TACO: Rapid Design Space Exploration for Protocol Processors

Seppo Virtanen, Johan Lilius, Tero Nurmi and Tomi Westerlund

Turku Centre for Computer Science (TUCS)
Lemminkäisenkatu 14 A, 20520 Turku, Finland
seaavi@utu.fi, jolilius@abo.fi, tnurmi@utu.fi, tovewe@utu.fi

Abstract

To meet the tightening requirements on network hardware, the design of programmable processors with network-optimized hardware, that is, network or protocol processors, has recently attracted interest. In this paper we address the problem of evaluating different architectural configurations for such processors. The proposed methodology enables easy and fast experimentation with different protocol processor hardware architecture configurations to estimate their performance characteristics at early stages in the design process. We conclude the paper with an example of designing processors using the methodology.

1 Introduction

At the moment increasing demands are put on network bandwidth and throughput requirements, and on the speed with which new devices can be put on the market. Using current standard techniques (general purpose microprocessors, ASIC's) these goals are difficult to reach simultaneously.

One solution to this problem that has recently attracted interest is the design of programmable processors with network-optimized hardware, that is, *network* or *protocol processors*. The challenge in protocol processor design is to find an architecture that is a good compromise between a general purpose processor and a custom, protocol-specific processor (ASIC): ideally the architecture should be programmable, and at the same time optimized for a family of protocols and tasks required in their processing.

In our protocol processor design framework TACO (Tools for Application-specific hardware/software CO-design) [16] we have developed tools and methods for helping the designer in specifying, simulating, evaluating and synthesizing a certain type of protocol processors, TACO processors [17]. The starting point in our design flow is a high level application description or specification. We let the development of the application software guide the processor design work so that the processing requirements of the target application determine the hardware architecture of a protocol processor. The approach is quite different from what is found in most commercial protocol processors available today, which are most often multiprocessors with high performance general purpose processing cores as the computing elements.

In this paper we present an approach to exploring the design space for protocol processors based on the TACO framework. With design space exploration we mean evaluating the quality of a design in terms of hardware architecture, application software, or a combination of both. This analysis is performed based on quality metrics specified by the target application; the metrics could typically be a combination of acceptable ranges of values for power consumption, chip size and allowable clock cycle duration. The hardware design space we consider is three-fold; design decisions are needed for the types of hardware blocks to be used, the number of such blocks, and the connections between them. Within this design space we first need to find a set of hardware architectures that are able to perform the required tasks correctly. Only a subset of these architectures will be able to function within the timing constraints set by the protocol processing application. Of these, again only a subset is feasible in terms of manufacturing costs, power consumption and circuit size.

The techniques presented in this paper reduce the amount of work needed in both setting up and carrying out evaluation experiments of different protocol processor hardware configurations, especially in terms of estimating their performance characteristics at early stages in the design process. TACO processors are simulated using a SystemC model, their physical parameters are estimated in a Matlab model and they are synthesized using a VHDL model. The TACO protocol processor architecture, SystemC simulator and Matlab estimation model themselves have been presented in previous work [12, 16, 17, 18]; therefore in this paper we limit ourselves to only giving a short update on the development status of these models.

The rest of the paper is organized as follows: after an overview of related work we give a brief introduction to the different research directions within the TACO project, i.e. the protocol processor architecture and the system level simulation and estimation models. Then we present our design methodology and its support for design space exploration. Finally, we conclude the paper with results of an experiment in design space exploration using the presented method.

1.1 Related Work

Currently two main branches for domain specific processor design methodologies can be identified:

1. Behavioral HDL based modeling flows For example in [4] a methodology is presented for designing a complex communication chip using behavioral VHDL as a starting point. In [13] a methodology called PRCLIB (parameterized re-usable component library) is proposed for constructing SoC's by specifying different hardware configurations of RTL level synthesizable and parameterizable IP blocks from a library, and exploring the design space for such SoC's.

2. Abstract specification based modeling flows With a trend towards increasing system complexities, this type of flows are constantly gaining in popularity. The idea is to start with a very abstract system description and then to incrementally refine this specification to contain more and more architectural details. In [1] an OCAPI [15] description of system behavior is used as a starting point in a design methodology for an ethernet packet decoder. In [10] a Y-chart based design methodology and its use for design space exploration is presented. In the Y-chart methodology the performance of a selected architecture is analyzed for a given set of applications. As a result of this analysis the designer

receives performance data, based on which decisions and design choices can be made to the architecture. The process is repeated iteratively until a satisfactory architecture for the target set of applications is found.

The TACO design methodology fits for the most part into the second category; our VHDL component development has followed more the conventions of the first category. There are similarities between the flows outlined in the second category above and the TACO flow, as one might expect: in TACO we work in a specific problem domain, it being protocol processing. In the TACO flow the hardware architecture is also represented as a specification and simulation model written in a high level language, but only prior to synthesis. And, lastly, from the high level simulations we obtain performance data such as clock cycle requirements and module utilization (in addition to the verification of correct processor functionality). However, our approach is very much library-based and allows extensive component re-use for both simulation and synthesis. Since we do not develop all modules in our models from scratch every time the actual hardware design times in our flow are quite short.

As a major difference to the flows in the second catogory, we develop TACO processor models in three different development environments at the same time: we have a model for system-level simulations written in SystemC, a model for estimating physical parameters (e.g. processor area and power consumption) at the system level written in Matlab, and a model for synthesizing architectures written in VHDL. The models are highly parameterizable, and hence top-level description files for a given architecture can be automatically generated for all three models using a single hardware design tool. We will present such a tool later in this paper.

