Operation Set Customization in Retargetable Compilers

Heikki Kultala, Pekka Jääskeläinen and Jarmo Takala
Department of Computer Systems
Tampere University of Technology
P.O. Box 553, FI-33101 Tampere, Finland
firstname.surname@tut.fi

***Abstract*—The core tool in Application-Specific Instruction Set Processor (ASIP) design toolsets is a retargetable compiler, which can generate efficient code to any processor developed with the toolset. Such a compiler must automatically adapt itself to the operation set supported by the designed processor by emulating missing instructions with other instructions and by selecting custom instructions automatically whenever possible.**

This paper proposes a simplified Directed Acyclic Graph-based recursive mechanism to support operation set customization. The proposed mechanism is capable of generating instruction selectors and architecture simulation models automatically, thus is suitable for fast design space exploration of ASIP operation sets.

I. Introduction

The set of operations (or instructions) the processor supports is an important customization point in *Application-Specific Instruction Set Processors (ASIP)*. For example, if some operations are used very rarely, it can be desirable to exclude them from the operation set and emulate them with other operations. On the other hand, if a complicated sequence of basic operations are used often in the program code, the processor could benefit from a custom hardware instruction for that operation sequence.

In addition to their reprogrammability aspect, ASIPs are often chosen as an accelerator implementation methodology instead of fixed function hardware in order to speed up the design process and to reduce the need for hardware design expertise. As producing new ASIPs from the scratch manually is a laborious and error-prone task, the customization process is usually supported by ASIP co-design toolsets. In order to make the operation set customization feasible in the rapid co-design of ASIPs using such toolsets, the methodology to define new custom operations in the toolset should be at the same time programmer-friendly while still being adequately expressive to derive all the information required to retarget the toolchain.

The programmer-friendliness aspect of an operation description language is not easily shown by scientific experiments. However, when designing a new language, one could try to consider the commonly used existing languages and work patterns of the target users of the new language. ASIPs are mostly used in the embedded systems industry. Therefore, an operation description language should feel familiar to the embedded engineers. The C language [1] is still a dominant programming language in the embedded system industry, thus the syntax of a programmer-friendly operation description language should be familiar to, if not compatible with, the C syntax.

Expressiveness of the operation description format is another point to consider. The description format should include enough information to retarget and provide information for at least the following tools in the ASIP design flow:

1) Instruction set simulator. The main use cases for instruction set simulators include preliminary performance evaluation, which requires cycle-accuracy, and software verification where functional accuracy suffices.

2) Compiler. Typical way of invoking custom operations from high-level source code has been explicit calls as pragmas, “intrinsics“ or inline assembly which require modifications to the source code of the program. This kind of custom operation call is inconvenient as it requires additional software development work, reduces portability, and might be a source of errors that only appear on some platforms. Ideally, the instruction selector of the compiler backend should be able to use the custom operations automatically to speed up the generated code without any changes to the source code. However, the reality is such that more complex the custom operations get, less likely it is that the instruction selection phase of the code generation can find the operation chains to replace automatically with the custom operation call without resorting to complex algorithms with long compilation times [2], so there should also be a feasible method for manually calling the custom operations.

3) Processor hardware generator. The final synthesizable implementation of the processor core includes the described custom operations in the function units of the designed processor’s datapath. Thus, the hardware description language (HDL) implementation of the special function units should be realizable from the operation language format.

This paper proposes a simplified operation set description format based on an imperative recursive *Directed Acyclic Graph (DAG)*-description language. The format can be used to describe the semantics of custom operations as *Data Flow Graphs (DFG)* that recursively refer to more primitive operations. The entry language to describe the operation semantics graphs is familiar to C programmers while the operation description format is expressive enough to retarget all the tools

in an ASIP co-design environment.

The operation definition methodology presented in this paper has been used successfully in the *TTA-based Co-design Environment (TCE) [3]*. The paper provides examples to illustrate the programmer-friendliness aspect and describes how the different tools are driven by these operation definitions.

The rest of the paper is organized as follows: Section II reviews some of the existing solutions to customizing instruction sets of processors. Section III describes the operation set abstraction layer of the TCE toolset. Section IV introduces the operation DAG language. Section V discusses the implementation of the toolchain retargeting based on the operation set descriptions. Section VI shows an example case of a program with automatically selectable custom operations. Section VII lists some ideas for improvement and Section VIII concludes the paper.

