Automatic Tool Generation from Structural Processor Descriptions

Florian Brandner

Institute of Computer Languages
Christian Doppler Laboratory** -
Compilation Techniques for Embedded Processors
Technische Universität Wien
brandner@complang.tuwien.ac.at

Abstract. Processor description languages are a promising approach to the rapid design of customized processors that are specialized for a given application domain. The xADL processor description language allows to derive a behavioral instruction set model from a structural hardware specification automatically. Traditional approaches often lead to redundant specifications due to the use of separate descriptions modeling these two views. The feasibility of our approach is demonstrated using two tool generators that retarget software development tools for a RISC and VLIW processor models. The compiler generator derives a highly optimizing code generator, while the simulator generator derives a cycle-accurate simulation engine. Experiments show that the derived tools compete with corresponding hand-crafted tools.

1 Introduction

The success of embedded systems in mobile communication and entertainment devices, in commodity appliances, the domestic environment, as well as in the (safety) critical control systems of cars and airplanes made these small computer systems an indispensable part of everybody's daily life. The demands on these systems in terms of reliability, efficiency, and computational power are steadily rising, while at the same time the physical dimensions are required to shrink and the costs per unit need to be reduced on every new product generation.

Application-Specific Instruction Processors (ASIP) have become a valuable tool to deliver high computing power under rigid power and chip area constraints. However, the development of such a processor is a delicate task that requires intimate knowledge of processor design, software development, compilers, and, of course, the particular problem domain at hand.

Processor Description Languages (PDL) – often referred to as *Architecture Description Languages* (ADL) – are a promising approach to capture the behavior, the hardware structure, and the instruction set of an ASIP using a compact

** This work is supported in part by OnDemand Microelectronics and the Christian Doppler Forschungsgesellschaft.

and concise specification. PDLs typically provide various views of the processor targeting different abstraction levels: (1) the *behavioral* level is primarily concerned with the abstract behavior of individual instructions, while (2) the *structural* view defines the processor's hardware organization and its computational resources. Languages focusing on the former processor view are called *behavioral* languages, those focusing on the latter *structural*. Sometimes a language allows to capture both views using a combined specification, it is then called a *mixed* language.

Processor models, specified using such a processor description languages, can be used in various ways. A very common task is to (semi-)automatically derive the software development tools, such as the compiler, the linker, and the assembler for the given processor. In addition, simulation tools, test cases, and even hardware prototypes can be derived. The quality of the obtained artifacts largely depends on the information that is specified by the processor description. *High-level* tasks, such as compiler generation, require an abstract model of the instruction behavior. Whereas *low-level* tasks, e.g., the generation of hardware models, require a detailed model of the processor's computational resources and their interaction. Mixed languages typically provide the highest flexibility in terms of these application tasks, because both a rather high-level behavioral and a rather low-level structural view of the processor is available.

In this work, the processor description language xADL [1] is presented, which allows to derive a corresponding behavioral model from a *structural* processor specification automatically using *instruction set extraction*. xADL models capture the behavioral as well as the structural details of a processor equally well and thus provides great flexibility with regard to verification, validation, and generation tasks. Redundancies known from mixed languages are, however, avoided, because only a single specification of the processor organization is required. We demonstrate the feasibility of our approach using two tool generators: (1) a compiler generator [2, 3] that automatically derives a highly optimizing code generator for the LLVM compiler infrastructure [4], and (2) a simulator generator [5, 6] that derives a highly accurate high-performance instruction set simulator based on dynamic binary translation. Experiments show that the derived tools can compete with hand-crafted tools for the MIPS RISC and CHILI VLIW processors. The code produced by the generated compilers achieves speedups of up to 20% in comparison to hand-crafted compilers. On average over all our benchmarks moderate slowdowns of only 5-15% have been observed for different processor models. The simulation framework is similarly competitive and achieves a simulation speed of up to 483 MHz for individual benchmarks. The average simulation speed over all our benchmarks for different processor models is in the range from 43 MHz to 79 MHz.

In the following section a short overview over the related work in the field of processor description languages is given. Followed by an introduction to the basic principles of the xADL language in Section 3. Section 4 shortly covers the tool generators that allow to derive an optimizing compiler backend and a cycle-accurate instruction set simulator from a given processor model. The results of the empirical evaluation are presented in Section 5 before concluding in Section 6.

2 Related Work

One of the most influential languages in the context of processor description is *nML* [7]. The processor is modeled using a behavioral description of the instruction set. The specification can be structured using an attributed grammar in order to reuse common information among individual instructions or instruction groups. New instructions are specified using either *AND*-rules or *OR*-rules. AND-rules combine information provided by independent rules, while OR-rules allow to compactly enumerate instruction variants. In its latest form, nML [8] also includes a basic skeleton that defines the internal organization of the processor hardware. In particular the early work on code generation using the retargetable compiler *Chess* is well described [9]. Programs that are compiled using Chess are represented using a *Control/Data-Flow Graph* (CDFG). Operations of the CDFG are matched with nodes of the instruction set graph (ISG), a representation of the target processor's instructions and storage elements.

