# Functional Abstraction driven Design Space Exploration of Heterogeneous Programmable Architectures

Prabhat Mishra
University of California, Irvine
444 Computer Science
Irvine, California, USA
pmishra@cecs.uci.edu

Nikil Dutt
University of California, Irvine
444 Computer Science
Irvine, California, USA
dutt@cecs.uci.edu

Alex Nicolau
University of California, Irvine
444 Computer Science
Irvine, California, USA
nicolau@cecs.uci.edu

## ABSTRACT

Rapid Design Space Exploration (DSE) of a programmable architecture is feasible using an automatic toolkit (compiler, simulator, assembler) generation methodology driven by an Architecture Description Language (ADL). While many contemporary ADLs can effectively capture one class of architecture, they are typically unable to capture a wide spectrum of processor and memory features present in DSP, VLIW, EPIC and Superscalar processors. The main bottleneck has been the lack of an abstraction underlying the ADL (covering a diverse set of architectural features) that permits reuse of the abstraction primitives to compose the heterogeneous architectures. We present in this paper the functional abstraction needed to capture such wide variety of programmable architectures. We illustrate the usefulness of this approach by specifying two very different architectures using functional abstraction. Our DSE results demonstrate the power of reuse in composing heterogeneous architectures using functional abstraction primitives allowing for a reduction in the time for specification and exploration by at least an order of magnitude.

## Keywords

Functional Abstraction, ADL, Design Space Exploration, DSP, VLIW, Superscalar, Programmable Architecture

## 1. INTRODUCTION

Contemporary processor architectures vary widely in terms of their architectural features: program address generation and instruction dispatch features are widely used in DSP processors; VLIW processors use strong compiler support to ensure correct execution of long instruction words; superscalar processors on the other hand, use hardware scheduling techniques, register renaming etc; and multimedia processors support SIMD operations. Furthermore, each architecture has a different type of branch prediction mechanism, different execution style (in-order/out-of-order), different way of detecting hazards, different way of handling interrupts/exceptions, and last but not least different memory subsystems[16]. Emerging architectures have combined features of classical architectures (DSP, VLIW and Superscalar). For example, the Intel Itanium combines features of VLIW and superscalar; the TI C6x family combines features of DSP and VLIW. Moreover, during design space exploration using customized IP cores, designers may want to add certain architectural features (e.g., some superscalar features to a VLIW processor core) to see how it impacts area, power, performance and other important design parameters. Similarly, to find the best match between the application characteristics and the memory organization features (caches, stream buffers, access modes, SRAM, DRAM etc.), the designer needs to explore different memory configurations in combination with different processor architectures, and evaluate each such system for a set of metrics (such as cost, power and performance). To enable this, designers need (i) a way of specifying wide variety of processor-memory features and (ii) automatic software toolkit generation to enable rapid design space exploration.

In this paper, we present a functional abstraction based specification technique, which is capable of capturing a wide variety of programmable architectures. The key advantage of this approach is that it allows designers to make fast design decisions by reusing the abstraction primitives while composing heterogeneous architectures. The rest of the paper is organized as follows. Section 2 presents related work addressing functional abstraction as well as ADL-driven DSE approaches. Section 3 outlines our approach and the overall flow of our environment. Section 4 presents the functional abstraction needed to capture the wide variety of architectural features and memory configurations. Section 5 illustrates how contemporary example architectures can be described using this functional abstraction based ADL approach. Section 6 presents the design space exploration results using this approach. Section 7 concludes the paper.

## 2. RELATED WORK

While code reuse is traditionally widespread in software engineering, the viability of this technique in the field of hardware design has also been demonstrated [10]. In [20] a database oriented reuse management system is presented. There has been work on parameterized [5], as well as functions and objects based [9], system design. In [18] a system design flow for fast design, prototyping and efficient IP reuse is presented. In [12] a complete codesign environment for embedded systems which combines automatic partitioning with reuse from a module database is presented. However, previous work has not addressed functional abstraction techniques for diverse processor-memory architectures, with the goal of allowing reuse and composability in the context of architectural design space exploration.

