Architecture Description Language (ADL)-driven Software Toolkit Generation for Architectural Exploration of Programmable SOCs

PRABHAT MISHRA
Department of Computer and Information Science and Engineering, University of Florida

AVIRAL SHRIVASTAVA and NIKIL DUTT
Center for Embedded Computer Systems, University of California, Irvine

The increasing complexity of system functionality and advances in semiconductor technology have resulted in system implementations using programmable Systems-on-Chips (SOCs). Furthermore as the software content of such SOCs dominates the design process, there is a critical need to support exploration and evaluation of different architectural configurations. The increasing software content coupled with decreasing time-to-market create a critical need for automated software generation tools to increase designer productivity. Traditional hardware-software codesign flows do not support necessary exploration and customization of the embedded processors. However the inherently application specific nature of the embedded processors and the stringent area, power and performance constraints in embedded systems design critically require fast and automated architecture exploration methodology. Architecture Description Language (ADL)-driven design space exploration and software toolkit generation strategies have become popular recently. This approach provides a systematic mechanism for a top-down design and validation framework, which is very important to develop complex systems. The heart of this approach lies in the ability to automatically generate software toolkit including a architecture-sensitive compiler, a cycle accurate simulator, assembler, debugger, and verification/validation tools. This paper presents a software toolkit generation methodology using the EXPRESSION ADL. Our exploration studies demonstrate the need and usefulness of this approach in compiler-in-the-loop design space exploration of reduced instruction-set architectures.

Categories and Subject Descriptors: D.3.4 [**Software**]: Programming Languages—*Processors*; I.6.7 [**Computing Methodologies**]: Simulation and Modeling—*Simulation Support Systems*

General Terms: Design, Languages

Additional Key Words and Phrases: Architecture Description Language, Design Space Exploration, Programmable Architecture, Retargetable Compilation, Embedded Processor

This work was partially supported by NSF grants CCR-0203813 and CCR-0205712. We would like to acknowledge the contributions of Prof. Alex Nicolau, Mehrdad Reshadi, Ashok Halambi, Dr. Peter Grun, Partha Biswas, Dr. Mahesh Mamidipaka, and Sudeep Pasricha.

Author's address: P. Mishra, Department of Computer and Information Science and Engineering, University of Florida, Gainesville FL 32611, email: prabhat@cise.ufl.edu; Aviral Shrivastava and Nikil Dutt, Center for Embedded Computer Systems, Donald Bren School of Information and Computer Sciences, University of California, Irvine, CA 92697, email: {aviral, dutt}@ics.uci.edu.

1. INTRODUCTION

Increasing complex system functionality and advances in semiconductor technology are changing how electronic systems are designed and implemented today. The escalating non-recurring engineering (NRE) costs to design and manufacture the chips have tilted the balance towards achieving greater design reuse. As a result, hardwired application specific integrated circuit (ASIC) solutions are no longer attractive. Increasingly we are seeing a shift toward systems implemented using programmable platforms. Furthermore, the high degree of integration provided by current semiconductor technology has enabled the realization of the entire system functionality onto a single chip, which we call a *Programmable System-On-Chip (SOC)*. Programmable SoCs are an attractive option not only because they provide a high degree of design reuse via software, but because they also greatly reduce the time-to-market.

With both the system complexity and time-to-market becoming the main hurdles for design success, a key factor in programmable SoC design is the designer's productivity, i.e., his ability to quickly and efficiently map applications to SOC implementations. Furthermore, the need for product differentiation necessitates careful customization of the programmable SOC – a task that traditionally takes a long time. Traditionally, embedded systems developers performed limited exploration of the design space using standard processor and memory architectures. Furthermore, software development was usually done using existing, off-the-shelf processors (with supported integrated software development environments) or done manually using processor specific low-level languages (assembly). This was feasible because the software content in such systems was low and the embedded processor architectures were fairly simple (e.g., no instruction level parallelism) and well-defined (e.g., no parameterizable components). The emergence of complex, programmable SOCs poses new challenges for design space exploration. To enable efficient and effective design space exploration, the system designer critically needs methodologies that permit: i) rapid tuning of the embedded processors for target applications, and ii) automatic generation of customized software for the tuned embedded processors.

Figure 1 describes a contemporary hardware/software co-design methodology for the design of traditional embedded systems consisting of programmable processors, application specific integrated circuits (ASICs), memories, and I/O interfaces [**?**]. This contemporary design flow starts from specifying an application in a system design language. The application is then partitioned into tasks that are either assigned to software (i.e., executed on the processor) or hardware (ASIC) such that design constraints (e.g., performance, power consumption, cost, etc.) are satisfied. After hardware/software partitioning, tasks assigned to software are translated into programs (either in high level languages such as C/C++ or in assembly), and then compiled into object code (which resides in memory). Tasks assigned to hardware are translated into Hardware Description Language (HDL) descriptions and then synthesized into ASICs.

In traditional hardware/software codesign, the target architecture template is

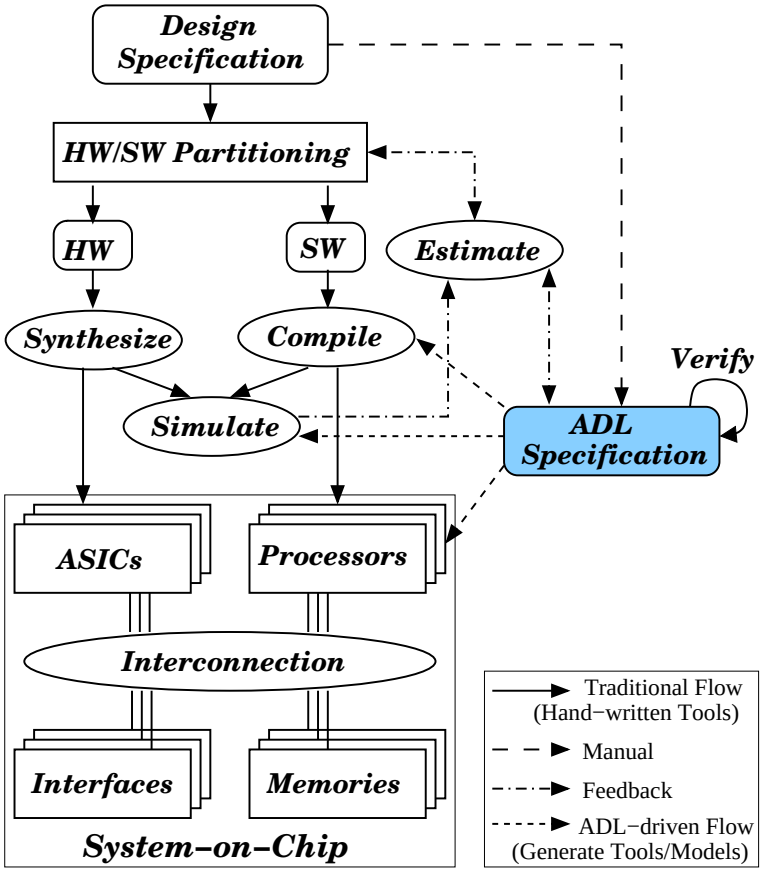

Fig. 1. Hardware/Software Co-Design Flow for SOC Design

pre-defined. Specifically, the processor is fixed or can be selected from a library of pre-designed processors, but customization of the processor architecture is not allowed. Even in co-design systems allowing customization of the processor, the fundamental architecture can rarely be changed. However the inherently application specific nature of the embedded processor and strict multi-dimensional design constraints (power, performance, cost, weight, etc.) on the SoC critically requires customization and optimization of the design, including the processor design, memory design and the processor-memory organizations. The contemporary co-design flow (which does not permit much customization) limits the ability of the system designer to fully utilize emerging IP libraries, and furthermore, restricts the exploration of alternative (often superior) SOC architectures. Consequently there is tremendous interest in a language-based design methodology for embedded SOC optimization and exploration.

Architectural Description Languages (ADLs) are used to drive design space exploration and automatic compiler/simulator toolkit generation. As with an HDL-based ASIC design flow, several benefits accrue from a language-based design methodology for embedded SOC design, including the ability to perform (formal) verification

and consistency checking, to easily modify the target architecture and memory organization for design space exploration, and to automatically generate the software toolkit from a single specification.

Figure 1 illustrates the ADL-based SOC co-design flow, wherein the architecture template of the SOC (possibly using IP blocks) is specified in an ADL. This template is then validated to ensure the specification is golden. The validated ADL specification is used to automatically generate a software toolkit to enable software compilation and co-simulation of the hardware and software. Another important and noticeable trend in the embedded SOC domain is the increasing migration of system functionality from hardware to software, resulting in a high degree of software content for newer SOC designs. This trend, combined with shrinking time-to-market cycles, has resulted in intense pressure to migrate the software development to a high-level language (such as C, C++, Java) based environment in order to reduce the time spent in system design. To effectively perform these tasks, the SOC designer requires a high-quality software toolkit that allows exploration of a variety of processor cores (including RISC, DSP, VLIW, Super-Scalar and ASIP), along with generation of optimized software, to meet stringent performance, power, code density, and cost constraints. Manual development of the toolkit is too time-consuming to be a viable option. An effective embedded SOC co-design flow must therefore support automatic software toolkit generation, without loss of optimizing efficiency. Such software toolkits typically include Instruction Level Parallelizing (ILP) compilers, cycle-accurate and/or instruction-set simulators, assemblers/disassemblers, profilers, debuggers etc. In order to automatically generate these tools, software toolkit generators accept as input a description of the target processor-memory system specified in an ADL.