An equally mature framework has been developed for the *LISA* language [10]. Instructions are composed of so-called *operations* that provide information on the behavior, the assembly syntax, and the binary encoding. The instruction behavior is described using C, C++, or SystemC, which prohibits high-level applications such as compiler generation. Ceng proposed an additional section to model the abstract behavior of instructions for the automatic generation of a compiler [11]. The LISA language further supports a wide range of applications, including the customization of compiler optimizations [12], efficient simulation using interpretation, compilation [13], and partial native execution [14].

The *EXPRESSION* language [15] is a typical mixed processor description language, i.e., a processor model consists of several views that capture the instruction set, the hardware structure, and abstract instruction semantics for retargetable compilation separately. The language has been used in a wide area of applications, among others for retargetable compilation [16], retargetable simulation [17], and verification as well as validation [18].

The *MIMOLA* language [19] and its software systems (MSS) is one of the few well known structural processor description languages. It originated from architecture synthesis and microprogramming of the synthesized hardware blocks. Several generations of hardware synthesis tools, compilers and test program generators have been developed within the MSS [18]. MIMOLA has been primarily designed for synthesis, however, the language semantics is also precisely defined for simulation. Later work even investigated the use of interpretation and compiled simulation [20] based on an instruction set abstraction that has been extracted from the processor model [21]. The extracted model can also be used to derive the instruction selector for a compiler based on tree pattern matching [22]. A similar extraction algorithm has been proposed by Akaboshi et al. for the *COACH* processor modeling system [23].

A recent book by Prabath Mishra and Nikil Dutt provides an excellent introduction to processor description languages and their applications [8]. The book also covers most of the languages and systems presented here in more detail.

3 Processor Description

The xADL language, in contrast to most contemporary PDLs, primarily captures a *structural* view of the processor's computational resources. The structure is modeled using a set of interconnected *components* that are enriched with semantic annotations about their behavior. All components are derived from *types*, which can be seen as *reusable* templates or blue prints of a given hardware resource. In addition, *meta-information* such as the assembly syntax and binary encoding of the processor's instruction is captured, along with *programming conventions*. Furthermore, processor models can be parametrized using *configurations*, which allow to control certain architectural features, e.g., the bit-width of the data path, the number of registers, the number of parallel computational units, et cetera.

The instruction set is *not explicitly* specified by an xADL processor description. Nevertheless, the instruction set abstraction is a fundamental concept of the language design. The instructions are *automatically* derived from the structural model using *instruction set extraction*.

3.1 Components and Types

Types and the corresponding component instances thereof are the building blockz of xADL processor models. The language provides abstractions for common hardware resources, such as registers, memories, and caches. In addition, constant immediate values that are embedded into the instruction word are modeled using specialized types and components. The respective types define ports that allow to interconnect the components among each other. Additional properties, such as the size and number of registers, features of caches and memories, etc. can be specified.

Components derived from the mentioned types only provide limited capabilities to describe arbitrary computations. Therefore, *functional unit* types can be defined, which provide ports to interface with other components along with a set of user-defined *operations*. Operations in turn consist of a sequence of *micro-operations*. The xADL language defines a rich set of *built-in* micro-operation, e.g. primitive arithmetic and logical operations, comparison operations, (conditional) copy operations, as well as debug and control operations.

The types of any given processor model can be reused in order to derive variations of a processor or combined to form completely new processors. The language also allows to customize existing types using type arguments, i.e., *generics*, and a form of *inheritance*, similar to templates and sub-classing in the programming language C++.

The component instances of a processor model are interconnected using *data* and *pipeline links*, which roughly correspond to wires and pipeline registers of the underlaying hardware implementation. These links are grouped by so-called *connects*, which allow a compact representation of valid link combinations to connect the various ports of components. In addition, *hazard links* and *signals* allow the modeling of bypasses and additional control logic to resolve data hazards as well as control hazards. It is important to note that the interaction among instructions in the pipeline through the various kinds of links and signals is well-defined and thus analyzable, which is particularly important when a behavioral view of the processors instruction set is to be derived.

The resulting network of components and data links corresponds to an enriched block diagram of the processor's hardware structure, that carries information about the pipeline, the data flow withing the pipeline, the computations of functional units, and the hazard resolution logic.

3.2 Instruction Set Extraction

The instruction set extraction operates on a slightly simplified form of the fine-grained structural processor model, which only captures the data flow between the ports of the processor's components. Information, such as the pipeline structure or the hazard detection logic is eliminated. This simplifies the extraction algorithm and leads to a very nice abstraction that can be easily followed by the processor designer.

