computers

MDPI

Article

Designing Domain-Specific Heterogeneous Architectures from Dataflow Programs

Süleyman Savas *, Zain Ul-Abdin and Tomas Nordström

School of Information Technology, Halmstad University, Halmstad, 301 18, Sweden; zain-ul-abdin@hh.se (Z.U.-A.); Tomas.Nordstrom@hh.se (T.N.)

* Correspondence: suleyman.savas@hh.se; Tel.: +46-72-977-36-07

Received: 1 March 2018; Accepted: 21 April 2018; Published: 22 April 2018

Abstract: The last ten years have seen performance and power requirements pushing computer architectures using only a single core towards so-called manycore systems with hundreds of cores on a single chip. To further increase performance and energy efficiency, we are now seeing the development of heterogeneous architectures with specialized and accelerated cores. However, designing these heterogeneous systems is a challenging task due to their inherent complexity. We proposed an approach for designing domain-specific heterogeneous architectures based on instruction augmentation through the integration of hardware accelerators into simple cores. These hardware accelerators were determined based on their common use among applications within a certain domain.The objective was to generate heterogeneous architectures by integrating many of these accelerated cores and connecting them with a network-on-chip. The proposed approach aimed to ease the design of heterogeneous manycore architectures—and, consequently, exploration of the design space—by automating the design steps. To evaluate our approach, we enhanced our software tool chain with a tool that can generate accelerated cores from dataflow programs. This new tool chain was evaluated with the aid of two use cases: radar signal processing and mobile baseband processing. We could achieve an approximately 4× improvement in performance, while executing complete applications on the augmented cores with a small impact (2.5–13%) on area usage. The generated accelerators are competitive, achieving more than 90% of the performance of hand-written implementations.

Keywords: heterogeneous architecture design; risc-v; dataflow; QR decomposition; domain-specific processor; accelerator; Autofocus; hardware software co-design

1. Introduction

Several highly demanding applications are currently in the process of being introduced, such as autonomous vehicles, 5G communication, and video surveillance and analytics. Many of these also include artificial intelligence and machine learning algorithms, which adds significantly to their computational demands. To support these applications on vehicles and other mobile devices, there is a need for embedded high-performance computing architectures to perform streaming computations in real time.

The present generation of multi/manycore architectures, along with general purpose graphics processing units (GPGPU), aim to address these computational demands by duplicating identical processing units or cores. Some companies are already pushing the number of cores on a chip as high as a thousand [1]. The first multi/manycores were produced by duplicating the processing cores, resulting in homogeneous architectures having several identical cores on the same die [2,3]. However, the streaming applications mentioned above are comprised of a large variety of tasks which are not necessarily identical. For example, a typical massive MIMO [4,5] application, which is the core of

5G communication technology, consists of a chain of tasks, including channel encoding/decoding, precoding, OFDM modulation, channel estimation, and MIMO detection, each of which performs different computations and requires different hardware resources for efficient performance and power consumption. Some tasks do not even perform any computation but consist of only memory operations such as shuffling or transposing a matrix. Techniques such as virtualization [6] and containerization [7] aim to execute heterogeneous tasks efficiently on processing systems by encapsulating them and minimizing runtime requirements. Additionally, technologies such as hyper-threading [8] provide simultaneous execution of threads on the same processor. However, the efficiency that can be achieved is still limited by the efficiency of the underlying hardware. To achieve the highest efficiency while executing the tasks on a manycore, it is necessary to optimize individual cores to the task at hand, thus introducing heterogeneity [9–12]. One core may be optimized for efficient fast Fourier transforms (FFT) used in OFDM modulation, whereas another core can be optimized for efficient QR decomposition for matrix inversion during precoding. However, designing heterogeneous architectures is a challenging task due to the complexity of these architectures. There can be many forms of heterogeneity based on the components and how they are inter-connected and used [12]. Therefore, finding the most suitable architecture for a target application requires design space exploration. This represents a further challenge, because of the diversity in the manycore design. In performing design space exploration, manycores are often simulated partially, or as a full system. Simulations allow hiding unnecessary architectural details when a certain component is tested and require no hardware development, which reduces costs. However, simulators are usually quite slow [13].

