Codesign toolset for application-specific instruction-set processors

Pekka Jääskeläinen, Vladimír Guzma, Andrea Cilio, Teemu Pitkänen, and Jarmo Takala
Tampere University of Technology, P.O. Box 553, FIN-33101 Tampere, Finland

*

ABSTRACT

Application-specific programmable processors tailored for the requirements at hand are often at the center of today's embedded systems. Therefore, it is not surprising that considerable effort has been spent on constructing tools that assist in codesigning application-specific processors for embedded systems. It is desirable that such design toolsets support an automated design flow from application source code down to synthesizable processor description and optimized machine code. In this paper, such a toolset is described. The toolset is based on a customizable processor architecture template, which is VLIW-derived architecture paradigm called Transport Triggered Architecture (TTA). The toolset addresses some of the pressing shortcomings found in existing toolsets, such as lack of automated exploration of the "design space", limited run time retargetability of the design tools or restrictions in the customization of the target processors.

Keywords: Application-specific instruction-set processors, codesign toolsets, retargetable compilers, machine descriptions, processor descriptions, instruction-level parallelism, transport-triggered architectures

1. INTRODUCTION

The complexity of today's embedded systems has made software implementation on a programmable processor more interesting over fixed-function hardware implementations due to its flexibility. However, software implementations typically require larger area, consume more power, and provide lower performance. In order to improve the efficiency of software implementations, the processor architecture can be customized according to the needs of the given application or application domain, hence the name application-specific instruction-set processor (ASIP). The instruction set of an ASIP can be extended with instructions that help the application in performing its task, and instructions which are underutilized can be removed from the instruction set.

Considerable effort has been spent on software tools that assist and automate the process of codesigning application-specific processors. Efficient hardware description and code generation on one side, and support for a vast range of different target architectures on the other side are contrasting goals. Existing solutions strike different trade-offs between these two requirements, from minimally customizable, predefined target processors and tools fully retargetable at run time, to very flexible and general architecture models and compile-time retargetability.

Several design environments supporting tailoring of processor architectures are already provided by commercial companies, e.g., Xtensa[1] by Tensilica, Atmosphere Development Environment by Adelante Technologies and

Further author information: (Send correspondence to P.J.)
P.J.: E-mail: pekka.jaaskelainen@tut.fi
V.G.: E-mail vladimir.guzma@tut.fi
J.T.: E-mail: jarmo.takala@tut.fi
T.P.: E-mail: teemu.pitkanen@tut.fi
*Copyright 2007 SPIE and IS&T.
This paper was published in Proceedings of Multimedia on Mobile Devices and is made available as an electronic reprint with permission of SPIE and IS&T. One print or electronic copy may be made for personal use only. Systematic or multiple reproduction, distribution to multiple locations via electronic or other means, duplication of any material in this paper for a fee or for commercial purposes, or modification of the content of the paper are prohibited.

the Jazz DSP Processor by Improv Systems. In these systems, the architecture customization is still rather limited. Design environments providing higher level architecture customization are, e.g., the LISA Processor Design Platform,[2] ASIP Meister,[3] and CHESS/CHECKERS.[4] The key element of these toolsets is the instruction set. That is, the customization starts by modeling the target's instruction set. The processor modeling languages, e.g., nML[5] and LISA [2] model the behavior, structural properties, instruction encoding, and even assembly syntax. Once the instruction set is defined, the processor architecture description is generated semiautomatically and application code is generated for the specific architecture either with a retargetable high-level compiler or an assembler. An alternative approach for processor customization is reported by Corporaal[6] where the customization is based on defining computational resources rather than the instruction set. The design method is based on the use of a architecture template, transport triggered architecture (TTA), and a toolset that includes code generation, simulation and semi-automatic design space exploration[7] using the template. However, the toolset is a bit inflexible in certain parts. For example, the model of function unit does not separate abstraction of operation from abstraction of function unit, the architectural simulator does not include debugging capabilities, and several algorithms that require improvements are quite fixed to the toolset code. These reasons, among others, led to the implementation of a new toolset that is presented in this paper.

In this paper, a new toolset for processor customization is presented which is based on the TTA template. The toolset called TTA-based Codesign Environment (TCE)[8] clearly separates structural properties from the operation semantics and the instruction encoding (a separate, configurable subsystem) unlike in previous systems. TCE can be used to codesign a vast range of target processors, with varying capabilities and performance levels from very simple processor architectures to the extent that the designed processor core can be seen as a complete system-on-chip. The problem of supporting design of a large and diverse set of highly customized processor architectures in an efficient way is addressed by instantiating the target architectures from a well-defined, highly parametrized architecture template. Main goal of TCE is to provide a solid platform for easy experimentation of research ideas around TTAs and toolsets for rapid codesign of application-specific processors.