The data flow of the processor is represented using a *directed hypergraph* [24]. A directed hypergraph $\mathcal{H} = (V, E)$ consists of a finite set V of vertices and a finite set E of *directed hyperedges*. An hyperedge is an ordered pair (X, Y), where $X \subseteq V$ and $Y \subseteq V$ are possibly disjoint sets of nodes in V. The vertices of the hypergraph correspond to the ports of the processor's components. The hyperedges are derived by reverting the *connects* of the structural processor model. Additional hyperedges are constructed to represent the internal behavior of computational resources.

Figure 1 presents the hypergraph of a simple processor model, consisting of an immediate, three register files, a cache and three functional units. The vertices are derived from the component ports which are further connected by the reverted *connects* of the structural model. Additional edges represent the internal computations of the functional units and the cache, i.e., edges $e2$, $e6$, $e7$, $e10$, and $e11$. Certain ports of caches and memories are not considered when these additional edges are constructed. For example, edge $e7$ does not include the two lower ports of the cache, which represent the address and data ports for store operations. The cache does not *compute* a value on store operations, the data flow thus stops at the involved ports.

The algorithm for instruction set extraction processes the hypergraph representation in two steps. First paths through the hypergraph are computed, which are referred to as *instruction paths*. Such a path represents a possible flow of an instruction through the processors' pipeline. Second, instructions are constructed by enumerating all possible combinations of the operations of the

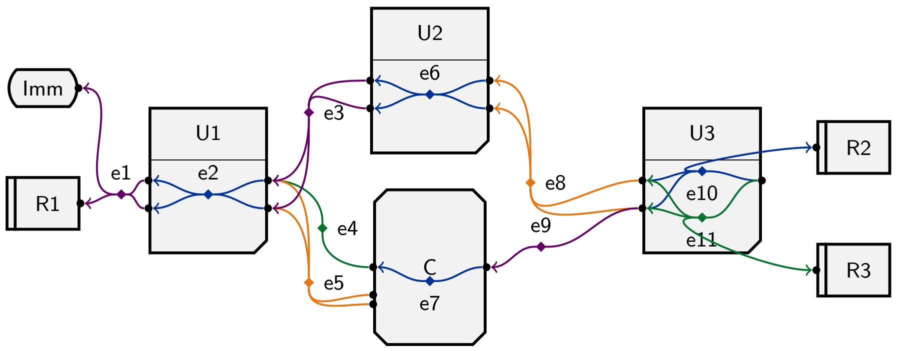

Fig. 1. Example of a simple data path represented by a directed hypergraph.

functional units along the individual paths. Instructions can be uniquely identified by their instruction path and the computed set of operations. The behavior of the instructions is finally represented as a linear sequence of micro-operations that are gathered from the instruction's operations. Furthermore, this representation is annotated with information about data dependencies, bypassing, and hazards, which is available from the structural processor model. The instruction representations encountered during the extraction process constitute the behavioral view of an xADL processor model.

Consider, for example, the processor from Figure 1. Assuming that the functional units are associated with one operation $\mathtt{op_i}$ respectively. The instruction set extraction computes five paths: $\{e1, e2, e3, e6, e8, e10\}$, $\{e1, e2, e4, e7, e9, e10\}$, $\{e1, e2, e3, e6, e8, e11\}$, $\{e1, e2, e4, e7, e9, e11\}$, and $\{e1, e2, e5\}$. Each path corresponds to exactly one instruction, e.g., the instruction corresponding the the first path can be modeled as a pair ($\{$ e1, e2, e3, e6, e8, e10 $\}, \{\mathtt{op_1}, \mathtt{op_2}, \mathtt{op_3}\}$), whereas the instruction of the last path is represented by ($\{e1, e2, e5\}, \{\mathtt{op_1}\}$).

4 Tool Generation

xADL processor models are processed by the *adlgen* tool [1], which not only provides a parser for the language itself, but also provides a rich interface for the development of analysis and generation tools. Various generators have already been developed within the *adlgen* framework, among them a compiler generator [2, 3] and a simulator generator [5, 6].

The compiler generator [2] derives a complete compiler backend, including register specifications for register allocation, an abstract resource model for instruction scheduling, and most importantly tree patterns for instruction selection. The initial generator [2] has been extended and is now able to derive a backend for the open-source LLVM compiler infrastructure [1] and a proprietary compiler. Deriving the instruction selection rules proceeds in two steps. First, an initial rule set is derived from the behavioral instruction models. This rule set heavily relies on the capabilities of the underlaying processor, and is in many cases not sufficient to derive machine code for all operations of the compiler's intermediate language. Thus, during a second phase *templates* and *specializations* are applied to the initial rule set in order increase the coverage by combining existing rules. However, even these additional rules may not be sufficient for a full covering, i.e., if the processor lacks certain fundamental capabilities or if template and specialization patterns are missing. We thus apply a formal completeness test [3] to the final rule set, which reports counter examples if the compiler intermediate language is not covered completely.

