High-Level Synthesis Developments in the Context of European Space Technology Research

Fabrizio Ferrandi
Politecnico di Milano, Italy

Michele Fiorito
Politecnico di Milano, Italy

Claudio Barone
Pacific Northwest National Laboratory, Richland, WA, USA

Giovanni Gozzi
Politecnico di Milano, Italy

Serena Curzel
Politecnico di Milano, Italy

Abstract

European efforts to boost competitiveness in the space services sector promote the research and development of advanced software and hardware solutions. The EU-funded HERMES project contributes to the effort by qualifying radiation-hardened, high-performance programmable microprocessors and developing a software ecosystem that facilitates the deployment of complex applications on such platforms. The main objectives of the project include reaching a technology readiness level of 6 (i.e., validated and demonstrated in relevant environment) for the rad-hard NG-ULTRA FPGA with its ceramic hermetic package CGA 1752, developed within projects of the European Space Agency, French National Centre for Space Studies and the European Union. An equally important share of the project is dedicated to the development and validation of tools that support multicore software programming and FPGA acceleration. The HERMES project selected the Bambu High-Level Synthesis tool to integrate capabilities to translate C/C++ code into Verilog/VHDL in its development ecosystem. In HERMES, Bambu has been and will be extended to support new FPGA targets, architectural models, model-based design, and input applications. The increased performance offered by FPGAs is thus made available also to software developers who do not have hardware design expertise.

2012 ACM Subject Classification Hardware → Methodologies for EDA; Hardware → High-level and register-transfer level synthesis; Computer systems organization → Architectures; Hardware → Very large scale integration design; Hardware → Reconfigurable logic and FPGAs

Keywords and phrases High-Level Synthesis, rad-hard FPGAs

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2024.1

Category Invited Talk

Funding This research has received funding from the EU Horizon 2020 Programme under grant agreement No 101004203.

1 Introduction

In space applications, the computational requirements for onboard computing are rapidly approaching the limits of what space-grade processors and microcontrollers can offer, mainly because radiation-hardened components are inherently slower than general-purpose CPUs. Hybrid platforms based on a mix of CPU and FPGA logic have become increasingly important. European institutions such as ESA, CNES, and the EU funded several efforts to develop a new generation of rad-hard FPGA platforms, including projects such as BRAVE [8],

VEGAS [13], OPERA [15], DAHLIA [14], and HERMES [12]. These platforms provide improved performance with acceptable overhead in size, power consumption, and cost. However, designing such systems is a challenging task.

High-level synthesis is a process that automatically generates an optimized hardware implementation from a high-level software description. HLS tools have changed the way we think about FPGA design by automating the most complex and time-consuming aspects of the development process: by eliminating the need for manual coding in VHDL/Verilog, users can now focus on providing a program written in a well-known software language such as C/C++ along with timing and resource utilization constraints that the final design must satisfy. This approach has become increasingly popular in the design of hybrid CPU-FPGA systems. It allows designers to explore different hardware/software partitionings and optimizations quickly and to generate efficient hardware implementations that meet stringent requirements. So, HLS in the design of hybrid CPU-FPGA systems for space applications has become a powerful tool for designers to optimize the performance of their systems while meeting the strict constraints imposed by space missions. As such, it is a crucial enabling technology for developing the next generation of rad-hard FPGAs for space applications.

The High-Level Synthesis flow begins with a compilation step to analyze data dependencies and loops in the input program, optimize the code, and generate a Control and Data Flow Graph (CDFG). The CDFG is then subjected to three core steps - resource allocation, scheduling, and binding - to define the structure of the output hardware. These steps involve assembling functional, storage, and communication units taken from a library of RTL components. Further optimization and analysis passes are carried out in the front-end, middle-end, and back-end of the tool to generate efficient accelerator designs. The result is HDL code ready to be used in a downstream FPGA or ASIC design tool for further analysis, logic synthesis, and deployment. In the past, the shorter development time offered by HLS tools used to come at the cost of reduced efficiency in the generated designs. However, with the availability of several commercial and open-source tools today, that is no longer the case. These tools can generate efficient designs that are competitive in terms of speed and resource utilization with hand-optimized RTL code.

This paper shows how the Bambu HLS tool [3] has been extended in the context of the HERMES project [12] (Qualification of High-pErformance pRogrammable Microprocessor and dEvelopment of Software ecosystem). The integration of the capabilities of Bambu in the space development ecosystem allows space application developers with no hardware design expertise to exploit the performance offered by FPGAs. The paper is divided into three additional sections. The first section presents Bambu, an open-source HLS tool adopted by the HERMES project. The second section discusses our most recent extensions to Bambu, which aim to enhance the tool's usability in an industrial context for space applications. The third section provides an overview of representative applications and corresponding experimental results, followed by the paper's conclusion.