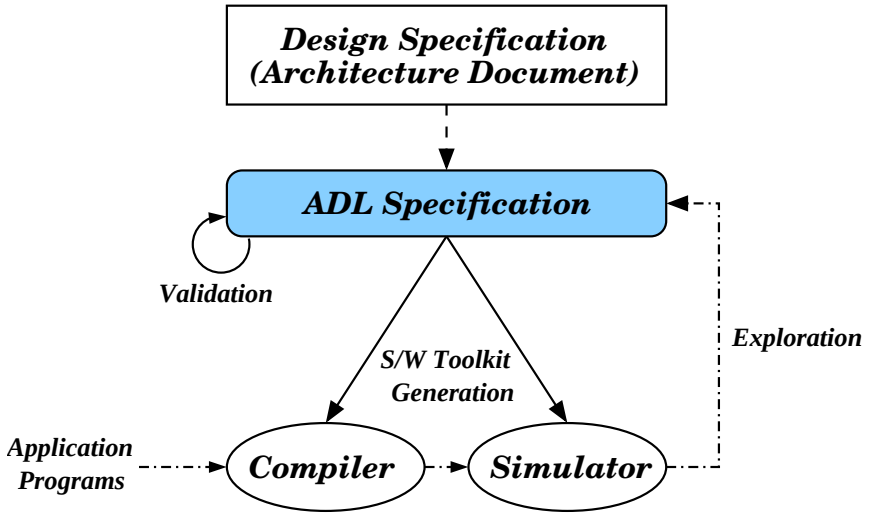

Fig. 2. ADL-driven Design Space Exploration

This article focuses on ADL-driven software toolkit generation and design space exploration. Figure 2 shows a simplified methodology for ADL driven exploration.

This methodology consists of four steps: design specification, validation of the specification, retargetable software toolkit generation, and design space exploration. The first step is to capture the programmable architecture using an ADL. The next step is to verify the specification to ensure the correctness of the specified architecture. The validated specification is used to generate a retargetable software toolkit that includes a compiler and a simulator. The generated software toolkit enables design space exploration of programmable architectures for the given application programs under various design constraints such as area, power and performance.

The rest of the paper is organized as follows. Section 2 briefly surveys current ADLs and describes how to capture processor, coprocessor and memory architectures using the EXPRESSION ADL. Section 3 presents validation techniques to verify the ADL specification. Section 4 presents the methodology for retargetable compiler generation. The retargetable simulator generation approach is described in Section 5 followed by a case study in Section 6. Finally, Section 7 concludes the paper.

2. ARCHITECTURE SPECIFICATION USING EXPRESSION ADL

The phrase "Architecture Description Language" (ADL) has been used in the context of designing both software and hardware architectures. Software ADLs are used for representing and analyzing software architectures [**?**]. Software ADLs capture the behavioral specifications of the components and their interactions that comprises the software architecture. However, hardware ADLs capture the structure (hardware components and their connectivity), and the behavior (instruction-set) of programmable architectures consisting of processor, coprocessor and memory subsystem. Computer architects have long used machine description languages for the specification of architectures. Early ADLs such as ISPS [**?**] were used for simulation, evaluation, and synthesis of computers and other digital systems. Contemporary ADLs can be classified into three categories based on the nature of the information an ADL can capture: structural, behavioral, and mixed. The structural ADLs (e.g., MIMOLA [**?**]) capture the structure in terms of architectural components and their connectivity. The behavioral ADLs (e.g., nML [**?**] and ISDL [**?**]) capture the instruction-set behavior of the processor architecture. The mixed ADLs (e.g., LISA [**?**] and EXPRESSION [**?**]) capture both structure and behavior of the architecture. There are many comprehensive ADL surveys available in the literature including ADLs for retargetable compilation [**?**], SOC design [**?**], and programmable embedded systems [**?**].

Our exploration framework uses the EXPRESSION ADL [**?**] to specify processor, coprocessor, and memory architectures. Figure 3 shows an example architecture that can issue up to three operations (an ALU operation, a memory access operation and a coprocessor operation) per cycle. The coprocessor supports vector arithmetic operations. In the figure, oval boxes denote units, dotted ovals are storages, bold edges are pipeline edges, and dotted edges are data-transfer edges. A data-transfer edge transfers data between units and storages. A pipeline edge transfers instruction (operation) between two units. A path from a root node (e.g., Fetch) to a leaf node (e.g, WriteBack) consisting of units and pipeline edges is called a *pipeline path*. For example, {*Fetch, Decode, ALU, WriteBack*} is a pipeline path. A path from a unit

to main memory or register file consisting of storages and data-transfer edges is called a *data-transfer path*. For example, {*MemCntrl, L1, L2, MainMemory*} is a data-transfer path. This section describes how the EXPRESSION ADL captures the structure and behavior of the architecture shown in Figure 3 [**?**].

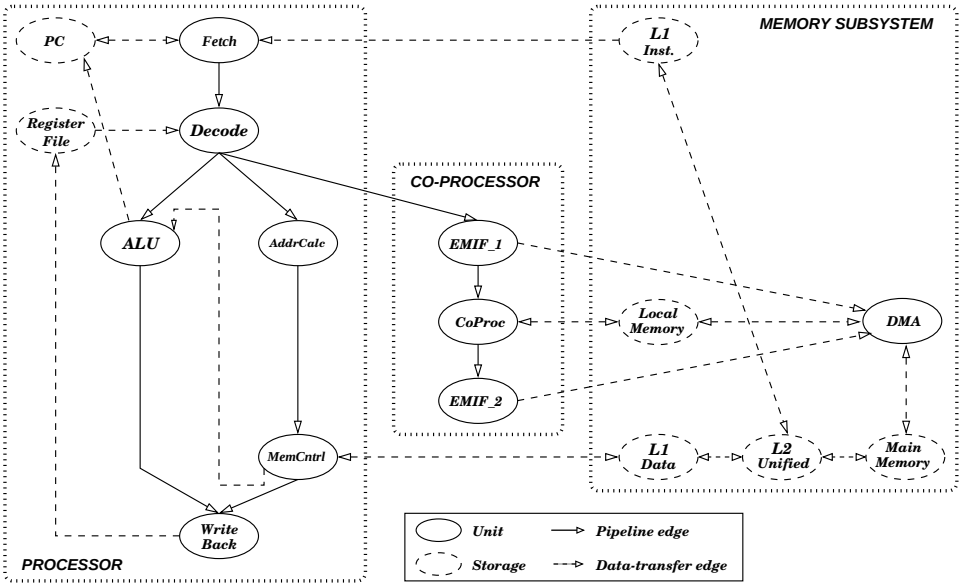

Fig. 3. An example architecture

2.0.1 *Structure.* The structure of an architecture can be viewed as a net-list with the components as nodes and the connectivity as edges. Similar to the processor's block diagram in its databook, Figure 4 shows a portion of the EXPRESSION description of the processor structure [**?**]. It describes all the components in the structure such as *PC*, *RegisterFile*, *Fetch*, *Decode*, *ALU* and so on. Each component has a list of attributes. For example, the *ALU* unit has information regarding the number of instructions executed per cycle, timing of each instruction, supported opcodes, input/output latches, and so on. Similarly, the memory subsystem structure is represented as a netlist of memory components. The memory components are described and attributed with their characteristics such as cache line size, replacement policy, write policy, and so on.

The connectivity is established using the description of pipeline and data-transfer paths. For example, Figure 4 describes the four-stage pipeline as {*Fetch, Decode, Execute, WriteBack*}. The *Execute* stage is further described as three parallel execution paths: *ALU*, *LoadStore*, and *Coprocessor*. Furthermore, the *LoadStore* path is described using pipeline stages: *AddrCalc* and *MemCntrl*. Similarly, the coprocessor pipeline has three pipeline stages: *EMIF_1*, *CoProc*, and *EMIF_2*. The architecture has fifteen data-transfer paths. Seven of them are uni-directional paths. For example, the path {*WriteBack* → *RegisterFile*} transfers data in one direction, whereas the path {*MemCntrl* ↔ *L1Data*} transfers data in both directions.

2.0.2 *Behavior.* The EXPRESSION ADL captures the behavior of the architecture as the description of the instruction set and provides a programmer's view of the architecture. The behavior is organized into operation groups, with each group containing a set of operations[1] having some common characteristics. For example, Figure 2.0.2 [**?**] shows three operation groups. The *aluOps* group includes all the operations supported by the *ALU* unit. Similarly, *memOps* and *cpOps* groups contains all the operations supported by the units *MemCntrl* and *CoProc* respectively. Each instruction is then described in terms of it's opcode, operands, behavior, and instruction format. Each operand is classified either as source or as destination. Furthermore, each operand is associated with a type that describes the type and size of the data it contains. The instruction format describes the fields of the instruction in both binary and assembly. Figure 2.0.2 shows the description of *add*, *store*, and *vectMul* operations. Unlike normal instructions whose source and destination operands are register type (except load/store), the source and destination operands of *vectMul* are memory type. The *s1* and *s2* fields refer to the starting addresses of two source operands for the multiplication. Similarly *dst* refers to the starting address of the destination operand. The *length* field refers to the vector length of the operation that has immediate data type.

```
# Components specification
( FetchUnit Fetch
  (capacity 3) (timing (all 1)) (opcodes all) ...) ...
)
( ExecUnit ALU
  (capacity 1) (timing (add 1) (sub 1) ...)
  (opcodes (add sub ...)) ...
)
( CPunit CoProc
  (capacity 1) (timing (vectAdd 4) ...)
  (opcodes (vectAdd vectMul))
)
......
# Pipeline and data-transfer paths
(pipeline Fetch Decode Execute WriteBack)
(Execute (parallel ALU LoadStore Coprocessor))
(LoadStore (pipeline AddrCalc MemCntrl))
(Coprocessor (pipeline EMIF_1 CoProc EMIF_2))
(dtpaths (WriteBack RegisterFile) (L1Data L2) ...)
......
# Storage section
( DCache L1Data
  (wordsize 64) (linesize 8) (associativity 2) ...
)
( ICache L1Inst (latency 1) ...)
( DCache L2 (latency 5) ...)
( DRAM MainMemory (latency 50) ...)
```

Fig. 4. Specification of Structure using EXPRESSION ADL

[1]In this paper we use the terms operation and instruction interchangeably.

```
# Behavior: description of instruction set
( opgroup aluOps (add, sub, ...) )
( opgroup memOps (load, store, ...) )
( opgroup cpOps (vectAdd, vectMul, ...) )
( opcode add
  (operands (s1 reg) (s2 reg/int16) (dst reg))
  (behavior dst = s1 + s2)
  (format 000101 dst(25-21) s1(21-16) s2(15-0))
)
( opcode store
  (operands (s1 reg) (s2 int16) (s3 reg))
  (behavior M[s1 + s2] = s3)
  (format 001101 s3(25-21) s1(21-16) s2(15-0))
)
( opcode vectMul
  (operands (s1 mem) (s2 mem) (dst mem) (length imm) )
  (behavior dst = s1 * s2)
)
...
# Operation Mapping
( target
  ( (addsub dest src1 src2 src3) )
)
( generic
  ( (add temp src1 src2) (sub dest src3 temp) )
)
...
```

Fig. 5. Specification of Behavior using EXPRESSION ADL

The ADL captures the mapping between the structure and the behavior (and vice versa). For example, the *add* and *sub* instructions are mapped to the *ALU* unit, the *load* and *store* instructions are mapped to the *MemCntrl* unit, and so on. The ADL also captures other information such as mapping between the target and generic instruction set to enable retargetable compiler/simulator generation. For example, the target instruction *addsub* in Figure 2.0.2 is composed of generic instructions *add* and *sub*. The detailed description of the EXPRESSION ADL is available in [**?**].