Another generator tool, the simulator generator [5, 6], is capable of deriving a cycle-accurate instruction set simulator from a given xADL processor description. In contrast to the compiler generator, the simulator requires additional information that is not entirely available through the behavioral instruction set model, it thus also relies on the structural processor view. The resulting instruction set simulator operates in three modes: (1) interpretation, (2) basic-block-level dynamic binary translation, and (3) *region-based* translation. The simulated program is initially executed and monitored using a simple interpreter. If the monitoring detects a region within the program that is executed frequently, the simulation of this region is optimized. At first, only basic blocks within the region are translated to machine code of the host computer and subsequently executed natively. Later, if these basic blocks are still executed frequently, possibly multiple basic blocks are recompiled into a *region*. Regions may contain arbitrary control flow, in particular loops, and may thus drastically improve the simulation speed. In many cases simulation proceeds within a region for thousands of simulated cycles before the fast native execution is exited and simulation has to fall back to the slower interpreter.

5 Evaluation

Several processor models have been developed with the xADL language ranging from very simple RISC processors to sophisticated VLIW processors supporting predicated execution. The resulting processor models are typically very compact and readable, and can easily be extended. The development of a new processor model is usually only a matter of a few days. This indicates that the structural specification style of the xADL language is well suited for the description of processors and their instruction set. The xADL language and the accompanying generators have been evaluated using three processor models: (1) a two-way configuration of the CHILI VLIW processor, (2) a four-way CHILI configuration, (3) a MIPS processor model.

5.1 Processor Models

The MIPS processor model closely follows the traditional five-stage pipeline implementation described by Hennessy and Patterson [25]. The processor description faithfully models the complete *MIPS1* integer instruction set, including the

		Syntax		Encoding		Types		Components	
Model	LOC	LOC	#Tmpl.	LOC	#Tmpl.	LOC	#Ty.	LOC	#Ists.
CHILI-v2	1580	191	12	141	6	800	20	350	14
CHILI-v4	1739	220	12	156	6	830	20	454	24
MIPS	1143	183	14	134	9	592	14	157	12

Table 1. Statistics on the MIPS and CHILI processor models.

mandatory branch and load delay slots of early MIPS implementations. In total 57 instructions are described by 1143 lines of xADL code (LOC). CHILI is a configurable VLIW processor developed by OnDemand Microelectronics. Its instruction set defines large load and branch delays to hide memory latencies. Branch latencies can further be eliminated using predicated execution, which occupies two slots of the VLIW bundle, one for the predicate and one for the predicated instruction. The rich set of predicated instruction variants is explicitly enumerated by the instruction extraction algorithm, which results in 886 and 1672 instructions for the two-way and four-way parallel configuration respectively.

Table 1 shows detailed statistics on the xADL specifications of the three processors. The models are very compact and consists of 1739 lines of xADL code at most. The table further shows the number of lines spent on the syntax and the binary encoding specifications, as well as on types and component instantiations, along with numbers on the respective templates, types and instantiations defined. The models further specify programming conventions and processor configurations, which occupy between 69 and 91 lines.[1] The component type definitions account for more than 50% of the code lines, the instantiations on the other hand account for just 26% for the four-way CHILI and less than 13% for the MIPS model. This indicates that the types help factoring out common hardware fragments.

The instruction set extraction algorithm relies only on the functional units and the operations associated with them. Table 2 relates the number of unit instantiations and operations statically defined in the xADL specification to the number of unit instances and operations present in the expanded hypergraph representation. The extraction algorithm is very powerful in enumerating instruction variants. From the four-way parallel CHILI configuration, for example, 1672 instructions are created, which corresponds to about one line of xADL code per instruction. Even for the MIPS model, only about 20 lines of code are spent

[1] The numbers in the table do not add up, due to these specifications.

	Definitions		Expanded		Instruction Set	
Model	#Uts.	#Ops.	#Uts.	#Ops.	#Paths	#Insts.
CHILI-v2	19	77	31	129	15	886
CHILI-v4	19	77	60	253	27	1672
MIPS	7	61	7	61	3	57

Table 2. Statistics on the MIPS and CHILI instruction set models.

		ISA		Behavior	Structure	Compiler	
Model	LOC	LOC	#Instrs	LOC	LOC	LOC	#Rules
R3000	2533	386	58	2121	–	–	–
acesMIPS	4184	828	85	–	533	2353	173

Table 3. Statistics on the ArchC MIPS R3000 and acesMIPS EXPRESSION descriptions.

per instruction. The results also show that the number of instruction paths is considerably smaller than the number of instructions. This helps during the development of xADL processor models, because the designer can focus on the overall structure, i.e., the instruction paths, first and later add operations as needed to realize instructions.