The rest of this paper is organized as follows: Section 2 describes the processor architecture paradigm used in the toolset. Section 3 describes the main file formats used in customizing the processors in TCE. Section 4 presents an overview of the design flow of TCE. Finally, Section 5 summarizes the paper.

2. TRANSPORT TRIGGERED ARCHITECTURES

The architecture template of the proposed toolset is based on a class of architectures called transport triggered architectures.[9] The structure of TTA is highly modular and scalable in nature, which is one of the main criteria in selecting an architecture template for a toolset that provides automated design space exploration and rapid design of new processors from existing "building blocks".

The following subsections briefly present the architectural structure, the unique programming model, the main drawbacks and limitations, and some experiences of the performance of TTA.

2.1. Architectural structure

The principal idea of TTA is to move complexity from the processor hardware to the compiler. The basic principle is the same as in VLIW which allows programs to define explicitly the operations executed in each function unit, only that TTA brings this idea one step further. In case of TTA, the control logic and interconnection complexity are simplified by making the fine-grained control of the processor data transports the programmer's responsibility. This implies, for example, the following improvements to the architecture in comparison to VLIW:[9]

- Reduced register file (RF) complexity.

  VLIW: Number of register file ports is dimensioned for the worst case (all function units simultaneously read their inputs from and write their outputs to the register file). In addition, adding a new function unit (FU) to the architecture increases the complexity of all connected RFs at least linearly with the count of FUs.

  TTA: All reads and writes to the RFs and FUs are programmer-visible, and all connections between units (RFs and FUs) can be defined differently for each designed processor according to the application's requirements.

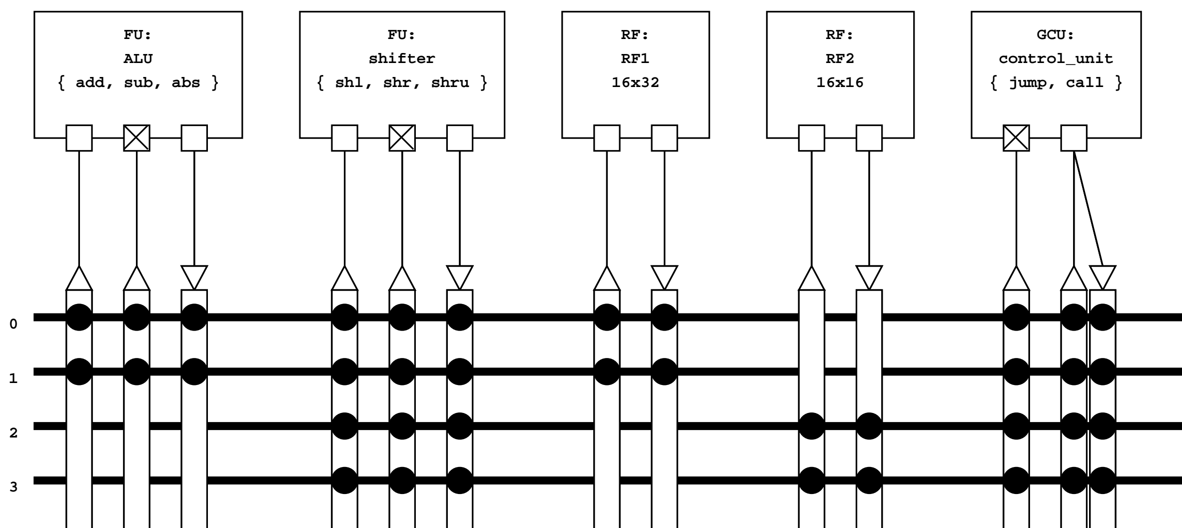

Figure 1. Example of a simple TTA processor with four buses, two function units and two register files.

- Reduced register bypass complexity.

  VLIW: Register bypass logic complexity increasing with $O(N^2)$, where N is the number of FUs.

  TTA: Connectivity between units (also between two FUs) can be fully specified for each processor design and registers can be bypassed in software due to explicit data transport programming.

TTA processors are built out of independent function units and register files connected by a transport network. Figure 1 depicts the data path of a TTA processor with two function units ('ALU' and 'shifter'), two register files (one containing 16 bit registers, the other 32 bit registers), and a control unit (implementing control flow operations 'jump' and 'call'). The four buses allows maximum of four parallel data transports.