1.2 TACO Processor Architecture

The TACO protocol processor architecture is based on the transport triggered architecture (TTA) [5, 14]. In TTA processors data transports are programmed and they trigger operations - traditionally operations are programmed and they trigger transports. A TTA processor is formed of functional units (FU's) that communicate via an interconnection network of data buses, controlled by an interconnection network controller unit. The FU's connect to the buses through modules called sockets.

A functional unit has input and output registers. FU operations are executed every time data is moved to a specific kind of FU input registers, the trigger register. Each FU has one such register. Each functional unit in a TACO processor performs a specific protocol processing task, and each FU has been designed to be able to complete the execution of its function in one clock cycle. The FU operations can be, among others and depending on the application to be implemented, bitstring matching, counting, timing, comparisons and random number generation. Figure 1 shows the functional view of one possible TACO protocol processor architecture for processing ATM cells. As can be seen in figure 1, there can be more than one of each kind of FU's in a TACO processor.

TTA's are in essence one instruction processors, as instructions only specify data moves between functional units. Thus, the instruction word of a TTA processor consists mostly of source and destination addresses of sockets called socket ID's. The socket ID's are transported on ID buses from the interconnection network controller. There are as many ID buses as there are data buses in the interconnection network. Upon finding its socket ID on one of the ID buses, a socket opens the connection between a FU and the corresponding data bus on the interconnection network. The maximum number of instructions (i.e. data transports) that can be carried out in one clock cycle is equivalent to the number of data buses in the interconnection network.

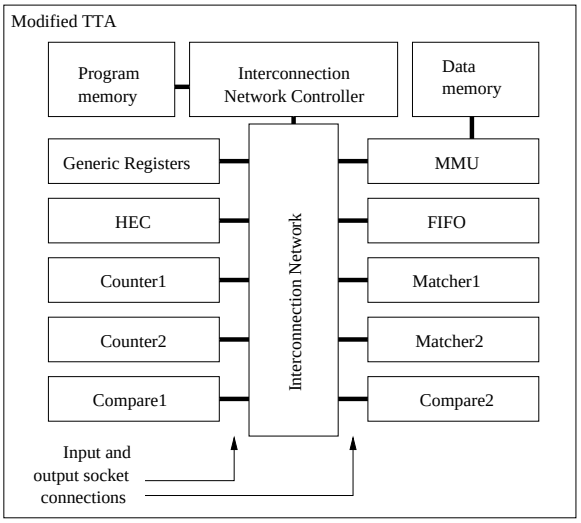

Figure 1: A TACO protocol processor for ATM.

To reduce clock cycle length, TACO processors have a four-stage pipeline with pipe stages *fetch* (fetch instruction from program memory), *decode* (decode source and destination socket addresses and dispatch them onto the ID buses), *move* (move data from an FU output register to an FU input register) and *execute* (execute FU operations). The TACO processor architecture is presented in detail in [17].

There are nowadays quite a few processor architectures designed and optimized for protocol processing, both as results of academic research and industrial product development. In the next few paragraphs we will go through the approaches we see most relevant to this work.

In [11] a protocol processor architecture optimized for internet protocols is proposed. The emphasis in the optimization is on the handling of the state table. The processor contains a special unit to handle jumps efficiently. In contrast the TACO processor is more like a pipelined processor. The programmer has to schedule the jumps. In [7] another programmable protocol processor architecture is proposed. This processor is designed for packet reception and pre-processing, i.e. delivering packet payloads to the host network processor. Here the idea is that each layer of the protocol is processed in a separate stage. The data flow is thus organized according to layers. In our approach something similar could be achieved by having one dedicated interconnection bus for each layer in the protocol stack.

The Intel IXP 1200 family and the Motorola C-5 are typical examples of modern commercial network processors. Most commercial protocol processors available today are multiprocessors with high performance general purpose processor cores as the computing elements. For example the Intel processors are built of a StrongArm microprocessor core and six programmable multithreaded "microengines", and the Motorola processors of a standard RISC core, 16 programmable "channel processors" and embedded coprocessors for table lookup, buffer memory and queue management. Compared to these, our approach has more in common with application specific processors (ASIP) in that we try to provide hardware implementations of frequently occurring operations. However, in contrast to ASIP, our hardware operations encapsulate much more functionality than the 2-3 assembler instructions typically found in ASIP operations.

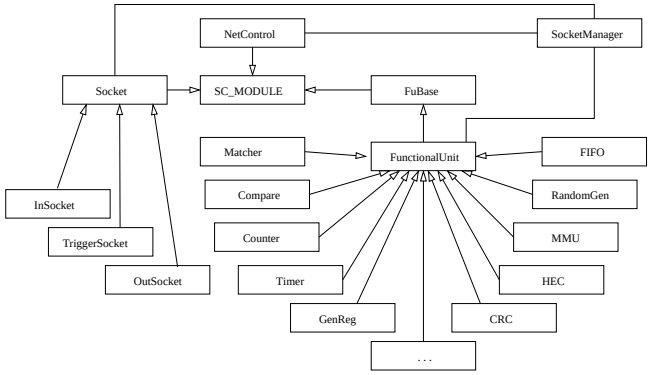

Figure 2: SystemC simulator class hierarchy. Arrows indicate inheritance (arrow head points to parent class), and lines indicate association.

1.3 The System-level Simulation Framework

To be able to rapidly evaluate different processor architectures designed for executing the same algorithm we have constructed an object oriented system-level protocol processor simulation framework. It is implemented as a library of components written in SystemC. The component library contains implementations of functional units, sockets, interconnection buses, and the program dispatch logic. Using this library it is possible to construct cycle accurate simulators of TACO processors.