To give an indication how the xADL language relates to other processor description language the MIPS model was compared against other publicly available MIPS-based specifications, namely a cycle-true MIPS-R3000 ArchC [26] model (version 0.7.2) and a MIPS-based VLIW *acesMIPS* specified using the EXPRESSION [15] language (version 0.99). Table 3 summarizes the number of lines spent on individual aspects of these two models.[2] Both models are considerably larger than the 1143 lines of the xADL MIPS model. Also note that the ArchC model is not suited for the automatic generation of a compiler backend, due to the use of the SystemC language, which does not provide an abstract model. The acesMIPS specification, on the other hand, includes a dedicated section providing an abstract view of the instruction set for compiler generation. However, the compiler specification alone requires more than twice the number of code lines.

5.2 Generated Compiler

In order to evaluate the code quality produced by the generated compilers, a subset of the MiBench benchmark suite, as provided by the LLVM test infrastructure, was compiled and subsequently executed using cycle-accurate simulators that were also generated from the respective processor descriptions [5, 6]. Table 4 lists the number of source lines including comments for each benchmark program. The benchmarks were run using the *small* input data sets in order to reduce the simulation time.

[2] The line numbers again do not add up, due to comments and empty lines.

Benchmark	LOC	Benchmark	LOC
automotive-bitcount	932	security-sha	269
consumer-jpeg	26,098	telecomm-crc32	284
network-dijkstra	187	telecomm-fft	476
office-stringsearch	3,250	telecomm-adpcm	304
security-blowfish	1,913		

Table 4. Size of the benchmark programs in source lines.

Model	#Instr. Def.	#Reg. Def.	#Reg. Cl.	#Res.	#Rules
CHILI-v2	817	64	1	2	1416
CHILI-v4	817	64	1	4	1416
MIPS	61	35	3	1	111

Table 5. Statistics on the generated LLVM backends for the MIPS and CHILI processor models.

The reference benchmarks for MIPS were compiled using GCC version 4.1.1 and the GNU Binutils version 2.19.1, both configured for the `mips-elf` target. The newlib system library provides a basic C library implementation. GCC version 4.2.0 together with GNU Binutils 2.16, both provided by OnDemand Microelectronics serve as a reference compiler for the CHILI architecture. The system libraries are based on newlib version 1.14.0. The automatically generated compilers rely on LLVM version 2.4, which uses a modified version of GCC 4.2 as frontend. The GNU Binutils and newlib libraries are shared among the respective reference compilers and the generated compilers. The reference compilers are invoked with aggressive optimizations (`-O3`), while the LLVM compilers use the standard optimization options (`-std-compile-opts`).

As can be seen in Table 5, the LLVM backends derived from the two-way and four-way configurations of the CHILI processor are virtually identical, except for the resources present in the reservation tables of the instruction scheduler. The table lists, from left to right, the number of instruction definitions, register definitions, register classes, abstract resources of the resource tables, and the number of instruction selection patterns generated from the respective processor descriptions. The number of instruction definitions differs from the instruction number listed by Table 2, because semantically equivalent instruction variants are merged. Hence, the number of instruction definitions is lower for the parallel CHILI models. On the other hand, instructions with multiple result values are duplicated, due to restrictions of the LLVM instruction selector. This leads to a higher number of instruction definitions for the MIPS processor model.

The backend generator quite successfully discovers translation patterns from the instruction sets. For every instruction definition of the MIPS and CHILI

	Code Size			Cycles		
Benchmark	GCC	xADL	%	GCC	xADL	%
automotive-bitcount	31,468	25,364	-19	726,162	991,642	+36
consumer-jpeg	245,148	161,648	-34	7,932,872	9221,371	+16
network-dijkstra	39,116	38,564	-1	342,801,926	314414,695	-8
office-stringsearch	27,672	25,700	-7	5,367,471	7274,936	+36
security-blowfish	36,768	26,544	-28	867,965	877,218	+1
security-sha	31,796	29,352	-8	13,270,780	17812,223	+34
telecomm-crc32	29,816	27,716	-7	7,464,707	8350,673	+12
telecomm-fft	45,172	44,976	–	140,766,729	137254,431	-2
telecomm-adpcm	27,588	27,372	-1	7,122,022	12125,699	+70

Table 6. Code size and execution time results for the MIPS processor.