There are software tools to simulate manycore architectures, such as Gem5 [14], ZSim [15], Graphite [16], Sniper [17], and PriME [18], however, most of them do not support the kind of complete configurable system that allows configuration of parameters such as the types and numbers of processing units, memory units, custom hardware blocks, or the network-on-chip structure. There are full system simulators such as Gem5 and SimFlex [19], which provide detailed data on different components. However, the simulation time increases beyond a feasible limit as the system grows. Additionally, the simulators do not provide timing and area information, which can be obtained by taking a design to RTL or FPGA implementation. As a result, exploring the design space of heterogeneous manycore architectures through simulations becomes increasingly challenging. Further discussion on manycore simulation can be found in [13].

In this paper, we propose a design approach for high-performance, domain-specific heterogeneous architectures based on simple cores with application-specific extensions in the form of custom hardware blocks. With this approach, instead of finding an efficient architecture to run a certain application, we aim to build the architecture automatically with a tool-chain starting from dataflow programs. The design approach can be summarized as identifying the extensions within an application domain, integrating these extensions to simple cores and, (as a future goal) generating heterogeneous manycore architectures by connecting these cores with a network-on-chip (NoC). The paper covers identification and integration of the extensions with case studies and provide the insight of connecting the extended cores with a NoC. However, it does not cover the steps needed to generate a manycore architecture. When all the design steps are automated, our approach can be used for exploring the design space of heterogeneous manycore architectures. We define the extensions (custom hardware blocks) as the compute-intensive parts (hot-spots) of the applications within a domain such as radar, baseband, or audio/video signal processing. These blocks are integrated to simple cores by extending the instruction set. The cores are tasked to execute the control flow of the application and delegate the compute-intensive parts to these blocks, using them as accelerators. The cores with accelerator extensions will be referred to as "tiles" in the rest of the paper.

As the first step towards automating the design approach, we developed a software tool to automatically generate custom hardware blocks from dataflow applications. We extended our code generation framework, Cal2Many [20], which takes dataflow applications and generates target specific code for different architectures, by adding a back-end to generate a combination of `C` and `Chisel` [21]

code. We considered two case studies to evaluate the performance and area usage of the generated code. The first case study was QR decomposition (QRD), implemented using Givens Rotations method [22]. The QRD operation is used in numerous signal processing applications including massive MIMO [23]. The second case study was an autofocus criterion calculation application, which is a key component of synthetic aperture radar systems [24]. The chosen method performs cubic interpolation based on Neville's algorithm [25] to calculating the autofocus criterion. The case studies are implemented in `CAL` dataflow programming language [26], which is a concurrent language with support for expressing parallelism. The compute-intensive parts of the case studies were identified through profiling and automatically generated as custom hardware (accelerators). The accelerators are integrated to a rocket core [27] that is based on RISC-V open-source instruction set architecture (ISA) [28]. Synthesizable designs for the integration of the rocket cores and the accelerators are generated via rocket chip system on chip generator [29]. The generated implementations were evaluated in terms of performance and area usage against hand-optimized implementations. The contributions made by this study can be summarized as:

- A generic method to design domain-specific heterogeneous manycore architectures with an emphasis on custom-designed tiles was proposed.
- An approach to design augmented cores (tiles) via instruction extension and hardware acceleration was realized, including development of a code generation tool to automate hardware accelerator generation directly from a dataflow application. This tool performs hardware/software codesign and generates `C` and `Chisel` code from `CAL` applications.
- The design method was evaluated using two case studies from baseband processing and radar signal processing. For these case studies, hand-written and automatically generated accelerators are used. The accelerators are integrated into a RISC-V core.

The remainder of the paper is structured as follows: Section 2 surveys the literature on related work. Section 3 describes the generic and realized versions of the proposed design approach. Section 4 introduces the case studies and provides details of how they are implemented. Section 5 presents the results of the case studies and a discussion of their results. Section 6 contains concluding remarks and some discussion of possible future works.

2. Related Works

In accordance with the focus of our work on generating heterogeneous manycore architectures, we provide here a review of related work on manycore design and custom hardware generation from high-level languages.