2 The Bambu Open-Source High-level Synthesis tool

Bambu is a command-line tool developed at Politecnico di Milano providing support to designers for the HLS of complex applications. Most C/C++ constructs are supported, including function calls, access to arrays and structs, parameters passed by reference or copy, pointer arithmetic, dynamic resolution of memory accesses, and module sharing. The flow resembles a software compilation process: it begins with a high-level specification and generates low-level code through a series of analysis and optimization steps. Like a standard

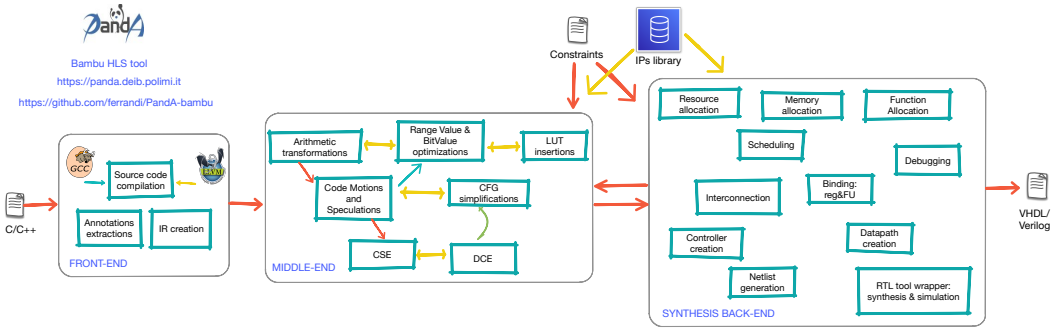

Figure 1 Bambu high-level synthesis flow.

software compilation flow, Bambu has three phases (Figure 1): front-end, middle-end, and back-end. In the front-end, the input code is parsed and translated into an intermediate representation (IR) that is used in the following parts of the flow. In the middle-end, target-independent analyses and optimizations are performed. The back-end performs the actual synthesis of Verilog/VHDL code ready for simulation, logic synthesis, and implementation through external tools.

Bambu front-end. Within Bambu, the user can choose several different front-end compilers, such as GCC and Clang. If GCC is selected, a plugin extracts the call graph and the Control Data Flow Graph of the functions under analysis from GCC's internal IR. Similarly, a Clang plugin extracts the same information from the input and serializes it into a textual format that is easy to parse. Bambu then parses all the compiler serialized information plus all the annotations to build a Static Single Assignment in-memory IR. This approach decouples the compiler front-end code from the rest of the HLS process. Localizing all the changes in a GCC or LLVM/Clang plugin allows rapid and easy integration of many different versions of the compilers. Bambu supports GCC versions from 4.9 to 8, and LLVM/CLANG versions from 4.0 to 16.

Bambu middle-end. Starting from the GCC/Clang IR, Bambu rebuilds data structures, such as the Call Graph and the Control Data Flow Graphs, and builds additional data structures, such as the Program Dependence Graphs. Next, it applies a set of device-independent analyses and transformations. Some of these steps are commonly used in a software compilation flow (e.g., data flow analysis, loop recognition, dead code elimination, constant propagation, LUT expression insertion, etc.). Multiplications and divisions by constant values are transformed into expressions that use only shifts and adders to reduce area utilization and improve timing. The resulting expression structure depends on the target device and technology since adders and multipliers may have different performances on different devices. Differently from general-purpose software compilers, designed to target a processor with a fixed-sized data-path (usually 32 or 64 bits), a HLS compiler can exploit custom-sized operators (e.g., a multiplier with the minimum number of I/O bits) and registers. Consequently, we can select the minimal number of bits required for the specific algorithm's operations and value storage, which leads to less area, less power, and shorter critical paths. At this stage, Bambu also performs Bitwidth and Range Analysis, aiming at reducing the number of bits required by data-path operators. This analysis is crucial during the optimization process because it impacts all non-functional requirements (e.g., performance, area, power) of a design without affecting its behavior.

