Support architecture for high level synthesis of algorithms strongly based on pointers

Alessandro Renzi

Mentor: Mario Casu

The exponential growth of the silicon technology allowed engineers to implement a very high and always increasing number of functionalities within integrated circuits.
Along with functionalities, also the overall complexity level increased exponentially too.
This forced designers to create new development methodologies in order to be able to effectively handle this growing complexity.

This need led to the creation of new tools and new paradigms which allowed to progressively rise the abstraction level, automating the management of underlying levels.
Specifically the abstraction level rising made able to switch from manual transistor drawing to HDL (Hardware Description Language) description, thanks to logic synthesis.
Later, when this approach became in turn insufficient too, the next step was to handle the development of the entire system by means of suitable languages, thanks to the introduction of high level synthesis (HLS), which allowed engineers to translate and automatically generate an HDL description of system model's algorithms.

High level synthesis usually takes as input a C/C++ or SystemC description and beyond a great portability, it is also able to make strong optimizations.
Obviously there are some limits to what the softwares are able to synthesize.
Specifically these limits are given by the fact that a software cannot be able to fully manage an input in a language like C/C++ and SystemC.
So it is necessary to put some constraints to what these languages are able to express, in particular these constraints result in strong limitations to the pointers usage.
There are cases where these constraints are too stringent, for example when the algorithm's logic is structurally based on pointers usage, so a complete rethinking of the code would be required.
But this requires also the knowledge of the specific field theory behind the algorithm, theory which usually is not known by the engineer who have to realize the hardware design.

Clearly this approach requires a huge amount of resources, which may not be available, so there is the need for a solution able to embed the pointers logic so that it is not necessary anymore to completely rewrite the existing code.

The proposed solution consists in a support architecture in the form of a C++ framework and a systematic procedure to apply it to the kind of algorithms under exam.
In order to better explain the use and the functionalities, the integration in a case study is shown, which consists in the hardware acceleration of a portion of the computer vision's algorithm: Canny Edge Detection, which software implementation comes from the open source library *OpenCV*.
This algorithm is a perfect example because includes a lot of implementation details which are not manageable by HLS softwares, in particular the inputs are too big to be entirely hosted on-chip and the outputs are both the data in a buffer and pointers to some of the data into the buffer.

From this example some requirements of the architecture can be extracted. The most important is that it must be able to host a lot of data, so it will need a memory element, but this memory will also need to be able to be accessed with the same absolute addresses which would be used in software, to allow the remaining code sections, which runs on the main CPU, to complete the work.
Moreover, since the data to work on is too much to be hosted all at the same time on the chip, it must be able to be updated on demand depending on the needs of the algorithm.

To realize all of this, first of all it is necessary to create an alternative model to the native C++ pointers, easier to handle by the HLS softwares, but at the same time able to keep the same functionalities.
This has been done in the *AddrPtr* class, which suitably exploiting the C++ features, allows to build a model that can be substituted to the native pointers almost transparently, requiring minimal code changes (and none to the functioning logic), but at the same time remains easily manageable by HLS tools.
Then it is necessary a Ram model able to reads and writes data of any size among the native C++ data types. This will be the fundamental layer of the architecture.
Upon this fundamental layer it is necessary to develop another one able to support the on demand updating feature, which overwrites the older data.
The efficient implementation of this feature recalls the ring buffer functioning, so it will be called *RingRam*. It has also, because of how it is implemented, the useful property of remapping the addresses so that the older data is always at the address zero.
The only remaining feature to implement is to be able to accept the same absolute addresses of the software domain.
This feature can be handled by a new address translation layer which defines, by means of two indexes, an operating window, which thanks to the update function, can advance through the address space, making possible to access memory locations virtually beyond the physical Ram boundary, this layer is called *VirtualBuffer*.

The initial offset is received from the software driver and applied to the starting index which at every reading and writing operation will be subtracted from the address given by the pointer (thanks to the further address translation of the *RingRam*, in this layer just a subtraction is needed).
In this way the software addresses and the on-chip buffer's ones are synchronized.
Every time that the update operation is performed, new data is imported overwriting the oldest data, and the *VirtualBuffer* mapping advance linearly in the address space, so that the new data can be accessed with an address which previously was outside the operating window.
Thanks to the update and translation operations it is possible to give the illusion to the algorithm of having all the data available as in the software execution.

The architecture is now complete, so substituting the pointers and the local buffers with the models just developed, and adding the necessary infrastructure for the data exchange with the CPU, it is possible to proceed with the high level synthesis.

In order to catch possible exceptions during the execution, also an exception handling infrastructure has been developed with the necessary care to make it synthesizable, unlike the native C++ constructs which are not supported for the same theoretical reasons as before.

A functional test is also showed, developed integrating the architecture in the case study algorithm's function and adding the simulation of the data exchange infrastructure's behavior, this allows to make sure that the final output produced by the modified version of the algorithm is identical to the one produced by the original version. This prove the architecture correctness.