The transport network consists of buses connected to ports of function units and register files by means of "sockets". Sockets provide means for programming TTA processors by allowing programmer to select which bus-to-port connections are enabled at each cycle. This way data transports taking place in each cycle are programmed by defining the source and destination socket/port connection to be enabled for each bus. The example architecture in Figure 1 is not fully-connected as can be seen from the black circles that mark the connections between the unit ports and the buses. In this processor, for example, it is not possible to directly transport data from 'RF1' to 'ALU' input ports due to missing connectivity.

A function unit implements a set of operations which the programs trigger through data transports. As explained in the next section, TTA programs transport the operands and the results of the operations through function unit ports and the triggered operation (opcode) is defined "as a side effect" in the move to the triggering function unit port. In Figure 1 the triggering ports of function units are marked with 'X'.

Off-core communication is handled by function units with external ports. For example, load-store units in TTA are function units that implement memory-accessing operations and are connected to a memory through an external port. Another example is an function unit that acts as a simple interface to external peripherals through custom operations. Other means for off-core communication supported by the TTA template of TCE include volatile registers and memory-mapped I/O.

A TTA control unit is a special kind of function unit that fetches and decodes instructions of the program and controls the execution flow by implementing jump operations.

Conditional execution is implemented with predicates (or guards). If the guard evaluates to non zero, the guarded data transport is committed, otherwise it is skipped. The predicate of guard can be any register in

```
add r3,r1,r2                                r1 -> add.1; r2 -> add.2;
add r4,r1,r3                                add.3 -> add.2; ...;
                                            add.3 -> r4; ...;
```

Figure 2. Two ADD operations in operation-triggered architecture (left) and in TTA (right).

a register file or directly an output port of an function unit. The latter case is used, for example, to implement conditional execution depending directly on the comparator unit's results, and the former can be used to implement predicate registers. Each transport bus can support an independent set of guards, thus every data transport can be made conditional independently of the others.

2.2. Programming model

TTA evolved as an answer to the complexity of scaling instruction-level parallel processors for very high performance.[10] TTA addresses the scalability problems by changing the programming paradigm. Instead of programming the processor by defining operations executed in function units and relying on hardware control logic to route the data transports from source to destination at run time, TTA programs explicitly control the transport network (effectively, a programmable bypass network). The operations are triggered as "a side effect" of writing "triggering input ports" of the function units that implement them. The compile-time control of transport resources enable to tailor the number of connections between units, buses and register files to the average demands of the application rather than the worst case.

Figure 2 shows an example of a short RISC program that adds up the content of two registers, r1 and r2, and adds the result to r1 again, and the corresponding TTA code, which takes four data transport definitions, commonly called "moves".

The unique programming model opens up new optimization opportunities to the compiler. For example, the result of an operation can be directly moved into an input of another operation, without storing the value in a register, given the result is not used elsewhere. In the example on Figure 2, this optimization is utilized in moving the result of the first addition operation directly into the input of the second addition operation. This TTA-specific optimization is called software bypassing,[11] as it effectively implements the register bypassing common in pipelined processors in software. Another example of TTA specific optimizations is called operand sharing in which operand of successive operations performed on the same function unit can be shared, eliminating an unnecessary data transport. This is exploited in the example of Figure 2: the value of r1 is written to the adder only once.

2.3. Limitations and drawbacks

One drawback of TTA considered serious by some is due to the abundance of programmer-visible details in the architecture. That is, a full context save in case of a TTA means in the worst case that all internal registers in all function units need to be saved. Thus, it is very expensive to implement a full context switch required by external interrupts and pre-emptive multitasking. Therefore, porting a pre-emptive multitasking operating systems into a TTA environment is currently considered not feasible. In addition, due to lack of external interrupt support, TTA processors are mainly used in the role of accelerators or "slave processors" that always "run to completion" and communicate by polling.

Another common criticism against TTA is its very large instruction width, even when compared to VLIW processors. A TTA instruction consists of data transport definitions, one for each bus in the machine. If high instruction level parallelism is desired, there has to be enough buses connecting the units in the machine to keep them busy. More buses leads to more "move slots" in the instruction, thus the instruction width can grow extremely large. Therefore, the use of the instruction compression or the variable length instruction encoding, to reduce the size of required instruction memory on chip, is basically a must in larger TTA-based designs. Instruction compression in TTA has been researched at TUT during recent years, and produced some promising solutions in terms of savings in chip area and energy consumption.[12]

Table 1. Parameters and "degrees of freedom" of the architecture template.

register files / immediate units:	# of registers, # of ports, register bit width
function units:	operation set, # of ports, port bit width
operations:	semantic properties, simulation behavior
for each operation in each FU:	resource vector, latency, bindings of operands to FU ports
buses:	bit width, set of supported guards, immediate width
sockets:	connections to ports of units (FUs and RFs), connections to buses
guards:	inverted or plain evaluation, guard term (FU output port or a register)

Finally, although the unique programming model of TTA clearly has its benefits, its flip side is that it is quite challenging to implement an efficient compiler for this kind of architecture with abundance of programmer-visible processor details.

