A Novel ADL-based Compiler-Centric Software Framework for Reconfigurable Mixed-ISA Processors

Timo Stripf, Ralf Koenig, Juergen Becker
Karlsruhe Institute of Technology, Karlsruhe, Germany
{stripf, ralf.koenig, becker}@kit.edu

Abstract—Reconfigurable processor architectures can dynamically switch their instruction set and instruction format at run time. They offer a new flexibility for adapting to changing applications’ requirements in order to optimize performance and enable resource-awareness. While programmability is a key issue of such architectures, today’s software toolchains are limited to static ISA architectures and must be extended to support reconfigurable processors that can expose different ISAs at runtime. In this paper, we address this shortcoming by presenting a novel retargetable software toolchain that is suitable for mixed-ISA application development as well as design space exploration (DSE). Therefore, we developed a novel mixed-ISA, compiler- and simulator-centric, behavioral architecture description language (ADL). The ADL provides the necessary flexibility to describe multiple ISAs for the software framework. The individual framework tools – the compiler, binary utilities, and instruction set simulator (ISS) – are generated from an ADL description. To realize the complex compiler inside the framework, we extended the LLVM compiler infrastructure by a mixed-ISA retargetable code generator (compiler back-end). To illustrate the flexibility of the ADL-based software toolchain, we performed a first DSE for application characterization of a variety of multi-domain applications. To show the feasible performance/resource benefits through dynamic reconfiguration, we further developed a mixed-ISA application that can dynamically change its instruction format at runtime.

I. INTRODUCTION

The design and implementation of new processor architectures is always a complex and time consuming task. In particular, early design decisions heavily influence the performance and power results of the architecture. The design space of a reconfigurable processor architecture increases since it must be decided which parts should be reconfigurable and which not. Therefore, the problem of early design decision is even worse.

Today’s reconfigurable processor architectures typically focus on dynamic extensions of their instruction set (the set of instruction the processor is capable to execute) by fine-grained reconfigurable logic while relying on a RISC instruction encoding. In contrast to that, we are researching a novel multi-grained, hypermorph reconfigurable processor architecture within the KAHRISMA project [1]. KAHRISMA features dynamic reconfiguration of the instruction set as well as instruction format (e.g. switching between RISC and n-issue VLIW) to execute a configurable number of statically-scheduled instructions in parallel. Multiple processor instances (each instance may be configured to execute a different instruction format) could co-exist in parallel. Each instruction format requires a different amount of resources and also provides different peak performance characteristics. This new degree of freedom leads to the decision problem which application should run on which instruction format to efficiently utilize the available reconfigurable hardware fabrics.

One major problem of a multiplicity of reconfigurable architectures is their programmability. That often arises from the methodology to first design the system from a hardware centric view and later concentrate on the problem how to program it. In contrast, in our research the programmability plays an important role. It is one of our major design goals to be programmable by the widely-used high-level general-purpose programming language C/C++ while improving application execution and architecture efficiency. To achieve this, a compiler-based software toolchain is designed and implemented in parallel to hardware architecture development. The requirements of the software framework can be respected in architectural design decisions from an early stage on to maintain programmability of the hardware architecture.

In contrast to a processor architecture with a fixed Instruction Set Architecture (ISA), the flexibility introduced by reconfigurability of the instruction set and instruction format requires especially a flexible code generator as part of a user-retargetable compiler. The instruction set and format is specified by an Architecture Description Language (ADL). In the past, several ADLs and thereon based retargetable compilers have been developed, but no available solution exists that fulfills the requirements introduced by our hypermorph reconfigurable architecture. Therefore, we started researching a novel retargetable software framework that is the focus of this paper. We selected the Low Level Virtual Machine (LLVM) compiler infrastructure as compiler for the following reasons: (1) the LLVM compiler is open-source and has a very active community, (2) it is already developer-retargetable and supports a broad range of processor architectures, and (3) the back-ends (the part of the compiler that is target dependent) are comparably small (in lines of code) compared to other open-source compilers.