The first set of related works are on manycore generation. There are FPGA based manycore architectures developed by Sano et al. [30], Tanabe et al. [31] and Schurz et al. [32]. However, these studies develop a single architecture and do not propose any generic method for doing so. However, celerity [33], an accelerator-centric system-on-chip (SoC) design based on a combination of three tiers, does closely resemble our work. The first tier consists of five rocket cores capable of running Linux, whereas 506 smaller RISC-V cores reside in tier 2, with an accelerator in tier 3. The tiers are connected to each other with parallel links. The accelerator is generated using SystemC and high-level synthesis tools. In their design, the accelerator and the cores are placed in different tiers and all cores share the accelerator tier. In contrast, in our design, each core can have its own tightly-coupled accelerator, making the accelerator an instruction extension to the simple core. Additionally, our design starts from application development and uses application requirements to configure the architecture in terms of the number of cores, memory size, accelerator types, etc.

There are several tools available that generate hardware description from high-level languages. These tools support a variety of languages [34] including `Lava` [35] and `Clash` [36] which are functional languages similar to `Chisel` [21]. `Clash` and `Lava` are based on `Haskell` and they have compilers and support simulations. Additionally, `Clash` supports generating `Verilog`, `VHDL` and `SystemVerilog`

code [37]. `Chisel` is based on `Scala` and has a compiler for simulation and generating `Verilog` code. However, the RISC-V tools that are used in this study require `Chisel` code for integrating accelerators into the rocket cores. Therefore, we have chosen `Chisel` as our high-level hardware description language. There are more common languages, which are `C` and `C`-like languages, that are used by the tools to generate a hardware description. Some of these tools are Xilinx Vivado [38], Calypto's Catapult C [39], CoDeveloper from Impulse [40], eXCite from Y Explorations [41], Stratus from Cadence [42], and Symphony C from Synopsis [43]. In these tools, the developer is usually required to change their `C`-like code by adding pragmas and rewriting some code snippets to change the structure of the generated RTL design. Both Catapult [44] and CyberWorkBench [45] deal with generating controller and data paths. They require modifications to the `C` code and the data paths depend on the controller generation. In our design method, we do not generate any controller. The accelerators we generate consist of only data paths and control is implemented on the processing core. This difference can also be seen when comparing our work with that of Trajkovic et al. [46]. Trajkovic et al. [46] automatically generate processor cores from `C` code, including separate generation of the data path and the controller.

The Tensilica XPRES (Xtensa PRocessor Extension Synthesis) Compiler [47] automatically generates extensions, which are combinations of existing instructions. It uses a C/C++ compiler to analyze the application and find the candidate instructions. The instructions can be in the form of VLIW, vector, fused operations, or combinations of these. Therefore, the generated extensions are within the bounds of combinations of the existing instructions. Instruction extensions require modifications to the compiler and the decoder. The compiler is generated automatically. However, having a combination of instructions, including VLIW-style instructions, requires multiple parallel instruction decoders [47]. This increases the hardware cost, which may affect the clock frequency, and also limits the number of instructions that may potentially be executed in parallel [46].

Clark et al. [48] automated custom instruction generation using dataflow graphs. They discovered custom instruction candidates by exploring the dataflow graphs. They used a hardware library to estimate timing and area for the the combined primitive operations (subgraphs). Combination of a few subgraphs, to be executed on the same hardware, is performed to form a set of custom function units (CFUs). Performance and area are estimated for the set members and fed to a selection mechanism to choose the custom instruction. They also generalized the custom instruction to be used by a set or all applications within the same domain. The compiler needs to create a dataflow graph of the application, perform pattern matching to find all the subgraphs matching the CFUs, and replace all the CFUs with the custom instructions.

Koeplinger et al. [49] automatically generated accelerators for FPGAs from applications described in a high-level language using patterns, such as *map*, *reduce*, *filter*, and *groupBy*. Their framework performs high-level optimizations and describes hardware using their own intermediate representation which is based on a set of architectural templates. Their method requires the parallel patterns and pragmas to be present in the input code, which in turn requires changes to the compiler/simulator. However, with our method, the only change in the code is applied to the name of the action to be accelerated.

The SDSoC environment from Xilinx [50] only provides automatic generation of accelerators running on their Znyq and Ultrascale+ platforms, which significantly limits portability across different target platforms.

