# Exploiting Model-Knowledge in High-Level Synthesis

Christian Zebelein, Christian Haubelt
Universität Rostock
{christian.zebelein,christian.haubelt}@uni-rostock.de

Joachim Falk, Jürgen Teich
Universität Erlangen-Nürnberg
{joachim.falk,juergen.teich}@informatik.uni-erlangen.de

**Abstract**

High-level synthesis tools are gaining more and more acceptance in industrial design flows. While they increase productivity in implementing a single complex hardware module, synthesizing and optimizing many hardware components simultaneously is still an open problem. Here, domain-specific models and specifications are seen as a key ingredient to raise the level of abstraction in future design flows. In this paper, we present a novel model-based synthesis framework which provides for efficient high-level-synthesis of streaming applications modeled as a set of communicating processes. The underlying formal dataflow model of computation enables model-based optimizations like efficient data caching, and naturally exposes parallelism contained in the application which can also be exploited by the proposed synthesis framework. Using a Motion-JPEG decoder as case-study, we will show how this model-based synthesis approach improves the overall quality of generated implementations in terms of performance and resource utilization.

## 1. Introduction

Due to the ever-increasing number and computational demands of functions performed by embedded software, multiprocessor system-on-chip (MPSoC) architectures are becoming more and more important in embedded systems. Such a heterogeneous architecture is typically composed of multiple microprocessors, dedicated hardware accelerators, as well as analog and mixed-signal components. However, without a design flow supporting these emerging complex platforms, the designer must perform the system synthesis, i.e., the mapping of functionality as specified by the application to resources available in the platform in an ad-hoc manner (cf. [Mar06]).
On the system level, choosing the right programming model is a challenging task. While more and more industrial design flows rely on high-level synthesis tools as can be seen by the broad availability of commercial HLS tools [For11, Cad11, Men11, NEC11], they do not support model-based optimizations. The Functional Oldenburg System SYnthesiser [GON$^+$08] processes object-oriented designs specified in a synthesizable subset of SystemC (OSSS). While this approach is expected to increase productivity, it does not perform high-level synthesis of functional blocks. Also, no explicit support for model-based optimizations is included.

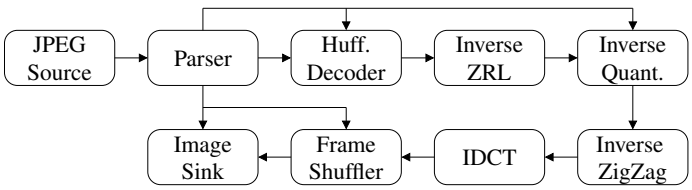



**Figure 1:** Example of a Motion-JPEG decoder modeled by a dataflow graph. Communication between actors (vertices) is realized via FIFO channels (directed edges).

Raising the level of abstraction, dataflow models of computation naturally expose the parallelism contained in the application and thus yield well to synthesis for such MPSoC platforms. In a dataflow model, concurrent processes (or actors) communicate with each other via packets transmitted over channels (cf. Figure 1). The behavior of an actor may be described by a set of firing rules which determine when an actor can be activated (cf. [LP95]). Often, these firing rules are described by means of *guarded atomic actions*.
Based on guarded atomic actions, the architecture-aware approach of Bluespec [ANRD04] promises good quality of results for the selected architecture. However, it prevents the same model from being mapped to a different architecture in the context of design space exploration. In contrast, our approach which is also based on guarded atomic actions separates the functionality of an application from the implementation-dependent behavior imposed by the platform onto which it will be mapped, thus enabling early analysis and optimizations in the design flow. In [BSS10], the synthesis of guarded atomic actions and scheduling is separated such that custom schedulers can be incorporated into the design. Although these approaches share many similarities with our proposed synthesis framework, explicit support for dataflow modeling is not reported. To the best of the authors' knowledge, CAL (California Actor Language) is the only framework which considers explicit support for dataflow models in high level synthesis [JMP+11]. However, no model-based optimizations for high-level synthesis are reported.
Other synthesis approaches, like [JH04] concentrate on static dataflow models of computation. While these models permit static scheduling, they are less expressive than the more general, dynamic dataflow model of computation supported by our proposed synthesis framework.