In this paper we present our novel ADL-based software framework for reconfigurable, mixed-ISA RISC processors. To support reconfigurable instruction set and format processors, the ADL is capable not only to describe one ISA but multiple ISAs within one description. Based on the ADL description our novel TargetGen utility partially generates the source code of our retargetable compiler, assembler, and simulator. All tools within the framework support multiple ISAs and the framework can be used to develop mixed-ISA applications that contain code of several ISAs in parallel as well as are able to dynamically switch their ISA at run time. The rest of this paper is organized as follows. Section II gives an overview of state-of-the-art ADLs as well as retargetable compilers. Section III describes the layout of the novel ADL. The ADL-based software framework is described next. First, a general overview is given (Section IV). Then, the TargetGen utility enabling the retargetability of the framework (Section V) and the novel LLVM code generator in Section VI are presented. In Section VII results gained by Design Space Exploration (DSE) and ISA reconfiguration are given. Finally, Section VIII concludes this paper.

II. OVERVIEW OF STATE-OF-THE-ART

A retargetable compiler is a compiler that has been designed to be relatively easy to modify to generate code for different ISAs. Retargetable compilers are classified into two categories: developer-retargetable and user-retargetable [2]. A developer-retargetable compiler can be retargeted to support a new ISA with limited effort by modifying its source code. A prominent example of this category is the GCC compiler that has been ported to various architectures. In contrast, a user-retargetable compiler
can be automatically generated by an ADL without manually modifying any source code.

An ADL [3], [4], [5] is a specification language that is used to model processors at a high level of abstraction enabling automatic analysis and generation of tools and processors especially in the field of Application Specific Instruction Set Processors (ASIPs).

In general, ADLs are classified into structural, behavioral, and mixed ADLs. The early ADLs were either structure-oriented (MIMOLA [6], UDL/I), or behavior-oriented (nML [7], ISDL [8]) leading to the problem that they are only suitable for specific tasks. Structure-oriented ADLs are applicable for hardware synthesis, but less suited for compiler generation. Similarly, behavior-oriented ADLs are appropriate for generating compiler and Instruction Set Simulators (ISSs), but not for generating Cycle-Accurate Simulators (CASs) or hardware implementation of the architecture. There also exist mixed ADLs (LISA [9], HMDES [10] and EXPRESSION [11]) capturing both structure and behavior of the architecture. Members of this class that are designed for a specific domain (such as DSP or VLIW) is 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) hardware and tools.

In the past, ADLs have been used for architecture exploration at design time but there has been less research in the field of ADLs targeting run-time reconfigurable architectures. In [12] a design flow for architecture exploration and implementation of partially reconfigurable processors is presented. Thereby, the LISA ADL has been extended to model partially Reconfigurable Application Specific Instruction Set Processors (rASIPs). rASIPs consist of a fixed part that cannot be changed after fabrication and a reconfigurable fabric that can be constructed as fine-grained Field Programmable Gate Array (FPGA) or Coarse-Grained Reconfigurable Architecture (CGRA). Within the ADL, the designer can mark units (a unit is a set of operations) as reconfigurable and these units are automatically placed on the reconfigurable fabrics on the processor. The instructions of the reconfigurable fabrics are available as Instruction Set Extensions (ISEs) within the ISA. The LISA ADL has been used and extended for automatic compiler generation [13]. Therefore, the modular compiler generation system CoSy from ACE is used [13]. CoSy offers numerous configuration possibilities, both at the level of the intermediate representation and the back-end. However, the approach is limited to ISE and does not focus on dynamic reconfiguration of the ISA.

In [14] the xMAML ADL (eXtended MACHine Markup Language) for designing and modeling of dynamically reconfigurable architectures from FPGAs to reconfigurable processors is presented. It is an extension of the XML-based MAML language that was originally developed for the design of massively parallel processor arrays. The language provides a structural-oriented approach to model the architecture's reconfigurability by introducing a flexible interconnection network to connect any kind of computing resources as well as configuration areas to model partial reconfiguration. However, there exist no compilers targeting the xMAML architecture model.

III. THE NOVEL ARCHITECTURE DESCRIPTION LANGUAGE

All parts of the software framework are based on a novel mixed-ISA, compiler- and simulation-centric, behavioral Architecture Description Language (ADL). It was designed for automatic back-end generation of the LLVM compiler infrastructure as well as automatic assembler and ISS generation. To support reconfigurable instruction formats and sets, the ADL is capable to specify multiple ISAs within one ADL description.

In contrast to mixed ADLs, we do not intend to include a detailed structural processor description, e.g. of the processor pipeline, into the ADL. The KAHRISMA pipeline is too specialized to describe it in a generic way inside the ADL and we would either loose flexibility in our hardware design or inside the ADL. Instead, we plan to include abstract hardware parameters into the ADL as well as to automatically generate selected hardware modules from the behavioral description. Especially modules that must be changed to reflect modifications to the instruction set should be generated, i.e. functional units, registers, and the decoder.