A TACO simulator is constructed by inserting and connecting modules to the processor with a few C++ instructions in the SystemC `sc_main` function. By connecting we mean creating sockets, assigning socket addresses, and connecting sockets to buses. An example of SystemC code required to connect a functional unit to the system is given in figure 6. Such code can also be automatically generated by a tool (discussed later).

The level of abstraction used in the TACO simulator implementations is heterogenous in the following sense: the inter-module communication is handled at the RTL level, whereas the internal functionality of the modules is implemented as higher level C++ using SystemC's fixed bitstring length data types (e.g. `sc_uint<32>`, 32-bit unsigned integer). The motivation for the use of heterogenous abstraction in the simulator implementations is that since we know the execution time (in cycles) for each module in the processor (based on VHDL synthesis of the individual modules), we can use high-level C++ inside the modules as long as we insert the correct amount of cycle delays into each module.

Since the functional units in a TACO processor (or more generally, in a TTA processor) have a very similar interface, we utilize inheritance in the simulator as seen in figure 2. This is done by gathering the behavior that is the same for every functional unit into a parent class, and placing only the additions to this port/signal configuration required by individual modules to the child classes. The approach has obvious benefits: the code is more compact and readable (the interface code is not repeated multiple times), there are less errors to debug (only additions to the interface are coded) and adding new functional units to the SystemC component library is faster (most FU's in our protocol processors differ only in the internal implementation, not the physical interface). For more details on the SystemC simulation framework and its implementation, see [16] and [18].

1.4 The Physical Estimation Model

The Matlab model for estimating physical characteristics of TACO protocol processors is implemented as scripts and functions in Matlab's M-language. The model contains a set of equations for estimating delay, area and power consumption. The Matlab protocol processor model has been introduced in earlier work [12]. It has been updated since to support estimation of power consumption and pipe stage delay. Similar approaches for modeling system level physical characteristics have been presented in e.g. [2, 6], but our estimation model has been extensively optimized to support the TACO protocol processor architecture.

The cycle time of a TACO architecture instance is defined by the delay produced by the slowest pipeline stage. The delay is calculated for a given technology generation (in this case 0.35 μm). Recall that TACO processors have a four-stage pipeline with pipe stages *fetch*, *decode*, *move* and *execute*. Our delay model assumes that the slowest pipeline stage is either the *move* or the *execute*, because the logic depth of the *decode* stage is shorter than in the *execute* stage, and the (on-chip) memory access time can be assumed to be small in our case.

***Move* stage delay** It is assumed that when socket IDs are sent to the corresponding input and output sockets the output driver of the sending FU drives the signal on the bus. The signal is received in the receiving FU's input register, either in an operand or a trigger register. Thus, the *move* stage delay is defined as a combined driver/bus delay from an output register to an input register.

The principle used in calculating the move stage delay is shown in figure 3: the *move* stage delay is proportional to the distance between two FUs. In this architecture the distance can be approximated with an equation

$$FU_{distance} = \lceil (\frac{N}{2}) \rceil \sqrt{area_{FU}} + total\_width_{buses},$$

where N is the total number of FUs around the buses, $area_{FU}$ is the area of the largest FU and $total\_width_{buses}$ is the total width of the bus structure. For $total\_width_{buses}$ it is assumed that the area required by the sockets and drivers/repeaters on silicon levels fits under the metal bus structure.

The *move* stage delay is then defined as RLC wire delay [9] and takes also into account the possible need of repeaters. The maximum length for global wires without the need to use repeaters is defined as [19]

$$L_c = 2th[(1+\alpha)\rho\sqrt{\varepsilon_k}\varepsilon_0 c K_c]^{-1},$$

where t and h are thicknesses of the conductor and the dielectric, α is a return/signal path resistance ratio (we have $\alpha = 1$), ρ is metal resistivity and K_c is a fringing factor for wire capacitance.

***Execute* stage delay** The *execute* delay equals a single gate delay multiplied by the worst case logic depth of a signal path in an FU. Using the Matlab model we have been able to conclude that the *execute* stage delay sets the cycle time for TACO processors in the 0.35 μm technology generation. This is due to the large logic depths in the TACO FU's. However, in future technology generations this will change due to diminishing wires. The physical estimation model will then have to be modified accordingly.

Area estimation In area estimations the Matlab model adds up the total area of the functional units (i.e. combinatorial logic) and the area for the bus structure (incl. drivers and sockets). The timing constraint used for area estimations is $1.20 \cdot f_{\text{clock}}^{\min}$, in which $f_{\text{clock}}^{\min}$ is the minimum clock frequency requirement for running the target algorithm on the target architecture (the value is obtained from SystemC simulations). The 20% increase is inserted to ensure adequate processing performance.

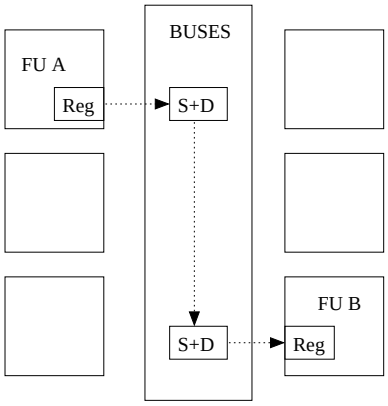

Figure 3: *Move* stage delay. The delay consists of socket delays and the bus wire delay including the sending socket's driver. *S* indicates a socket, *D* a driver and *Reg* a register.

After establishing the delay constraint the wire dimensions and/or repeaters in the global bus are set up so that the delay formed in the bus is shorter than the *execute* stage delay. The delay constraint determines the size (width and length) and thus the area of a logic gate. The area for the bus structure depends on the number of buses and the wire widths of the buses. Also the number of FU's has an effect on the size of the bus structure: the more there are FU's the longer buses are needed and hence the bus structure area increases. The number of gates in an FU is estimated by Rent's rule; we have used the value 0.464 for Rent's exponent p as suggested in [6] for inner structures of microprocessors.