## 2. Modeling Framework

In this section, we will briefly review the dataflow model of computation upon which the proposed synthesis framework is based. The application model is given as a general *dataflow graph*. A dataflow graph consists of processes (in the following called *actors*) and abstract communication channels with FIFO semantics. Actors are supposed to communicate with each other via packets (in the following called *tokens*) transmitted over these channels. More formally, a dataflow graph is defined as follows:

**Definition 2.1 (Dataflow Graph)** *A* dataflow graph *is a directed graph* $g = (A, C)$*, where the set of vertices* $A$ *represents the* actors *and the set of edges* $C \subseteq A.I \times A.O$ *represents the* channels*. Each channel* $c = (a_{src}.o, a_{dst}.i) \in C$ *connects an* output port $o$ *of an actor* $a_{src}$ *to an input port* $i$ *of an actor* $a_{dst}$*. Additionally, a* delay function $d : C \to \mathbb{N}_0$ *is given. It assigns to each channel* $c \in C$ *a non-negative number of initial tokens.*

Note that due to space constraints, we refrain from extending this definition towards hierarchical dataflow graphs, which can be easily accomplished, and is supported in our modeling framework. Actors in a dataflow graph perform the actual computation by so called *firings*. When an actor fires, it consumes tokens from its input channels and produces tokens onto its output channels (outgoing edges). The behavior of an actor might be either *static* or *dynamic*. Static actors consume and produce tokens with constant (or periodically constant) rates, as known from synchronous dataflow (SDF) graphs [LM87] and cyclo-static dataflow (CSDF) graphs [BELP96]. In contrast to this, dynamic actors have variable consumption and production rates. The precondition, which decides if an actor can fire or not, is called the *firing rule*. For static actors, the firing rule consists only of the required number of tokens per incoming channel. For dynamic actors, the firing rule may consist of more general predicates based on token values or the state of an actor. More formally, an actor is defined as follows:

**Definition 2.2 (Actor)** *An* actor *is a tuple $a = (I, O, F_g, F_a, R)$, where $I$ and $O$ represent the set of input and output ports, respectively. The set of functions $F_g$ and $F_a$ represent* guard functions *and* actions*. A guard function $f_g \in F_g$ corresponds to a general predicate checked by a firing rule, whereas an action $f_a$ describes the state transformation of an actor during a* firing*. Finally, $R$ represents the set of firing rules: A firing rule $r = (c, p, f_g, f_a) \in R$ consists of functions $c : I \mapsto \mathbb{N}_0$ and $p : O \mapsto \mathbb{N}_0$ which specify the number of consumed/produced tokens for each input/output port, a (possibly empty) set of guard functions $f_g \in \mathcal{P}(F_g)$, and an associated action $f_a \in F_a$.*

During the execution of an actor $a \in A$, each firing rule $r \in a.R$ determines the availability of sufficient tokens and space on channels as specified by $r.c$ and $r.p$, respectively. If this check is successful, the guard functions $r.f_g$ are invoked (if any). Note that these functions may access the tokens on channels as requested by $r.c$ and read the internal state (but not modify it). If these general predicates also evaluate to true, the associated action $r.f_a$ is executed atomically. Thereby, it possibly modifies the internal state and consumes and produces tokens as specified by $r.c$ and $r.p$, respectively.

## 3. Synthesis Framework

In this section, we will introduce the proposed synthesis framework based on the formal model of computation presented in the last section. Please note that the framework presented in this section does not make any assumptions about the target platform (hardware/software, or mixed). In the next section, we describe how the framework can be specialized in order to support efficient high-level synthesis.

At a glance, the framework works in three steps as shown in Figure 2: 1) The front end parses the application and extracts the dataflow graph as specified in Def. 2.1. 2) The resulting dataflow graph is then processed by so-called *generators* which synthesize the different components of the dataflow graph. 3) The various back ends collect the data generated during step 2) and finally assemble all files which make up the synthesized application. In the following, we will discuss these steps in more detail.