The ADL is based on a special markup language for coding hierarchical structured data in a text document. It is comparable to XML and JSON [15] but offers the flexibility to use variables as well as constant mathematical expressions. This enables the flexibility to describe an ISA in a more abstract model, e.g. by setting central parameters like register count or architecture bit-width in a variable and calculating all related parameters. After variable propagation and mathematical expression calculation, the format can be converted to an XML or JSON representation and is thus further reusable.

This markup language describes a tree consisting of scalar data types (string, boolean, integer, and floating-point number) as leaf nodes and container data types (vector and associative array) as inner nodes. Based on the tree of this markup language, the structure of the ADL description is specified. Figure 1 shows the overall structure of an ADL description. It is composed out of six major sections: five target-independent sections (Global, Nodes, RegisterFile, FieldFormat, and InstrFormat) and a Target section. Target describes a list of available targets. Each target contains two separate target-dependent subsections (Instruction and CallingConvention). Instruction defines for each target a list of instructions. Each instruction references one instruction format (the instruction’s machine word) that is defined in InstrFormat. Each format in turn is composed out of fields that are specified in FieldFormat. The CallingConvention section describes target-specific conventions on the stack layout and register content during a function call. The RegisterFile defines all registers of the architecture that could be referenced by a field (to model register addressing), an instruction (for direct register access), or a calling convention. In the following, all sections and subsections are described in detail.

The Global section contains several globally defined parameters of the specified processor like the name, description, endianness, pointer size, pointer alignment, stack growth direction, and stack alignment of the architecture.

The Nodes section contains a list of all architecture independent instructions of the LLVM back-end that are used by the compiler for instruction selection. The nodes are used within the Instruction

---

**Fig. 1. Overview of the ADL structure**
section to specify the behavior of instructions. Within the list it is outlined if the node is a load, store, call, return, jump, or commutative operation. This information is exploited during ADL processing for optimization and term rewriting.

The **RegisterFile** section defines the register file of the architecture. It is subdivided into physical, logical, and semantic description of registers. The physical register file defines the memories for storing the registers. Each memory is specified by an element count and element width in bits. Within the logical description, a list of register names is defined. Each is mapped to a physical register memory location and has a well-defined type (e.g. 16-bit integer or 32-bit floating-point), identifier name, and assembly name. The semantic subsection defines the behavior for some logical registers. This way instruction, stack, and frame pointer are defined for the compiler and simulator.

The **FieldFormat** section specifies user-defined formats of fields used within the instruction word. Within the ADL there exist predefined fundamental field formats like signed/unsigned integer immediate values of any bit-width. A field definition is typically used to encode register numbers or condition codes (for comparison or conditional branch operations). It has a well-defined encoding bit-width and target format. The encoding bit-width represents the space in number of bits the fields consumes within the instruction word. In contrast, the target format represents the format of the data the field expresses, i.e. a field for addressing 16 32-bit registers has a target format of 32-bit integers but consumes only 4 bits within the instruction word. A field format consuming \( n \) bits can encode \( 2^n \) states. The field definition allows assigning each state to a register, immediate number, code snippet, or leaves the state unused. A code snippet consists of an assembly string, a simulation code snippet, and a LLVM pattern fragment for instruction selection within the compiler.

The **InstrFormat** section is used to declare several instruction formats. Each instruction format describes one structure of the instruction word. It has a fixed bit-width and is composed out of fields. Each field is specified by a name, field format (fundamental or user-defined), bit position, bit-width, data direction, and an optional default value. Depending on the data direction the field’s value can be read (immediates or registers) or written (registers only).

To avoid redundant field specifications within various instruction formats, one format is able to inherit, modify, and extend the values from a previous format. In this way, the formats can be built hierarchically. Furthermore, multiple encoding formats can be expressed within one instruction format. That avoids multiple specifications of instructions that differ only in its instruction format. For an ISA it is common that, e.g. one source operand of an addition could be either an immediate or a register. In that case, the behavior of the operation is identical but only the instruction encoding and source operand differs. By defining multiple encoding formats in one instruction format only one instruction definition is later necessary.