Also the inter-gate wiring is taken into account in the area estimations. A method for estimating the area of a logic gate is defined in [3]. However, it doesn't take into account neither the power distribution network nor the clock tree. To compensate, we increase the resulting logic area by 20%.

Power estimation The dynamic power consumption is defined by the capacitance of the logic gates and inter-gate wiring as well as global bus structure including driver/repeater capacitance and wire capacitance, power supply voltage (3.3 V), cycle time and switching activity of electrical nodes in different logic units (FUs) and bus drivers. The supply voltage is the key factor to overall power consumption, but it is pre-defined by the technology used (e.g. 0.35 μm).

The designer can affect dynamic power consumption mostly by preferring designs that operate on lower clock frequencies. This way the driving ability requirements for gates are less stringent and therefore gates can be smaller. Minimum gate size is defined by the technology used. However, the timing requirements of the application may not allow the use of minimum size gates. By co-analyzing the results from SystemC simulations and Matlab estimations the designer should be able to find a power-efficient architecture configuration that satisfies the timing requirements of the application.

2 Design Methodology

In this section we present a protocol processor design flow in which a gate-level synthesized model of a protocol processor is derived from a high level application description using a set of tools that enable rapid architecture design, simulation, physical parameter estimation and hardware synthesis. Figure 4 gives an overview of the design flow we suggest. It is to be noted that the flow is not a completely automated process; rather it is a set of consecutive procedures the designer

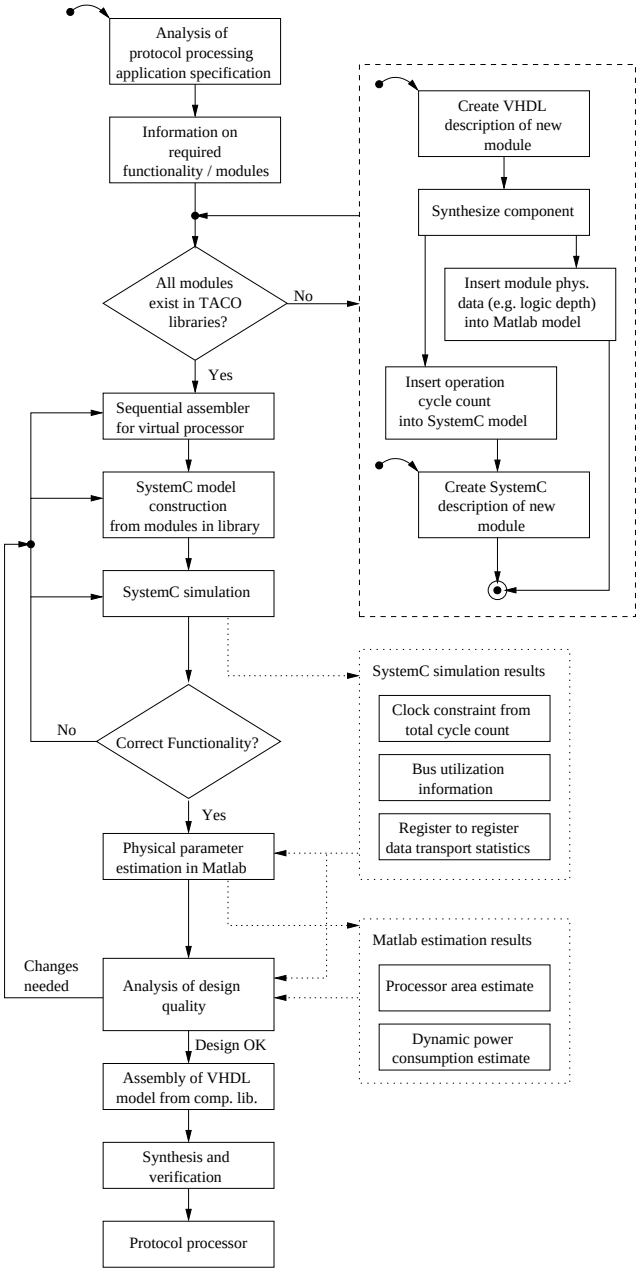

Figure 4: Protocol processor design flow.

needs to follow and complete. The designer is required to make certain design decisions along the way, and to evaluate the quality of solutions found by means of combining and evaluating results from e.g. simulations and physical parameter estimation.

2.1 The Design Flow

In the design of optimized protocol processors one of the most important tasks in our view is to identify frequently occurring protocol processing operations that vary little between different protocols and could be implemented in hardware to increase execution speed and reduce code size. The reduction in code size follows from these operations becoming primitive instructions of the processor (i.e. the operations become functional units of the processor). The operations are currently identified in an ad-hoc manner, that is, by reading protocol specifications, but in another line of research we are currently seeking ways of making this process more formal and to at least some extent automated. The

identification of these operations is an ongoing process and a key foundation of the TACO processor design flow.

Our design flow starts from a high-level protocol processing application specification. The specification can be of any kind, e.g. written or executable. This specification is analyzed to determine the types of operations and data transports needed to perform the application (first two boxes in figure 4). For the time being, this analysis is carried out manually. Once the required operations to perform the application have been determined, they are compared to operations already existing in the TACO module library. Currently the library contains a multitude of implementations of different operations, e.g. CRC, bitstring matching, timers, counters, Boolean logic units and so on.