3. VALIDATION OF ADL SPECIFICATION

After specification of full programmable SoC architecture in an ADL, the next step is to verify the specification to ensure the correctness of the specified architecture. Although many challenges exist in specification validation, a particular challenge is to verify the pipeline behavior in the presence of hazards and multiple exceptions. There are many important properties that need to be verified to validate the pipeline behavior. For example, it is necessary to verify that each operation in the instruction set can execute correctly in the processor pipeline. It is also necessary to ensure that execution of each operation is completed in a finite amount of time. Similarly, it is important to verify the execution style (e.g., in-order execution) of the architecture.

We have developed validation techniques to ensure that the architectural specifi-

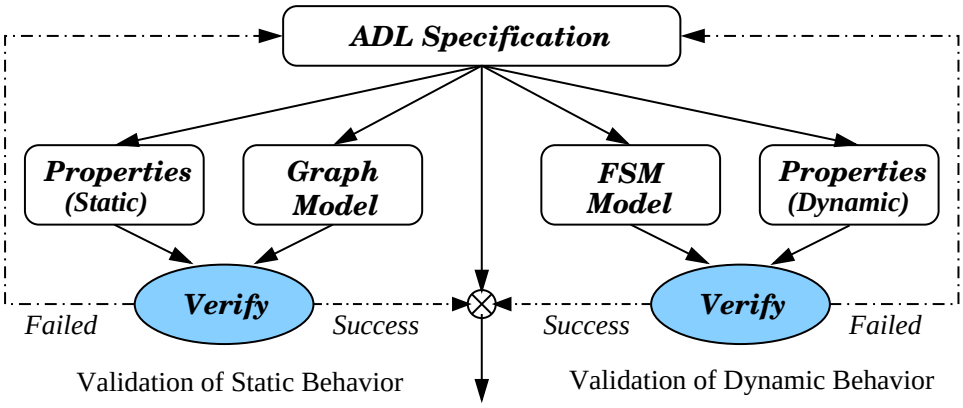

Fig. 6. Validation of ADL Specification

cation is well formed by analyzing both static and dynamic behaviors of the specified architecture. Figure 6 shows the flow for verifying the ADL specification [**?**]. The graph model as well as the FSM model of the architecture are generated from the specification. We have developed algorithms to verify several architectural properties, such as connectedness, false pipeline and data-transfer paths, completeness, and finiteness [**?**]. The dynamic behavior is verified by analyzing the instruction flow in the pipeline using a finite-state machine (FSM) based model to validate several important architectural properties such as determinism and in-order execution in the presence of hazards and multiple exceptions [**?**; **?**]. The property checking framework determines if all the necessary properties are satisfied. In case of a failure, it generates traces so that a designer can modify the architecture specification.

4. RETARGETABLE COMPILER GENERATION

As mentioned earlier, embedded systems are characterized by stringent multi-dimensional constraints. To meet all the design constraints together, embedded systems very often have non-regular architectures. Traditional architectural features employed in high-performance computer systems need to be customized to meet the power, performance, cost, weight needs of the embedded system. For example, an embedded system may not be able to implement complete register bypassing because of its impact on the area, power and complexity of the system. As a result the embedded system designer may opt for partial bypassing, which can be customized to meet the system constraints. Further customization is possible in embedded systems due to their application specific nature. Some architectural features, which are not "very" useful for the given application set may be absent in the embedded processor.

In this highly-customized world of embedded architectures, the role of the compiler is very crucial and important. The lack of regularity in the architecture, poses significant challenges for compilers to exploit these features. However, if the compiler is able to exploit these architectural features, it can have a significant impact on the power, performance and other constraints of the whole system. As a result, compiler development is a very important phase of embedded processor design. A

lot of time and effort of experienced compiler-writers is invested in developing an optimizing compiler for the embedded processor. Given the significance of the compiler on the processor power and performance, it is only logical that the compiler must play an important role in embedded processor design.

Although a compiler's effects can be incorporated into the design of an embedded processor, this process is often ad-hoc in nature and may result in conservative, or worse yet, erroneous exploration results. For example, designers often use the code generated by the "old compiler", or "hand-generated" code to test the new processor designs. This code should faithfully represent the code that the future compiler will generate. However, this approximation, or "hand-tuning" generates many inaccuracies in design exploration, and as a result, may lead to sub-optimal design decisions. A systematic way to incorporate compiler hints while designing the embedded processor is needed. Such a schema is called a Compiler-Assisted, or Compiler-in-the-Loop design methodology. The key enabler of the "Compiler-in-the-Loop" methodology is an ADL-driven retargetable compiler. While a conventional compiler takes only the application as input and generates the executable, a retargetable compiler also takes the processor architecture description as an input. The retargetable compiler can therefore exploit the architectural features present in the described system, and generate code tuned to for the specific architecture.

Whereas there has been a wealth of research on retargetable compilers[**?**; **?**], contemporary research on ADL-driven retargetable compilers has focused on both the design abstraction levels: architecture (instruction-set) and micro-architecture (implementation, such as pipeline structure). Traditionally there has been more extensive research on the architecture-level retargetability. Our EXPRESSION-based compiler-in-the-loop exploration framework employs and advances compiler retargetability at both the abstraction levels. At the Processor Pipeline (microarchitecture) level , decisions on which bypasses should be present in the processor greatly impacts the power, performance and the complexity of the processor. Indeed, our recent research results [**?**], [**?**], [**?**] show that deciding the bypasses in the processor by a traditional "simulation-only" exploration leads to incorrect design decisions and may lead to sub-optimal designs. A Compiler-in-the-Loop exploration can be used to suggest pareto-optimal bypass configurations.

In this section we will focus on the use of our "Compiler-in-the-Loop" exploration methodology at the architectural level, investigate Instruction-Set Architecture extensions for code size reduction.

4.1 Instruction Set Level Retargetability (rISA)

rreduced bit-width **I**nstruction **S**et **A**rchitecture, (**rISA**) is a popular architectural feature to reduce the code size in embedded systems. Architectures with rISA support two instruction sets, one is the "normal" set, which contains the original 32-bits instructions, and the other is the "reduced bit-width" instruction set, which encodes the most commonly used instructions in 16-bit instructions. If an application is fully expressed in terms of "reduced bit-width" instructions, then 50% code size reduction is achieved as compared to when it is expressed in terms of normal instructions.

Several embedded processors support this feature. For instance, the ARM processor supports rISA with 32-bit normal instructions and narrow 16-bit instructions.

While the normal 32-bit instructions comprise the *ARM* instruction set, the 16-bit instruction form the *Thumb* Instruction Set. Other processors with a similar feature include the MIPS32/16 [?], TinyRISC [?], STMicro's ST100 [?] and the ARC Tangent [?] processor.

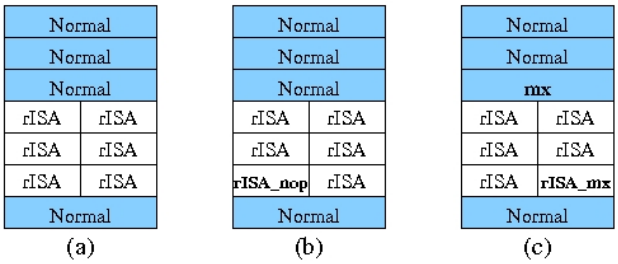

Fig. 7. rISA normal instructions co-reside in memory

The code for a rISA processor contains both normal and rISA instructions, as shown in Figure 7 from [?]. The fetch mechanism of the processor is oblivious of the mode of execution of the processor: regardless of the processor executing rISA instruction, or normal instruction, the instruction fetch of the processor remains unchanged.

The processor dynamically converts the rISA instructions to normal instructions before or during the instruction decode stage. Figure 8 shows the dynamic translation of Thumb instructions to ARM instructions, as described in [?].

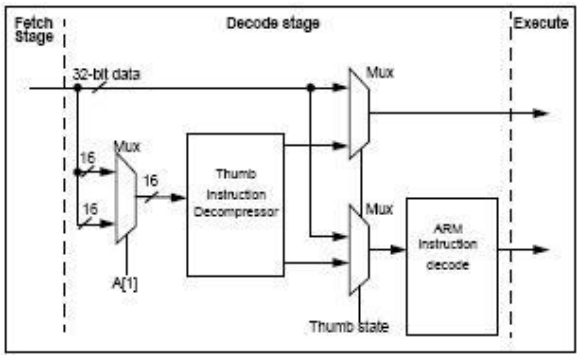

Fig. 8. Translation of Thumb instruction to ARM instruction

Typically, each rISA instruction has an equivalent instruction in the normal instruction set. This is done to ensure that the translation from a rISA instruction to a normal instruction (which has to be performed dynamically) is simple, and can be done without performance penalty.

After conversion, the instruction executes as a normal instruction, and therefore no other hardware change in required. Thus, the main advantage of rISA lies in achieving good code size reduction with minimal hardware additions.

However, some instructions, (for example, an instruction with a long immediate operand), cannot be mapped into a single rISA instruction. It takes more than one rISA instructions to encode the normal instruction. In such cases, more rISA instructions are required to implement the same task. As a result, rISA code has slightly lower performance as compared to the normal code.