The **Instruction** subsection describes the instruction set, a list of instructions that is available in the processor architecture. Each instruction is specified by an identifier, a reference to an instruction format (defined in **InstrFormat** section), a list of implicit registers, simulation code, assembly syntax, and LLVM pattern for instruction selection. By setting one or more immediate fields (e.g. the opcode) of the instruction format to fixed numbers, it must be ensured that the instruction encoding is unique.

Implicit registers are automatically read or written without flexibility to choose the target or source register number inside the instruction word. In contrast, explicit registers are specified as part of the instruction format as user-defined fields. On the X86 ISA e.g., the compare instruction implicitly writes the flags register and the corresponding conditional branch instruction implicitly reads the flags register to decide if the jump should be taken or not.

The simulation code of one instruction is expressed as a restricted C++ code fragment. The fragment is later embedded into a function body within the ISS. The implicit and explicit registers are available as variables. They have a special data type to emulate arbitrary bit-width integers realized by C++ templates. The functionality is expressed by using functions and overloaded operators working on the special data type that are available within the code fragment.

The assembly syntax of an instruction is specified by an assembly string containing the operands as placeholders. It is used inside the compiler for generating the assembler output and inside the retargetable assembler for parsing the assembler files. It is allowed that multiple instructions use the same mnemonic (assembly name representing the opcode) as long as instructions can be distinguished by arguments or parameters following the opcode.

For instruction selection inside the LLVM compiler, the semantic of instructions is expressed as LLVM patterns. A LLVM pattern is a Directed Acyclic Graph (DAG) consisting of architecture-independent LLVM-specific inner nodes as specified in the **Nodes** section. Leaf nodes could be either implicit registers, explicit registers, or numbers.

The **CallingConvention** subsection specifies the calling convention as used by the compiler. It is also used by the simulator of the software framework e.g. to emulate the calling convention of built-in functions like fopen. The calling convention describes (1) how parameters are passed to a function, (2) how the result is returned from a function, and (3) which registers must be preserved by the callee function.

Within the **Target** section a list of target architectures is specified. Each target architecture is identified by a unique key. The identifier is used inside the compiler, assembler, and simulator to specify the target e.g. as command line parameter. Each target contains the target-dependent subsections **Instruction** and **CallingConvention**. Although all other sections are defined globally, they can contain target-dependent information. One instruction format is only used inside a target if it is referenced in the target’s instruction list. This allows reusing instruction formats between targets. In the same way, only instruction fields are used which are referred by used instruction formats.

Sharing instruction formats between targets is optional and helps avoiding redundant definitions. In contrast, sharing registers between targets is necessary to model the behavior when the ISA is switched at run time. The register content is preserved during ISA reconfiguration and all registers being accessible by two targets can be used to copy data between two targets during reconfiguration. Especially, the content of the stack pointer should be preserved during reconfiguration. All other registers can be stored on the stack.

A reconfiguration between two targets can be triggered by any instruction inside the ADL. A **SWITCHTARGET** command inside the simulation code is used to change the ISA. This allows modeling a special reconfiguration instruction that continues execution on the following instruction using the new ISA. It is also possible to combine reconfiguration with a branch instruction, e.g. “Thumb-aware” ARM processors can switch their ISA to the space optimized Thumb ISA by jumping to an unaligned target address using a regular branch instruction.
A. Example

In Listing 1 an example of an instruction definition is shown. It describes the LD32 operation that loads a 32-bit word from memory. The Format key refers to an instruction format with one data destination register (DDst) and two data source registers (DSrc1 and DSrc2). The Opcode and CtrlSignals field of the format are set to fixed numbers to make instruction encoding unambiguous. The C++ code fragment is given in the Code field. ASMName and ASMParam specify the assembler mnemonic as well as the parameters. In LLVM one LLVM pattern is provided. During the expansion pass of the TargetGen utility (see Section V) additional LLVM patterns are derived by input specialization. E.g. DSRC2 is set to 0 to derive a load operation without calculating the address by an addition. Also DSRC2 is set to DSRC1 to derive a load operation that multiplies the address by two.

```cpp
[ 'LD32' ] = {
  [ 'Format' ] = 'FORMAT_D_DD';
  [ 'Field1' ][ 'Opcode' ][ 'Value' ] = 10;
  [ 'Field1' ][ 'CtrlSignals' ][ 'Value' ] = 0b1000;
  [ 'Code' ] = q {
    DDst = Mem. Read32(DSsrc1 + DSsrc2);
  };
  [ 'ASMName' ] = 'LD32';
  [ 'ASMParam' ] = "$DDst, [$DSrc1 + $DSrc2]";
  [ 'LLVM' ] = {
    [] = ('set', 'DDst',
      ('load', ('add', 'DSRC1', 'DSRC2')));
  };
};
```

Listing 1. Example of an instruction definition inside the ADL.