If the operations are found in the library or they can conveniently be performed by using the existing operations in the library, the library will remain unchanged. If however the application requires an operation of a type that is not directly mappable to the modules available in the TACO library, the operation is added to the library. This is done by creating SystemC and VHDL modules of the operation as seen in the dashed box at the top right of figure 4. Once the VHDL module has been synthesized, the physical parameters of the module are inserted into the Matlab estimation model. These characteristics can be e.g. the logic depth of the module and the ratio of combinatorial vs. non-combinatorial logic in the module. Also the SystemC module description needs to be updated with information from the VHDL synthesis: the number of clock cycles required to perform the operation needs to be specified, because to speed up simulations the SystemC model uses high-level C++ functions inside the modules. Therefore additional delay statements are needed in the SystemC module description to represent the execution time.

After establishing that all the operations in the application can be performed by modules available in the TACO libraries, the application specification is refined until it becomes a list of consecutive data moves between elementary TACO protocol processing operations (i.e. operations offered by the modules in the TACO libraries). A data move of this kind could be e.g. to move counter output value to the input of a greater-than operation. The list of data moves is the sequential assembler code for a virtual processor shown in figure 4. By virtual processor we mean a TACO processor with one functional unit of each needed type and one bus in the interconnection network.

The assembler code for a virtual processor specifies the types of functional units needed to perform the target application and is thus used as a basis for deriving different architecture instances. This is the SystemC model construction phase in figure 4. An experienced designer can derive a set of good candidates for architecture instances quickly, since such a designer is able to predict what kinds of effects adding or removing buses or functional units to/from the architecture will probably have on overall performance.

To exploit the increased parallelism offered by concurrent data transports in TACO processors the virtual assembler code of the target application must be organized in an optimal manner for each given architecture instance. This optimization will in the future be carried out by a compiler that takes in program code written in a high-level language and yields optimized assembler code. However, currently the virtual assembler code is optimized manually for each architecture instance. After the architecture instance has been constructed and the application assembler code has been prepared for the specific architecture instance, the system can be simulated (SystemC simulation in figure 4).

If the SystemC simulation reveals incorrect or otherwise unwanted functionality in the processing of the application, the program code and/or the hardware architecture need to be modified. It is up to the designer's experience to decide at this point whether to modify only the application code or the entire system.

If the designer is satisfied with the overall system functionality in the SystemC simulation, the physical characteristics of the simulated architecture are estimated in the MatLab model with some of the SystemC simulation results as parameters (as shown in figure 4).

As the last system-level phase in our design flow the Matlab and SystemC results are combined and analyzed to determine whether the system meets the requirements of the original specification. The number of clock cycles it takes for the given application to be performed on the simulated architecture is obtained from the SystemC simulation, and an estimate of the worst case clock cycle using a certain manufacturing technology (0.35 μm in this paper) is obtained from the Matlab estimator. By multiplying the number of required clock cycles with the estimated cycle time the designer knows already at the system level whether the architecture instance is able to fulfill the timing requirements of the application. If the design was not able to perform the given application correctly or not able to match the design constraints (timing, power, area), the designer returns to earlier stages in the flow as indicated in figure 4.

Once the designer has found a satisfactory architecture instance in terms of functionality, performance and physical characteristics, it is possible to generate a synthesizable VHDL model for the processor using the VHDL modules in the TACO libraries. The VHDL model generation requires a top-level VHDL file to specify the components of the processor and their interconnections. The file is constructed either manually or using a design tool like the one in figure 5 (discussed below). After the synthesis the architecture and the program code are verified by post-synthesis simulation.

If all requirements are fulfilled the design is ready for placement and routing. In these steps detailed information of the physical placement of the functional units and interconnections between and within them are derived. Successful completion of the routing finishes the design process.

2.2 Turn-around time

Since parts of the SystemC simulator are implemented in high-level C++ as described earlier, simulations run very fast. One run of a packet processing loop can be performed in less than a second or at most a few seconds on a standard PC. Also the MatLab model executes fast, a typical execution takes 3-4 seconds on a standard PC. Thus the design iteration cycle (from box *SystemC model construction* to box *Analysis of design quality* in figure 4) is also fast.

The most time consuming process in the flow is synthesizing the architecture. Depending on the optimization constraints, already the gate-level VHDL synthesis of an architecture instance can be an overnight process on a typical workstation (we use Cadence Ambit Buildgates 4.0 and Alcatel 0.35 μm technology libraries on a SUN Ultra 10 workstation). The process of placement and routing consumes even more time. The time-wise benefits of rapid system-level design space exploration are obvious, and well supported by our design flow.

2.3 Design tool

To speed up the setup of SystemC simulations, Matlab estimations and VHDL synthesis for rapid design space exploration, we have implemented an architecture design tool that creates top level SystemC, Matlab and VHDL files for our models. Figure 5 shows the design tool with a design of a possible TACO protocol processor. The tool in figure 5 does not show all the hardware modules needed in a TACO processor; the ones that are not shown are the same for all TACO architecture instances and are automatically produced into the top level files by the design tool.

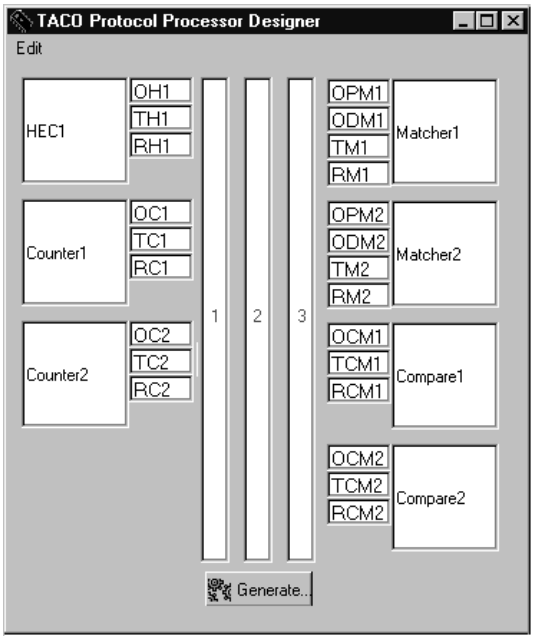

Figure 5: Protocol processor design tool.