In our approach the viability and advantages of reuse in the context of software toolkit generation is studied. Recent approaches on language-driven design space exploration ([1], [2], [11], [4], [6], [19], [22]), use ADLs to capture the processor architecture, generate automatically a software toolkit for that processor, and provide feedback to the designer on the quality of the architecture.

nML [4] has been used by the retargetable code generation environment CHESS [1] to describe DSP and ASIP processors. In ISDL [2], constraints on parallelism are explicitly specified through illegal operation groupings. This could be tedious for complex architectures like DSPs which permit operation parallelism (e.g. Motorola 56K) and VLIW machines with distributed register files (e.g. TI C6X). MIMOLA [11] descriptions are structure-based, and generally very low-level, and laborious to write. MDes [22] allows only a restricted retargetability of the simulator to the HPL-PD processor family. MDes permits the description of the memory system, but is limited to the traditional cache hierarchy. LISA [3] and RADL [21] capture VLIW DSP processors. The approach of [8] uses LISA and SystemC based framework for fast hardware-software co-simulation. These previous ADL-based approaches have, in general, been targeted towards a specific class of architectures, with limited descriptive facilities for complex memory organizations.

EXPRESSION [6], on the other hand, is an ADL designed to capture a wide range of programmable architectures, including DSP, VLIW, and Superscalar, together with their distinct architectural features. This is possible due to the functional abstractions we have developed to support such an ADL-driven approach. Indeed, an ADL such as EXPRESSION critically needs the power of reuse in composing heterogeneous architectures using functional abstraction primitives; this facilitates rapid generation of software toolkits for a wide range of architectures, thus allowing effective design space exploration of heterogeneous processor-memory architectures.

## 3. BACKGROUND

In order to understand and characterize the diversity of contemporary architectures, we surveyed each major architectural domain viz., RISC, DSP, VLIW, Superscalar, and EPIC [13]. We studied the similarities and differences of each architectural feature in different architectural 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 etc. 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 etc. Moreover, each unit has different parameters for different architectures (e.g., number of fetches per cycle, levels of cache, cache line size etc.). Depending on the architecture a functional unit may perform the same operation at different points in time. For

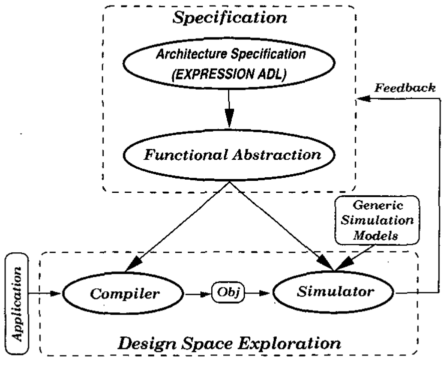

**Figure 1: The Flow in our approach**

example, read-after-write(RAW) hazard detection followed by operand read occurs in the decode unit for some architectures (e.g., DLX [7]), 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). On the other hand, some architectures do not issue operations if RAW hazard is detected while others issue the operation in spite of RAW hazard ( use snooping to read the data at execution stage using feedback paths). In other words, the same functionality is used at different points in the pipeline for different architectures.

We can observe some fundamental differences from the study above; the architecture may use: (i) the same functional or memory unit with different parameters, (ii) the same functionality in different functional or memory unit, (iii) 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 points in time. 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. Section 4 presents the functional abstraction needed to capture the wide variety of programmable architectures. We have developed the associated generic simulation models, which is a one-time activity and independent of the architecture; these can be stored in a library and reused in an ADL to compose and evaluate new architectures.

Figure 1 shows the flow in our approach. In our IP library based design space exploration scenario, the designer starts by specifying the design using functional abstractions using the EXPRESSION ADL. The software toolkit including compiler, simulator, and assembler can be automatically generated from the functional abstraction. The input application program is compiled and simulated and the feedback is used to modify the specification.