IV. ADL-BASED SOFTWARE FRAMEWORK

In Figure 2 an overview of the ADL-based software framework is given. The software framework uses the ADL description and the application's C/C++ source code as input. The ADL is processed by the TargetGen utility which is explained in Section V. The C/C++ front-end of the LLVM compiler infrastructure translates the source code into the LLVM Intermediate Representation (LLVM-IR). The LLVM-IR describes the source code by a language-independent, RISC-like instruction set in Static Single Assignment (SSA) form including type and dataflow information. The SSA form is very beneficial for compiler optimization done in the middle- and back-end.

The LLVM front-end and the IR are not completely independent of the target architecture since C/C++ is an inherently platform-dependent programming language. The fundamental C data type “int” typically reflects the target architecture’s general purpose register size. This influences among other things especially predefined preprocessor macros (e.g. INT_MAX). Also the byte order of the target architecture is defined in the preprocessor and is required for writing portable code. During preprocessing macros are resolved and portable code gets irrecoverably target dependent. Therefore, the front-end requires high-level information including the size and alignment of fundamental C data types, the byte order, and the floating-point format from the ADL.

The LLVM middle-end optionally performs platform-independent optimizations on the LLVM-IR, e.g. dead code elimination or constant propagation. It is currently independent of the architecture description of the ADL. However, for some architectures it might be beneficial to active or deactivate some optimization passes and that could be later integrated into the ADL.

Afterwards the LLVM-IR is used as input to the back-end. It translates the IR into machine code of the target architecture.
simulations of partial implementations of the ISA and is therefore very useful for early evaluation of hardware components.

The simulator is capable to switch the simulated ISA at run time. It contains for each ISA a separate instruction tables for decoding and execution of instructions. The initial active ISA can be selected by a command line argument. If not given, the default ISA is used. During simulation the processor state stores the currently active ISA. It can be changed by any instruction using the SWITCHTARGET command inside the simulation code. The following instruction is then decoded using the instruction table of the new ISA.

V. THE NOVEL TARGETGEN UTILITY

The TargetGen utility (target generator) compiles the ADL description into different output formats which are used to realize the retargetable tools (LLVM compiler back-end, assembler, and simulator). The concept of metaprogramming is used by the utility as it generates source code being used to build the different tools of the framework. This implies that the tools must be partially recompiled before a change within the ADL description takes effect.

A general overview of the TargetGen utility is given in Figure 3. It is organized in five passes that are explained next. At first the markup language is parsed into simple tree based data structures. ADL Parsing analyses the content of this tree and converts its high-level content into internal data structures that are organized similar to the sections available within the ADL. Thereby, it is checked if the ADL is semantically correct, especially the references between the sections must be correct.

Expansion is the most complex part of the TargetGen utility. The instructions inside the ADL are expressed in a compact, non-redundant way that is not beneficial for further processing. The instructions description is expanded by rewriting the semantic representation as well as duplicating instructions.

For each instruction (the semantic representation of one instruction) is transformed into semantically equivalent LLVM patterns. Each instruction contains a set of LLVM pattern and for each LLVM pattern a set of rewriting rules is applied. A new LLVM pattern is added to the set until all LLVM pattern have been processed. The LLVM pattern can be rewritten by utilizing commutative law and input operand specialization.

Input operand specialization sets the input operands to constant values or equal parameters. So is it possible to eliminate one binary node within a LLVM pattern by setting the left or right operand to the left or right neutral element of the node. This plays an important role for specification of load/store operations. A typical load instruction is able to load data from a memory address calculated by adding a register with an immediate value. The instruction selector would only match the instruction if an add follows a load node. By setting the immediate value fix to zero the addition is eliminated and the address is equal to the register. This is very generic for the compiler since the result of any address calculation can be stored inside a register. It is also very useful settings two parameters to identical registers. Several RISC ISAs don’t provide any register transfer instruction (MOV), instead OR x, y, y could be used since it provides the same functionality. Also on X86 XOR eax, eax is used to set the eax register to zero since this opcode encoding requires only one byte in comparison to the immediate move instruction which requires two bytes.