The rISA instruction-set (IS), because of bit-width restrictions, can encode only a subset of the normal instructions and allows access to only a small subset of registers. Contemporary rISA processors (such as ARM/Thumb, MIPS32/16) incorporate a very simple rISA model with rISA instructions that can access 8 registers (out of 16 or 32 general-purpose registers). Owing to tight bit-width constraints in rISA instructions, they can use only very small immediate values. Furthermore, existing rISA compilers support the conversion only at a routine-level (or function-level) of granularity. Such severe restrictions make the code-size reduction obtainable by using rISA very sensitive to the compiler quality and the application features. For example, if the application has high register pressure, or if the compiler does not do a good job of register allocation, it might be better to increase the number of accessible registers at the cost of encoding only a few opcodes in rISA. Thus, it is very important to perform our "Compiler-in-the-Loop" design space exploration (DSE) while designing rISA architectures.

4.1.1 *rISA Model.* The rISA model defines the rISA IS, and the mapping of rISA instructions to normal instructions. Although typically each rISA instruction maps to only one normal instruction, there may be instances where multiple rISA instructions map to a single normal instruction.

rISA processors can operate in two modes: "rISA mode" and "normal mode". Most rISA processors (e.g. ARM) have a mode bit in the CPSR (Current Process State Register), which identifies whether the processor is in rISA mode or normal mode. When the processor is in rISA mode, it breaks the instruction into two halves and decodes them separately and sequentially. The processor needs to find out whether the instructions it is receiving are normal instructions or rISA instructions. Many rISA processors accomplish this by using explicit instructions that change the mode of execution. We label an instruction in the normal IS that changes mode from normal to rISA the *mx* instruction, and an instruction in the rISA instruction set that changes mode from rISA to normal the *rISA_mx* instruction. Every sequence of rISA instructions starts with an mx instruction and ends in a rISA_mx instruction. Such a sequence of rISA instructions is termed as *rISABlock*.

In order to avoid changing the instruction fetch mechanism of rISA processors, it is important to ensure that all the normal instructions align to the word boundary. However, an odd number of instructions in a rISABlock result in the ensuing normal instruction not being aligned to the word boundary. Therefore a padding *rISA_nop* instruction is required to force the alignment to the word boundary.

Due to bit-width constraints, a rISA instruction can access only a subset of registers. The register accessibility of each register operand must be specified in the rISA model. The width of immediate fields must also be specified. In addition, there may be special instructions in the rISA model to help the compiler generate better code. For example, a very useful technique to increase the number of registers accessible in rISA mode is to implement a rISA move instruction that can access all

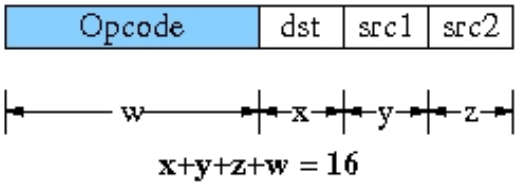

Fig. 9. Bitwidth constraints on rISA instructions

registers [2]. Another technique to increase the size of the immediate operand value is to implement a rISA extend instruction that completes the immediate field of the succeeding instruction. Numerous such techniques can be explored to increase the efficacy of rISA architectures. Next we describe some of the more important rISA design parameters that can be explored using our framework.

4.1.2 *rISA Design Parameters.* The most important consequence of using a rISA instructions is the limited number of bits available to encode the opcode, the register operands and the immediate values – resulting in a large space of alternative encodings that the rISA designer needs to explore and evaluate. For instance, most existing architectures the register operand field in the rISA instructions is 3-bit wide, permitting access to only 8 registers. For 3-address instructions, $(3 \times 3) = 9$ bits are used in specifying the operands. Therefore only (16 - 9) = 7 bits are left to specify the opcode. As a result only $2^7 = 128$ opcodes can be encoded in rISA. The primary advantage of this approach is the huge number of opcodes available. Using such a rISA most normal 32-bit instructions can be specified as rISA instructions. However, this approach suffers from the drawback of increased register pressure possibly resulting in poor code size.

One modification is to increase the number of registers accessible by rISA instructions to 16. However, in this model, only a limited number of opcodes are available. Thus, depending on the application, large sections of program code might not be implementable using rISA instructions. The design parameters that can be explored include the number of bits used to specify operands (and opcodes), and the type of opcodes that can be expressed in rISA.

Another important rISA feature that impacts the quality of the architecture is the "implicit operand format" feature. In this feature, one (or more) of the operands in the rISA instruction is hard-coded (i.e. implied). The implied operand could be a register operand, or a constant. In case a frequently occurring format of add instruction is *add* R_i R_i R_j (where the first two operands are the same), a rISA instruction $rISA\_add1$ R_i R_j , can be used. In case an application that access arrays produces a lot of instructions like $addr = addr + 4$ then a rISA instruction $rISA_{add4}$ *addr* which has only one operand might be very useful. The translation unit, while expanding the instruction, can also fill in the missing operand fields. This is a very useful feature that can be used by the compiler to generate high

[2]This is possible because a move has only two operands and hence has more bits to address each operand.

quality code.

Another severe limitation of rISA instructions is their inability to incorporate large immediate values. For example, with only 3 bits available for operands, the maximum unsigned value that can be expressed is 7. Thus, it might be useful to vary the size of the immediate field, depending on the application and the values that are (commonly) generated by the compiler. However, increasing the size of the immediate fields will reduce the number of bits available for opcodes (and also the other operands). This trade-off can be meaningfully made only with a Compiler-in-the-Loop DSE framework.

Various other design parameters such as partitioned register files, shifted/padded immediate fields, etc. should also be explored in order to generate a rISA architecture that is tuned to the needs of the application and to the compiler quality. While some of these design parameters have been studied in a limited context, there is a critical need for an ADL-driven framework that uses architecture-aware compiler to exploit and combine all of these features. We now describe our rISA compiler framework, which enables compiler-in-the-loop exploration framework to quantify the impact of these features.

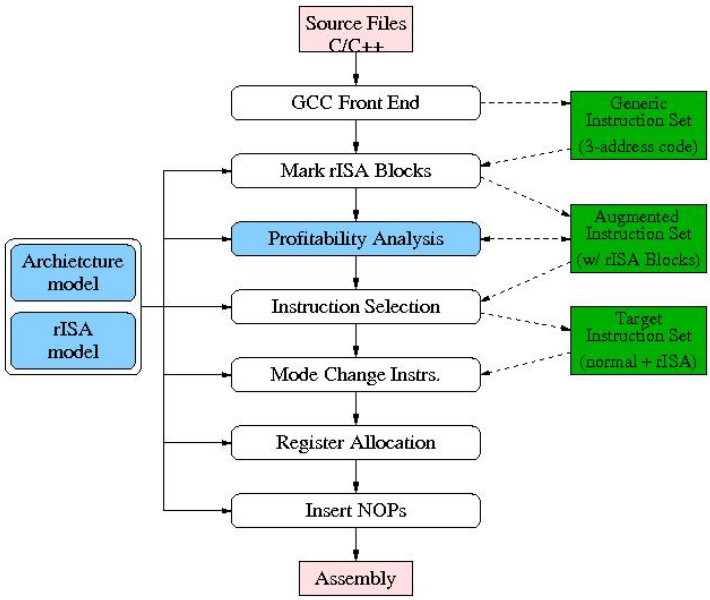

Fig. 10. rISA Compiler flow for DSE

4.1.3 *rISA Compiler.* Figure 10 shows the phases of our EXPRESS compiler [**?**] that are used to perform rISAization. rISAization is the process of converting normal instruction to rISA instructions. We use the GCC front end to output a sequence of generic three-address MIPS instructions, which in turn are used to

generate the CFG, DFG and other compiler data structures comprising the internal representation of the application.

Mark rISA Blocks. Due to the restrictions discussed in the previous subsection, several types of instructions, such as those with many many operands, instructions with long immediate values etc. may not be convertible to rISA instructions. The first step in compilation for a rISA processor marks all the instructions that can be converted into rISA instructions. A contiguous list of marked instructions in a basic block is termed a **rISABlock**. Owing to the overhead associated with rISAization, it may not be profitable to convert all rISABlocks into rISA instructions.

Profitability Analysis. This step decides which rISABlocks to convert into rISA instructions, and which ones to leave in terms of normal instructions. A mode change instruction is needed at the beginning and at the end of each rISA Block. Further, in order to adhere with the word boundary, there should be an even number of instructions in each rISABlock. Therefore, if a rISABlock is very small, then the mode change instruction overhead could negate gains from code compression achieved through rISAization. It should be noted here that if all the predecessors of a basic block are also decided to be encoded in rISA mode, then the mode change instructions may not be needed. We will perform and discuss such optimizations in a later stage.

Similarly if the rISABlock is very big, the increased register pressure (and the resulting register spills) could render rISAization unprofitable. Thus an accurate estimation of code size and performance trade-off is necessary before rISAizing a rISABlock. In our technique, the impact of rISAization on code size and performance is estimated using a profitability analysis function.

The profitability analysis function estimates the difference in code size and performance if the block were to be implemented in rISA mode as compared to normal mode. The compiler (or user) can then use these estimates to trade-off between performance and code size benefits for the program. The profitability heuristic is described in greater detail in [**?**].

Instruction Selection. Our compiler uses a tree pattern matching based algorithm for Instruction Selection. A tree of generic instructions is converted into a tree of target instructions. In case a tree of generic instructions can be replaced by more than one target instruction tree, then one with lower cost is selected. The cost of a tree depends upon the user's relative importance of performance and code-size.

Normally, during instruction selection, a tree of generic instructions has trees of target instructions as possible mappings. Our rISA framework provides trees of rISA instructions also as possible mapping. As a result the instruction selection to rISA instructions becomes a natural part of the normal instruction selection process. The Instruction Selection phase uses a profitability heuristic to guide the decisions of which section of a routine to convert to rISA instructions, and which ones to normal target instructions. All generic instructions within profitable rISABlocks are replaced with rISA instructions and all other instructions are replaced with normal target instructions.

Replacing a generic instruction with a rISA instruction involves two steps, first converting to the appropriate rISA opcode, and second, restricting the operand

variables to the set of rISA registers. This is done by adding a restriction on the variable, which implies that this variable can be mapped to the set of rISA registers only.