## 4. FUNCTIONAL ABSTRACTION

Functional abstraction allows the system designer to describe a wide variety of architectures in a hierarchical fashion. 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.

## 4.1 Structure of a Generic Processor

We capture the structure of each functional unit using parameterized functions. In the following specific example, the fetch unit functionality contains several parameters, such as the 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. The fetch unit is described using sub-functions. Each subfunction is defined using appropriate parameters. For example, *ReadInstMemory* reads $n$ operations from the 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 a BTB based branch prediction mechanism.

```
FetchUnit(# read per 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 (next_PC)
  } ELSE
    IncrementPC(x)
}
```

We have defined parameterized functions for all functional units present in contemporary programmable architectures, such as the fetch unit, branch prediction unit, decode unit, issue unit, execute unit, completion unit, interrupt handler unit, PC Unit, Latch, Port, Connection etc. We have defined sub-functions for all the common activities e.g., ReadLatch, WriteLatch, ReadOperand, RenameRegister etc. We have also defined a few sub-functions e.g., RenameRegister, GraduateOperation using sub-functions to allow a finer granularity of architectural exploration. The notion of generic sub-function allows the flexibility of specifying the system in finer detail. It also allows reuse of the components. Furthermore, these components can be pre-verified. Thus the task of verification will reduce mainly to performing interface verification at all levels.

## 4.2 Behavior of a Generic Processor

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 performs the intended functionality. The parameter list includes source and destination operands, necessary control and data type information. We have defined a set of common sub-functions e.g., ADD, SUB, SHIFT etc. The opcode functions may use one or more sub-functions. For example, the MAC (multiply and accumulate) uses two sub-functions. These opcode functions are used as a parameter for the generic function of the execution unit.

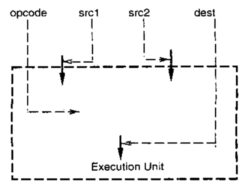

Figure 2: Example of distributed control

## 4.3 Structure of a Generic Memory Subsystem

The memory represents a major bottleneck in modern embedded systems. Each type of memory module, such as the SRAM, cache, DRAM, SDRAM, stream buffer, victim cache etc., is modeled using a function with appropriate parameters. For example, the cache function has parameters: cache size, line size, associativity, word size, replacement policy, write policy, read/write access times etc. These functions also have parameters for specifying pipelining, parallelism, access modes (e.g., normal read, page mode read, burst read) etc. Again, each function is composed of sub-functions.

## 4.4 Generic Controller

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 a 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 as shown in Figure 2. 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. We have shown here only the control information of the latch. The latch contains data values and predicate registers (if applicable) as well.

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 also contains information specific to that unit e.g., busy bit (BB), stall bit (SB), list of children, list of parents, opcodes supported etc. The control table captures all the necessary details to perform selective or complete stalling of the pipelines. Stalling happens due to three kinds of hazards: structural hazards, data hazards and control hazards.

## 4.5 Interrupts and Exceptions

Another major challenge for functional abstraction is the modeling of interrupts and exceptions. We now briefly describe the abstraction needed to capture the wide variety of exceptions and interrupts possible in programmable architectures. Each exception is captured 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.

We model an interrupt handler unit that services these exceptions. It has information regarding the priority of interrupts and which exceptions generate what interrupt. The generic interrupt handler has a parameterized priority table. The interrupt handler unit generates one particular interrupt based on the priority. Before execution of that particular interrupt service routine, context saving and complete/partial flushing occurs. The specific type of flushing is decided by the semantics of that interrupt: complete flushing clears the entire pipeline; partial flushing means flushing only the instructions behind the interrupted instruction and allowing the previous instructions to continue using the program order information available in completion queue. Again, these actions are part of parametric sub-functions that allow a finer grain of microarchitectural exploration.