Another task of Expansion is the duplication of instructions. To provide a compact representation to express instructions, it is possible to define multiple instruction word encodings within one instruction format. For further processing it is important that one instruction has exactly one instruction encoding, so the instruction using an instruction format with multiple encodings are duplicated accordingly into multiple instructions. A second reason for duplicating instructions is user-defined field formats using code snippets. For each code snippet a new instruction is generated by replacing the fields within the simulation code, assembly syntax, and LLVM pattern by corresponding code snippets defined in the field format.

Correctness Check performs two independent checks. Firstly, it is verified that every instruction has a unique encoding, i.e. two instructions cannot use the same encoding within the instruction word. The unique check is not a trivial task since every instruction may consist of fields with different field types, bit positions, and bit length. Also two fields may only differ because their sets of allowed encodings are disjoint. Secondly, the completeness of the instruction selector is rudimentary checked. Therefore, at least the elementary target-independent LLVM nodes (e.g. add, and, sub) must be covered by a target instruction otherwise the instruction selection may fail. Non-elementary LLVM nodes are replaced by semantically equivalent subgraphs within the Legalize pass (see Section VI) if they are not covered by the target architecture. However, we cannot prove that the completeness check is complete but it gives good hints to ADL description authors.

The last pass, Output Generation, outputs the information stored in internal data structures as C++ and LLVM-specific Target Description (.td) files for the compiler and C++ files for the assembler and simulator. These dynamically generated source files are then used together with static source files to compile the compiler, assembler, and simulator.

VI. LLVM BACK-END GENERATION

In LLVM the back-end translates the LLVM-IR into a target-dependent assembly file of the target architecture. Figure 4 shows a typical design of an LLVM back-end. It is organized in several passes that are sequentially applied to the input data until the last pass generates the final assembly output. Due to the highly modular design of the LLVM compiler only relatively small modifications to the back-end (compared to other compilers) are necessary to support a new ISA. All passes that have been extended to enable user-retargetability are marked within Figure 4.

The source-code of an architectural LLVM back-end is based on so called Target Description (.td) files. The .td files abstract the architecture description within the LLVM code generator and their main purpose is to significantly simplify back-end development. Among other characteristics they describe the register set, instruction set, and calling conventions of the target architecture. During
LLVM compilation the LLVM utility *tblgen* (Target Description To C++ Code Generator) parses the .td files and generates C++ files that are included within several passes inside the code generator. The .td files give the advantage that registers and instructions can be defined in a centralized, pass-independent way to avoid duplicated, error-prone definitions within several back-end passes. The *TargetGen* utility tries to express as much information about the target architecture as possible within the .td files. However, since the .td files are not intended to generate the complete LLVM back-end, additionally C++ source files must be generated by our *TargetGen* utility.

To support multiple ISAs within the back-end we utilized the subtarget feature of LLVM. In general, the subtarget feature is used to inform the code generation process of instruction set variations (e.g. MMX support) for a given architecture (e.g. Pentium II). The architecture can be set by a command line parameter. We used the subtarget feature and generated for each ISA available in the ADL a new architecture. The active architecture/ISA is less than accessible within the individual passes. In the following we explain the passes of our back-end and our extensions enabling user-retargetability and support for multiple ISAs.

In a first pass, *BuiltDAG*, the LLVM-IR in SSA-form is translated into a DAG. This DAG has no restrictions related to data types and instructions. It is referred as illegal since it may contain data types and instructions that are not natively supported by the target architecture and would be unselectable within the instruction selector.

The following *Lowering* pass performs the *Legalize* phase as well as handling of calling conventions. During the *Legalize* phase the illegal DAG is turned into a legal one by replacing unsupported data types and instructions by supported ones. Therefore, the pass has been extended to enable a simple specification of the supported data type and operation. The specification for each ISA is automatically generated by the *TargetGen* utility and is embedded into the source code. The specification of the active ISA is then used during *Lowering*.

To eliminate unsupported data types only a list of supported types has to be provided to the LLVM parent class. Unsupported smaller data types are automatically promoted by LLVM to larger data types (e.g. 1/8/16-bit integer to 32-bit) or larger data types are broken up into smaller ones (e.g. 64-bit integer to two 32-bit) by adding sign/zero extension or truncate operations into the DAG as required to preserve the semantic.