Nevertheless, we see these limitations and drawbacks as challenges that are superable and look forward to tackle them in our research. For example, the need for full context save can be eliminated with compiler assisted instrumentation of the program, which is a sort of trickery that is feasible only in the case the program and the processor are codesigned, as is the case in our toolset.

2.4. Experiences on performance

Our experience of TTA is that it's a very promising architecture paradigm for implementing DSP applications, or as a hardware accelerator implementation technology. It is especially suitable for applications which require speed, low power, or the combination of the two. TTA is capable of lowering the speed and power dissipation cap between DSP processors and ASIC designs. The power of TTA is emphasized when tailoring a TTA processor for a certain DSP application, or a set of DSP applications. The following presented two processor design cases show that TTA is capable of competing with fixed ASIC designs in speed and power dissipation.

One of the experimental processor designs[13] was built for computing the 1024-point FFT. The core uses two-ported memory. The execution of 1024-point FFT takes 5234 instruction cycles, thus the overhead to the theoretical lower bound with a dual-port data memory (5120 cycles) is only 2% (114 cycles). The designed processor consumes 74 mW of power while running at 250 Mhz. These rates are comparable to the ones that can be achieved with ASIC designs.

Another design experiment[14] presents an implementation of the mixed radix algorithm, which uses both radix-4 and radix-2 butterflies to generate a FFT processor capable for all transforms of power of two from 32 to 16384 points. The processor uses two memory ports, and with larger FFTs (above 256 points), the overhead when compared to the theoretical lower bound is quite low, ranging from 5,5% to 0,02%. The power dissipation for 1024-point transformation is 82 mW when executing at 250 Mhz clock frequency.

3. PROCESSOR CUSTOMIZATION IN TCE

The TTA processors supported by TCE are instantiations of a predefined template. The number of parameters and the degree of customization within the template is quite high. Table 1 lists the most important template parameters and customization points. For each type of processor component (bus, function unit, register file, socket, and so on) there can be multiple instances in the architecture. The listed parameters apply independently to every instance of a given type.

An important point to keep in mind while customizing processors in TCE is that we have separated the abstraction of hardware operations to two parts: Operation semantics and implementation of an operation in each FU. This is derived from the observation that the same operation semantics (e.g., addition of two integers) can be implemented in multiple ways in multiple FUs, and thus have different latencies and resource usage patterns. Therefore, processor architecture designs in TCE are defined using two main databases: Architecture Definition File (ADF) and Operation Set Abstraction Layer (OSAL). The resource usage patterns and the latencies of operation implementations in each function unit of the designed processor are defined in ADF,

whereas all target-independent properties of operations, including their architectural simulation behavior, are stored in the system-wide OSAL database. This division allows reusing operation definitions stored in OSAL in multiple ADF designs.

The operation execution pipeline model of ADF is very general, and allows expressing complex resource usage patterns for the operation implementations in each function unit. Each operation implementation can have an independent pipeline, or share hardware resources with other operations in the same function unit. For the operation semantics part, OSAL lets the designer freely customize the operation set of the target processor with arbitrary functionality. The usual use case is to convert longer operation sequences to a composite custom operation to boost performance.

The main file edited during the processor design cycle in TCE is the ADF, which is used to define all the structural properties of the designed processor, such as types and counts of function units, operation pipelines implemented by the units, types and counts of register files, and the connectivity. An ADF provides enough details of the target to allow programmer (or compiler) to write correctly functioning programs for the processor and simulate the program in the architecture simulator.

ADFs are generated either automatically by the automated design space exploration tool or manually using a graphical tool called Processor Designer. One thing worth noting is that as ADF describes only architectural properties, in order to generate the final synthesizable processor description, each function unit and register file in ADF needs to be mapped to an implementation. For this, we use an additional file format called Implementation Definition File (IDF) which is used to describe the location of implementation data for each component in the architecture. This kind of division between architectural and implementation details allows certain level of architectural processor design space exploration without implementations available for all components in the architecture.

The IDF is constructed by selecting implementations for each architectural component from a database storing information of available implementations for function units and register files. This database is called the Hardware Database (HDB) and is essential in the final phase of the design flow which generates the synthesizable hardware description. The database includes synthesizable descriptions and properties of the interfaces of the implementations required for connecting the implementation with the rest of the processor components.