We have also defined the functional abstraction needed for the DMA, co-processor and external interface. The detailed description of generic abstractions for all of the microarchitectural components are too long to describe in this paper, and can be found in [13].

## 5. CONTEMPORARY EXAMPLE ARCHITECTURES

Using the functional abstraction approach outlined above, we have been able to describe the DLX [7], TI C6x, MIPS R4000, MIPS R10K, Intel Itanium and PowerPC architectures representing a diverse set of processor-memory styles. In this section, we demonstrate the ability to describe two architectures with radically different processor and memory styles using functional abstraction: MIPS R10K is a superscalar processor with two level of cache; TI C6x is a hybrid processor containing both DSP and VLIW features with a *novel memory organization* (partitioned register file, cache hierarchy and configurable scratch pad SRAM etc.).

### 5.1 MIPS R10K Architecture

The MIPS R10000 is a dynamic, superscalar microprocessor that implements the 64-bit Mips-4 instruction set architecture. It fetches and decodes four instructions per cycle and dynamically issues them to five fully-pipelined, low-latency execution units. Instructions can be fetched and executed speculatively beyond branches. Instructions graduate in order upon completion. Although execution is out of order, the processor still provides sequential memory consistency and precise exception handling. With speculative execution, it calculates memory addresses and initiates cache refills early. In this section, we outline how we specify components of the MIPS R10K using the functional abstraction and the generic parameters described in Section 4.

The fetch unit function is invoked with the appropriate parameter values. Both the number of instructions fetched per cycle and the number of instructions sent to decode stage per cycle are set to four. The number of entries in reservation station is set to zero. The number of entries in completion queue (Active List in R10K terminology) is set to 32.

The decode functionality is instantiated with read connection from fetch latch and write connections to MemIssue, IntIssue and FloatIssue units. It uses the register renaming sub-function while decoding instructions. It inserts the decoded instruction in the completion queue (ActiveList) which maintains the program order. The decode logic decides where to dispatch (MemIssue, IntIssue or FloatIssue) a particular instruction based on the supported opcodes information available in the control unit. Table 1 shows the simplified control table for the MIPS R10K architecture, where the rows indicate the pipeline stages and columns represent parallel functional units. For illustration, we do not show all the details, such as the opcodes supported by each unit, list of children, list of parents etc., although we model them.

The IntIssue, FloatIssue and MemIssue functions are instantiated with a reservation station size of 16 entries. Each issue unit performs operand read and RAW hazard detection (using appropriate sub-functions) before performing out-of-order issue.

Execution units are instantiated using appropriate opcode functionalities. The Address Queue (MemIssue unit) reads data and tag using virtual address while the physical address is computed. It checks whether the load is a hit or miss once the physical address is available. This is different than the conventional way of hit or miss detection. In conventional architectures the load request is done using physical address and hit or miss detection is done inside the memory subsystem. This illustrates our ability to reuse the hit or miss detection sub-functions in the processor side (although conventionally it remains on the memory side).

**Table 1: Control table for R10K architecture**

| | | | | |
|---|---|---|---|---|
| . | | Fetch BB:0 SB:0 | | |
| | | Decode BB:0 SB:0 | | |
| MemIssue BB:0 SB:0 | IntIssue BB:0 SB:0 | | FloatIssue BB:0 SB:0 | |
| AddrCalc BB:0 SB:0 | ALU1 BB:0 SB:0 | ALU2 BB:0 SB:0 | FADD1 BB:0 SB:0 | FMPY1 BB:0 SB:0 |
| TLB BB:0 SB:0 | FADD2 BB:0 SB:0 | FMPY2 BB:0 SB:0 | SQRT BB:0 SB:0 | FDIV BB:0 SB:0 |
| | FADD3 BB:0 SB:0 | | FMPY3 BB:0 SB:0 | |

The memory hierarchy consists of two levels of cache. The parameters for the primary cache are: associativity - 2, cache size - 2K double-words, line size - 4 , word size - 64 bits, replacement - LRU, write policy - write back, number of lines - 512, access time - 1 cycle. Similarly secondary cache functionality is instantiated with the appropriate parameters: associativity - 2, cache size - 512K double-words, line size - 16, word size - 64 bits, replacement - LRU, write policy - write back, number of lines - 32K, access time - 2 cycles.