Mode Change Instructions. After Instruction Selection the program contains sequences of normal and rISA instructions. A list of contiguous rISA instructions may span across basic block boundaries. To ensure correct execution, we need to make sure that whenever there is a switch in the instructions from normal to rISA or vice versa (for any possible execution path), there is an explicit and appropriate mode change instruction. There should be a *mx* instruction when the instructions change from normal to rISA instructions, and a *rISA_mx* instruction when the instructions change from rISA instructions to normal instructions. If the mode of instructions change inside a basic block, then there is no choice but to add the appropriate mode change instruction at the boundary. However when the mode changes at basic block boundary, the mode change instruction can be added at the beginning of the successor basic block or at the end of the predecessor basic block. The problem becomes more complex if there are more than one successors and predecessors at the junction. In such a case, the mode change instructions should be inserted so as to minimize the performance degradation, i.e., in the basic blocks which execute the least. We use a profile based analysis to find where to insert the mode change instructions [**?**].

Register Allocation. Our EXPRESS compiler implements a modified version of Chaitin's solution [**?**] to Register Allocation. Registers are grouped into (possibly overlapping) register classes. Each program variable is then mapped to the appropriate register class. For example, operands of a rISA instruction belong to the rISA register class (which consists of only a subset of the available registers). The register allocator then builds the interference graph and colors it honoring the register class restrictions of the variables. Since code blocks that have been converted to rISA typically have a higher register pressure (due to limited availability of registers), higher priority is given to rISA variables during register allocation.

Insert NOPs. The final step in code generation is to insert *rISA_nop* instruction in *rISABlocks* that have an odd number of rISA instructions.

We have briefly described how the EXPRESSION ADL-driven EXPRESS compiler can be used to generate architecture-sensitive code for rISA architectures by considering the compiler effects during DSE, the designer is able to accurately estimate the impact of the various rISA features. Section 6 presents some exploration results for rISA exploration using the MIPS 32/16-bit rISA architecture.

5. RETARGETABLE SIMULATOR GENERATION

Simulators are indispensable tools in the development of new architectures. They are used to validate an architecture design, a compiler design as well as to evaluate architectural design decisions during design space exploration. Running on a host machine, these tools mimic the behavior of an application program on a target machine. These simulators should be fast to handle the increasing complexity of processors, flexible to handle all features of applications and processors, e.g., runtime self modifying codes, and retargetable to support a wide spectrum of architectures.

The performance of simulators vary widely depending on the amount of information it captures (abstraction level), as determined by its abstraction level. *Functional* simulators only capture the instruction-set details. The *Cycle-accurate* simulators capture both instruction-set and micro-architecture details. As a result, cycle-accurate simulators are slower than functional simulators but provide timing details for the application program. Similarly, *bit-* and *phase-accurate* simulators model the architecture more accurately at the cost of simulation performance.

Simulators can be further classified based on the model of execution: *interpretive* and *compiled*. Interpretive simulation is flexible but slow. In this technique, an instruction is fetched, decoded, and executed at run time. Instruction decoding is a time consuming process in a software simulation. Compiled simulation performs compile time decoding of application program to improve the simulation performance. The performance improvement is obtained at the cost of flexibility. Compiled simulators rely on the assumption that the complete program code is known before the simulation starts and is furthermore run-time static. Compiled simulators are not applicable in embedded systems that support runtime self modifying codes or multi-mode processors. There are various ADL-driven simulator generation frameworks in the literature including GENSIM using ISDL [**?**], MSSB/MSSU using MIMOLA [**?**], CHECKERS using nML [**?**], SIMPRESS/RexSim using EXPRESSION [**?**], HPL-PD using MDES [**?**], and fast simulators using LISA [**?**] and RADL [**?**]. This section briefly describes retargetable simulator generation using the EXPRESSION ADL for instruction-set architecture (ISA) simulation as well as cycle-accurate simulation.

5.1 Instruction-Set Simulation

In a retargetable ISA simulation framework, the range of architectures that can be captured and the performance of the generated simulators depend on three issues: first, the model based on which the instructions are described; second, the decoding algorithm that uses the instruction model to decode the input binary program; and third, the execution method of decoded instructions. These issues are equally important and ignoring any of them results in a simulator that is either very general but slow or very fast but restricted to some architecture domain. However, the instruction model significantly affects the complexity of decode and the quality of execution. We have developed a generic instruction model coupled with a simple decoding algorithm that lead to an efficient and flexible execution of decoded instructions [**?**; **?**].

Figure 11 shows our retargetable simulation framework that uses the ADL specification of the architecture and the application program binary (compiled by gcc) to generate the simulator [**?**]. Section 5.1.1 presents our generic instruction model that is used to describe the binary encoding and the behavior of instructions [**?**]. Section 5.1.2 describes the *instruction decoder* that decodes the program binary using the description of instructions in the ADL. The decoder generates the optimized source code of the decoded instructions [**?**]. The *structure generator* generates the structural information to keep track of the state of the simulated processor. The target independent components are described in the *library*. This library is finally combined with the *structural information* and the *decoded instructions* and is compiled on the host machine to get the final ISA simulator.

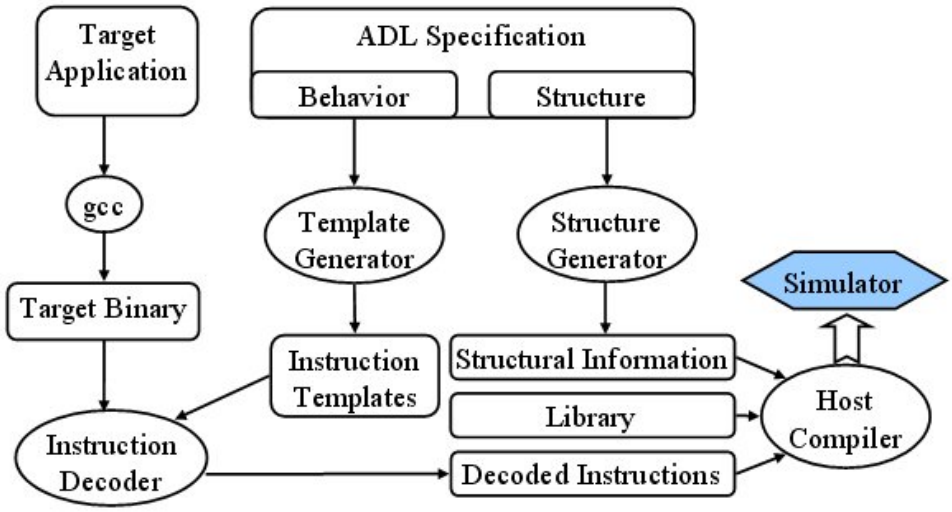

Fig. 11. ADL-driven Instruction-Set Simulator Generation

5.1.1 *Generic Instruction Model.* The focus of this model is on the complexities of different instruction binary formats in different architectures [**?**]. As an illustrative example, we model the integer arithmetic instructions of the Sparc V7 processor which is a single-issue processor with 32-bit instruction [**?**]. The integer-arithmetic instructions, *IntegerOps* (as shown below), perform certain arithmetic operations on two source operands and write the result to the destination operand. This subset of instructions is distinguished from the others by the following bit mask:

```
Bitmask:     10xxxxx0  xxxxxxxx  xxxxxxxx  xxxxxxxx
IntergerOps: < opcode     dest      src1     src2 >
```

A bit mask is a string of '1', '0' and 'x' symbols and it matches the bit pattern of a binary instruction if and only if for each '1' or '0' in the mask, the binary instruction has a 1 or a 0 value in the corresponding position respectively. The 'x' symbol matches with both 1 and 0 values. In this model, an *instruction* of a processor is composed of a series of slots, $I =< sl_0, sl_1, \ldots >$ and each slot contains only one operation from a subset of operations. All the operations in an instruction execute in parallel. Each operation is distinguished by a mask pattern. Therefore, each slot (sl_i) contains a set of operation-mask pairs (op_i, m_i) and is defined by the format: $sl_i =< (op_i^0, mi_0)|(op_i^1, mi_1)| \ldots >$.

An operation class refers to a set of similar operations in the instruction set that can appear in the same instruction slot and have similar format. The previous slot description can be rewritten using an operation class clOps: $sl_i =< (clOps_i, m_i) >$. For example, integer arithmetic instructions in Sparc V7 can be grouped in a class (IntegerOps) as: $I_{SPARC} =< (IntegerOps, 10xx - xxx0xxxx - xxxxxxxx - xxxxxxxx - xxxx)| \ldots >$. An operation class is composed of a set of symbols and

an expression that describes the behavior of the operation class in terms of the values of its symbols. For example, the operation class has four symbols: opcode, dest, src1 and src2. The expression for this example will be: dest = f_{opcode}(src1, src2). Each symbol may have a different type depending on the bit pattern of the operation instance in the program. For example, the possible types for src2 symbol are register and immediate integer. The value of a symbol depends on its type and can be static or dynamic. For example, the value of a register symbol is dynamic and is known only at run time, whereas the value of an immediate integer symbol is static and is known at compile time. Each symbol in an operation has a possible set of types. A general operation class is then defined as: $clOps =< (s_0, T_0), (s_1, T_1), \ldots | exp(s_0, s_1, \ldots) >$, where ($s_i$, T_i) are (symbol, type) pairs and exp(s_0, s_1,...) is the behavior of the operations based on the values of the symbols.

The type of a symbol can be defined as a register ($\in$ *Registers*) or an immediate constant ($\in$ *Constants*) or can be based on certain micro-operations ($\in$ *Operations*). For example, a data processing instruction in ARM (e.g., add) uses shift (micro-operation) to compute the second source operand, known as Shifter-Operand. Each possible type of a symbol is coupled with a mask pattern that determines what bits in that operation must be checked to find out the actual type of the corresponding symbol. Possible types of a symbol are defined as: $T = \{(t, m) | t \in Operations \cup Registers \cup Constants, m \in (1|0|x)*\}$. For example, the opcode symbol can be any of valid integer arithmetic operations (*OpTypes*) as shown in Figure 12.

Note that this provides more freedom for describing the operations because here the symbols are not directly mapped to some contiguous bits in the instruction and a symbol can correspond to multiple bit positions in the instruction binary. The actual register in a processor is defined by its class and its index. The index of a register in an instruction is defined by extracting a slice of the instruction bit pattern and interpreting it as an unsigned integer. An instruction can also use a specific register with a fixed index, as in a branch instruction that update the program counter. A register is defined by: $r = [regClass, i, j]|[regClass, index]$, where i and j define the boundary of index bit slice in the instruction. For example, if the dest symbol is from the 25th to 29th bits in the instruction, and is an integer register, its type can be described as: $DestType = [IntegerRegClass, 29, 25]$. Similarly a portion of an instruction may be considered as a constant. For example, one bit in an instruction can be equivalent to a Boolean type or a set of bits can make an integer immediate. It is also possible to have constants with fixed values in the instructions. A constant type is defined by where i and j show the bit positions of the constant and type is a scalar type such as integer, Boolean, float, etc. The complete description of integer-arithmetic instructions in SPARC processor is shown in Figure 12.