Bambu synthesis back-end. In this phase, Bambu performs the actual architectural synthesis of the specification. The synthesis process works on each function separately, and the resulting architecture reflects the structure of the call graph. A single function is implemented through at least two sub-modules: the control logic and the data-path. Control logic modeled as a Finite State Machine handles the routing of the data values and the temporal execution of the operations. The data-path is a custom mux-based architecture with optimized data types to reduce the number of flip-flops and bit-level multiplexers, implementing all the operations and memories required during the function execution. The following paragraphs describe the sequence of steps that Bambu follows to generate control and data-path modules.

Function Allocation. Function Allocation associates the high-level functions with specific resources available in the technology library associated with the target device. The technology library coming with Bambu integrates standard functions described in Verilog or VHDL, standard system libraries such as `libc` and `libm`, and designer-defined components written in Verilog or VHDL. Bambu supports function pointers and sharing of (sub)modules across module boundaries [7]. Sharing of functions is achieved using function proxies that act as intermediaries between function calls in the original specification and shared modules. This method of sharing results in significant area savings when dealing with complex call graphs, without any notable impact on execution delays.

Memory Allocation. Memory Allocation refers to the storage of aggregate variables such as arrays and structures, global variables, and the implementation of dynamic memory allocation. Bambu adopts an architecture for memory accesses that supports a wide range of cases. Statically analyzing the memory accesses, Bambu builds a hierarchical data-path where memories can be classified as read-only, local, with aligned or unaligned memory accesses, or those requiring dynamic resolutions. The memory interconnection defines multiple buses connecting load/store components to their respective memories. Dual-port BRAMs or memory controllers with complex parallel channels are supported by replicating such memory interconnections as needed. The same memory infrastructure can connect to external components (e.g., scratchpads, caches, and DRAMs) or directly to the bus to access off-chip memory. Supporting protocol-based accesses (e.g., FIFO or stream-based access) is obtained by generating specific components that replace load/store units.

Resource Allocation. Resource allocation associates operations not mapped on a function to resource units (RU) available in the resource library. During the middle-end phase, the specification is inspected to identify the characteristics of the operations: these include the type of the operation (e.g., addition, multiplication, etc.) and the types of the operands (e.g., integer, float, etc.). Floating-point operations are supported through the HLS of a soft-float library containing basic soft-float operators [4] or alternatively by exploiting the FloPoCo software [2], a generator of arithmetic Floating-Point Cores. The allocation step maps operations on the set of available RUs; their characterization includes latency, area, and the number of pipeline stages. Usually, more operation/RU matchings are feasible; in this case, selecting a proper RU is driven by design constraints. The library of RUs used by Bambu is quite rich and may include several implementations for the same operation. Furthermore, the library includes RUs that are provided as templates in a standard hardware description language, such as Verilog or VHDL. These templates can be customized and retargeted in accordance with the characteristics of the target technology. In this case, the underlying logic synthesis tool will determine the best architecture to implement each operation (for example, multipliers can be mapped on dedicated DSP blocks or implemented with LUTs). To perform aggressive optimizations, each library component is annotated

with helpful information during the entire HLS process, such as resource occupation and latency. Bambu adopts a pre-characterization approach through a tool called Eucalyptus to synthesize different configurations of library components and collect their resulting latency and resource consumption metrics as XML files in the Bambu library. The configurations are obtained by specializing a generic template of the resource component, such as a multiplier or an adder, according to the bit widths of its input and output arguments and the number of pipeline stages.

Scheduling. By default, Bambu employs a list-based scheduling algorithm. In its basic formulation, list-based scheduling associates each operation with a priority according to particular metrics. Scheduling proceeds iteratively, associating a set of operations to be executed with each control step. Ready operations (i.e., whose dependencies have been satisfied in previous iterations of the algorithm) can be scheduled in the current control step, considering the availability of the resources. If multiple ready operations compete for a resource, then the one having a higher priority is scheduled. In addition to this old but efficient algorithm, Bambu also features a more aggressive scheduling algorithm, the speculative scheduling algorithm based on System of Difference Constraints [6]. This algorithm builds an integer linear programming formulation of the scheduling problem, allowing code motion and speculation of operations that belong to different basic blocks.

Module Binding. Within the computed schedule, operations that execute concurrently are not allowed to share the same resource instance. In Bambu, binding of operations to resources is performed through a clique covering algorithm on a weighted compatibility graph [11]. The compatibility graph is built by analyzing the schedule: operations scheduled on different control steps are marked as compatible. Weights express how valuable it is for two operations to share the same hardware resource. They are computed considering area/delay trade-offs caused by sharing; for example, RUs that occupy a large area will be more likely shared. Weights computation also considers the cost of interconnections required by the steering logic. Bambu also offers several other algorithms for solving the covering problem on compatibility/conflict graphs.