II. Related Work

There exists several commercialized customizable architectures which include operation set customization capabilities without requiring modifications to the compiler. In addition, there are compilers that target multiple non-customizable architectures, but are not "runtime retargetable". Such compilers often provide languages for operation set abstraction in order to make the compiler backend easier to port to different architectures.

Tensilica Xtensa is a customizable processor core that has its own operation set customization language, *TIE*, which can be used for automatic hardware generation. The operation description language is also independent from the actual implementation of the processor, which makes it easy to use the same custom operations in multiple processor designs. The compiler, however, cannot use this language in instruction selection to automatically use custom instructions. [4]

nML [5] is a machine description language which can be used to model the architecture of a processor, including its operation set. A compiler called CBC [6] can automatically use custom operations based on their nML-based specifications.

The *LISA* language, which is used in *Synopsys Processor designer*, also allows specifying custom operations and it can be directly used for hardware generation and simulator retargeting [7]. There is also a compiler which uses LISA language as a partial input for the instruction selection, but in addition to the LISA language operation specifications it needs extra mapping information given as tree-like patterns [8].

LISA, nML and TIE languages, however, have some properties which reduce their usability for quick operation set exploration: They are all relatively low-level, optimized for automatic hardware implementation. The operation definitions are somewhat lexically complex and existing custom operations cannot be used as building blocks for more complex custom operations. In LISA and nML languages, the operations are tightly tied to the given architecture: there is no option to define a "custom operation library" and easily share the custom operation definitions between processor architectures.

GCC and *LLVM* compilers both have specified their own pattern language, which can be used to define semantics of the operation set as input to the instruction selection [9], [10]. The languages are used solely for making the compiler backend easier to port into different architectures.

The pattern formats of both compilers are based on functional language expressions which call functions that return a single value, having no way to represent intermediate variables. GCC, however, can have multiple of these expressions for single operations, allowing multiple return values. But as these expressions cannot share temporary variables, the definitions of more complex operations may get longer, and practically leads to "copy-paste programming".

Most operation set extension languages mentioned above do not differentiate the concept of "instruction" from "operation"; That is, they are also used to define where the data can come from, and where it must go. This limits the customization flexibility by forcing extra information about the architecture implementation to be stored into the operation definition. This makes it harder to share same custom operation descriptions between different architectures, and makes it impractical to use smaller custom operation definitions as building blocks for larger operations.

III. Operation Set Abstraction Layer

TTA-based Co-design Environment (TCE) [3] is a toolkit for designing ASIPs based on the *Transport Trigged Architecture (TTA)* [11] template. The toolkit contains a graphical processor designer program, a simulator which can simulate any processor designed with the toolkit, a retargetable compiler which can target any processor designed with the toolkit, programs to generate the final VHDL description of the processor and also some automatic tools for optimizing the processor design.

TCE is driven by an *Architecture Description File (ADF)* which contains all the programmer-visible aspects of the designed TTAs. The operation set information is separated to a reusable database called *Operation Set Abstraction Layer (OSAL)*. OSAL contains information about all the operation descriptions currently available to the toolset. These operations are read from XML files which contain the information of the operations required by the various retargetable tools. The user of the toolset can define his own operations to the OSAL database using a graphical tool.

The OSAL *OPeration Property (OPP)* XML files contain vital information about the operations such as the number of operands and results. OPP contains information needed by the compiler to produce valid code that uses the operations. For example, whether the operation reads or modifies memory or has other side effects, which operand of a memory operation is an address, which is the data, etc. These properties are mainly used by the instruction scheduler to better optimize the code and to avoid illegal code motions.

The OSAL database does not differentiate the operations by their complexity. A custom operation and a basic operation are described similarly at the database level. That is, both simple operations such as integer addition and more complex

application-specific operations are described with the same format. However, in the point of view of the several tools and the end user of the co-design toolset, the operations stored in the OSAL DB can be categorized as follows:

1) Required operations. If the target machine does not contain all these operations, the compiler cannot guarantee that it can compile all code written in the supported programming languages for it. This set is dependent on the capabilities of the compiler and the requirements placed by the supported programming languages.
2) Base operations. The set of operations that are distributed with the toolset. The idea of base operations is that their semantics is known by the compiler. Therefore, if available in the target machine, the compiler might be able to exploit them to accelerate or enable specific pieces of code.
3) Custom operations. User-defined operations with specific purpose. These can or can be selected automatically, depending on how the execution semantics is described.