### 3.1. Front End

An application in our modeling framework basically consists of a *dynamic part* and a *static part*. The dynamic part corresponds to the dataflow graph as specified in Def. 2.1. We refer to it as dynamic part because it is dynamically constructed during the SystemC elaboration phase, i.e., the initialization phase prior to simulation. Note that the dataflow graph is static after the elaboration phase, i.e., during simulation. This allows the user, e.g, to procedurally generate the firing rules during elaboration, or to instantiate actors based on some configuration parameter. Due to the nature of this dynamic part, it cannot be extracted by a parser and is generated by the simulation itself in the form of an XML file describing the dataflow graph.
The static part on the other hand corresponds to information which can be determined at compile time, like guard functions and actions of actors, token types, etc. This information is extracted in the form of an abstract syntax tree (AST) by parsing the source code of the application. For this purpose, we use the LLVM/Clang modular compiler framework [LA04] which provides access to the C++ AST in an object-oriented way. The static part of the application may be used by generators for example to extract token sizes, or to perform source code transformation of guard functions and actions.

### 3.2. Synthesis

For each element of the dataflow graph as extracted in the previous step, exactly one so-called *generator* exists which synthesizes this particular element. Basically, a generator is associated with a specific element of the dataflow graph (an actor, a port, a channel, etc.) and transforms this element into the target representation. For example, the actor's target representation may be a C++ class, in which case the generator would create a corresponding C++ AST snippet.
Generators may require information from other generators. In order to be able to mix generators for different synthesis targets, generators implement a certain interface corresponding to their associated dataflow element. For example, the generator associated with a guard function or action can be queried to return the name of the synthesized function. This name can then be used, e.g., by the generator responsible for synthesizing a firing rule $r$ to generate the call to $r.f_a$. As can be seen from this example, generators usually have state. Due to this reason, there exists only one generator for a specific element during the whole synthesis. This is enforced by the *generator pool* which acts as a generator cache: The pool can be queried for a generator for a given dataflow

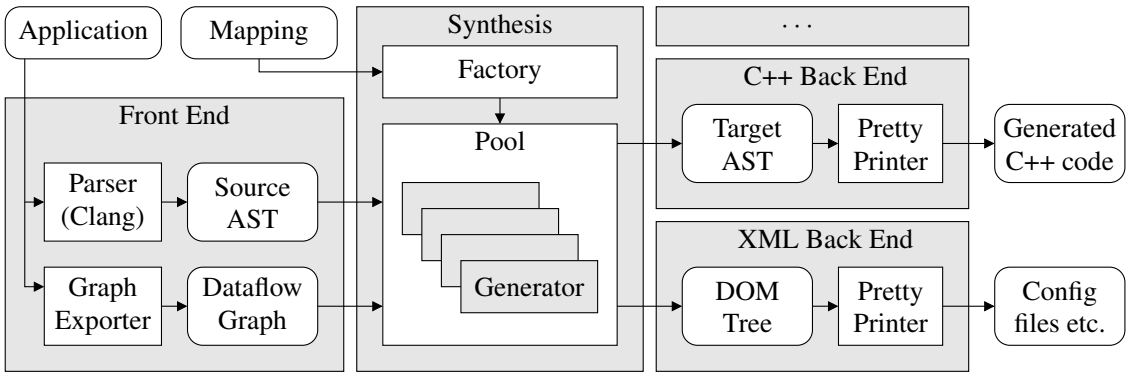



**Figure 2:** Synthesis framework overview.

element. If the generator already exists, it is returned. If it is not contained in the cache, it has to be created (and subsequently inserted into the generator cache).
The creation of generators is the task of the *generator factory*: Based on the mappings of dataflow elements to platform resources as specified by the user (possibly during design space exploration), the factory instantiates the corresponding generator. For example, if a certain actor is mapped onto a CPU, the factory instantiates a generator suitable for software synthesis. However, if the actor should be implemented, e.g., as a TLM module for a virtual prototype, the factory will instantiate a different generator. As the exact type of generator is not needed by other generators (the interface methods should be used), this approach provides for mixing of generators for different synthesis targets.
Given a hierarchical definition of dataflow graphs, elements may contain other elements (e.g., an actor contains input and output ports). Without going into details, this induces a similar concept for generators, i.e., a generator may be *instantiated* by another generator. A so-called *platform generator* is the top-level generator in the hierarchy representing the overall synthesized application. It basically instantiates the generator for the root dataflow graph, which in turn instantiates the generators for its nested subgraphs and actors, and so on. In this way, the dataflow graph is traversed and the application is synthesized in the process.