Register Binding. Register binding associates storage values to registers and requires a preliminary analysis step, the liveness analysis [11]. Liveness analysis starts from the schedule to identify each variable's life intervals, i.e., the sequence of control steps in which a temporary value needs to be stored. Variables with non-overlapping life intervals may share the same register.

Interconnection Binding. Interconnections are bound according to the outcome of the previous steps: if a functional or memory resource is shared, then the algorithm introduces steering logic on its inputs. It also identifies the set of control signals that will be driven by the controller.

Netlist Generation. The final architecture is then generated and represented through a hyper-graph, highlighting the interconnection between modules. The netlist generation step translates such representation in a register transfer-level (RTL) description in Verilog or VHDL. The process accesses the resource library, which embeds the RTL implementation of each resource. This process is target-dependent, and the hardware descriptions may differ for different technologies (e.g., ASIC or FPGA) or target devices.

Generation of Synthesis and Simulation Scripts. Bambu automatically generates synthesis and simulation scripts that can be customized via XML configuration files. The RTL-synthesis tools currently supported are AMD/Xilinx ISE, AMD/Xilinx Vivado, Yosis-Vivado, Intel/Altera Quartus, Lattice Diamond, NanoXplore, and OpenRoad. Supported simulators are Mentor Modelsim, Xilinx XSIM, and Verilator.

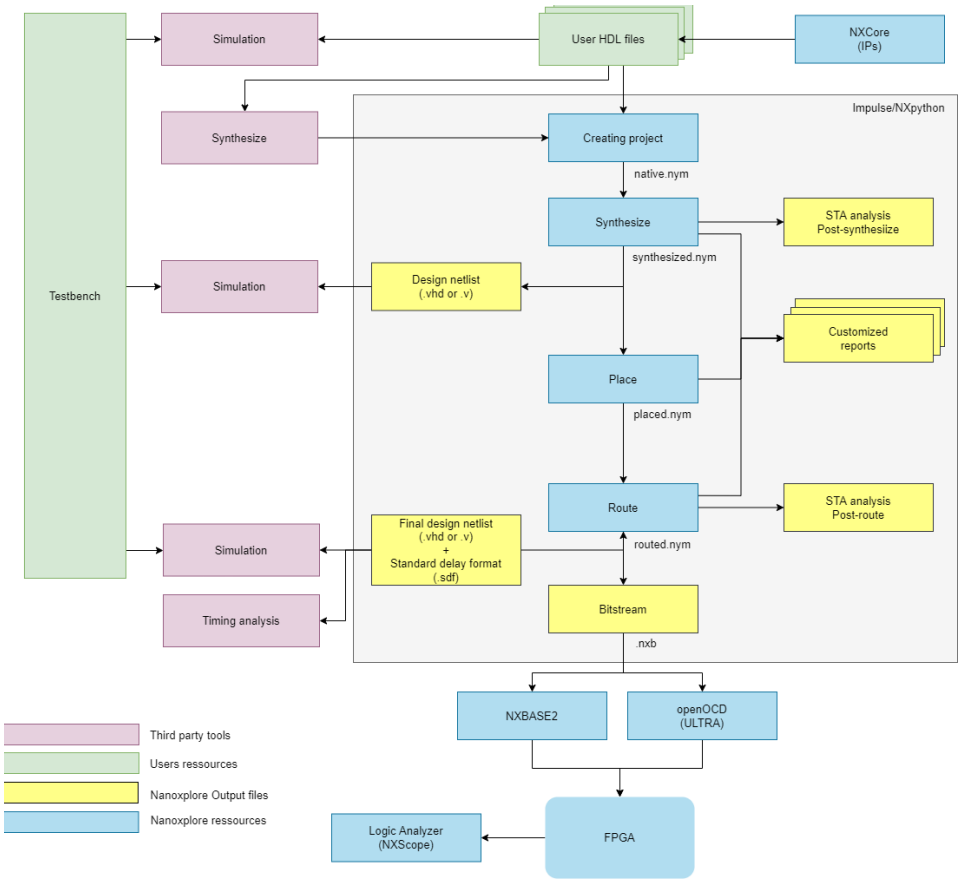

Figure 2 NanoXplore Impulse design flow.

3 New HLS Features

3.1 NanoXplore Logic Synthesis Integration

The first step in the integration of Bambu into the HERMES design flow for space applications was to add support for the NanoXplore synthesis tool Impulse (Figure 2). The Impulse design suite translates HDL code into a device-specific bitstream for NanoXplore radiation-hardened FPGAs through logic synthesis, place-and-route compilation steps, and static timing analysis tools for performance estimation. Seamless integration of Bambu and Impulse is achieved by automatically generating backend synthesis scripts after the generation of the RTL code.