The semantics information of the operations can be described with two main methods:

1) Operations with DAG definitions. These operations might be selectable automatically by the compiler, depending on the complexity of the DAG pattern and the capabilities of the implemented instruction selection algorithm.
   Operations that are chains of Required Operations should describe DAGs as there is high probability the compiler instruction selector can select them automatically.
2) Operations not containing a DAG. Compiler cannot automatically use these. A separate simulation behavior description (in C++) is required to simulate the operation in the instruction set simulation. These operations are usually complex operations with state data or I/O operations.

IV. Operation Semantics as Recursive DAGs

The operation semantics description language was chosen to be based on a recursive *Directed Acyclic Graph (DAG)* description language. The DAGs form the *Data Flow Graph (DFG)* of the operation with the possibility to refer to any other operation in the OSAL, thus forming a database of DAGs that can be recursively matched and interchanged with each other.

A common form of internal representation (IR) for instruction selection in compilers is a DFG for each basic block. The instructions in the basic block have a strict weak ordering, which can be represented as a DAG. The input DAG for the basic block and the operation DAGs for the operations can then be pattern-matched to select operations which implement the IR with the target's operations.

The OperationDAG language contain just two kind of statements: temporary variable declarations and operation execution calls to smaller operations. Inputs to operation execution calls may be either inputs of the operation, temporary variables, or constants. Output destinations of operation execution calls may be either temporary values of output values.

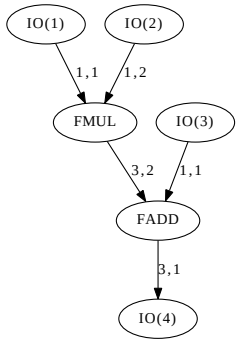

Fig. 1. DAG of the multiply-addition operation.

Example code of a floating-point multiply-addition operation is shown below and the corresponding DAG is shown in Figure 1.

```
SimValue mulRes;
EXEC_OPERATION(FMUL, IO(1), IO(2), mulRes);
EXEC_OPERATION(FADD, mulRes, IO(3), IO(4));
```

A more complex operation description language language supporting unlimited looping etc. could allow more complex operations to be described, but would have severe drawbacks:

- Operation descriptions with loops could not be selected by an instruction selector which only operates one basic block at time, and global instruction selector which operates on multiple basic blocks simultaneously would be more complex, leading to increased compilation times.
- Simplicity was one of the major goals in the operation definition language, and adding loops would have made it more complex to use.
- Loops would be problematic if the same operation descriptions are used for automatically generating the hardware for the custom function unit.

Language with limited looping with fixed loop counts could allow some operations to be defined with smaller code length, but it would be computationally equally expressive as the simple DAG based model, as the loops could be completely unrolled into the DAG form.

In addition, most of the description code size benefits from a cyclic operation definition language would offer can be realized with the recursion between operation definitions. One operation definition can be a subgraph of another graph, so that in the description language the operation definition of the bigger operation can just call the definition of the smaller operation. Operation, however, cannot call itself, or any other operation that contais a call to the original operation, as it would produce the unsupported cyclic graphs.

Compared to tree-like expression-based languages, such as the LLVM's instruction patterns, the DAG-based imperative language allows using same temporary value multiples times,

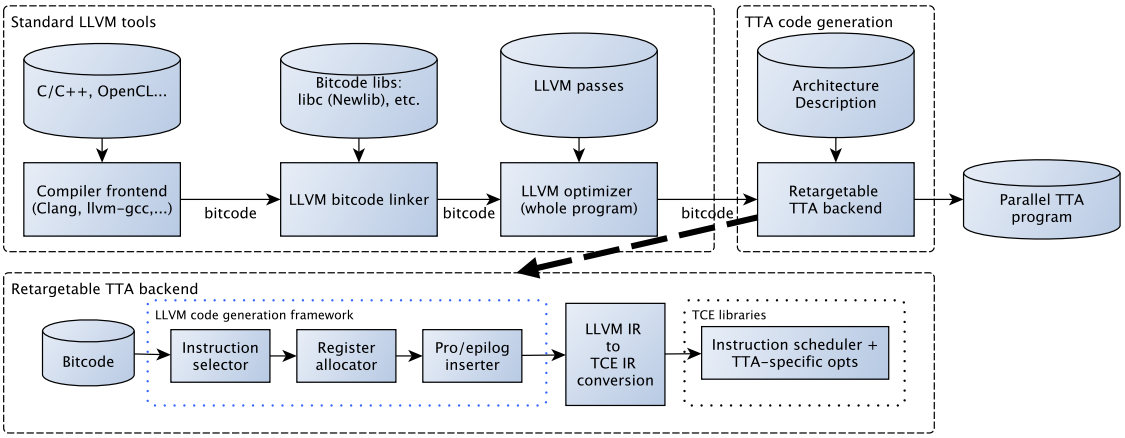

Fig. 2. Overview of tcecc internals.

and allows multiple outputs. For example, the “addsub” operation which computes both the sum and the difference of the two input values.