### 3.3. Back End

The data produced by generators in the previous step is collected by various back ends in order to assemble the files which make up the synthesized application. For generators which produce C++ AST snippets, these are combined into a C++ AST and pretty-printed into one or more source files to be processed by downstream synthesis steps. For this task, we also employ the LLVM/Clang compiler framework. Other back ends are available or can be implemented within the framework in order to produce, e.g., XML configuration files for virtual prototypes or VHDL/Verilog files for logic synthesis.
In the next section, we will present a specialization of the framework for high-level synthesis which supports model-based optimizations.

## 4. High-Level Synthesis

In this section, we show how high-level synthesis can be realized within the framework presented in the previous section. At a glance, we transform the untimed SystemC application into a bus-cycle accurate SystemC representation suitable for downstream synthesis steps, in this case, high-level synthesis. During this transformation, we perform certain optimizations based on the underlying dataflow model of computation in order to improve the performance of the synthesized model. The downstream high-level synthesis of the remaining unscheduled functionality is performed by a commercial tool.

### 4.1. Bus-Cycle Accurate Model

In the untimed model (also known as Programmer's View (PV) model), both the communication between actors and the behavior of an actor is described in a purely functional way. Such a model is used during the early stages of the design flow in order to verify the functional correctness of the

application. In contrast, a bus-cycle accurate model implements the communication transactions between actors in a cycle-accurate way, whereas the remaining behavior of an actor is still untimed. Bus-cycle accurate models may be used to explore the communication design space in order to decide how the abstract FIFO channels should be mapped onto resources available in the platform (e.g., shared memory vs. dedicated HW FIFOs).

In our dataflow model, communication between actors is performed via tokens transmitted over channels. Actors may consume tokens from channels via input ports and produce tokens onto channels via output ports. Thus, an input port represents the read interface of the corresponding FIFO channel, whereas an output port represents the write interface of the corresponding FIFO channel (cf. Figure 3).

An actor may randomly access any valid token in the FIFO by asserting the RD_ENABLE signal and supplying the desired offset via RD_OFFSET. This read offset will be added to the value of the read pointer internal to the FIFO channel, resulting in an absolute address RD_ADDR. In the case of the given HW FIFO implementation, this absolute address is simply supplied to the dual-ported memory contained in the FIFO channel which returns the corresponding token data in the next cycle. The RD_TOKENS signal provides the number of valid tokens in the channel. In order to consume one or more tokens, the RD_COMMIT signal must be set to the corresponding number. The channel's control logic checks this signal each cycle and advances the read pointer accordingly. The write interface behaves analogously.

The input/output port generators are responsible for implementing these access/commit protocols. To this end, the generator interface for output ports provides (amongst others) a method which accepts the token to be written and the corresponding offset, and generates the corresponding cycle-accurate transaction in SystemC. The generator interface for input ports provides similar wrapper functions.

The FIFO channel generator is responsible for synthesizing the access to the token storage and the read/write pointer management. In this case, dedicated HW FIFOS are generated for the channels in the model (cf. Figure 3). Reading and writing a token takes exactly one cycle in this case, assuming the availability of single-cycle memory in the target platform. However, for variable latency memory accesses, an ACK signal could be implemented with little effort.

Ports are accessed by guard functions and actions. The corresponding function generator traverses the source AST of the function body and generates the target AST by replacing the port accesses with function calls to the cycle-accurate transaction wrapper functions provided by the port generator associated with the port in question. The remaining statements in the function body are left unscheduled for later high-level synthesis steps.