5.1.2 *Generic Instruction Decoder.* A key requirement in a retargetable simulation framework is the ability to automatically decode application binaries of different processors architectures. This section describes the generic instruction decoding technique that is customizable depending on the instruction specifications captured through our generic instruction model.

```
SPARCInst = $ (InegerOps, 10xx-xxx0 xxxx-xxxx xxxx-xxxx xxxx-xxxx) | .... $;
IntegerOp = <
    (opcode, OpTypes), (dest, DestType), (src1, Src1Type),  (src2, Src2Type)
    | {    dest = opcode(src1, src2); }
>;
OpTypes = {
    (Add, xxxx-xxxx 0000-xxxx xxxx-xxxx xxxx-xxxx),
    (Sub, xxxx-xxxx 0100-xxxx xxxx-xxxx xxxx-xxxx),
    (Or , xxxx-xxxx 0010-xxxx xxxx-xxxx xxxx-xxxx),
    (And, xxxx-xxxx 0001-xxxx xxxx-xxxx xxxx-xxxx),
    (Xor, xxxx-xxxx 0011-xxxx xxxx-xxxx xxxx-xxxx),
    ....
};
DestType = [IntegerRegClass, 29, 25];
Src1Type = [IntegerRegClass, 18, 14];
Src2Type = {
    ([IntegerRegClass,4,0], xxxx-xxxx xxxx-xxxx xx0x-xxxx xxxx-xxxx),
    (\#int,12,0\#, xxxx-xxxx xxxx-xxxx xx1x-xxxx xxxx-xxxx)
};
```

Fig. 12. Description of integer-arithmetic instructions in SPARC processor

Algorithm 1 describes how the instruction decoder works [**?**]. This algorithm accepts the target program binary and the instruction specification as inputs and generates a source file containing decoded instructions as output. Iterating on the input binary stream, it finds an operation, decodes it using Algorithm 2, and adds the decoded operation to the output source file [**?**]. Algorithm 2 also returns the length of the current operation that is used to determine the beginning of the next operation. Algorithm 2 gets a binary stream and a set of specifications containing operation or micro-operation classes. The binary stream is compared with the elements of the specification to find the specification-mask pair that matches with the beginning of the stream. The length of the matched mask defines the length of the operation that must be decoded. The types of symbols are determined by comparing their masks with the binary stream. Finally, using the symbol types, all symbols are replaced with their values in the expression part of the corresponding specification. The resulting expression is the behavior of the operation. This behavior and the length of the decoded operation are produced as outputs.

Consider the following SPARC Add operation example and its binary pattern:

```
Add g1, \#10, g2   1000-0100  0000-0000  0110-0000  0000-1010
```

Using the specifications described in Section 5.1.1, in the first line of Algorithm 2, the (InegerOps, 10xx-xxx0 xxxx-xxxx xxxx-xxxx xxxx-xxxx) pair matches with the instruction binary. This means that the *IntegerOps* operation class matches this operation. Next the values of the four symbols are determined: opcode, dest, src1, src2. Symbol opcode's type is OpTypes in which the mask pattern of Add matches the operation pattern. So the value of opcode is Add function. Symbol dest's type is DestType which is a register type. It is an integer register whose index is bits 25th to 29th (00010), i.e. 2. Similarly, the values for the symbols src1 and src2 can be computed. By replacing these values in the expression part of the IntegerOps the final behavior of the operation would be: g2 = Add(g1, 10); which

```
Algorithm 1: StaticInstructionDecoder
Inputs: i)  Target Program Binary (Application)
        ii) Instruction Specifications (InstSpec)
Output: Decoded Program DecodedOperations
Begin
    Addr = Address of first instruction in Application
    DecodedOperations={};
    while (Application not processed completely)
        BinStream = Binary stream in Application starting at Addr;
        (Exp, AddrIncrement) = DecodeOperation (BinStream, InstSpec);
        DecodedOperations  = DecodedOperations  U <Exp, Addr>;
        Addr = Addr + AddrIncrement;
    endwhile;
    return DecodedOperations ;
End;
```

means g2 = g1 + 10.

```
Algorithm 2: DecodeOperation
Input: Binary Stream (BinStream), Specifications (Spec)
Output: Decoded Expression (Exp), Length (DecodedStreamSize)
Begin
    (OpDesc, OpMask) = findMatchingPair(Spec, BinStream)
    OpBinary = initial part of BinStream whose length is equal to OpMask
    Exp = the expression part of OpDesc;
    foreach pair of (s, T) in the OpDesc
        Find t in T whose mask matches the OpBinary;
        v = ValueOf(t, OpBinary);
        Replace s with v in Exp;
    endfor
    return (Exp , size(OpBinary));
End;
```

5.2 Cycle-Accurate Simulation

Automatic simulator generation for a class of architecture has been addressed in the literature. However, it is difficult to generate simulators for a wide variety of processor and memory architectures including RISC, DSP, VLIW, superscalar, and hybrid. The main bottleneck is the lack of an abstraction (covering a diverse set of architectural features) that permits the reuse of the abstraction primitives to compose the heterogeneous architectures. In this section, we describe our simulator generation approach based on functional abstraction. Section 5.2.1 presents functional abstraction needed to capture a wide variety of architectural features. Section 5.2.2 briefly describes ADL-driven simulator generation using the functional abstraction.

5.2.1 *Functional Abstraction.* In order to understand and characterize the diversity of contemporary architectures, we have surveyed the similarities and differences of each architectural feature in different architecture domains [**?**]. Broadly speaking, the structure of a processor consists of functional units, connected using

ports, connections and pipeline latches. Similarly, the structure of a memory subsystem consists of SRAM, DRAM, cache hierarchy, and so on. Although a broad classification makes the architecture look similar, each architecture differs in terms of the algorithm it employs in branch prediction, the way it detect hazards, the way it handle exceptions, and so on. Moreover, each unit has different parameters for different architectures (e.g., number of fetches per cycle, levels of cache, and cache line size). Depending on the architecture, a functional unit may perform the same operation at different stages in the pipeline. For example, read-after-write(RAW) hazard detection followed by operand read happen in the decode unit for some architectures (e.g., DLX [**?**]), whereas in some others these operations are performed in the issue unit (e.g., MIPS R10K). Some architectures even allow operand read in the execution unit (e.g., ARM7). We observed some fundamental differences from this study; the architecture may use:

—the same functional or memory unit with different parameters

—the same functionality in different functional or memory unit,

—new architectural features

The first difference can be eliminated by defining generic functions with appropriate parameters. The second difference can be eliminated by defining generic sub-functions, which can be used by different architectures at different stages in the pipeline. The last one is difficult to alleviate since it is new, unless this new functionality can be composed of existing sub-functions. Based on our observations we have defined the necessary generic functions, sub-functions and computational environment needed to capture a wide variety of processor and memory features. In this section we present functional abstraction by way of illustrative examples. We first explain the functional abstraction needed to capture the structure and behavior of the processor and memory subsystem, then we discuss the issues related to defining generic controller functionality, and finally we discuss the issues related to handling interrupts and exceptions.

We capture the structure of each functional unit using parameterized functions. For example, a fetch unit functionality contains several parameters, viz., number of operations read per cycle, number of operations written per cycle, reservation station size, branch prediction scheme, number of read ports, number of write ports etc. These ports are used to connect different units. Figure 13 shows a specific example of a fetch unit described using sub-functions. Each sub-function is defined using appropriate parameters. For example, *ReadInstMemory* reads n operations from instruction cache using current PC address (returned by *ReadPC*) and writes them to the reservation station. The fetch unit reads m operations from the reservation station and writes them to the output latch (fetch to decode latch) and uses BTB based branch prediction mechanism. We have defined parameterized functions for all functional units present in contemporary programmable architectures viz., fetch unit, branch prediction unit, decode unit, issue unit, execute unit, completion unit, interrupt handler unit, PC Unit, Latch, Port, Connection, and so on. We have also defined sub-functions for all the common activities e.g., ReadLatch, WriteLatch, ReadOperand, and so on.

The behavior of a generic processor is captured through the definition of opcodes. Each opcode is defined as a function, with a generic set of parameters, which per-

```
FetchUnit ( # of read/cycle, res-station size, ....)
{
   address = ReadPC();
   instructions = ReadInstMemory(address, n);
   WriteToReservationStation(instructions, n);
   outInst = ReadFromReservationStation(m);
   WriteLatch(decode_latch, outInst);

   pred = QueryPredictor(address);
   if pred {
      nextPC = QueryBTB(address);
      SetPC(nextPC);
   } else
      IncrementPC(x);
}
```

Fig. 13. A Fetch Unit Example

forms the intended functionality. The parameter list includes source and destination operands, necessary control and data type information. We have defined common sub-functions e.g., ADD, SUB, SHIFT, and so on. The opcode functions may use one or more sub-functions. For example, the MAC (multiply and accumulate) uses two sub-functions (ADD and MUL) as shown in Figure 14.

```
ADD (src1, src2) {
  return (src1 + src2);
}
```

```
MUL (src1, src2) {
  return (src1 * src2);
}
```

```
MAC (src1, src2, src3) {
   return ( ADD( MUL(src1, src2), src3) );
}
```

Fig. 14. Modeling of MAC operation

Each type of memory module, such as SRAM, cache, DRAM, SDRAM, stream buffer, and victim cache, is modeled using a function with appropriate parameters. For example, a typical cache function has many parameters including cache size, line size, associativity, word size, replacement policy, write policy, and latency. The cache function performs four operations: read, write, replace, and refill. These functions can have parameters for specifying pipelining, parallelism, access modes (normal read, page mode read, and burst read), and so on. Again, each function is composed of sub-functions. For example, an associative cache function can be modeled using a cache function.