V. Toolchain Retargeting

The TCE compiler (tcecc) uses the LLVM [12] compiler infrastructure extensively. The internals of the tcecc are illustrated in Fig. 2. Tcecc uses the LLVM’s Clang frontend for parsing the high-level language code into LLVM’s internal representation, after which LLVM passes are used for various optimizations.

The target-specific backend code is loaded from a dynamic library which is generated for the target architecture by tcecc. LLVM uses code from this library to perform instruction selection and register allocation for the architecture. After these phases the code is converted from an LLVM internal representation into the internal representation of the TCE toolset and the final instruction scheduling and bundling is done by the TCE instruction scheduler.

The processor simulator can adapt itself to simulate any custom operation available in the processor if the behaviour description of the operation is provided to the simulator. The two different ways described in the previous section to describe the operation semantics can be used by the simulator to simulate the behavior of each operation. In case of recursive OperationDAGs, the simulator executes the DAG recursively, and in case of the C++ described simulation code, the behavior code is loaded as a dynamic plugin to the simulator engine.

Regardless of which of the two ways to describe operation simulation behavior, the simulation is always instruction cycle-accurate; the timing (the pipeline model) of the operations is not handled by the operation behaviour code, but by the simulation model of the execution pipeline of the processor. When the operation is triggered, the operation behaviour model is called to determine the results of the operation, and the result is written to the result port at time decided by the execution pipeline model of the function unit described using the ADF format.

The compiler is retargeted by generating new ”backend plugins“ from the ADF and the OSAL. All available operation definitions and the target architecture definition file are read by a tool called TDGen. This tool generates the LLVM target description format tables and source code for LLVM target classes, which describe all operations available in the arhitecture to the LLVM code generation framework, and creates emulation patterns for operations that are not available in the target architecture.

If an operation contains a DAG description of the semantics of the operation as a DFG of multiple operations, this is converted into an LLVM instruction selection pattern. The pattern is then picked up by the instruction selection algorithm implementation which might or might not be able to detect a matching pattern in the program code.

VI. Case Study

The sha program of the CHStone benchmark [13] was chosen as the case study program. It was observed that the program spends most of its time in loops containing repeated sequences of simple logic operations.

Such chains of logic operations can be easily executed in single clock cycle in hardware, reducing both operation count and the total latency of the calculations. Thus, for this experiment, we created a set of custom operations for simple chains of logic operations:

- 3-operand xor operation, shown in Figure 3
- 4-operand xor operation, shown in Figure 3
- 3-operand and operation
- 3-operand or operation
- bit selection operation
- bitwise 3-operand majority instruction

Source code of the program was not modified to use the custom operations. The compiler automatically managed to use the other instructions except the 3-operand “and” and the “majority” instructions, even though the original C code contained code which looked like the majority operation. It is assumed that some optimization made by the compiler before

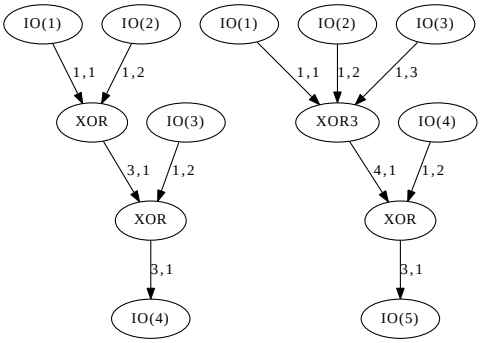

Fig. 3. DAGs of the 3-operand XOR operation and the 4-operand XOR operation which is defined using the 3-operand xor operation.

the instruction selection transformed this operation sequence into a form that it could not be matched by the DAG.

The program was simulated with two processor architectures, the first one contained only basic logic operations, the second one contained also the custom operations mentioned above.

TABLE I
BENCHMARK RESULTS IN THE SHA CASE WITH AND WITHOUT AUTOMATICALLY USED CUSTOM OPERATIONS.