During the HLS process, Bambu applies optimizations that are specific to a target, and therefore, its backend has been customized to support three new types of space-grade FPGAs: NG-MEDIUM, NG-LARGE, and NG-ULTRA. Before integrating the logic synthesis backend based on Impulse and running characterization through Eucalyptus, it was necessary to map Bambu library components correctly to the actual DSPs and True Dual Port RAMs available on the NG-ULTRA fabric. Since the mapping occurs through behavioral HDL templates, the components used by Bambu for arithmetic operations and storage modules have been customized to comply with the Impulse synthesis guidelines.

3.2 AXI Protocol Interfaces

The NG-ULTRA board's ARM processor uses the AXI4 protocol interfaces to communicate with the rest of the system. AXI4 is a standard that includes AXI4, AXI4-stream, and AXI4-Lite protocols. These are used to access memory banks, streaming channels, and memory-mapped registers.

Bambu was therefore extended to offer the possibility to access data outside of the accelerator over an AXI4 bus, which is the standard also for other on-chip communications (e.g., to communicate with HBM on AMD/Xilinx FPGAs). This can be used to connect modules created with Bambu with modules created by other sources, or with external memories. AXI4 is a master/slave protocol that defines a series of channels, i.e., independent groups of signals exchanged between the master and slave devices. As shown in figure 3, AXI4 defines five channels: Address read, Read data, Address write, Write data and Write response, and collects them into bundles associated with different parameters. Each channel contains a set of information signals and two handshake signals, xVALID and xREADY, where x depends on the actual channel. These signals indicate the availability of one of the endpoints to exchange the channel information. In particular, the xVALID signal indicates that the source of the information on the bus has provided valid data, and the xREADY signal indicates that the recipient is ready to accept the information. The handshake can only be completed once both signals are active simultaneously. The address read/write channels contain the signals needed to define the transaction, such as the data address, the burst type that should be used, the length of the burst, and more. The read channel contains the data requested by the transaction and a read response, indicating whether the request was successful. The write channel contains the data that must be passed to the slave device and a write strobe signal, specifying which data bytes are valid. The write response channel contains a signal indicating whether the write transaction was successful. AXI supports unaligned operations: it can exchange data even if the requested value has a size different from the bus or its address is not a multiple of the bus size. The AXI protocol is burst-based, which means the master must only specify the initial address and the burst information. The slave will then compute the correct addresses for each subsequent data transfer, or beat, autonomously based on the information passed when the burst was defined. Using pragmas, Bambu can add an AXI4 controller module inside the hardware design linked to a specific memory parameter or a set. Each master module is responsible for the communications on a bundle of AXI4 channels and can act independently from the others. When a memory operation is issued, the finite state machine activates only the AXI master module related to the requested memory parameter. This allows the execution of parallel memory accesses when there are multiple AXI bundles.

Bambu can also create a testbench that includes the AXI4 slave counterparts of the master interfaces. This enables users to simulate data exchange and verify its correctness. Users can configure memory delay estimates to assess the application's performance, taking into account an estimated latency for data transfers.

3.3 AXI Caches

When requesting an AXI interface, Bambu offers the possibility to add a customizable cache that can intercept or forward memory access requests coming from the memory controller to the AXI slave. These caches can help reduce the average memory access latency by accessing the data present in the cache rather than performing the full transaction over the AXI bus. Caches are requested by the user through pragma annotations, and several different options can be specified to generate caches with e.g., different write policies, replacement policies, and cache line sizes.

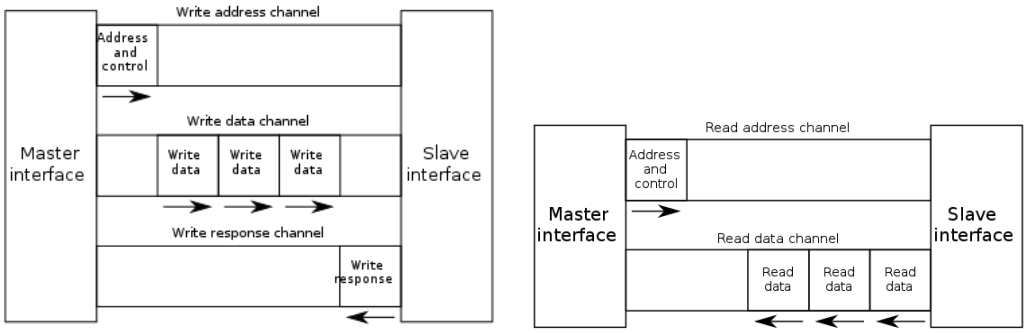

Figure 3 AXI4 channels descriptions: on the left the Write address, Write data and Write response channels, while on the right the Read address and Read data channels.