A major challenge for functional abstraction of programmable architectures, is the modeling of control for a wide range of architectural styles. We define control

in both distributed and centralized manner. The distributed control is transfered through pipeline latches. While an instruction gets decoded the control information needed to select the operation, the source and the destination operands are placed in the output latch. These decoded control signals pass through the latches between two pipeline stages unless they become redundant. For example, when the value for *src1* is read that particular control is not needed any more, instead the read value will be in the latch. The centralized control is maintained by using a generic control table. The number of rows in the table is equal to the number of pipeline stages in the architecture. The number of columns is equal to the maximum number of parallel units present in any pipeline stage. Each entry in the control table corresponds to one particular unit in the architecture. It contains information specific to that unit e.g., busy bit (BB), stall bit (SB), list of children, list of parents, opcodes supported, and so on.

Another major challenge for functional abstraction is the modeling of interrupts and exceptions. We capture each exception using an appropriate sub-function. Opcode related exceptions (e.g., divide by zero) are captured in the opcode functionality. Functional unit related exceptions (e.g., illegal slot exception) are captured in functional units. External interrupts (e.g., reset, debug exceptions) are captured in the control unit functionality. The detailed description of functional abstraction for all the micro-architectural components is available in [**?**].

5.2.2 *Simulator Generation.* We have developed C++ models for the generic functions and sub-functions described in Section 5.2.1. The development of simulation models is a one-time activity and independent of the architecture. The simulator is generated by composing the abstraction primitives based on the information available in the ADL specification. The simulator generation process consists of three steps. First, the ADL specification is read to gather all the necessary details for the simulator generation. Second, the functionality of each component is composed using the generic functions and sub-functions. Finally, the structure of the processor is composed using the structural details. In the remainder of this section we briefly describe the last two steps.

To compose the functionality of each component, all necessary details (such as parameters and functionality) are extracted from the ADL specification. First, we describe how to generate three major components of the processor: instruction decoder, execution unit and controller, using the generic simulation models. Next, we describe how to compose the functionality A generic instruction decoder uses information regarding individual instruction format and opcode mapping for each functional unit to decode a given instruction. The instruction format information is available in ADL specification. The decoder extracts information regarding opcode and operands from input instruction using the instruction format. The mapping section of the ADL captures the information regarding the mapping of opcodes to the functional units. The decoder uses this information to perform/initiate necessary functions (e.g., operand read) and decide where to send the instruction.

To compose an execution unit, it is necessary to instantiate all the opcode functionalities (e.g, ADD, SUB etc. for an ALU) supported by that execution unit. The execution unit invokes the appropriate opcode functionality for an incoming operation based on a simple table look-up technique as shown in Figure 15. Also,

if the execution unit is supposed to read the operands, the appropriate number of operand read functionalities need to be instantiated unless the same read functionality can be shared using multiplexers. Similarly, if the execution unit is supposed to write the data back to register file, the functionality for writing the result needs to be instantiated. The actual implementation of an execute unit might contain many more functionalities such as read latch, write latch, modify reservation station (if applicable), and so on.

The controller is implemented in two parts. First, it generates a centralized controller (using a generic controller function with appropriate parameters) that maintains the information regarding each functional unit, such as busy, stalled etc. It also computes hazard information based on the list of instructions currently in the pipeline. Based on these bits and the information available in the ADL, it stalls/flushes necessary units in the pipeline. Second, a local controller is maintained at each functional unit in the pipeline. This local controller generates certain control signals and sets necessary bits based on the input instruction. For example, the local controller in an execute unit will activate the add operation if the opcode is *add*, or it will set the busy bit in case of a multi-cycle operation.

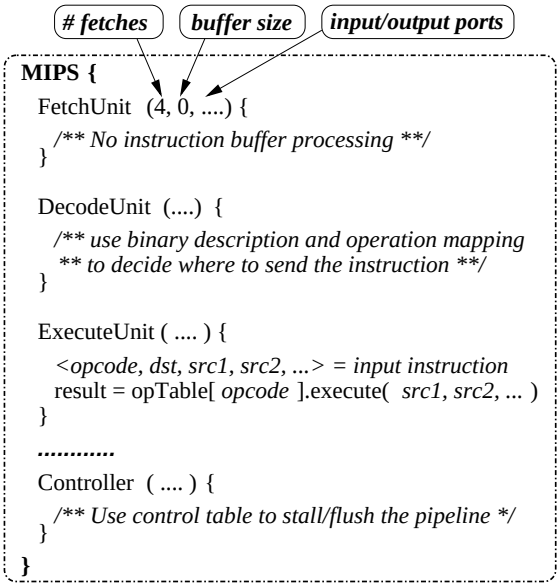

Fig. 15. An example simulation model for MIPS architecture

It is also necessary to compose the functionality of new instructions (behavior) using the functionality of existing instructions. This operation mapping (described in Section 2) is used to generate the functionality for the target (new) instructions using the the functionality of the corresponding generic instructions. The final implementation is generated by instantiating components (e.g., fetch, decode, register files, memories etc.) with appropriate parameters and connecting them using the information available in the ADL. For example, Figure 15 shows a portion of the simulation model for the MIPS architecture. The generated simulation models

combined with the existing simulation kernel creates a cycle-accurate structural simulator.

6. DESIGN SPACE EXPLORATION

Having established the need for, and some techniques using an ADL-driven approach for toolkit generation, we now show some sample results of design space exploration of programmable SoCs. We have performed extensive architectural design space exploration by varying different architectural features in the EXPRESSION ADL. We have also performed micro-architectural exploration of the MIPS 4000 processor in three directions: pipeline exploration, instruction-set exploration, and memory exploration [**?**]. Pipeline exploration allows addition (deletion) of new (existing) functional units or pipeline stages. Instruction-set exploration allows the addition of new instructions or formation of complex instructions by combining the existing instructions. Similarly, memory exploration allows modification of memory hierarchies and cache parameters. The system architect only modifies the ADL description of the architecture, and the software toolkit is automatically generated from the ADL specification using the functional abstraction approach. The public release of the retargetable simulation and exploration framework is available from *http://www.cecs.uci.edu/~express*. This release also supports a graphical user interface (GUI). The architecture can be described (or modified) using the GUI. The ADL specification and the software toolkit are automatically generated from the graphical description. In the rest of this section, we present sample experimental results demonstrating the need and usefulness of Compiler-in-the-Loop design space exploration of rISA designs which discussed in Section 4.

6.1 Instruction-Set Exploration (rISA Design Space)

The experimental results of the Compiler-in-the-Loop Design Space Exploration of rISA designs is divided into two main parts. First we demonstrate the goodness of our rISA compiler. We show that as compared to existing rISA compilers, our instruction-level, register pressure sensitive compiler can consistently achieve superior code compression over a set of benchmarks. In the second part, we develop rISA designs, and demonstrate that the code compression achieved by using different rISA designs is very sensitive on the rISA design itself. Depending on the specific rISA design chosen, there can be a difference of up to 2X in the code compression achievable. This clearly demonstrates the need to very carefully select the appropriate rISA design, and shows that a Compiler-in-the-Loop exploration greatly helps the designer choose the best rISA configuration.

6.1.1 *rISA Compiler Comparison.* For the baseline experiments, we compile the applications using GCC for MIPS32 ISA. Then we compile the same applications using GCC for MIPS32/16 ISA. We perform both the compilations using -Os flags with the GCC to enable all the code size optimizations. The percentage code compression achieved by GCC for MIPS16 is computed and is represented by the light bars in Figure 16 taken from [**?**]. The MIPS32 code generated by GCC is compiled again using the register pressure based heuristic in EXPRESS. The percentage code compression achieved by EXPRESS is measured and plotted as dark bars in Figure 16.

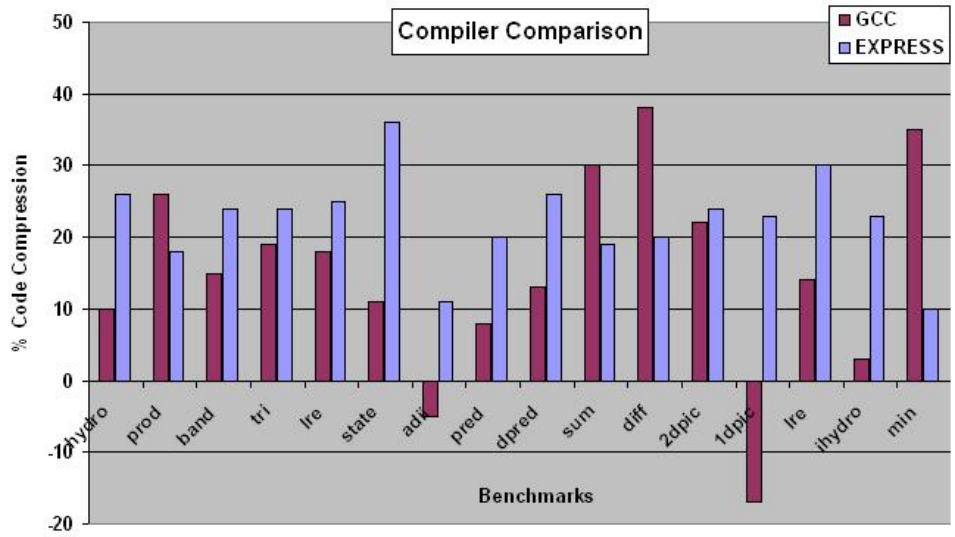

Fig. 16. Percentage code compressions achieved by GCC and EXPRESS for MIPS32/16

Figure 16 shows that the register pressure based heuristic performs consistently better than GCC and successfully prevents code size inflation. GCC achieves on an average 15% code size reduction, while EXPRESS achieved an average of 22% code size reduction. We simulated the code generated by EXPRESS on a variant of the MIPS R4000 processor that was augmented with the rISA MIPS16 Instruction Set. The memory subsystem was modeled with no caches and a single cycle main memory. The performance of MIPS16 code is on average 6% lower than that of MIPS32 code, with the worst case being 24% lower. Thus our technique is able to reduce the code size using rISA with a minimal impact on performance.