Finally, the firing rules generator synthesizes code for checking the token requirements of a firing

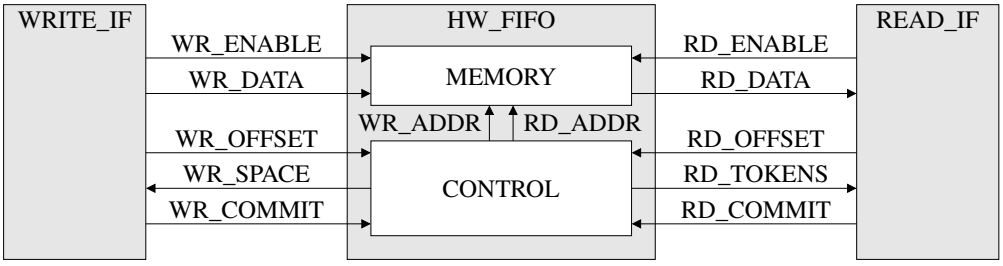
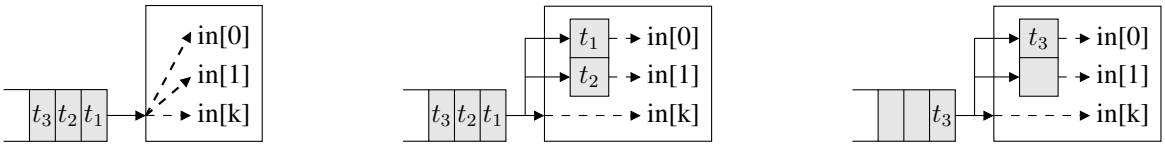



**Figure 3:** Read/Write interfaces with a corresponding HW FIFO implementation.




a) No token cache b) Token cache before commit c) Token cache after commit

**Figure 4:** Token access with cache module

rule $r$ as specified by $r.c$ prior to the execution of action $r.f_a$, as well as token commit code as specified by $r.c$ and $r.p$ after the execution of action $r.f_a$. For example, for each input port $p$ where $r.p(c) > 0$, the firing rules generator fetches the corresponding port generator from the pool and retrieves the token check/commit code via interface methods.
Concerning the overall graph synthesis, graphs and actors are synthesized as nested concurrent SystemC modules. Each actor generator instantiates the corresponding firing rules generator, which, in the baseline high-level synthesis, checks the firing rules sequentially in a round-robin fashion. Thus, all port accesses are automatically inlined into the main thread of an actor, which is synthesized by the actor generator as a SystemC *clocked thread*. In such a process, the user is only allowed to wait on the static sensitivity of the process, which is defined in the constructor and corresponds to the rising or falling edge of a clock signal.
After the platform generator has instantiated the top graph, the application has been synthesized into a bus-cycle accurate representation suitable for further processing by a high-level synthesis tool, like the ForteDS Cynthesizer [For11]. However, by exploiting some model-based knowledge, the quality of result of this baseline high-level synthesis can be improved. These optimizations will be discussed in the next section.

### 4.2. Optimizations

In this section, we show how model-based optimizations can be performed during synthesis in order to improve the performance of the baseline high-level synthesis presented in the previous section. Two optimizations are described, namely token caching and parallel guard function evaluation.

**Token Caching** The token access via input ports as described above requires the user to provide an offset for each token to be read. While this permits random access to any token in the channel, it also sequentializes token accesses, as only one token can be read at a time. Thus, if a guard function or action accesses multiple tokens (possibly multiple times), latency may increase. For example, the three port accesses in Figure 4a) read tokens from the same channel via input port $in$. Thus, they are sequentialized and take three cycles to complete.
In order to hide this token access latency, a token cache can be added to the read interface (cf. Figure 4b). As described in the previous section, tokens are removed from incoming channels and made visible on outgoing channels explicitly via commit signals. This commit phase is performed after the execution of an action. Thus, token values do not change during the execution of a guard function or action, assuming point-to-point FIFO channels as specified by the underlying dataflow model of computation.
In order to prevent the generation of large multiplexer structures, the cache module is only used for port accesses with a constant index. For port accesses with a variable index, the random-access