The CAL2HDL tool, developed by Janneck et al. [51], was the first attempt to generate hardware descriptions from `CAL` actor language. This tool transforms `CAL` application into an XML language-independent model before the OpenForge synthesizer is used to translate the application to a hardware description in `Verilog`. CAL2HDL supports a subset of `CAL` and the time for `Verilog` design generation increases dramatically with the design complexity, which limits its wider applicability. It generates a platform-dependent design that can be used only on Xilinx FPGAs. Another hardware description generator from `CAL` actor language is developed by Siret et al. [52]. Their tool generates VHDL, however, it lacks loop support. Bezati et al. [53] developed Xronos, aiming to support the ISO

subset of CAL actor language to generate RTL code. Xronos uses OpenForge and generates Verilog, similar to CAL2HDL, however the authors claimed that Xronos operates faster and the generated hardware uses fewer resources because of changes in the transformations applied to the IR that is used as the input to the OpenForge. In our work, we did not generate hardware to cover the whole application but only the tagged (hot-spot) actions. Therefore, our back-end does not perform all of the transformations that are covered by the other CAL tools. Additionally, we used different IRs, and generated hybrid code (C+Chisel) that embeds custom instructions (in assembly) into the C code to communicate with the generated hardware blocks.

The generated heterogeneous architecture does not require an explicit mapping/resource-discovery process to map the application onto corresponding components (core or accelerator) as our code generation framework already takes care of this by generating the code to fire the accelerator wherever necessary. However, if many of these extended cores (core + accelerator) are to be used in the same architecture, there is a need for a mapping approach such as HARD (Hybrid adaptive resource discovery) [54]. This approach might be somewhat too advanced due to its functionality support. The only significant information for a mapping approach would be the capability of the accelerator and the hot-spots of the applications. Hence, a simpler approach can be applied to map the hot-spots to the accelerators.

To summarize, we present a generic methodology to design domain-specific heterogeneous manycore architectures based on accelerators integrated into simple cores. We reveal the steps undertaken to integrate the accelerators into an open source core and use them via custom instructions. We automate custom hardware generation to facilitate design space exploration of heterogeneous architectures.

3. Design Approach

Our approach for designing domain-specific heterogeneous architectures was based on instruction augmentation through the integration of hardware accelerators added into simple cores. We based these accelerators on an analysis of the application where hot-spots suitable for hardware acceleration are identified. The overall design flows consisted of the following steps:

- Application development
- Analysis and code generation
- Accelerator integration
- System integration

In the following section, we describe a generic design flow, and then give a description of its realization, starting from CAL dataflow applications in Section 3.2. The generic design flow and its specific realization are illustrated in Figure 1. On the left hand side of the figure, where the generic flow is shown, one can see that the application is fed to the code generation and analysis tools. The analysis tool provides hot-spot information to the code generation tool, which sends feedback data to the developer. The generated hardware code is passed down to an accelerator integration tool to be integrated to a core and form a tile. These tiles are fed to a system integration tool to be connected together with a NoC to form a manycore. The generated software code is passed down to a native compiler. Together with a mapping tool, the compiler maps executables onto the generated architecture. The configuration parameters are used in different tools to determine the features of the hardware components. The right hand side of the figure shows the tools, languages and hardware components used in the realization of the design flow. Note that the system integration step is grayed out and the tools are not yet specified, because this step of the design flow is an ongoing work and is not covered in this study.