3.1. Operation Set Abstraction Layer

The description of operation semantics in Operation Set Abstraction Layer (OSAL) is divided into two parts. The static properties of the operations are stored in XML data files and the architecture simulation behavior of an operation is defined as a C++ class. The behavior models are compiled into dynamic libraries that can be loaded into the architecture simulator without recompilation of the toolset.

Static properties of operations includes the number of inputs and outputs, and important details that are needed by the compiler backend to produce semantically correct optimized parallel programs. These properties include, for example, whether the value of an operation input is used to compute a memory address, whether inputs are commutative, and whether the operation uses state data. This is largely the same type of data that is stored in the "Opcode Attribute Descriptors" part of machine description database in the Elcor compiler research infrastructure.[15]

Figure 3 shows the operation property definition for the ADD operation. The input operands of an addition are commutative, which is marked by the *can-swap* property. The example in Fig. 4 describes the architectural simulation behavior of the operation: the first and the second operand (id 1 and id 2), treated as integers, are added up, and the result is written to the output with id 3. It should be noted that operation latencies are not part of OSAL architectural behavior definitions. The programmer-visible latencies of operations are modeled in the architectural simulation according to the function unit pipeline definitions in the simulated ADF files.

One of the most important "degrees of freedom" of the TCE architecture template is the extensible operation set. TCE allows to choose which operations of a predefined pool are supported by the target processor, and to add new, user-defined custom operations. The custom operations are defined in exactly the same way as the predefined "base operations" such as ADD or JUMP.

```
<operation>
  <name>ADD</name>
  <inputs>2</inputs>
  <outputs>1</outputs>
  <in id="1">
    <can-swap> <in id="2"/> </can-swap>
  </in>
  <in id="2">
    <can-swap> <in id="1"/> </can-swap>
  </in>
  <out id="3"/>
</operation>
```

Figure 3. Properties of operation ADD.

```
#include "OSAL.hh"

OPERATION(ADD)
TRIGGER
    IO(3) = INT(1) + INT(2);
    RETURN_READY;
END_TRIGGER;
END_OPERATION(ADD);
```

Figure 4. Simulation behavior definition of operation ADD.

```
result.sum = a + b;
result.diff = a - b;
```

```
#include "userdef.h"
#define addsub(a,b,c,d) \
  __userdef_22__("addsub",a,b,c,d)
. . .
addsub(a,b,result.sum,result.diff);
```

Figure 5. Piece of plain C code (left) and equivalent with ADDSUB custom operation (right).

It is well known that the use of custom operations is essential in boosting ASIP performance. For example, if profiling data suggests that most of program execution time is spent adding and subtracting the same operands, one could implement a custom operation ADDSUB which computes both the sum and the difference. Finding beneficial composite operations manually is sometimes burdensome and difficult, therefore there is ongoing research in our group on automatic operation set extension for TCE, which hopefully will result in easy-to-use exploration of custom operation candidates.

One thing worth noting is that custom operations supported by TCE are not limited to simple operations like ADDSUB. Theoretically, there is no upper-limit for the complexity of the implemented operations, thanks to the unique programming model and the independence of function units from the rest of the architecture in TTA. Types of operations in TCE can range from computation of arithmetic and logic operations with latencies of few cycles, to I/O functionality and co-operation with other processors with latencies in hundreds of cycles. Indeed, a TTA processor core with this type of "long latency operations" can implement the feature set of a full system-on-chip.

Adding a new operation to TCE is straightforward. First, in order to simulate a program that uses a custom operation in our architecture simulator, an OSAL simulation behavior definition for the operation is written in C++. In order to use the custom operation in the application under design, a simple piece of inline assembly is inserted into its source code. Figure 5 clarifies this with an example. A piece of plain C code is replaced with code that utilizes a user-defined custom operation. When invoked, the macro '`__userdef_22__`', defined in header file 'userdef.h', "injects" a piece of assembly code corresponding to a TTA operation into the source code. In the future, we plan to add automatic insertion of custom operations in our compiler backend, which in some cases removes the need for manually using the defined custom operations in C code this way.

4. TCE DESIGN FLOW

The slowness of simulation when compared to native execution makes software development within TCE uncomfortable, especially in case of complex applications such as video coding. Therefore, the software development cycle is usually isolated from the TCE design flow. Modifications to the software source code during TCE design flow are required mainly in case custom operations are to be used manually.