Each functional unit invokes the appropriate sub-functions to capture exception conditions. The interrupt handler function captures the priority table of 19 interrupts. In this manner we are able to concisely capture a state-of-the-art dynamic superscalar architecture using the functional abstraction approach. Furthermore, we now have the ability to quickly change its architectural parameters and view their effects on performance, code size etc.

### 5.2 TI C6x Architecture

We now demonstrate the ability to capture components of a hybrid VLIW DSP architecture using our functional abstraction technique. TI C6x is an 8-way VLIW DSP processor with a novel memory subsystem (cache hierarchy, configurable SRAM, partitioned register file). TI C6x processor has a deep pipeline, composed of 4 fetch stages (PG, PS, PR, PW), 2 decode stages (DP, DC), followed by the 8 functional units.

The fetch functionality consists of four stages viz., program address generation, address send, wait, and receive. Each of the four stages is modeled using respective sub-functions with appropriate parameters. The architecture fetches one VLIW instruction (eight parallel operations) per cycle.

The decode function decodes the VLIW word and dispatches upto eight operations per cycle to eight execution units. Each execution unit performs operand read and hazard checks (using sub-functions). At the end of computation each execution unit writes back (using sub-function) the result to register file.

The TI C6x architecture has a novel memory organization, comprised of a 2-level cache hierarchy and a programmable SRAM space. The L1 program cache is 4K bytes, direct mapped with line size 64 bytes. The L1 data cache is 4K bytes, associativity 2 and line size 32 (bytes). The L2 cache is 64K bytes and depending on the mode of configuration, the memory space is divided between SRAM and associative cache. Memory modules are instantiated with appropriate parameters for capturing the memory subsystem.

Each functional unit invokes the appropriate sub-functions to capture exception conditions. The interrupt handler function captures the priority table of 14 interrupts. Reset and NMI has higher priority than INT4 to INT15 interrupts. In this manner, we are able to capture a hybrid DSP/VLIW architecture using our functional abstraction. Furthermore, we again have the ability to tweak architectural parameters (and compose new features) using the functional abstraction approach.

## 6. EXPERIMENTS

We performed extensive architectural design space exploration by varying different architectural features, achieved by reusing the abstraction primitives with appropriate parameters. In this section we illustrate the usefulness of our approach in three architecture exploration dimensions.

### 6.1 Exploration varying Processor Features

Contemporary superscalar processors use in-order completion/graduation to ensure sequential execution behavior in the presence of out-of-order execution. Here, we explore the MIPS R10K processor in the presence of out-of-order graduation without violating functional correctness. We described the MIPS R10K architecture (with in-order graduation and 8 entry Active List) using the functional abstraction approach and generated the software toolkit. We modified the description to perform out-of-order graduation and generated the software toolkit rapidly. We used a set of benchmarks from the multimedia and DSP domains. Figure 3 presents a subset of the experiments we ran to study the performance improvement due to out-of-order graduation. The light bar presents the number of execution cycles when in-order graduation is used whereas the dark bar presents the number of execution cycles when out-of-order graduation is used. We observe an average performance improvement of 10%. During in-order graduation certain instructions (independent of the instructions above in the Active List) complete execution but are not allowed to graduate since some long latency operations are on top of the Active List and are yet to complete. As a result, the Active List becomes full soon and the decode stalls. This situation becomes more prominent when the top instruction is a load and the load misses. We modified the memory subsystem to study the impact of cache misses along with out-of-order graduation and observed upto 27% performance improvement (in benchmark StateExcerpt when hit ratio is zero). The complete study of the out-of-order graduation for the MIPS R10K processor can be found in [14].