protocol is still used, and thus, the cache module also has to support this protocol. The cache module basically works as follow: If an external (i.e., variable) read access is active, it is forwarded by the cache module to the FIFO channel in order to support the random-access protocol. If no external read access is active and the cache is not yet full, the cache module generates a read request with the corresponding offset and stores the result in the cache.
In order to prevent a function from reading invalid token values from the cache, the number of available tokens as reported by the cache module is calculated as follows: If the cache is not yet full, the number of cached tokens is returned. Otherwise, the cache is full and the actual number of tokens available in the FIFO (i.e., RD_TOKENS) is returned. Note that this does not introduce deadlocks into the model (without proof).
For example, consider Figure 4b), where a cache module with space for two tokens has been inserted. Assuming the cache is full, the two port accesses with a constant offset can concurrently access tokens $t_1$ and $t_2$, while the port access with a variable offset can randomly access token $t_1$, $t_2$ or $t_3$. Thus, in this case, only one cycle is required to perform all three port accesses.
The cache is cleared when tokens are consumed and is subsequently refilled (cf. Figure 4c). Note that if enough tokens are available in the FIFO, the first token may be already fetched into the cache while the commit is still active by modifying the request offset accordingly.
The port generator interface is extended accordingly by providing a transaction wrapper function for cache accesses, which basically returns the value of the register with the given index. Finally, whether or not a port index is static (i.e., a so-called *integer constant expression*) is determined via the Clang API when traversing the function AST.

**Parallel guard function evaluation** The synthesis generates code which checks the firing rules sequentially in a round-robin fashion. While this is a sensible approach for software synthesis, the high-level synthesis should exploit the fact that the order of firing rule checks is non-deterministic, and thus can be performed in parallel. Note that in order to achieve a parallel RTL implementation after high-level synthesis, the guard functions have to be evaluated in concurrent SystemC processes: Simply performing all guard checks prior to selecting a firing rule based on the results does not guarantee a parallel RTL implementation (at least with the high-level synthesis tool employed in our design flow), and thus it may even increase the latency because all guard functions are always evaluated, even if the selected firing rule does not depend on some guard functions. However, evaluating the guard functions in concurrent SystemC processes raises some problems, as discussed next.
First, port accesses performed via transaction wrapper functions are driving some signals like the read/write enable and offset signals. Thus, the port accesses from different processes have to be multiplexed (i.e., sequentialized). Note that port accesses may take longer than one cycle in this case. However, token requests from different processes may be served simultaneously if they refer to the same token (i.e., specify the same port and offset). Moreover, this mechanism is only required for variable read offsets, as for static offsets the token is fetched from the cache without driving any signals.
Second, transferring the guard function into a separate process requires the introduction of enable and ready signals. The enable signal can be implemented as a reset signal which is active until the guard function should be executed. The ready signal is asserted when the multi-cycle guard function has been evaluated, and stays active until the guard function is re-evaluated. Thus, the process which invokes the actions can check the ready signals of the various guard functions at