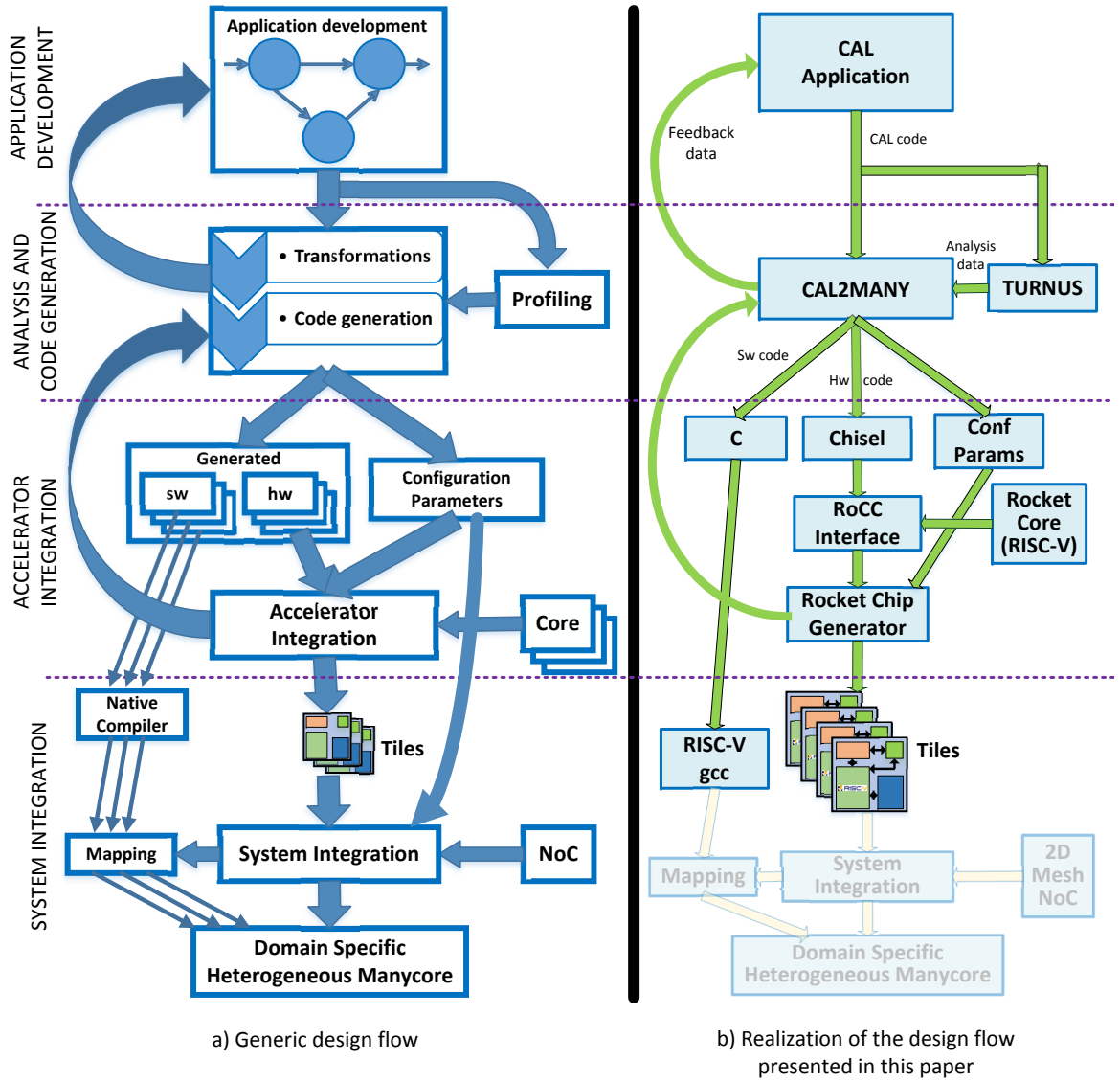

Figure 1. Illustration of the design flow and its realization. The grayed out components are not addressed by this study.

3.1. Generic Design Flow

The generic design flow consists of the following steps: (1) application development in a suitable programming language; (2) analysis and code generation to generate hardware and low-level software for the intended architecture; (3) accelerator integration where the hardware accelerator is integrated with the basic core; and (4) system integration where all the accelerated cores are integrated and connected through a NoC. This generic description is independent of the programming language, programming model, tool and hardware.

3.1.1. Application Development

The design method aims to build an efficient architecture for a specific domain of applications by integrating task-specific custom hardware into simple cores. The architecture configurations are based on the requirements of the applications. Therefore, the design method starts from the application description.

From an application programming perspective, programming languages in which concurrency is inherent in the language are attracting increased attention in mainstream parallel computing compared to sequential languages. These concurrent languages make explicit the inherent parallelism of the applications. In the proposed method, to target a manycore architecture, the application development should be performed in a programming language that can express different levels of parallelism, such as instruction-level, task-level, and data-level parallelism. This facilitates generation of the task-specific cores, which are the fundamental components of the target architectures.

A developer can use the feedback from the analysis and code generation tools to improve the application. Additionally, the feedback can be used to adjust the application to make it more suitable for use in architecture development, especially in cases where acceleration of the application is desired, but the implementation lacks a distinguishable hot-spot.