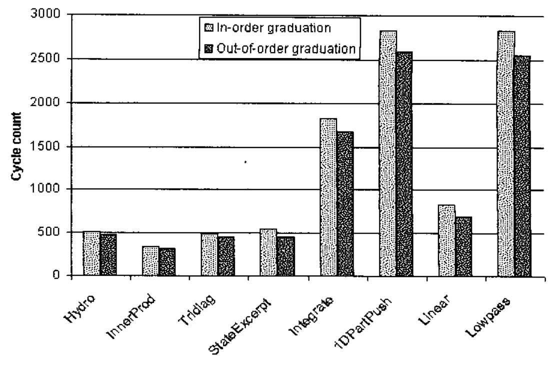

**Figure 3: Cycle counts for different graduation styles**

Due to the high modeling efficiency of functional abstraction the original description and toolkit generation took less than a week; the graduation style modification and toolkit generation took less than a day; the experiments and analysis took few hours; the complete exploration experiment took approximately one week.

### 6.2 Co-processor based Exploration

In the context of co-processor codesign for programmable architectures we have explored the performance impact using a co-processor for the TI C62x environment. First, we described the TI C62x architecture in the EXPRESSION ADL (where multiplication is done in the functional unit) using functional abstraction and generated the software toolkit. Next, we modified the description by adding a co-processor (with DMA controller and local memory) that supports multiplication and generated the software toolkit rapidly. This co-processor has its own local memory and uses DMA to transfer data from main memory. We then used a set of DSPStone fixed point benchmarks to explore and evaluate the effects of adding a coprocessor. Figure 4 presents a subset of the experiments we ran to study the performance improvement due to the co-processor. The light bar presents the number of execution cycles when the functional unit is used for the multiplication whereas the dark bar presents the number of execution cycles when the co-processor is used. We observe an average performance improvement of 22%. The performance improvement is due to the fact that the co-processor is able to exploit the vector multiplications available in these benchmarks using its local memory. Moreover, the functional units operate in register-to-register mode whereas the co-processor operates on its memory-memory mode. As a result the register pressure and thereby spilling gets reduced in the presence of the co-processor. However, the functional unit performs better when there are mostly scalar multiplications. The complete study of the co-processor based design space exploration can be found in [17].

### 6.3 Memory Subsystem Exploration

Another important dimension for architectural exploration is the investigation of different memory configurations for a programmable architecture. We explored different memory

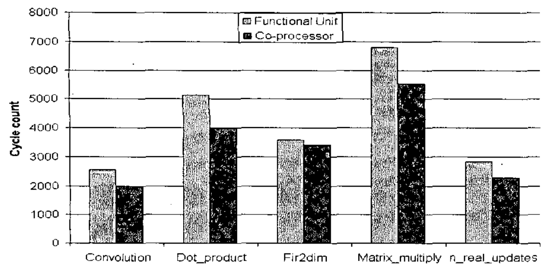

**Figure 4: Functional unit vs. Co-processor**

configurations for the TI C6x architecture with the goal of studying the trade-off between cost and performance. Detailed experiments can be found in [15].

## 7. SUMMARY

This paper proposed a functional abstraction based design space exploration methodology which is capable of capturing a wide variety of programmable architectures. A key advantage of the generic function based design space exploration is that it allows designers to rapidly compose new architectures (by reusing the generic abstractions within an ADL such as EXPRESSION), generate automatically a customized software toolkit and quickly perform a compiler-in-the-loop exploration exercise that allows comparative evaluation of different architectural features. The notion of generic sub-functions allows the flexibility of specifying the system in finer detail. Furthermore, since these components can be pre-verified, the task of verification will reduce mainly to performing interface verification at all levels of the design hierarchy.