The caches provided by Bambu are largely based on the work done in [10], with some modifications that allowed it to be integrated into our tool and to improve its performance and customization. The basic element of the Bambu cache is a single datum of the same size as the data type of the kernel argument being stored. These are grouped in cache lines, i.e., a sequence of elements that are contiguous in memory. When there is a cache miss, the cache reads an entire line from memory, so whenever a datum is requested all the content of the line is immediately available for future requests. Cache lines can also be grouped in ways. Unlike elements in a line, lines in a way are not necessarily contiguous. Inside each way, a cache line can only be stored in a single position, depending on the starting address of the line. When more than one way is present, each of them provides a position to hold the line. Multiple memory areas are mapped to the same cache lines, so while populating the cache it is common that new data must be placed in an already populated line. In this case, the line is simply overwritten and requests regarding the old data will need to go through the main memory again.

While caches typically have the same behavior with read operations, different policies are available when performing writes. Bambu caches offer two different policies: write through no allocate and write back allocate. When using the first policy, the cache always immediately forwards the write operation to the main memory. If the address that was written is already present in the cache it is updated in the internal memory too, otherwise, no other action is performed. When the write back policy is selected, the data is not transferred to the main memory: only the internal state of the cache is updated. In case the element that was written was not already present in the cache the line in which it is contained is first read from memory, then the data is updated. The write policy also impacts what happens when a cache line needs to be replaced: in the case of write through policy, the data in the main memory and the cache are always consistent, so no action is needed when replacing lines. However, write back caches need to keep track of modified lines using a dirty bit that is set whenever any element of the line is modified. When a line must be replaced, if the dirty bit is set, the entire line is first written to the main memory, then it can be replaced. Otherwise, no action is needed, and the line can immediately be overwritten. Finally, another difference is that at the end of the computation write back caches need to write all their dirty lines back to main memory, while write though caches do not need this operation. In general, write through policies perform better when the time between two write operations is greater than the latency of the memory because by the time the second write is received, the cache is already done with the previous operation and its state is consistent with the main memory.

On the other hand, write back caches are more useful when performing a lot of writes to the same cache lines in a small period, because the whole line is then transferred in a single memory transaction either when it is replaced or at the end of the computation.

One of the modules used internally by the caches is the write buffer. This is a data structure that holds write transactions to main memory that must be performed but have not yet been completed. It is similar to a circular buffer and is handled by three indexes: a write index, indicating the next slot where data can be written, a read index, indicating the next write transaction that must be initiated, and a backup index, indicating the first transaction that has been started but has not yet completed. The write index is increased whenever a new item is inserted in the buffer, i.e., when a cache line should be written to memory. As long as the buffer is not empty and the AXI bus is available, the controller handling the bus will pick up the element marked by the read index and increase it, while beginning the write transaction on the bus. The controller also monitors the write responses of the AXI slave device and forwards them to the queue. The response indicates whether the transaction was successful or not. In the first case, the write operation has been completed and is no longer needed in the buffer, so the backup index in the write buffer is increased. In case the write response indicated an error, the transaction must be repeated. This is done by reading the element in the buffer marked by the backup index and writing it at the next available position in the buffer. Both the backup index and the write index are then increased. By using this buffer, it is possible to perform pipelined write transactions to the main memory, exploiting the AXI bus more efficiently. While waiting for the response of a transaction (if the buffer is not full), new write operations can be performed without stalling the entire cache. Increasing the size of the buffer is especially useful for write through caches since they typically perform a higher number of smaller data transfers with respect to write back caches, which perform a smaller number of larger transfers. While it can be useful even in this case, it should be noted that while write through caches transfer data one element at a time, write back caches transfer entire lines, so the buffer will use more resources when using the same size parameter.

Caches can be classified as direct-mapped, n-way set-associative, and fully associative. All three types of caches are made available by Bambu by selecting appropriate values for the n_ways and way_size parameters in the pragmas that instruct the tool to generate a cache. Greater associativity can improve the effectiveness of the cache, as it provides more options for storing cache lines. However, associativity has a great cost in terms of resource usage, so it should only be used as needed.