Once the designer has "drawn" the architecture in the program window, clicking the *Generate* button in the tool generates the necessary files for simulation, physical parameter estimation and synthesis. Figure 6 shows the SystemC, Matlab and VHDL code for Matcher1 of figure 5 generated by the design tool. Note that the Matlab code actually covers all the Matchers in the architecture (more generally, each functional unit type has its own line, on which the information is given for all of the FU's of that kind). Also note that the SystemC and VHDL code excerpts also contain code for modules and signals that are used by other functional units as well, such as buses, system clock and the interconnection network controller; these have been left into the code examples for purposes of clarity.

As can be seen in the generated SystemC code, many hardware modules (like the sockets) are automatically instantiated and connected to other modules by the simulator. This reduces significantly the complexity of the SystemC code needed to be generated by the design tool.

The first version of the tool is implemented in Delphi 2.0 (object Pascal). We intend to develop the tool further by separating the code generation functionality into a text-mode code generator and making the graphical part act only as a front-end to this generator. To make this possible, we will specify a simple architecture representation language. The generator then converts files in this language into the top level files required by the three design environments. We also intend to enhance the graphical front-end to act as a front-end for SystemC simulations and Matlab estimations and thereby to support analysis of simulation and estimation results. The front-end will be enhanced to support register-to-register type programming, and a feature for component library browsing is in the plans. This feature would be used for reviewing component characteristics directly in the tool. The goal is to make at least the front-end platform-independent, and therefore the next version will be transferred to Java.

3 Design Example

To test the interaction capabilities of our tools and methods we decided to apply the design methodology to a part of ATM AIS (Alarm Indication Signal [8]) processing. We implemented an algorithm that analyzes incoming ATM cells to find out if a cell is a regular user cell, an empty cell or an AIS operations and maintenance (OAM) cell. In the following we describe the design steps to reach six TACO architecture instances with the necessary functionality to perform AIS processing. We estimate their physical parameters and evaluate their capability to perform in a 622 Mbps ATM network, where a new cell arrives for processing approximately $1.47 \cdot 10^6$ times per second. Finally, of the simulated and estimated architectures we choose two best candidates for gate-level synthesis to verify the results of our system-level simulations and physical parameter estimations.

Our analysis of ATM AIS processing resulted in adding a HEC (ATM header error check) unit into the 0.35 μm SystemC and VHDL component libraries and the Matlab model. After this addition, all operations needed for performing the target application were available for use in the libraries. The sequential assembler code for a virtual processor was then constructed as explained in section 2.2.

Next we started to explore different architectural alternatives. The architectures were specified and the top level description files (as in figure 6) were generated using the design tool described in section 2.3. Each instance was then simulated in SystemC and the physical parameters were estimated in Matlab. The results from both environments are presented in tables 1 and 2.

The different architecture instances were constructed by varying the number of buses and functional units in the processor. The first architecture instance we explored was the one at the top of table 2. This trivial case is basically an implementation of the virtual processor for which the sequential assembler code was previously constructed. After simulating and estimating this instance we started to incrementally add complexity to the instances. As can be seen in table 2, increasing the number of functional units and buses in the architecture decreases the minimum required clock speed. As our final instance, we simulated and estimated the last architecture listed in table 2. As we expected based on our previous experiments in e.g. [17], it turned out to be the least clock cycle consuming architecture of those we explored for the target algorithm.

The bus utilization data from the SystemC simulation (table 1) indicates the amounts of data transports on the interconnection network buses. Each data bus can be considered to have a data transport slot during each clock cycle. Therefore, the total number of available data transport slots (*move slots* in table 1) is $n_{slots} = n_{buses} \cdot n_{cycles}$, where n_{cycles} is the number of clock cycles required to perform the algorithm. In the trivial case with only one bus, bus utilization is naturally 100% (the only bus is used all the time until the algorithm finishes). The more interesting result is that with double FU's the bus utilization remains very high when adding more buses. This is a major contributor to overall processor performance as seen in table 1; as the number of functional units of the same kind increases, more buses are needed to fully utilize the functional units and thus gain even more in processing speed. This utilization information is also used in the Matlab model to estimate the total energy consumed during the execution of the algorithm.

The SystemC simulation of an architecture instance yields a minimum value for the required clock frequency of the processor, $f_{\text{clock}}^{\min}$. This value multiplied by 1.2 (as explained in section 1.4) is then used as a timing constraint in the Matlab model when estimating the area use and the power consumption of the given architecture instance. By joining the bus utilization results from SystemC simulation and power analysis results from Matlab estimation we reach an estimate of the energy consumed by a task running in the processor. The Matlab estimation results are shown in table 2.