The initial input to TCE is a generic sequential TTA program generated by a high level language compiler. We refer to this tool as the "frontend compiler". Currently, a frontend compiler based on the GCC [16] version

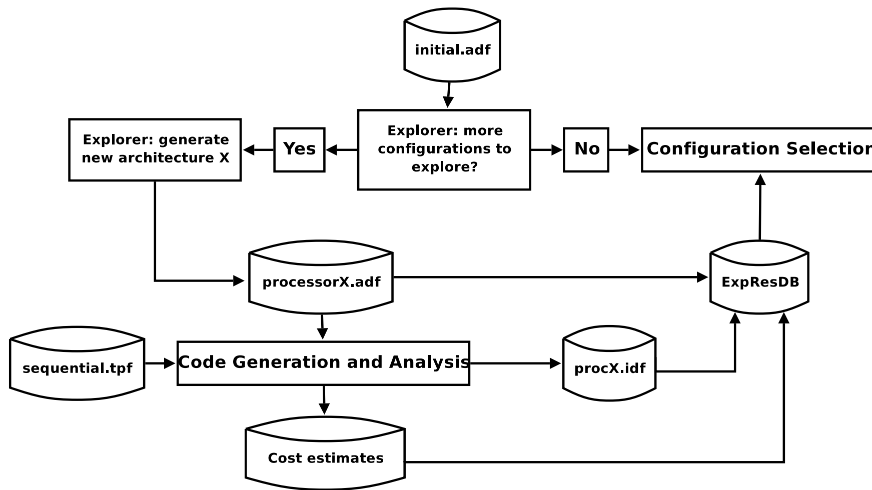

Figure 6. Design Space Exploration.

2.7.0 is used for compiling C programs to sequential TTA programs. More advanced frontend compiler based on the LLVM[17] is under construction. The generic sequential TTA programs are mapped to the explored TTA processors by a tool we call "compiler backend".

The TCE design flow can be divided into the following main phases:

Processor design space exploration is the process of finding a suitable processor architecture for the application at hand, either manually or automatically. This phase includes the following phase as a subphase.

Code generation and analysis is the phase in which target-specific code transformations, instruction scheduling and resource assignment are performed with the compiler backend. The performance of the resulting parallel program running on the target architecture is analyzed using the architecture simulator and the cost estimator.

Program image and processor generation is the phase in which the final products of the design flow are generated: a synthesizable hardware description and the binary executable code.

These phases are briefly discussed in the following sections.

4.1. Processor design space exploration

We define design space exploration as the process of finding target processor architectures with desired performance for a given set of applications. In TCE, thanks to the modularity of TTAs, this process can be largely automated, but it can be a manual trial-and-error process as well.

The automated design space exploration process of TCE is illustrated in Fig. 6. The abbreviations and shorthands used in the pictures of the design flow are as follows. TPF is a suffix of the files used to store TTA programs stored in our TTA Program Exchange Format files. ADF and IDF are the two file formats for describing the architecture and implementation data of the processor, respectively, as presented in the previous section. A processor configuration consists of an ADF/IDF pair.

The design space explorer, which can be thought of as the automated tool, or the designer manually exploring the design space using helper tools, modifies resources of a given architecture and passes the modified architecture to the code generation and analysis phase for evaluation. The evaluation produces the estimates of the die area of the processor core, the consumed energy, and the runtimes of the target applications. This process is repeated for each modified architecture, resulting in a set of candidate target processor configurations. Each explored configuration is stored in the Exploration Result Database with their cost estimates for later retrieval. Exploration is finished at the point a configuration with satisfying performance is found.

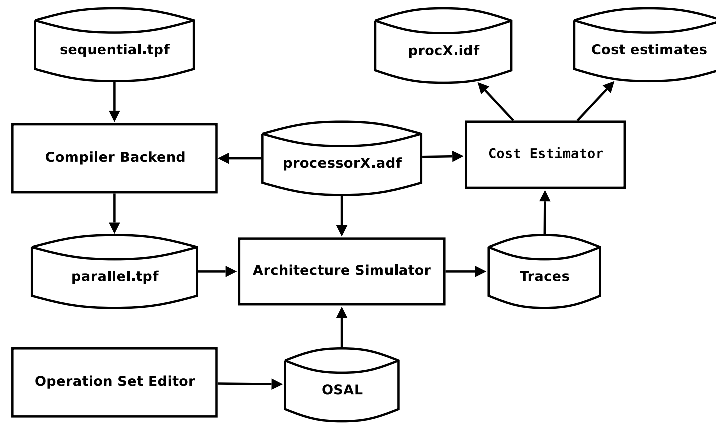

Figure 7. Code Generation and Analysis.

4.2. Code generation and analysis

The code generation and analysis phase, illustrated in Fig. 7, is the most challenging part of the TCE design flow. This phase includes mapping the generic sequential program to a target-specific parallel program which utilizes its resources efficiently. The quality of the toolset is largely dependent on the quality of the compiler backend which performs the mapping by assigning registers and other machine resources, and schedules the sequential instructions to parallel instructions.

