressing. Orion [17] provides this critical capability, with a dynamic power simulation environment for a wide range of interconnection networks, including OCAs. Orion extends LSE, building a library of router and link modules, each instantiated with functional, timing and power models. A user first *picks* modules to assemble the network he/she wishes to simulate, the modules are then *plugged* into LSE that builds a network simulator automatically, and a communication workload *played* on the network to evaluate its power and performance. This "pick, plug-and-play" environment allows users to rapidly explore the design space of interconnection networks.

MESCAL's methodology for OCAs (see Section 3) guided the selection of the building blocks of Orion. For each building block, the functional and timing behavior follows that characterized in [18] closely, while power modeling is carried out as outlined in Section 4. Orion has since been used to model a variety of network architectures and workloads, ranging from networks connecting PEs on a single chip to microprocessors with integrated routers and complex InfiniBand switches, providing valuable insights [19].

Orion forms a key piece in MESCAL's development of a complete tool suite for exploring application-specific embedded processors. While Orion is currently a stand-alone platform for investigating networks, we are in the process of tying it with PE power simulation within the LSE framework, so designers can explore interconnected processors in tandem with the network, in a single coherent environment.

6 Architectural Description

The MESCAL Architecture Description [20] (MAD) is designed as a unified architecture representation for MESCAL's retargetable software tool-chain including the optimizing C compiler and the simulators at various abstraction

levels. The unified scheme eases the work for the description writers and ensures that all parts of the tool-chain share a consistent view of the architecture. To transform the single description to naturally-fit data models for individual components of the tool-chain, the MAD compiler analyzes the description and generates optimized view files, as shown in Figure 1. The current focus of MAD is the specification of individual PEs, future work will include OCAs in the same specification.

Architecture description languages (ADLs) have been the research focus of several past and ongoing projects [21–25]. However, we see no sign of convergence of the field for two main reasons. First, the extensive space of computer architectures and microarchitectures is difficult to capture accurately and efficiently with a single model. Second, the diverse requirements imposed by different software tools are hard to satisfy effectively with a single description scheme. Bearing in mind the difficulties, we currently confine MAD to a limited architecture scope including in-order RISC processors and statically scheduled VLIW processors. Within this restriction though, MAD can handle a wide range of application specific customizations - specialized functional units, memories, restrictions on Instruction Level Parallelism (ILP) etc. This enables it to model a fairly wide range of PEs. We also try to balance MAD between support for the retargetable compiler and support for the retargetable simulators.

For computer architectures, two abstraction levels are well understood: the instruction set architecture (ISA) and the micro-architecture. Since both provide useful knowledge to the optimizing compiler and the simulators, MAD includes the two in its behavioral part and its structural part, respectively. To bridge the gap between the two abstraction levels, MAD also provides a mapping part.

The Behavioral Part: The ISA is modeled in three sections: operand, operation and instruction. Similar to nML [22] and ISDL [23], MAD utilizes attribute grammar [26] to organize the semantics, binary encoding and assembly format across the layers.

The operand section describes the addressing modes. Two types of primitive operands can be defined: immediate and register. The immediate operand describes the constant values encoded in instruction words. The register operand describes the logical registers exposed to the ISA. Most PEs have complex addressing modes as the combinations of two or more primitive operands. One example is the shifting operand of the ARM [27] architecture. Since such complex addressing modes are often shared by many operations, a flat operation description scheme based purely on primitive operands will result in lengthy descriptions with much redundancy. To avoid this, we introduce an intermediate level "composite" operand to capture the complex addressing modes. A composite bases its semantics, encoding and syntax on those of its children primitive operands. Similar hierarchical schemes can also be found in nML and ISDL.

The operation section describes operations based on the operands. Besides encoding, assembly syntax and semantics, operations have two optional attributes: predicate and side_effects. The former specifies the predicate operand and the latter defines side effects like the altering of machine flags.

The instruction section describes possible bundling schemes of the operations for VLIW architectures. Irregular operation packing rules which often appear in low cost DSP designs can be described in the section and will be converted into resource constraints [28] for use in the optimizing compiler. For RISC processors, an instruction simply contains one operation.

The Structural Part: This part models the micro-architecture in the form of a coarse-grained netlist. MAD distinguishes between two categories of hardware units: pipeline stages and special function units. Pipeline stages are regular building fabrics of a PE as simple place-holders for operations. Their actual semantics as an ALU or a multiplier is ignored here since operation semantics are covered in the behavioral part. Special function units have heterogeneous semantics. Units like registers, memories, branch predictors are all treated as special function units. A library of basic special function units is built into MAD.

Each MAD hardware unit has input and output ports. Connections can be specified between ports. Essential connections such as those connecting pipeline stages and register files or memories are important for the MAD compiler to understand important architecture properties such as data path organization and memory banking.