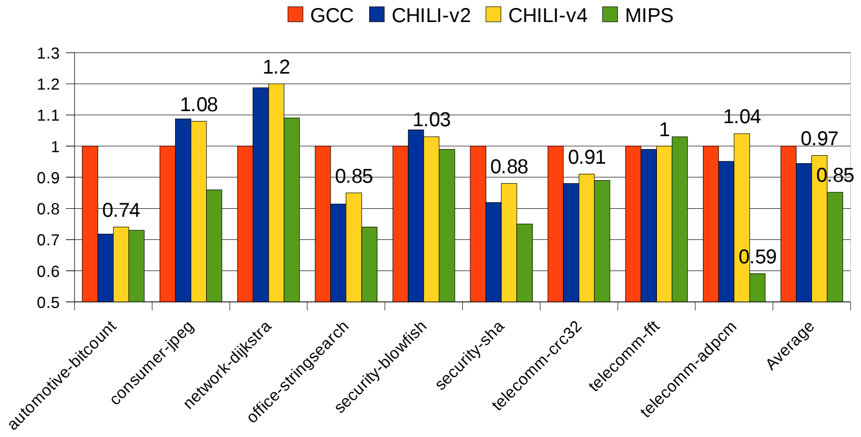

Fig. 2. Performance improvement of the generated CHILI and MIPS backends in comparison to the GCC reference compiler.

models almost two instruction selection patterns are generated, leading to the huge number of 1416 rules for the CHILI models.

The measured execution times and the code size of the stripped benchmark programs for the MIPS instruction set are shown in Table 6. The results indicate that the automatically generated MIPS compiler is competitive to the well-tuned production compiler GCC, in particular, when code size is taken into account. The *fft* and *dijkstra* benchmarks show a reduction of the execution time by 2% and 8% respectively. The severe increase in execution time of 70% in the case of the *adpcm* benchmark is caused by useless branches generated late during the compilation process from conditional assignments. The branch optimization of the LLVM framework runs earlier and thus misses these cases. The relative performance is depicted by Figure 2. On average, a slowdown of only 15% has been observed over all benchmarks.

The performance results obtained for the two CHILI processor configurations are very close to the handcrafted production compilers – see Figure 2.

	Code Size			Cycles		
Benchmark	GCC	xADL	%	GCC	xADL	%
automotive-bitcount	348,892	296,376	-15	881,144	1,183,104	+34
consumer-jpeg	2,341,904	1,241,408	-47	10,794,047	9,976,958	-8
network-dijkstra	485,560	470,744	-3	2,894,236	2,414,124	-17
office-stringsearch	334,004	303,384	-9	624,087	738,406	+18
security-blowfish	400,160	306,192	-23	1,541,883	1,491,519	-3
security-sha	351,860	327,608	-7	10,791,045	12,215,822	+13
telecomm-crc32	353,980	327,132	-8	8,637,327	9,520,911	+10
telecomm-fft	415,184	400,884	-3	187,968,275	188,243,462	+1
telecomm-adpcm	338,988	324,728	-4	10,116,131	9,755,433	-4

Table 7. Code size and execution time results for the four-way parallel CHILI configuration.

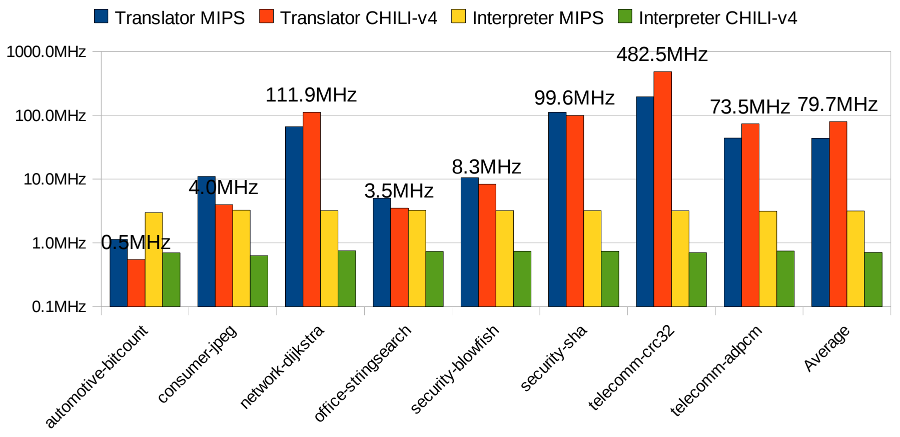

Fig. 3. Simulation speed in MHz for CHILI and MIPS with compilation enabled and disabled – note the logarithmic scale.

Several benchmarks show considerable speedups, in particular the *dijkstra* and the the *jpeg* benchmarks show an improvement of up to 20%. In contrast to the MIPS model, the generated compilers for the CHILI models do not show such severe slowdowns. The *adpcm* benchmark performs much better for the CHILI, because the conditional assignments are directly supported by the processor, useless branches are thus avoided. The four-way parallel configuration even outperforms the production compiler by 4%. The code produced by the generated compilers for the *bitcount* benchmark, however, performs poorly. Address calculations for memory operations accessing an array are not optimally translated and cause some extra instructions. Due to the small loops of the bit counting algorithms this has a large impact. Nevertheless, slight slowdowns of only 5% and 3% respectively have been observed over all benchmarks for both CHILI configurations. Considering the code size, these results are motivating for future work. On average the code size of the stripped executables produced by the xADL-based backends is reduced by 15%. The size of the *jpeg* benchmark program, for example, is reduced by 47%, for *blowfish* benchmark the reduction amounts to about 25%.