We demonstrated the power of our approach in the context of three very different architectural exploration scenarios. We are able to rapidly generate a software toolkit for each instance of a configured architecture and perform detailed simulations of the compiled application to evaluate the effects of different processor and memory configurations. It took less than a week to complete the specification and derive an optimized toolkit for each class of explored architectures, and another week to perform detailed exploration experiments by tuning the architecture. Feedback from our industrial sponsors indicate that each such exploration experiment typically takes on the order of 6 months to 1 year for a compiler-in-the-loop exploration even for a fairly "standard" architectural family. Thus we are able to reduce the exploration time by at least an order of magnitude. Therefore, we believe this functional abstraction approach solves a critical bottleneck for rapid architectural exploration; the ability to reuse parameterized functions and sub-functions enables true heterogeneous processor-memory architectural exploration particularly in the context of IP-based System-on-Chip(SOC) design.

Our ongoing work targets the use of this functional abstraction based design space exploration for generating synthesized hardware automatically. Furthermore, we plan to extend this specification technique to generate FSM automatically and perform property checking during rapid design space exploration.

## 8. ACKNOWLEDGMENTS

This work was partially supported by grants from NSF (MIP-9708067), DARPA (F33615-00-C-1632) and Motorola Corporation. We would like to gratefully acknowledge Jonas Astrom for his contribution to the functional abstraction work.

## 9. REFERENCES

[1] G. Goosens et al. CHESS: Retargetable code generation for embedded DSP processors. In *Code Generation for Embedded Processors.* Kluwer, 1997.
[2] G. Hadjiyiannis et al. ISDL: An instruction set description language for retargetability. *DAC*, 1997.
[3] V. Zivojnovic et al. LISA - machine description language and generic machine model for HW/SW co-design. In *VLSI Signal Processing*, 1996.
[4] M. Freericks. The nML machine description formalism. TR SM-IMP/DIST/08, TU Berlin, 1993.
[5] T. Givargis and F. Vahid. Parameterized system design. In *CODES Workshop*, 2000.
[6] A. Halambi et al. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proc. DATE*, Mar. 1999.
[7] J. Hennessy and D. Patterson. *Computer Architecture: A quantitative approach.* Morgan Kaufmann Publishers Inc, San Mateo, CA, 1990.
[8] A. Hoffmann et al. A framework for fast hardware software co-simulation. *DATE*, 2001.
[9] A. Jantsch and I. Sander. On the roles of functions and objects in system specification. *CODES*, 2000.
[10] M. Koegst et al. A systematic analysis of reuse strategies for design of electronic circuits. *DATE*, 1998.
[11] R. Leupers and P. Marwedel. Retargetable code generation based on structural processor descriptions. *Design Automation for Embedded Systems*, 3(1), 1998.
[12] M. Meerwein et al. Linking codesign and reuse in embedded systems design. In *CODES Workshop*, 2000.
[13] P. Mishra et al. Functional abstraction of programmable embedded systems. TR UCI-ICS 01-04, 2001.
[14] P. Mishra et al. A study of out-of-order completion for the MIPS R10K superscalar processor. TR UCI-ICS 01-06, University of California, Irvine, 2001.
[15] P. Mishra et al. Memory subsystem description in EXPRESSION. TR UCI-ICS 00-31, 2000.
[16] P. Mishra et al. Processor-memory co-exploration driven by an architectural description language. *VLSI Design 2001.*
[17] P. Mishra et al. Coprocessor codesign for programmable architectures. TR UCI-ICS 01-13, 2001.
[18] F. Pogodalla et al. Fast prototyping: a system design flow for fast design, prototyping and efficient ip reuse. *CODES*, 1999.
[19] V. Rajesh et al. Processor modeling for hardware software codesign. In *VLSI Design 1999*
[20] A. Reutter and W. Rosenstiel. An efficient reuse system for digital circuit design. In *Proc. DATE*, 1999.
[21] C. Siska. A processor description language supporting retargetable multi-pipeline dsp program development tools. In *Proc. ISSS*, Dec. 1998.
[22] The MDES User Manual, 1997. *www.trimaran.org*