Unsupported operation handling is more complex. For each possible instruction and data type combination it is specified whether it is supported. Additionally, the result data type of supported operations is given. The result data type is important for operations where the result type is independent of the operand’s type. E.g. the results type of a comparison operation could be a 1 or 32-bit integer depending on the target architecture. Based on the specification one of four legalize actions is performed: The operation is supported and retained (legal), replaced by an alternative sequence of operations by the LLVM infrastructure (expand), replaced by the same operation on a large data type (promote), or custom handled using a hook within the *Legalize* pass of the back-end (custom). We use the last action to apply a generic model to express control-flow inside the ADL for instruction selection.

Furthermore, the *Lowering* pass handles the calling conventions. At the beginning and end of a function as well as before and after a function call operations enforcing the calling conventions of the particular function are inserted. The calling conventions for each ISA are partially specified in the .td files that are automatically generated by the *TargetGen* utility. When a function is called, the location (either a register or a stack slot) of each function parameter is calculated from the specification inside the .td files and the corresponding code to transfer the parameters to their location is inserted.

The *Instruction Selection* pass translates the DAG containing target-independent operations into a new DAG consisting of target instructions as defined within the ADL. Therefore, an instruction list including LLVM patterns are provided for each ISA by the *TargetGen* utility as part of the generated .td files. The LLVM patterns represent the behavior of each instruction and are matched to the DAG during instruction selection. Tblgen supports to restrict an instruction to a given subtarget, so only the instructions of the active ISA are considered during instruction selection. One LLVM pattern consists of a DAG including node types and restrictions on the nodes it matches. The restrictions are additionally generated by *TargetGen* and are used to model immediate fields that could encode any arbitrary range or set of numbers (e.g. a 13 bit immediate). During compilation of the back-end tblgen translates the LLVM patterns into C++ code performing the matching operation.

The next pass after *Instruction Selection* is *Scheduling*. *Scheduling* assigns an order to the instructions and transforms the DAG into an instruction list. Within the DAG all necessary dependencies information to perform the scheduling are available. Therefore, no additional code generation by *TargetGen* is required.

The *Register Allocation* pass maps the unbound number of virtual registers that are used within the LLVM-IR to a limited number of physical registers. Therefore, the *TargetGen* utility must express the available registers and register classes for each ISA within the .td files. A register class contains all registers of an ISA of a specific type, e.g. 32-bit integer registers. For each register class and ISA the set of registers available for *Register Allocation* is calculated from the ADL. These are all accessible registers of a register class of an ISA except the instruction pointer, stack pointer, and frame pointer. Additionally, three C++ functions for inserting load/store operations for accessing the stack as well as register move operations are generated by *TargetGen*. Since *Register Allocation* is performed after instruction selection the operations must be created using the native LLVM API without help of the abstract formulation within the .td files. The load/store operations are required for register spilling. If the virtual registers do not fit into the physical registers, some virtual registers will be mapped to the stack frame and temporarily loaded into a physical register before being used.

The *Prologue/Epilogue Insertion* pass generates instructions at the beginning and the end of a function as well as finalizes the stack access instructions. After *Register Allocation* the usage of physical registers is known and the set of physical registers that must be saved/restored within the prologue/epilogue code to match the calling convention can be determined. Therefore, the load/store stack operations are generated by using the C++ functions provided by *TargetGen* for *Register Allocation*. After *Register Allocation* it is also known which virtual registers are spilled to the stack frame and therefore the stack size and layout
can be calculated. The prologue/epilogue code is extended by instructions that decrease/increase the stack pointer by the stack size. Since the instructions are target dependent they must be created by TargetGen generated C++ functions depending on the active ISA. Also since the stack layout is known, all load/store instructions accessing the stack can be finalized by setting their immediate value to the stack position.

The final Code Emission pass converts the instruction list into an assembly file. Therefore, each instruction and register within the TargetGen generated .td files contains an assembly string. The instruction assembly string contains the operands as placeholders that are replaced by the register assembly string or immediate value on output generation. Each operand has a format as specified within the ADL. User-defined fields in the ADL (e.g. used to model condition codes) output no immediate values but assembly strings dependent on their immediate value. These field formats are custom handled inside the code emitter by TargetGen generated C++ functions. On the beginning of each assembly file the target ISA is announced to the assembler by the .target directive. Additionally, we prefix all function names (symbol references and definitions) by the identifier of the active ISA. This enables to compile one function to different ISAs and link all into one executable without producing duplicated symbol errors. The prefix is completely transparent to the C/C++ developer and is only visible when writing native assembly code.