Co-analyzing the results from SystemC and Matlab re-

a) Generated SystemC code

```
sc_clock clk("clock",20);
NetControl nc("NetCtrl1");
nc.clk(clk);
Bus* bus1 = new Bus("Bus1");
Bus* bus2 = new Bus("Bus2");
Bus* bus3 = new Bus("Bus3");
Matcher* m1 = new Matcher("Mtchr1", clk);
bus1->insertOperand(m1);
bus2->insertOperand(m1);
bus3->insertOperand(m1);
bus1->insertData(m1);
bus2->insertData(m1);
bus3->insertData(m1);
bus1->insertTrigger(m1);
bus2->insertTrigger(m1);
bus3->insertTrigger(m1);
bus1->insertResult(m1);
bus2->insertResult(m1);
bus3->insertResult(m1);
nc.initialize();
```

b) Generated Matlab code

```
matcher = [2 4 129 0.5]
```

c) Generated VHDL code

```
signal highway :  tacobus_matrix(2 downto 0);
signal matcher1_T, matcher1_O1 :  std_ulogic_vector(w-1 downto 0);
signal matcher1_O2, matcher1_R : std_ulogic_vector(w-1 downto 0);
signal matcher1_IStm1_act, matcher1_ISopm1_act :  std_ulogic;
signal matcher1_ISodm2_act, matcher1_OSrm1_act :  std_ulogic;
--matcher 1
TM1:  input_socket
generic map(idi_matcher1tm1, 1, 0, 1, w, "111")
port map(clk, rst, Glock, squash, dst_id, highway, matcher1_IStm1_act, matcher1_T, open);
OPM1:  input_socket
generic map(idi_matcher1opm1, 1, 0, 1, w, "111")
port map(clk, rst, Glock, squash, dst_id, highway, matcher1_ISopm1_act, matcher1_O1, open);
ODM1:  input_socket
generic map(idi_matcher1odm1, 1, 0, 1, w, "111")
port map(clk, rst, Glock, squash, dst_id, highway, matcher1_ISodm2_act, matcher1_O2, open);
matcher1:  matcher_fu
port map(clk, rst, Glock, matcher1_T, matcher1_O1, matcher1_O2,
          matcher1_IStm1_act,matcher1_ISopm1_act,
          matcher1_ISodm2_act, matcher1_OSrm1_act, matcher1_R, a);
RM1:  output_socket
generic map(ido_matcher1rm1, 1, 0, 1, "111")
port map(clk, rst, Glock, squash, src_id, matcher1_R, matcher1_OSrm1_act, highway, open);
```

Figure 6: Code examples. These excerpts represent the code generated for Matcher1 of figure 5. The Matlab code is a vector with the following fields: [#FUs #FU_registers #I/O_wires %Clk_nodes].

vealed that in two and three bus configurations the double-FU instances require larger area than the single-FU- instances. This is not surprising; in the 0.35 μm technology logic gates are still relatively large and the more there are functional units the more there are gates which increase the total area.

The cases with only one bus need further analysis. As seen in table 1, these instances have the most stringent clock frequency demands. The Matlab estimations showed that minimum-sized gates are too slow in these cases. The first instance (*single-1*) needed gates of scaling factor 3 and the second one (*double-1*) gates of scaling factor 2 to reach the target clock frequency. Because an average size gate was used for all FU logic, this difference in gate scaling factor was enough to compensate the difference in the number of FU's between these two instances. For all the rest architecture instances minimum-sized gates could be used in order to meet the target frequency constraints and thus only the number of FU's affected the logic area.

The same analysis can be extended to average power consumption, or more precisely, task energy consumption (see table 2). Task energy is the total amount of energy needed to execute a task once and is directly proportional to the average power consumed by the corresponding architecture instance. For the two 1-bus instances, the single-FU instance consumes more energy than its double-FU counterpart; for the two- and three-bus instances the situation is vice versa. The total task energy consumed is smaller for single-FU instances in the case of two or three buses and nearly the same for both FU configurations in the case of one bus. This is because double-FU instances have much longer cycle time and thus the energy consumed during *one* clock cycle is larger for all double-FU instances. Only in the case of one bus, the bigger number of clock cycles for single-FU instances compensates the smaller energy/cycle.

We decided to run two of the six architecture instances through gate-level synthesis to verify our simulations and estimations. We chose to synthesize *single-3* because of its excellent power and area characteristics, and *double-3* because of its ability to operate at a low clock frequency (this allows the most cost-efficient co-circuitry and the best chip manufacturing yield). The gate-level synthesis yields the logic area of an instance as seen in table 2 (*actual* column) for the two synthesized instances. As the Matlab model is capable of estimating both the logic area and the entire processor area, we have included also the logic area estimates in table 2. As can be seen in table 2, the logic area estimates and the actual values are quite close. Thus, we were able to verify that the system level simulations and estimations provided reliable results and that the suggested methodology was well applicable to this kind of system design.

Archit. instance	Exec. cycles	Req. clock	Move slots	Unused slots	Bus util.
single-1	121	178 MHz	121	0	100%
single-2	68	100 MHz	136	15	89%
single-3	49	72 MHz	144	42	71%
double-1	90	132 MHz	90	0	100%
double-2	53	78 MHz	106	1	99%
double-3	36	53 MHz	108	2	98%

Table 1: Worst case clock frequencies and data bus utilizations. *Single-2* indicates the use of one FU of each needed type and two buses in the interconnection network, *double-3* the use of two FU's of each needed type and three buses.

Archit. instance	Energy estim.	Logic area estimate	Logic area actual	μP area estimate
single-1	187 nJ	2.08 mm^2	-	2.63 mm^2
single-2	74 nJ	0.89 mm^2	-	1.20 mm^2
single-3	58 nJ	0.89 mm^2	0.87 mm^2	1.27 mm^2
double-1	179 nJ	1.88 mm^2	-	2.39 mm^2
double-2	105 nJ	1.41 mm^2	-	1.96 mm^2
double-3	81 nJ	1.41 mm^2	1.25 mm^2	2.09 mm^2

Table 2: Estimated task energies, estimated and actual logic areas, and estimated processor area.