Associative caches can have multiple positions in which a cache line can be located. For this reason, a policy must be enacted that decides which way will store data whenever a new line is read from memory. Bambu offers two different replacement policies: least recently used (LRU) and a tree-structure-based pseudo-LRU.

4 Applications

In the context of the HERMES project, Bambu has been used to automate the design of space-related applications. In particular, the project use cases cover image and vision processing algorithms, software-defined algorithms, and artificial intelligence applications. In this section, as an example of the results achievable with Bambu, we provide some simple benchmarks showcasing the features described in this paper. All the benchmarks have been synthesized using Bambu to create the hardware description and NXmap to map it on the FPGA.

Table 1 Comparison between different Boards for the Sparse Matrix-Vector Multiplication benchmark.

Target	Latency(µs)	Cycles	Frequency	LUTs	Registers	DSPs
nx1h35S	2109	75586	35.85 MHz	1453	1669	10
nx1h140tsp	2083	85960	41.28 MHz	1508	1603	10
nx2h540tsc	1190	81020	68.11 MHz	1664	1642	10

Table 2 Effect of caches on Matrix-Matrix Multiplication benchmark.

Target	Latency(µs)	Cycles	Frequency	MEM	LUTs	Registers	DSPs
nx2h540tsc	5945	302160	50.82 MHz	0	2719	3312	6
nx2h540tsc Cache	3415	129706	37.98 MHz	112	5806	3832	6

4.1 Sparse Matrix-Vector Multiplication

The first benchmark is an implementation of a double precision sparse matrix-vector multiplication (SPMV) from the MachSuite benchmark suite [9]. The results are reported in Table 1 and focus on comparing three boards from Nanoexplore: the first one is the NG-MEDIUM (nx1h35S), the second one the NG-LARGE (nx1h140tsp), and the last one the NG-ULTRA (nx2h540tsc). The considered application implements a sparse matrix-vector multiplication using fixed-size neighbour lists: the matrix has 494 rows and columns, but only 10 elements are assumed to differ from zero for each row vector multiplication.

As shown in the table, there is almost no difference in the number of resources used on the different boards: the number of LUTs, Registers and DSPs is approximately the same on all the proposed designs. This is also true for the number of cycles, as there is only a tiny difference between the fastest and slowest board. However, using larger boards has an advantage: the frequency that NXmap can achieve on nx2h540tsc is two times faster than the one achieved on nx1h35S.

4.2 Matrix-Matrix Multiplication

The second benchmark is an implementation of a single precision dense matrix-matrix multiplication with tiling to increase the locality of the requested data. The results are reported in Table 2 and focus on comparing the effect of the introduction of caches around a kernel synthesized from a regular application. We assumed a memory delay of 20 cycles for both read and write operations to simulate the delay of an external memory. The matrices used in the computations have 32 rows and columns for a total of 1024 elements. In this benchmark, we used AXI4 to exchange data between the external memory and the accelerator; we used Bambu to create three different AXI4 bundles, one for each input matrix and one for the output, to parallelize the memory operations. The difference between the two configurations is that in the one with the caches, we added a 16 elements line cache for each bundle, which allows burst operations and reduces the latency of the external memory.

This benchmark shows the effect of the caches on the performance of kernels synthesized by Bambu. In this application, which is highly regular and with data locality, the configuration with caches can achieve a 2.5 times speed-up while only using 2.1 times LUTs, which is an efficient trade-off. Another critical factor when evaluating the performances of the caches is that the number of physical resources used does not depend on the specific design but is constant. This means that the cost of the caches proportionally decreases with larger and more complex designs as the area used by the kernel increases.

Table 3 High-level synthesis of the quantized digit classifier.

Target	Latency(µs)	Cycles	Frequency	MEM	LUTs	Registers	DSPs
NG-ULTRA Embedded	3712	169649	45.7 MHz	34	4627	5714	54

4.3 Quantized Digit Classifier Synthesis

In the third example, we consider a simple MNIST model for digit classification taken from one of the TensorFlow tutorials [16], demonstrating how to convert a TensorFlow model from 32-bit floating-point to the nearest 8-bit fixed-point model using post-training integer quantization. The aim of this process is to encode the model weights with fewer bits, thereby increasing the inference speed. This is particularly useful for low-power devices such as microcontrollers or edge devices, and the same is true for FPGA-equipped space missions.