6.1.2 *Sensitivity on rISA Designs.* Owing to various design constraints on rISA, the code compression achieved by using a rISA is very sensitive on the rISA chosen. The rISA design space is huge and several instruction set idiosyncrasies make it very tough to characterize. To show the variation of code compression achieved with rISA, we take a practical approach. We systematically design several rISAs, and study the code compression achieved by them. Figure 17 taken from [**?**] plots the code compression achieved by each rISA design. We start with the extreme rISA designs, *rISA_7333* and *rISA_4444* and gradually improve upon them.

rISA_7333. The first rISA design is (*rISA_7333*). In this rISA, the operand is a 7-bit field, and each operand is encoded in 3-bits. Thus there can be $2^7 = 128$ instructions in this rISA, but each instruction can access only 8 registers. Furthermore, they can support unsigned immediate values from 0 to 7. However, instructions that have 2 operands (like move) have 5-bit operands. Thus they can access all 32 registers. Owing to the uniformity in the instruction format, the translation unit is very simple for this rISA design. The leftmost bar in Figure 17 plots the average code compression achieved when we use *rISA_7333* design on all our

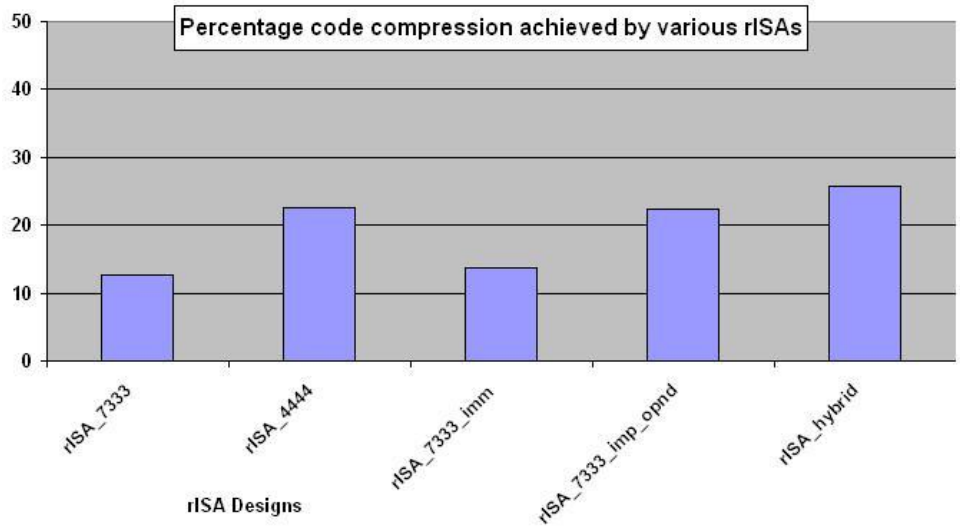

Fig. 17. Code size reduction for various rISA architectures

benchmarks. On average *rISA_7333* achieves 12% code compression. *rISA_7333* is unable to achieve good code compressions for applications that have high register pressure, e.g., *adii*, and those with large immediate values. In such cases, the compiler heuristic decides not to rISAize large portions of the application to avoid code size increase due to extra spill/reload and immediate extend instructions.

rISA_4444. *rISA_4444* is the instruction set on the other end of the rISA design spectrum. It provides only 4-bits to encode the operand, but has 4-bits to specify each operand as well. Thus there are $2^4 = 16$ instructions in *rISA_4444*, but each instruction can access $2^4 = 16$ registers. We profiled the applications and incorporated the 16 most frequently occurring instructions in this rISA. The second bar from the left in Figure 17 shows that the register pressure problem is mitigated in the *rISA_4444* design. It achieves better code size reduction for benchmarks that have high register pressure, but performs badly on some of the benchmarks because of its inability to convert all the normal instructions into rISA instructions. *rISA_4444* achieves about 22% improvement over the normal instruction set.

rISA_7333_imm. We now attack the second problem in *rISA_7333* – small immediate values. For instructions that have immediate values, we decrease the size of opcode, and use the bits to accommodate as large an immediate value as possible. This design point is called *rISA_7333_imm*. Because of the non-uniformity in the size of the opcode field, the translation logic is complex for such a rISA design. The middle bar in Figure 17 shows that *rISA_7333_imm* achieves slightly better code compressions as compared to the first design point since it has large immediate fields, while having access to the same set of registers. *rISA_7333_imm* achieves about 14% improvement over the normal instruction set.

rISA_7333_imp_opnd. Another useful optimization that can be performed to save precious bit-space is to encode instructions with the same operands using a different opcode. For example, suppose we have a normal instruction *add R1 R2*

R2, and suppose we have a rISA instruction of the format *rISA_add1 R1 R2 R3*. The normal instruction can be encoded into this rISA instruction by setting the two source operands same (equal to *R*2). However, having a separate rISA instruction format of the form *rISA_add2 R1 R2*, to encode such instructions may be very useful. This new rISA instruction format has fewer operands, and will therefore require fewer instruction bits. The extra bits can be used in two ways: first, directly by providing increased register file access to the remaining operands and, second indirectly, since this instruction can afford a longer opcode, another instruction with tighter constraints on the opcode field (for example an instruction with immediate operands) can switch opcode with this instruction.

We employ the implied operands feature in *rISA_7333* and generate our fourth rISA design, *rISA_7333_imp_opnd*. This rISA design matches the MIPS16 rISA. It can be seen from Figure 17, the *rISA_7333_imp_opnd* design achieves, on average, about the same code size improvement as the *rISA_4444* design point; it achieves about 22% improvement over the normal instruction set.

rISA_hybrid. Our final rISA design point is *rISA_hybrid*. This is a custom ISA for each benchmark. All the previous techniques, i.e., long immediate values, implied operands, etc. are employed to define the custom rISA for each benchmark. In this rISA design instructions can have variable register accessibility. Complex instructions with operands having different register set accessibility are also supported. The register set accessible by operands varies from 4 to 32 registers. We profiled the applications and manually (heuristically) determined the combinations of operand bit-width sizes that provide the best code size reduction. The immediate field is also customized to gain best code size reduction. The *rISA_hybrid* achieves the best code size reduction since it is customized for the application set. As seen in Figure 17, *rISA_Hybrid* achieves about 26% overall improvement over the normal instruction set. The code compression is consistent across all benchmarks.

In summary, our experimental results for rISA-based code compression show that the effects of different rISA formats are highly sensitive to the application characteristics: the choice of a rISA format for different applications can result in up to 2X increase in code compression. However, the system designer critically needs a "Compiler-in-the-Loop" approach to evaluate, tune and design the rISA instruction set to achieve the desired system constraints of performance, code size and energy consumption.

In a similar manner, the ADL-driven "Compiler-in-the-Loop" approach is critical for exploration of programmable SoCs at both the architectural and microarchitectural abstraction levels. While our experimental results have only demonstrated the need for software toolkit generation using a "Compiler-in-the-Loop" exploration of rISA formats, it should be clear that a similar ADL-driven software toolkit generation approach is critical for comprehensive exploration of programmable SoCs.

Experimental results verify that the code compression achieved is very sensitive to the application characteristics and the rISA itself. Choosing the correct rISA for the applications can result in a 2X difference in code compression. Thus it is very important to design and tune the rISA to the applications. Compiler-in-the-Loop exploration can help designers to choose the optimal rISA design.

7. CONCLUSIONS

The complexity of programmable architectures (consisting of processor cores, co-processors and memories) are increasing at an exponential rate due to the combined effects of advances in technology as well as demand from increasingly complex application programs in embedded systems. The choice of programmable architectures plays an important role in SOC design due to its impact on overall cost, power and performance. A major challenge for an architect is to find out the best possible programmable architecture for a given set of application programs and various design constraints. Due to the increasing complexity of programmable architectures, the number of design alternatives is extremely large. Furthermore, shrinking time-to-market constraints make it impractical to explore all the alternatives without using an automated exploration framework. This paper presented an Architecture Description Language (ADL)-driven exploration methodology that is capable of accurately capturing a wide variety of programmable architectures and generating efficient software toolkit including compilers and simulators.

ADLs have been successfully used in academic research as well as industry for embedded processor development. The early ADLs were either structure-oriented (MIMOLA [**?**], UDL/I [**?**]), or behavior-oriented (nML [**?**], ISDL [**?**]). As a result, each class of ADLs are suitable for specific tasks. For example, structure-oriented ADLs are suitable for hardware synthesis, and unfit for compiler generation. Similarly, behavior-oriented ADLs are appropriate for generating compiler and simulator for instruction-set architectures, and unsuited for generating cycle-accurate simulator or hardware implementation of the architecture. However, a behavioral ADL can be modified to perform the task of a structural ADL (and vice versa). For example, nML is extended by Target Compiler Technologies to perform hardware synthesis and test generation [**?**]. The later ADLs (LISA [**?**], HMDES [**?**] and EXPRESSION [**?**]) adopted the mixed approach where the language captures both structure and behavior of the architecture. ADLs designed for a specific domain (such as DSP or VLIW) or for a specific purpose (such as simulation or compilation) can be compact and it is possible to automatically generate efficient (in terms of area, power and performance) tools and hardware. However, it is difficult to design an ADL for a wide variety of architectures to perform different tasks using the same specification. Generic ADLs require the support of powerful methodologies to generate high quality results compared to domain-specific or task-specific ADLs.

This paper presented the four important steps in ADL-driven exploration methodology: architecture specification, validation of specification, retargetable software toolkit generation, and design space exploration. The first step is to capture the programmable architecture using an ADL. The next step is to verify the specification to ensure the correctness of the specified architecture. The validated specification is used to generate a retargetable software toolkit including a compiler and a simulator. This paper presented sample experiments to illustrate the use of an ADL-driven architectural exploration methodology for the exploration of reduced bit-width Instruction Set Architectures (rISA) on the MIPS platform. Our experimental results demonstrate the need and utility of the compiler-in-the-loop exploration methodology driven by an ADL specification. The ADL specification can also be used for rapid prototyping ([**?**], [**?**], [**?**]), test generation ([**?**], [**?**], [**?**]),

and functional verification of programmable architectures [**?**].

As SOCs evolve in complexity to encompass high degrees of multiprocessing coupled with heterogeneous functionality (e.g., MEMS and mixed signal devices) and new on-chip internconnection paradigms (e.g., Networks-on-Chip), the next generation of Multi-Processor SOCs (MPSOCs) will similarly require a language-driven methodology for the evaluation, validation, exploration and codesign of such platforms.