4 Conclusions

In this paper we have presented a design space exploration methodology for a family of protocol processors. By adhering to the conventions of this methodology, a synthesizable processor model and its program code are developed with an initial application specification as a starting point. Our method reduces the amount of time and work needed in both setting up and carrying out evaluation experiments of different protocol processor hardware configurations, especially in terms of estimating their performance characteristics at early stages in the design process.

The results obtained in this paper showed that if there is more than one bus in an architecture instance, the instances with single-FU configuration are more area- and power/energy-efficient than their double-FU-counterparts. However, in future technology generations global (bus) wire delay will be more dominant in the overall delay. Then it might be reasonable to have a reserve for delay constraint if the delay formed in the bus starts to exceed or at least equal the delay formed in FUs when a signal is going through the ever faster logic path. The logic gates are also smaller in the future, and the size (and also the number) of repeaters along the global bus relatively increases. Thus we may face a situation where stringent delay constraints can't be met with physical or architectural design alone but rather with architectural/physical co-design. The instruction-level parallelism (ILP) in instances that have more than one of at least some types of functional units will be an important help for the designer: the less demanding timing constraints of ILP instances provide a solution to the problem. More coherent architectural-physical co-design will be needed in the future in order to meet the specifications of the increasingly demanding target applications.

With the proposed techniques the designer is able to tell already at early stages of the design work whether an architecture will be able to perform the target application within a given set of constraints, such as clock frequency, power consumption and circuit area. This information is obtained by combining and analyzing results from system level simulation and physical parameter estimation. Once design feasibility has been verified at the system level, tools can be used to generate a gate-level synthesizable VHDL model of the architecture configuration.

Acknowledgements Seppo Virtanen gratefully acknowledges financial support for this work from the Nokia foundation and the HPY research foundation.

References

[1] M. Attia and I. Verbauwhede. Programmable gigabit ethernet packet processor design methodology. In *Proceedings of the European Conference on Circuit Theory and Design (ECCTD'01*, pages III:177–180, Espoo, Finland, August 2001.

[2] H. Bakoglu. *Circuits, Interconnections and Packaging for VLSI.* Addison-Wesley, 1990.

[3] T. Bednar, R. Piro, D. Stout, L. Wissel, and P. Zuchowski. Technology-migratable ASIC library design. *IBM Journal of Research and Development*, 40(4):377–386, July 1996.

[4] R. Clauberg, P. Buchmann, A. Herkersdorf, and D. J. Webb. Design methodology for a large communication chip. *IEEE Design and Test of Computers*, pages 86–94, July-September 2000.

[5] H. Corporaal. *Microprocessor Architectures - from VLIW to TTA.* John Wiley and Sons Ltd., Chichester, West Sussex, England, 1998.

[6] B. Geuskens and K. Rose. *Modeling microprocessor performance.* Kluwer, 1998.

[7] T. Henriksson, U. Nordqvist, and D. Liu. Specification of a configurable general-purpose protocol processor. In *Proceedings of Second International Symposium on Communication Systems, Networks and Digital Signal Processing*, pages 284–289, Bournemouth, UK, July 2000.

[8] International Telecommunication Union, Telecommunication Standardization Sector. *ITU-T Recommendation I.610: B-ISDN Operation and Maintenance Principles and Functions*, 1993.

[9] Y. I. Ismail and E. G. Friedman. Effects of inductance on the propagation delay and repeater insertion in VLSI circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(2):195–206, April 2000.

[10] B. Kienhuis, E. F. Deprettere, P. van der Wolf, and K. Vissers. *A Methodology to Design Programmable Embedded Systems*, volume 2268 of *LNCS*, pages 18–37. Springer-Verlag, 2001 (to appear).

[11] Y. Ma, A. Jantsch, and H. Tenhunen. A programmable protocol processor architecture for high speed internet protocol processing. In *Proceedings of the 18th IEEE NORCHIP Conference*, pages 212–216, Turku, Finland, November 2000.

[12] T. Nurmi, S. Virtanen, J. Isoaho, and H. Tenhunen. Physical modeling and system level performance characterization of a protocol processor architecture. In *Proceedings of the 18th IEEE NORCHIP Conference*, pages 294–301, Turku, Finland, November 2000.

[13] R. S. Shelar, S. Nath, and J. S. Nanaware. Parameterized reusable component library methodology. In *Proceedings of the 26th EUROMICRO Conference (EUROMICRO'00*, Maastricht, The Netherlands, September 2000.

[14] D. Tabak and G. J. Lipovski. MOVE architecture in digital controllers. *IEEE Transactions on Computers*, 29(2):180–190, February 1980.

[15] S. Vernalde, P. Schaumont, and I. Bolsens. An object oriented programming approach for hardware design. In *IEEE Computer Society Workshop on VLSI'99*, Orlando, FL, USA, April 1999.

[16] S. Virtanen and J. Lilius. The TACO protocol processor simulation environment. In *Proceedings of the 9th International Symposium on Hardware/Software Codesign (CODES'01)*, pages 201–206, Copenhagen, Denmark, April 2001.

[17] S. Virtanen, J. Lilius, and T. Westerlund. A processor architecture for the TACO protocol processor development framework. In *Proceedings of the 18th IEEE NORCHIP Conference*, pages 204–211, Turku, Finland, November 2000.

[18] S. Virtanen, D. Truscan, and J. Lilius. SystemC based object oriented system design. In *Proceedings of the 2001 Forum on Design Languages (FDL'01)*, Lyon, France, September 2001.

[19] L. R. Zheng, B. Li, and H. Tenhunen. Global interconnect design for high speed ULSI and system-on-package. In *Proceedings of the 12th Annual IEEE ASIC/SOC conference (ASIC/SOC'99)*, Washington DC, USA, September 1999.