VII. RESULTS

We described the current version of the RISC-like 32-bit ISA of our reconfigurable hardware architecture within the ADL. The ISA was designed very homogeneously to allow efficient execution of compiler generated code. Like IA-64, the ISA consists of two independent register files: one 32-bit data register file and one 1-bit event register file. The event register file is mainly used by the compiler to store results of compare instructions as well as carry flags of additions and subtractions. Conditional jumps expect an event register as parameter and decide by the content of the event register if the jump is taken or not. We use the ISA for evaluation of the software framework and for Design Space Exploration (DSE).

Fig. 5. Influence of the number of registers on executed cycles

Based on the ADL description we retargeted the software framework by generating the LLVM compiler back-end, assembler, and simulator. We used LLVM 2.8 together with the LLVM-GCC 4.2 front-end. We validated the complete framework with a set of applications from different domains by comparing their simulator output to natively compiled and executed versions. The set of applications comprises the JPEG encoder/decoder (used from the MiBench), a fixed-point Fast Fourier Transform (FFT) implementation, a Quicksort sorting algorithm, the Dhrystone benchmark, a fully-unrolled Advanced Encryption Standard (AES) implementation, and a 4x4 integer Discrete Cosine Transform (DCT) approximation as used in H.264. All applications were compiled with maximum performance optimization.

Based on the ISA we performed an initial DSE by varying the number of available physical registers of the ISA. Therefore, we generated the ADL description and software framework automatically by a script. Figure 5 presents the performance impact of the applications when limiting the number of registers stepwise from 48 to 14 (x-axis). On the y-axis the relative number of executed instructions is given in percent. The best (minimum) value per application is set to 100% and any higher value represents the overhead in percent when reducing the register count. As seen in the figure the performance impact is highly dependent on the application. The control-flow dominant Quicksort is only able to utilize up to 19 registers while the data-flow oriented FFT benefits from up to 46 registers. Data-flow oriented applications have larger basic blocks and tend to use more variables and have a higher register pressure. Applications with higher register pressure can benefit more from larger register files on the LLVM compiler.

![Fig. 6. Percentage time requirement for performing DSE](image)

For automatic DSE one important factor is the time requirement for evaluating one design. On a 2x4 core Intel Xeon X5355 CPU running with 2.66 GHz we required 277 seconds in average to compile the modified parts of the software toolchain (compiler, assembler, simulator) and to compile and simulate all five applications. In Figure 6 the percentage time requirements of the individual steps are shown. In our case, 48% (109s) are needed to retarget the software toolchain. The largest application (in respect of lines of code) is the JPEG encoder/decoder which alone required 66s (29%) to compile. Since the simulator reaches 11-13 million instructions per seconds the simulation time is comparatively short.

To demonstrate the capability of the framework to support mixed-ISA applications, we developed an ADL description including two ISAs: one with 32 registers and one with 16 registers. We wrote a synthetic application that performs a Quicksort and FFT. The application is organized in three phases: Initialization, Quicksort, and FFT. Initialization generates the input values for both algorithms as well as the twiddle factors for FFT. We compiled each phase for both architectures and linked all together into one executable. On startup the boot code and main function is always executed with 32 registers. It executes successively the three phases by calling one function per phase. Each phase can be either executed on the 32 register or 16 register architecture. When executing a phase with 16 registers a wrapper function is called that switches the ISA before the phase is executed. The wrapper function is hand-written in assembler and uses the SWITCHTARGET opcode to change to 16 registers, calls the function that executes the phase with 16 registers, and switches the ISA back to 32 registers after return. We can dynamically control the ISA per phase by a command line parameter without recompilation.

Figure 7 shows the relative number of executed load/store instructions of the application for three execution scenarios (7(a) -
The presented ADL and software framework is the fundamental work to enable programmability of our hypermorphic reconfigurable architecture. Our next steps are to extend the ADL and software framework to support clustered VLIW ISAs as well as switching between RISC and clustered VLIW. We also plan to extend the compiler to automatically generate wrapper code for changing the ISA dynamically at run time. Furthermore, we will research automatic generation of selected modules of our hardware architecture from the ADL in order to reflect changes to the instruction set.

ACKNOWLEDGMENT

We thank the German Research Foundation (DFG) for funding this work within the KAHRISMA project.

REFERENCES