The Mapping Part: Two types of mappings are described in this part: operand mapping and operation mapping. The operand mapping specifies the port through which an operand is accessed and the time of the access. The port information allows the compiler to schedule the operations properly to avoid port resource contentions. It also enables the simulator to interlock operations properly when such contentions occur. The operation mapping specifies the pipeline stages that an operation will flow through from its issue to completion. The description style in this section is similar to that of Maril [29] or EXPRESSION [25]. It describes the paths of an operation.

7 Retargetable Compiler

The compiler is a critical tool when designing systems with ASIPs. Along with tools such as execution-time estimators [30] and good input sets for performance evaluation, a high quality compiler allows designers to meet tight real-time deadlines and price/performance constraints without writing large amounts of assembly code. The compiler is also essential in properly evaluating the effectiveness of the hardware early in the design of the ASIP itself, since a microarchitecture cannot be properly evaluated without high quality benchmark code. Thus, the requirements for a compiler in the MESCAL environment are three-fold, *viz.* it must be highly retargetable to cover a wide range of architectures; it must have a wide variety of optimizations to produce good quality code; and it must be highly configurable to allow for different compile time vs. efficiency trade-offs during various stages of design. These requirements often conflict with each other, thus providing challenges that differentiate this compiler work from others.

The basic structure of the compiler is fairly standard. The front end is based on *lcc* [31] which is light-weight, documented and publicly available. The back-end, based on the Liberty-IR infrastructure [32], consists of three phases, namely, the code generation phase; the optimization phase including register allocation and instruction scheduling; and the code emission phase. With retargetability as one of the main thrusts of MESCAL, the compiler is automatically driven by the MAD specification described in Section 6. Minimal compiler knowledge is needed to retarget the back-end.

Embedded processors such as DSPs often contain non-orthogonal instruction sets and irregular data-paths like multiple register banks to reduce code size; to optimize area/power; to increase ILP; and to offer a wide set of addressing modes. These features pose a variety of problems to the compiler such as selecting the optimal addressing mode by the code generator; supporting diverse register banks efficiently during register allocation; handling irregular ILP in the scheduler, etc. Retargetable solutions to many of the problems are still being worked on.

To support architectures with irregular constraints within an algorithmic framework, we seek low-cost retargetable solutions rather than architecture (family) specific optimizations [33]. Here we briefly describe one such method that enables the use of resource based VLIW schedulers for processors with irregular ILP where the ISA restricts the sets of operations that can be issued in parallel, even though the physical resources may not impose them. We have developed the Artificial Resource Allocation (ARA) [28] algorithm which takes the set of all possible combinations of operations that can be issued in parallel from the machine description and assigns artificial resource (AR) usages to each operation such that, an AR is assigned to every pair of operations that cannot be issued in parallel; an AR is not assigned to every pair of operations that can be issued in parallel; and the total number of ARs is minimum. This is achieved by constructing a compatibility graph that has the operations in the ISA as vertices and an edge is drawn between every pair of vertices that can be issued in parallel. The ARA problem then translates to labeling the complement of the compatibility graph with the minimum number of labels (ARs) such that each pair of vertices (operations) connected by an edge in the complement graph is assigned at least one label (AR). For further details, we refer the readers to [28].

8 Conclusion

We are seeing a significant move from application development in dedicated hardware on ASICs, to programmable solutions on Application Specific Embedded Processors. However, these processors must achieve high power and performance efficiency in order to replace ASICs. This is accomplished through application specific customization in the form of specialized hardware resources. This customization places significant requirements on the software development environment needed for both processor development, as well as application deployment.

We have described a set of tools used for simulation and compilation with the following key characteristics:

- ability to handle a wide range of architecture customization
- fully retargetable, starting from a unified architectural specification
- ability to handle on-chip communication architectures in addition to processing element architectures
- ability to model and simulate power consumption for both PEs and OCAs

We believe that the above infrastructure will significantly enable this transition from ASICs to Application Specific Embedded Processors.