Measurement	no custom ops	with custom ops	difference
Executed operations	559913	502361	-10.3 %
Instruction cycles	467042	434417	-6.9 %

From the benchmarks we see that the total operation execution count drops by 10.3 % when the custom operations are in use. This shows that the program still contains lots of other operations than the operations replaced by the custom operations, and that there are severe performance bottlenecks elsewhere. The processor with the custom operation spends 6.9% less instruction cycles executing the program. The cycle count reduction is smaller than the operation count reduction, because custom operations require more operands, which have to be read from the registers and moved to the function units. However, this benchmark proved that the compiler can use custom operations automatically based on the OperationDAG descriptions.

VII. Future Work

TCE's OSAL DAG language supports multiple outputs, but as these cannot be converted into LLVM instruction selector patterns in its current state, the TCE compiler cannot currently automatically use these operations. In addition to improving the LLVM instruction selector's support for MIMO operations, another interesting point of improvement is to implement compiler transformations that try to transform the input code to a format where the more complex OperationDAGs can be matched.

The basic operation set of TCE currently does not define a select operation which selects value from two sources based on some condition value. This prevents the instruction selector to be able to match custom operations which contain parts that conditionally select data from multiple data sources.

Hardware implementation generation of special function units (SFU) based on the OperationDAG and the ADF is not yet implemented, but should be relatively trivial to add.

VIII. Conclusions

The proposed Dynamic Acyclic Graphs based operation set description model allows defining the operation semantics in a format that can be used to automatically select custom operations by the compiler. We demonstrated that the automatic instruction selection works in practice for small custom operations. The same model can also be used for retargeting the instruction set simulator.

Separating the abstractions of "operation semantics" from the "architecture" and the actual instructions which are used to execute operations allows more complex custom operations to be easily defined using simpler custom operations as building blocks.

References

[1] D. M. Ritchie, "The development of the C language," in *ACM SIGPLAN Conf. History of Programming Languages*, Cambridge, Massachusetts, United States, Apr. 20–23 1993, pp. 201–208.

[2] D. Ebner, F. Brandner, B. Scholz, A. Krall, P. Wiedermann, and A. Kadlec, "Generalized instruction selection using ssa-graphs," in *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, ser. LCTES '08. New York, NY, USA: ACM, 2008, pp. 31–40.

[3] P. Jääskeläinen, V. Guzma, A. Cilio, and J. Takala, "Codesign toolset for application-specific instruction-set processors," in *Proc. SPIE Multimedia on Mobile Devices*, San Jose, CA, Jan. 29–30 2007, pp. 65 070X–1 – 65 070X–11.

[4] R. E. Gonzalez, "Xtensa — A configurable and extensible processor," *IEEE Micro*, vol. 20, no. 2, pp. 60–70, 2000.

[5] A. Fauth, J. V. Praet, and M. Freericks, "Describing instruction set processors using nML," in *Proc. 1995 European Conference on Design and Test*, Paris, France, March 6–9 1995, p. 503.

[6] A. Fauth, G. Hommel, A. Knoll, and C. Müller, "Global code selection of directed acyclic graphs," in *Proceedings of the 5th International Conference on Compiler Construction*, ser. CC '94. London, UK: Springer-Verlag, 1994, pp. 128–142.

[7] O. Schliebusch, A. Hoffmann, A. Nohl, G. Braun, and H. Meyr, "Architecture implementation using the machine description language lisa," in *Design Automation Conference, 2002. Proceedings of ASP-DAC 2002. 7th Asia and South Pacific and the 15th International Conference on VLSI Design. Proceedings.*, 2002, pp. 239 –244.

[8] M. Hohenauer, H. Scharwaechter, K. Karuri, O. Wahlen, T. Kogel, R. Leupers, G. Ascheid, H. Meyr, G. Braun, and H. van Someren, "A methodology and tool suite for c compiler generation from adl processor models," in *Proceedings of the conference on Design, automation and test in Europe - Volume 2*, ser. DATE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 21 276–.

[9] The LLVM Team, "Writing an LLVM Compiler Backend." [Online]. Available: http://llvm.org/docs/WritingAnLLVMBackend.html

[10] FSF, "RTL Template - Gnu Compiler Collection Internals." [Online]. Available: http://gcc.gnu.org/onlinedocs/gccint/RTL-Template.html

[11] H. Corporaal, *Microprocessor Architectures: From VLIW to TTA*. John Wiley & Sons, 1997.

[12] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Generation Optimization*, Palo Alto, CA, March 20–24 2004, p. 75.

[13] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis," *Journal of Information Processing*, vol. 17, pp. 242–254, 2009.