While its feasibility has been amply proven by previous research,[18] effective instruction scheduling for TTAs is a subject which deserves extensive further research. Therefore, TCE provides a software framework for experimenting new ideas in the area of TTA instruction scheduling, resource assignment and code transformations.

The analysis part of the design flow includes simulation of the parallel program generated by the compiler backend on the architecture simulator. The default simulation model provides cycle and data path bit accuracy in case ideal data memory (no wait states) and instruction-addressed instruction memory (one address per instruction) are used. If the actual target differs from this default configuration, it is possible to customize the simulation model to improve the accuracy of the simulation according to the target specifics.

The architecture simulator produces an execution trace which includes processor utilization data, necessary for calculating the energy consumption estimates and for guiding the design space exploration, and the cycle count used to compute the runtime of each evaluated application. The architecture simulator also provides program debugging capabilities such as breakpoints and inspection of the processor architectural state at any cycle of simulation. This is an useful feature especially while debugging the developed compiler backend algorithms. The graphical user interface of the simulator aids “manual architecture exploration” by color coding the utilization rates of simulated processor’s components, and highlighting the frequently executed parts of the simulated program. Such a visual feedback simplifies search for sequences of operations that are profitable to convert to composite operations, and allows manually tuning the counts of function units, register files and buses of the processor according to the utilization rates.

Finally, the simulator can be used in verifying the hardware description language (HDL) implementation of the designed processor by producing a bus trace and comparing it to the one produced by an HDL simulator.

4.3. Program image and processor generation

The final phase of TCE design flow, illustrated in Fig. 8, includes generation of HDL files of the selected TTA designs and bit images of the programs. Synthesizing the processor from a set of HDL files is a task out of scope of TCE, it is performed with third-party tools.

Program Image Generator (PIG) processes a scheduled program stored in a TPEF file and generates bit images of the programs that can be uploaded into the instruction memory of the target processor.

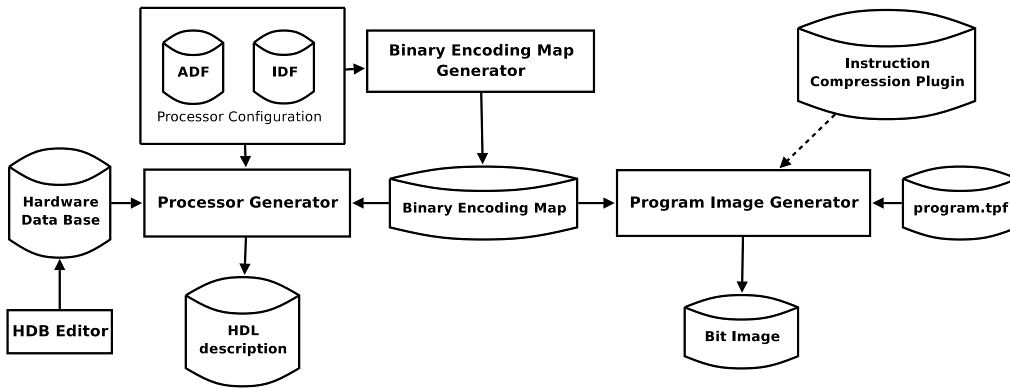

Figure 8. Program Image and Processor Generation.

What is notable is that the actual instruction encoding is dealt with only in this phase. The previous phases treat the programs as higher-level objects without considering their presentation in the instruction memory, but only their meaning. The separation of instruction encoding to a separate phase in the design flow allows fine-tuning the encoding parameters according to the specifics of the instruction memory architecture. In addition, this results in faster architectural simulation as no time is consumed on interpreting the instruction encoding during simulation. On the other hand, this complicates implementing programs that manipulate the instruction memory contents directly (e.g., self-modifying code).

The instruction encoding is defined in a file called Binary Encoding Map (BEM), which can be manually generated by the user or can be obtained from a tool called Binary Encoding Map Generator. Due to the lengthy instruction encoding typical of TTA programs, instruction compression is often desired. The binary images of programs can be compressed automatically if the designer provides PIG with a plug-in that implements an instruction compression algorithm. The instruction compressor plugins are able to compress program images, and generate a corresponding decompression block to the control unit of the target processor. An instruction compression algorithm plug-in implementing a dictionary-based compression algorithm[12] is provided as part of the toolset.

The Processor Generator reads a processor configuration consisting of the architecture (ADF) and the implementation (IDF) definition files of the target processor. IDF describes the locations of implementations for each component in the architecture in a Hardware Database (HDB). Using the HDB files, the Processor Generator picks the HDL files (currently only VHDL is supported) of the implementations and generates the interconnection network and the control logic.