any time. It remains the question when to reset a guard function. Due to the underlying model of computation, guard functions only depend on the variables of an actor and tokens contained on incoming channels. Thus, a conservative approach is to reset all guard functions after the execution of an action and the token consumption phase afterwards. However, this approach may be overly pessimistic as a guard function may not read any of the variables/tokens which have been modified/consumed by an action. Thus, we reset a specific guard only if the action which has been executed modifies/consumes a variable/token which is read by the guard. This requires the introduction of synchronization signals for variable writes. Note that similar synchronization signals for consumed token already exist, namely the RD_COMMIT signals. In order to determine which variables are read/written by a function, we evaluate the statements comprising the function AST. This analysis traces function calls and references, but not accesses through non-constant pointers, i.e., pointers which can be re-assigned during program execution.
Finally, as guard functions should operate only on valid tokens in order to avoid spurious results, the availability of tokens has to be checked by the guard process prior to evaluating the guard function itself. The number of required tokens from a specific input port $p \in a.I$ has to be inferred from the token consumption rate $r.c(p)$ as specified in the firing rule $r$ where this guard function is invoked. However, a guard function $g \in a.F_g$ may be used by multiple firing rules, i.e., $r_1 \in a.R$ and $r_2 \in a.R$ such that $r_1 \neq r_2$ and $g \in r_1.f_g \wedge g \in r_2.f_g$. If the token requirements for a specific port $p \in a.I$ are different in $r_1$ and $r_2$, i.e., $r_1.c(p) \neq r_2.c(p)$, it is not clear for how many tokens the guard process should wait before evaluating the guard function: Waiting for $\min(r_1.c(p), r_2.c(p))$ tokens may result in invalid token values being used during function evaluation, while waiting for $\max(r_1.c(p), r_2.c(p))$ tokens may introduce deadlocks into the synthesized model because the process possibly waits for more tokens than necessary. In this case, we duplicate the guard process for each different set of token requirements encountered during synthesis of the firing rules. However, as these different token requirements for the same guard function strongly indicate the merging of multiple guard functions into a single guard function, this situation should be encountered rarely in real-world designs.

## 5. Results

In order to show the applicability of the proposed synthesis framework, high-level synthesis and optimizations, we synthesized the Motion-Jpeg decoder depicted in Figure 1. Basically, the JpegSource and ImageSink actors have been mapped to SystemC-TLM representing the testbench, while the remaining actors have been mapped to SystemC-RTL, representing the high-level synthesis presented in the previous section. After transforming the corresponding actors into a bus-cycle accurate representation, we used the ForteDS Cynthesizer high-level synthesis tool to further process the synthesized model. For this purpose, we selected a Xilinx Virtex-5 XC5VLX110T FPGA as target platform, with a clock period of 10ns. For further synthesis steps, we used Synopsys Synplify Pro for logic synthesis, and the Xilinx tools for place and route.
The overall (normalized) results are shown in Figure 5. Note that the absolute latency of the baseline synthesis (i.e., inline guards without token caching) is approx. 28ms, and results from averaging the latencies for decoding three subsequent frames of a QCIF Motion-JPEG image (i.e., dimensions 176x144).
First, it can be observed that the latency is improved significantly by the presented optimizations: Token caching results in an approx. 40% latency reduction compared to the baseline synthesis,

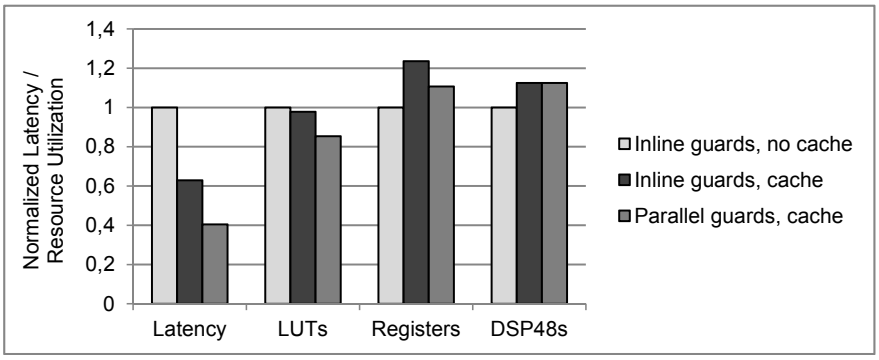



**Figure 5:** End-to-end latency and resource utilization for baseline synthesis and optimizations

while token caching combined with parallel guard function evaluation reduces the latency even more by approx. 60% w.r.t. the baseline synthesis. Due to space constraints, we refrain from giving detailed latency numbers for each synthesized module. At a glance, the largest latency savings can be observed in modules with a majority of port accesses with static indices and many firing rules, like the InvQuant actor. On the other hand, actors with a majority of variable port accesses and less firing rules do not contribute much to the latency improvement, like the InvZigZag actor, which basically reorders incoming tokens, and thus, performs only variable port accesses and consists of only two firing rules.