References

1. Paulin, P.: What is the next EDA driver? Design Automation Conference Panel (2002)
2. Pai, V.S., Ranganathan, P., Adve, S.V.: RSIM reference manual, version 1.0. Technical Report 9705, Department of Electrical and Computer Engineering, Rice University (1997)
3. Vachharajani, M., Vachharajani, N., Penry, D., Blome, J., August, D.: Architectural exploration with Liberty. Technical Report Liberty-02-01, Liberty Research Group, Princeton University (2002)
4. The Liberty Research Group: http://liberty.princeton.edu/ (2002)
5. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Proceedings of the 11th European Conference on Object-Oriented Programming. (1997) 220–242
6. SystemC Community: http://www.systemc.org (2002)
7. IBM Corp.: The CoreConnect™ bus architecture. Technical White Paper (1999)
8. ARM Holdings PLC: Advanced microcontroller bus architecture (AMBA) specification rev 2.0. http://www.arm.com/Documentation/UserMans/AMBA (2001)
9. Dally, W.J., Towles, B.: Route packet, not wires: On-chip interconnection networks. In: Proceedings of Design Automation Conference. (2001)
10. Sgroi, M., Sheets, M., Mihal, A., Keutzer, K., Malik, S., Rabaey, J., Sangiovanni-Vincentelli, A.: Addressing the system-on-a-chip interconnect woes through communication-based design. In: Proceedings of Design Automation Conference. (2001)
11. Taylor, M.B., et. al.: The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. IEEE Micro **22** (2002)
12. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: Object-Oriented Modeling and Design. Prentice-Hall, New York, NY (1991)
13. Davis, J., et. al.: Ptolemy II - heterogeneous concurrent modeling and design in Java. Technical Report UCB/ERL M01/12, Dep. of EECS, Univ. of California at Berkeley (2001)
14. Zhu, X., Malik, S.: A hierarchical modeling framework for on-chip communication architectures. In: Proceedings of International Conference on Computer-Aided Design. (2002)
15. Katevenis, M., Vatsolaki, P., Efthymiou, A.: Pipelined memory shared buffer for VLSI switches. In: Proceedings of Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication. (1995)

16. Yang, X., Lee, R.B.: Fast subword permutation instructions using omega and flip network stages. In: Proceedings of International Conference on Computer Design. (2000)
17. Wang, H.S., Zhu, X.P., Peh, L.S., Malik, S.: Orion: A dynamic power simulator for interconnection networks – enabling power-performance tradeoffs for emerging microprocessor systems. Technical Report PU-02-06, Department of Electrical Engineering, Princeton University (2002)
18. Peh, L.S., Dally, W.J.: A delay model and speculative architecture for pipelined routers. In: Proceedings of International Symposium on High-Performance Computer Architecture. (2001)
19. Wang, H.S., Peh, L.S., Malik, S.: A power model for routers: Modeling Alpha 21364 and InfiniBand routers. In: Proceedings of Hot Interconnects 10. (2002)
20. Qin, W.: Mescal architecture description. http://www.ee.princeton.edu/~mescal/mad.html (2002)
21. Zimmerman, G.: The MIMOLA design system: a computer aided processor design method. In: Proceedings of Design Automation Conference. (1979) 53–58
22. Freericks, M.: The nML machine description formalism. Technical Report 1991/15, Technische Universität Berlin, Fachbereich Informatik, Berlin, DE (1991)
23. Hadjiyiannis, G., Hanono, S., Devadas, S.: ISDL: An instruction set description language for retargetability. In: Proceedings of Design Automation Conference. (1997) 299–302
24. Pees, S., Hoffmann, A., Zivojnovic, V., Meyr, H.: LISA – machine description language for cycle-accurate models of programmable DSP architectures. In: Proceedings of Design Automation Conference. (1999) 933–938
25. Halambi, A., Grun, P., Ganesh, V., Khare, A., Dutt, N., Nicolau, A.: EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In: Proceedings of Conference on Design Automation and Test in Europe. (1999) 485–490
26. Paakki, J.: Attribute grammar paradigms – a high-level methodology in language implementation. ACM Computing Surveys **27** (1995) 196–255
27. ARM Ltd.: ARM architecture reference manual. http://www.arm.com/arm/documentation (1996)
28. Rajagopalan, S., Vachharajani, M., Malik, S.: Handling irregular ILP within conventional VLIW schedulers using artificial resource constraints. In: Proceedings of International Conference on Compilers, Architectures and Synthesis for Embedded Systems. (2000)
29. Bradlee, D.G., Henry, R.R., Eggers, S.J.: The marion system for retargetable instruction scheduling. In: Proceedings of Conference on Programming Language Design and Implementation. (1991) 229–240
30. Chen, K., Malik, S., August, D.I.: Retargetable static timing analysis for embedded software. In: Proceedings of International Symposium on System Synthesis. (2001)
31. Fraser, C.W., Hanson, D.R.: A Retargetable C Compiler : Design and Implementation. Addison-Wesley, Menlo Park, CA (1995)
32. Triantafyllis, S., Vachharajani, M., August, D.: The Liberty Compiler intermediate representation. Technical Report Liberty-02-02, Liberty Research Group, Princeton University (2002)
33. Marwedel, P., Goossens, G.: Code Generation for Embedded Processors. Kluwer Academic Publishers (1995)