5. CONCLUSIONS

An overview of a new codesign toolset has been presented. The toolset utilizes transport triggered Architectures as a processor architecture template, which the paper briefly introduced. The main phases of the processor design flow of the toolset have been described, and the main file formats for defining and customizing processor architectures in the toolset were introduced. The main advantages of the proposed toolset have been identified in automated design space exploration, the seamless run-time retargetability of the tools, and the vast and diverse set of supported target architectures.

The toolset is a work-in-progress. Most of the applications are completed, and the future work will be directed to the implementation of an efficient retargetable compiler backend. At its current state, the compiler backend of TCE provides a framework for implementing new code analysis and transformation passes, including algorithms for instruction scheduling, and register allocation. Further research on retargetable TTA compiler backend algorithms and other algorithms relevant to this kind of codesign toolset is an ongoing activity.

ACKNOWLEDGMENTS

This work has been supported by the National Technology Agency of Finland under research funding decision 40153/05 and Academy of Finland project number 205743.

REFERENCES

1. R. E. Gonzalez, "Xtensa — A configurable and extensible processor," *IEEE Micro* **20**(2), pp. 60–70, 2000.
2. A. Hoffmann, H. Meyr, and R. Leupers, *Architecture Exploration for Embedded Processors with Lisa*, Kluwer Academic Publishers, Norwell, MA, USA, 2002.
3. S. Kobayashi, K. Mita, Y. Takeuchi, and M. Imai, "Rapid prototyping of JPEG encoder using the ASIP development system: PEAS-III," in *IEEE Int. Conf. Acoustics, Speech, and Signal Processing*, **2**, pp. 485–488, July.
4. Target Compiler Technologies, "CHESS/CHECKERS: A retargetable tool-suite for embedded processors." Technical white paper, 2003. http://www.retarget.com/doc/target-whitepaper.pdf.
5. A. Fauth, J. V. Praet, and M. Freericks, "Describing instruction set processors using nML," in *Proc. 1995 European Conference on Design and Test*, p. 503, (Paris, France), March 6–9 1995.
6. H. Corporaal and M. Arnold, "Using transport triggered architectures for embedded processor design," *Integrated Computer-Aided Eng.* **5**(1), pp. 19–38, 1998.
7. H. Corporaal and H. J. Mulder, "Move: a framework for high-performance processor design," in *Proc. ACM/IEEE Conf. Supercomputing*, pp. 692–701, (Albuquerque, NM), Nov. 18–22 1991.
8. Tampere Univ. of Tech., "TCE project at TUT." http://tce.cs.tut.fi.
9. H. Corporaal, *Microprocessor Architectures: from VLIW to TTA*, John Wiley & Sons, Chichester, UK, 1997.
10. H. Corporaal, "TTAs: missing the ILP complexity wall," *Journal of Systems Architecture* **45**(12-13), pp. 949–973, 1999.
11. J. Hoogerbrugge, *Code Generation for Transport Triggered Architectures.* PhD thesis, Delft University of Technology, The Netherlands, 1996.
12. J. Heikkinen, A. Cilio, J. Takala, and H. Corporaal, "Dictionary-based program compression on transport triggered architectures," in *Proc. IEEE Int. Symp. on Circuits and Systems*, pp. 1122–1124, (Kobe, Japan), May 23–26 2005.
13. T. Pitkänen, R. Mäkinen, J. Heikkinen, T. Partanen, and J. Takala, "Low-power, high-performance TTA processor for 1024-point fast Fourier transform.," in *Embedded Computer Systems: Architectures, Modelling, and Simulation*, S. Vassiliadis, S. Wong, and T. Hämäläinen, eds., *Lecture Notes in Computer Science* **4017**, pp. 227–236, Springer-Verlag, Heidelberg, Germany, 2006.
14. T. Pitkänen, R. Mäkinen, J. Heikkinen, T. Partanen, and J. Takala, "Transport triggered architecture processor for mixed-radix FFT," in *Asilomar Conf. Signals, Systems, and Computers*, (Pacific Grove, CA), Oct. 30–Nov. 1 2006.
15. B. R. Rau, V. Kathail, and S. Aditya, "Machine-description driven compilers for EPIC and VLIW processors," *Design Automation for Embedded Systems* **4**(2-3), pp. 71–118, 1999.
16. Free Software Foundation, "Gcc, the gnu compiler collection." http://gcc.gnu.org.
17. C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Generation and Optimization*, p. 75, (Palo Alto, CA), March 20–24 2004.
18. H. Corporaal and J. Hoogerbrugge, "Code generation for transport triggered architectures," in *Code Generation for Embedded Processors*, pp. 240–259, Springer-Verlag, Heidelberg, Germany, 1995.