Concerning the lookup table (LUT) utilization, we can also observe a reduction when applying the presented optimizations. For the token caching alone, the slight reduction probably stems from the fact that more port accesses are performed via static indices than variable indices. In this case, less logic is needed to implement the random-access protocol for multiple port accesses, due to the fact that a majority of port accesses is directly wired to the token cache registers. In case of parallel guard evaluation, the LUT utilization is reduced even more, simply due to the fact that guard functions are now separate processes, and no longer inlined (and possibly duplicated) into the main process. Note that the duplication of guard function processes as described in the previous section had to be performed only once in the whole model.

As could be expected, the register utilization in case of inline guards with token caching increases (by approx. 23%) compared to the baseline synthesis, due to the additional registers used by the token caches. However, this increase in register usage is alleviated by the parallel guard evaluation, which reduces the number of registers used for implementing the guard functions, due to the same reason as described for the LUTs.

Finally, the optimized variants require nine DSP48 resources instead of eight required by the baseline synthesis. Concerning the parallel guard evaluation, this may be caused by less possibilities for resource sharing when functionality is split into separate SystemC processes.

## 6. Summary

In this paper, we proposed a synthesis framework based on a formal dataflow model of computation, which allows efficient and effective mapping of a class of multimedia and signal processing applications onto emerging multiprocessor platforms in a systematic and automated way. Due to the component-based nature of the proposed framework, it provides for various synthesis targets

encountered during system synthesis and design space exploration. We showed how high-level synthesis can be realized within the framework, and described two optimizations based on the underlying model of computation, namely token caching and parallel guard evalution. A Motion-JPEG decoder case-study has been presented which clearly showed the applicability of our approch.

## References


[ANRD04] Arvind, Rishiyur S. Nikhil, Daniel L. Rosenband, and Nirav Dave: *High-level Synthesis: An Essential Ingredient for Designing Complex ASICs*. In *Proceedings of CAD*, pages 775–782, 2004.

[BELP96] Bilsen, Greet, Marc Engels, Rudy Lauwereins, and Jean Peperstraete: *Cyclo-Static Dataflow*. IEEE Transaction on Signal Processing, 44(2):397–408, February 1996.

[BSS10] Brandt, J., K. Schneider, and S.K. Shukla: *Translating Concurrent Action Oriented Specifications to Synchronous Guarded Actions*. In *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 47–56, 2010.

[Cad11] http://www.cadence.com/products/sd/silicon_compiler, 2011.

[For11] http://www.forteds.com, 2011.

[GON+08] Grüttner, Kim, Frank Oppenheimer, Wolfgang Nebel, Fabien Colas-Bigey, and Anne Marie Fouilliart: *SystemC-based Modelling, Seamless Refinement, and Synthesis of a JPEG 2000 Decoder*. In *Proceedings of DATE*, pages 128–133, 2008.

[JH04] Jung, Hyunuk and Soonhoi Ha: *Hardware Synthesis From Coarse-Grained Dataflow Specification for Fast HW/SW CoSynthesis*. In *Proceedings of CODES+ISSS*, pages 24–29, New York, NY, USA, 2004.

[JMP+11] Janneck, Jörn, Ian Miller, David Parlour, Ghislain Roquier, Matthieu Wipliez, and Mickaël Raulet: *Synthesizing Hardware from Dataflow Programs*. Journal of Signal Processing Systems, 63:241–249, 2011.

[LA04] Lattner, Chris and Vikram Adve: *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*. In *Proceedings of CGO*, Palo Alto, California, Mar 2004.

[LM87] Lee, Edward A. and David G. Messerschmitt: *Synchronous Data Flow*. Proceedings of the IEEE, 75(9):1235–1245, September 1987.

[LP95] Lee, Edward A. and Thomas Parks: *Dataflow Process Networks*. In *Proceedings of the IEEE*, pages 773–799, 1995.

[Mar06] Martin, Grant: *Overview of the MPSoC Design Challenge*. In *Proceedings of DAC*, pages 274–279, New York, NY, USA, 2006.

[Men11] http://www.mentor.com/esl/catapult, 2011.

[NEC11] http://www.nec.com/global/prod/cwb/, 2011.