The process followed to synthesize the quantized model into an FPGA accelerator is the MLIR-based SODA methodology, which is described in detail in [1]. The Multi-Level Intermediate Representation (MLIR) [5] is a flexible and reusable infrastructure available within the LLVM project for building domain-specific compilers. MLIR allows the creation of specialized intermediate representations (IRs), known as *dialects*, which can implement analysis and transformation passes at different levels of abstraction. It can interface with multiple software programming frameworks, including the ones used for implementing deep learning algorithms.

The MNIST model is first described in Python, trained, and then quantized using the tutorial directives. Instead of running the TensorFlowLite model using the standard TensorFlowLite runtime, we output the MLIR description with the quantized weights and activations. Then, we follow a slightly modified SODA flow that translates the MLIR representation into a low-level IR that is understood by Clang/LLVM and, consequently, by Bambu, which is used to generate a corresponding hardware accelerator. Performance and area consumption metrics are shown in Table 3.

5 Conclusions

The HERMES project has extended an open-source High-Level Synthesis tool to suit the specific needs of space applications, as described in this paper. FPGAs are highly versatile and can be used in many different applications, including the space market, where HLS can help reduce the burden of developing hardware accelerators for radiation-hardened FPGAs by raising the level of abstraction required. The HERMES project has many use cases, including machine learning models, and the paper demonstrates how dedicated design methodologies can be integrated into the hardware acceleration process to improve the quality of results of the resulting FPGA design. Future developments will focus on more optimization techniques and architectural templates that can address the specific needs and requirements of aerospace applications, including both general computational demand and artificial intelligence applications.

References

1 Nicolas Bohm Agostini, Serena Curzel, Ankur Limaye, Vinay Amatya, Marco Minutoli, Vito Giovanni Castellana, Joseph B. Manzano, Antonino Tumeo, and Fabrizio Ferrandi. The SODA approach: leveraging high-level synthesis for hardware/software co-design and hardware specialization: invited. In Rob Oshana, editor, *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 – 14, 2022*, pages 1359–1362. ACM, 2022.

2 Florent de Dinechin et al. Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers*, 28(4):18–27, July 2011.

3 F. Ferrandi, V. G. Castellana, S. Curzel, P. Fezzardi, M. Fiorito, M. Lattuada, M. Minutoli, C. Pilato, and A. Tumeo. Bambu: an open-source research framework for the high-level synthesis of complex applications. In *Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1327–1330, 2021.

4 Michele Fiorito, Serena Curzel, and Fabrizio Ferrandi. Truefloat: A templatized arithmetic library for hls floating-point operators. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 486–493, 2023.

5 Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, 2021.

6 Marco Lattuada and Fabrizio Ferrandi. Code transformations based on speculative SDC scheduling. In *IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '15, pages 71–77, November 2015.

7 M. Minutoli et al. Inter-procedural resource sharing in high level synthesis through function proxies. In *International Conference on Field Programmable Logic and Applications, FPL*, pages 1–8, September 2015.

8 STMicroelectronics (FR) NanoXplore (FR). High Density European Rad-Hard SRAM-Based FPGA – First Validated Prototypes – BRAVE, 2017. URL: `https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Shaping_the_Future/High_Density_European_Rad-Hard_SRAM-Based_FPGA-_First_Validated_Prototypes_BRAVE`.

9 Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. Machsuite: Benchmarks for accelerator design and customized architectures. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 110–119. IEEE, 2014.

10 João V. Roque, João D. Lopes, Mário P. Véstias, and José T. de Sousa. Iob-cache: A high-performance configurable open-source cache. *Algorithms*, 14(8), 2021.

11 Leon Stok. Data path synthesis. *Integration*, 18(1):1–71, 1994.

12 supported by H2020 under grant agreement n. 101004203. Qualification of High-pErformance pRogrammable Microprocessor and dEvelopment of Software ecosystem (HERMES), 2021. URL: `https://dahlia-h2020.eu/`.

13 supported by H2020 under grant agreement n. 687220. Validation of European high capacity rad-hard FPGA and software tools (VEGAS), 2016. URL: `http://vegas.us.es/`.

14 supported by H2020 under grant agreement n. 730011. Space Qualification and Validation of High Performance European Rad-Hard FPGA (OPERA), 2017. URL: `https://dahlia-h2020.eu/`.

15 supported by H2020 under grant agreement n. 821969. Space Qualification and Validation of High Performance European Rad-Hard FPGA (OPERA), 2019. URL: `https://operahorizon2020.eu/`.

16 TensorFlow tutorial. Post-training integer quantization, 2023. URL: `https://www.tensorflow.org/lite/performance/post_training_integer_quant`.