5.3 Generated Simulators

The performance of the generated simulators was similarly evaluated using a subset of the MiBench benchmark suite. The benchmarks were compiled with optimization enabled (-O) using *GCC version 3.4.6* for MIPS, which is part of the official development kit SDE Lite 6.06, and *GCC 4.2.0* for the four-way parallel CHILI processor. All measurements were performed on a single core *AMD Athlon(tm) 64 Processor 3500+* with 2200 MHz and 1 GB of RAM running a 32-Bit Linux operating system.

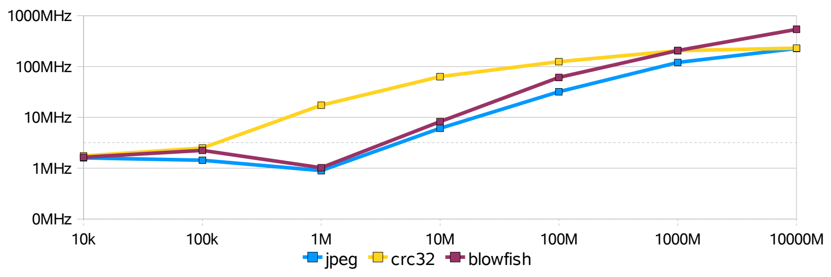

Fig. 4. Simulation speed over time for the MIPS architecture.

We compared the simulation speed of the interpreter and translator for both architectures – see Figure 3. The MIPS simulator reaches about 3.2 MHz, the CHILI simulator about 0.7 MHz. The translator is up to 500 times faster for the longer running benchmarks and reaches up to 480 MHz. On average the MIPS simulator executes at a speed of 43 MHz, the CHILI simulator even reaches 79 MHz on average. Figure 4 shows the peak simulation speed over time for three benchmarks for the MIPS architecture. With all optimizations enabled a peak simulation speed of 800 MHz can be reached for the *blowfish* benchmark. For the very short running *bitcount* benchmark the translator is slower since the compile time cannot be compensated.

Figure 5 shows the relative number of cycles simulated using interpretation or execution of JIT-compiled code in basic blocks and regions. Except for *bitcount* interpretation is only used to simulate a small fraction of the overall cycles, on average 11.7% for CHILI and 11% for MIPS. For *crc32* the complete main loop is compiled to a single region resulting in very high simulation speed.

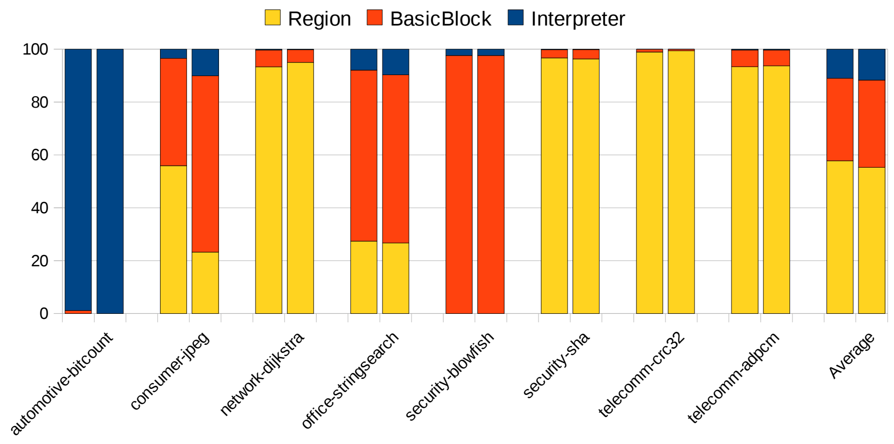

Fig. 5. Ratio of simulated cycles using the interpreter, JIT-compiled code in basic blocks, and compiled code in regions for the MIPS (l.) and CHILI (r.) architecture.

6 Conclusion

This work highlighted the basic design principles of the novel xADL processor description language, which is based on a structural specification of the processor's hardware organization. The instruction set abstraction, even though not described explicitly, is provided by an easy to follow extraction algorithm that provides a behavioral view of the processor. Due to the used abstractions and types, xADL specifications are short and comprehensible compared to other approaches.

Experiments have shown that the structural and behavioral view provided by an xADL processor model allows the automatic generation of competitive software development tools. It is possible to derive an optimizing compiler backend, which generates fast and compact code for the modeled processor that nearly reaches the quality of hand-crafted compilers. Similarly, accurate simulation tools can be derived that use mixed interpretation and dynamic binary translation of basic blocks and regions to improve simulation speed.

References

1. Brandner, F.: Compiler Backend Generation from Structural Processor Models. PhD thesis, Institut für Computersprachen, Technische Universität Wien (2009)
2. Brandner, F., Ebner, D., Krall, A.: Compiler generation from structural architecture descriptions. In: CASES '07: Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems. (2007) 13–22
3. Brandner, F.: Completeness of instruction selector specifications with dynamic checks. In: COCV '09: 8th International Workshop on Compiler Optimization Meets Compiler Verification. (2009)
4. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO '04: Proceedings of the International Symposium on Code Generation and Optimization. (2004) 75–86
5. Brandner, F.: Fast and accurate simulation using the LLVM compiler framework. In: RAPIDO '09: 1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools. (2009)
6. Brandner, F.: Precise simulation of interrupts using a rollback mechanism. In: SCOPES '09: Proceedings of the 12th International Workshop on Software and Compilers for Embedded Systems. (2009) 71–80
7. Fauth, A., Praet, J.V., Freericks, M.: Describing instruction set processors using nML. In: EDTC '95: Proceedings of the 1995 European Conference on Design and Test. (1995) 503–507
8. Mishra, P., Dutt, N.: Processor Description Languages. Volume 1. Morgan Kaufmann Publishers Inc. (2008)
9. Van Praet, J., Lanneer, D., Geurts, W., Goossens, G.: Processor modeling and code selection for retargetable compilation. ACM Transactions on Design Automation of Electronic Systems (TODAES) **6**(3) (2001) 277–307
10. Pees, S., Hoffmann, A., Živojnović, V., Meyr, H.: LISA – machine description language for cycle-accurate models of programmable DSP architectures. In: DAC '99: Proceedings of the 36th Conference on Design Automation. (1999) 933–938

11. Ceng, J., Hohenauer, M., Leupers, R., Ascheid, G., Meyr, H., Braun, G.: C compiler retargeting based on instruction semantics models. In: DATE '05: Proceedings of the Conference on Design, Automation and Test in Europe. (2005) 1150–1155
12. Hohenauer, M., Engel, F., Leupers, R., Ascheid, G., Meyr, H.: A SIMD optimization framework for retargetable compilers. ACM Transactions on Architecture and Code Optimization (TACO) **6**(1) (2009) 1–27
13. Nohl, A., Braun, G., Schliebusch, O., Leupers, R., Meyr, H., Hoffmann, A.: A universal technique for fast and flexible instruction-set architecture simulation. In: DAC '02: Proceedings of the 39th Conference on Design Automation. (2002) 22–27
14. Gao, L., Kraemer, S., Leupers, R., Ascheid, G., Meyr, H.: A fast and generic hybrid simulation approach using C virtual machine. In: CASES '07: Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems. (2007) 3–12
15. Halambi, A., Grun, P., Ganesh, V., Khare, A., Dutt, N., Nicolau, A.: EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In: DATE '99: Proceedings of the conference on Design, Automation and Test in Europe. (1999) 485–490
16. Halambi, A., Shrivastava, A., Dutt, N., Nicolau, A.: A customizable compiler framework for embedded systems. In: SCOPES '01: Proceedings of the 5th International Workshop on Software and Compilers for Embedded Systems. (2001)
17. Reshadi, M., Mishra, P., Dutt, N.: Hybrid-compiled simulation: An efficient technique for instruction-set architecture simulation. ACM Transactions on Embedded Computing Systems (TECS) **8**(3) (2009) 1–27
18. Mishra, P., Dutt, N.: Specification-driven directed test generation for validation of pipelined processors. ACM Transactions on Design Automation of Electronic Systems (TODAES) **13**(3) (2008) 1–36
19. Marwedel, P.: The MIMOLA design system: Tools for the design of digital processors. In: DAC '84: Proceedings of the 21st Conference on Design automation. (1984) 587–593
20. Leupers, R., Elste, J., Landwehr, B.: Generation of interpretive and compiled instruction set simulators. In: ASP-DAC '99: Proceedings of the 1999 Asia and South Pacific Design Automation Conference. (1999) 339–342
21. Marwedel, P., Leupers, R.: Instruction set extraction from programmable structures. In: EURO-DAC '94: Proceedings of the Conference on European Design Automation. (1994) 156–161
22. Leupers, R., Marwedel, P.: Retargetable generation of code selectors from HDL processor models. In: EDTC '97: Proceedings of the 1997 European Conference on Design and Test. (1997) 140–144
23. Akaboshi, H., Yasuura, H.: Behavior extraction of MPU from HDL description. In: APCHDL '94: Proceedings of the 2nd Asia Pacific Conference on Hardware Description Languages. (1994) 67–74
24. Bondy, J.A., Murty, U.S.R.: Graduate texts in mathematics - Graph theory. Volume 244. Springer (2007)
25. Patterson, D.A., Hennessy, J.L.: Computer Organization & Design: The Hardware/Software Interface. 3rd edn. Morgan Kaufmann (2007)
26. Azevedo, R., Rigo, S., Bartholomeu, M., Araujo, G., Araujo, C., Barros, E.: The ArchC architecture description language and tools. International Journal of Parallel Programming **33**(5) (2005) 453–484