3.1.2. Analysis

The most compute-intensive parts of the application (hot-spots) were implemented as custom hardware to be integrated into the base core. To identify the compute-intensive parts of the application, analysis of the application is required (often referred as profiling). Analysis methods are usually divided into static and dynamic analysis methods [55–57]. Static analysis can be used to highlight possible coding errors, and mathematically prove properties of a program via formal methods, or to count the number of operations and operands. It can also be used to estimate execution times for applications which have constant or static behavior. However, dynamic analysis is required to obtain execution measures for applications with dynamic behavior, and to define the most frequently executed, computationally-intensive code blocks. Analysis information useful for hot-spot identification includes the execution rate, the number of operations and operands, the complexity of the operations, and the execution time.

During the analysis, architectural features can also be generated, including memory requirements, communication characteristics, and the required number of cores. These features can be adjusted during code generation for optimization purposes.

3.1.3. Code Generation

Once identified, hot-spots are candidates for implementation as accelerators. These were implemented as custom hardware, which can be performed using a high-level language such as `C` before using HLS tools to generate the RTL design, or directly, using a hardware description language. The code generation step consisted of both software and hardware generation. Software code generation resulted in native code with embedded instructions for the accelerator. This native code can then be translated to the target machine code using proprietary tools. Hardware generation involved generating hardware descriptions for the custom hardware blocks.

3.1.4. Accelerator Integration

The custom hardware block, which was generated in the previous step, needs to be integrated to the base core so that the hot-spot can be delegated to the accelerator while the rest of the application is executed on the core. The accelerator can be connected to the data bus and be memory mapped, it can be connected through custom interfaces, or even act as an instruction extension. However, instruction extensions need changes in the compiler unless the instructions are already in the instruction set and supported by the compiler. During integration, the custom hardware can be interfaced to memory to provide direct memory access. Moreover, the custom hardware can be connected to a network-on-chip through a network interface, enabling the other cores on the network to make use of the custom hardware. In-short, the integration method can vary based on different aspects, such as the features of the base core, architectural requirements, and application requirements.

3.1.5. System Integration

The prior steps of the design flow produce tiles for individual tasks in the applications. To execute the entire application, or a set of applications, the tiles need to be connected to each other to form a manycore architecture. In particular, for dataflow applications that have significant amount of core-to-core (or tile-to-tile) communication, the tiles must be connected with a proper infrastructure that supports tile-to-tile communication. The architectural information gathered during the analysis and code generation stages was used in this step while configuring the network-on-chip (NoC), and when determining whether to add tiles containing only memory to provide more on-chip memory.

In this step, an important decision is the choice of NoC topology. The efficiency of the topology might change based on the domain of the applications However, for dataflow applications, we suggest a 2D mesh structure based on our experience during previous work [22,58] with the Epiphany architecture [59]. This structure provides efficient core-to-core communication for dataflow applications in terms of bandwidth and latency. With a proper routing algorithm, such as X-Y routing, it becomes deadlock free. In addition, it is scalable, which enables different sized manycore architectures to be built. However, there are several other NoC topologies, such as torus, ring, star, tree, crossbar, and hypercube [60–62]. A torus has wrap-around connections between the opposite edges of the mesh. However, it would be smart to save the edge connections for extensions to the NoC and external connections. The crossbar topology establishes all-to-all connections, however, this becomes quite expensive in terms of area usage when the size of the network increases. In a star topology, all messages pass through a central router. This router is a potential congestion point, especially for applications with intensive communication needs. In tree topologies, the root node and the nodes close to it become a bottleneck. In the hypercube topology, the number of neighbours of each node is equal to the degree of the topology. To increase the number of nodes in such topologies, one needs to increase the degree of the topology, and hence increase the number of connections to each node. From a scalability point of view, this is an obstacle, as the structure of the NoC needs to be changed with each dimension increase.

3.2. Realization of the Design Flow

We now describe the realization of the design flow in terms of the tools, languages, and other components that have been applied in our proposed approach.

3.2.1. Application Development

Applications that process a continuous stream of data and have high performance requirements, such as video coding/decoding, radar signal processing, and wireless communication, can be implemented efficiently using a dataflow programming model [63]. In this model, applications consist of a number of tasks, which are connected to one another to form the execution flow. They can execute concurrently and express parallelism. We adopted such a dataflow programming model, and developed our applications as streaming applications. An example streaming application can be seen in Figure 2, in which the raw input data flow through different tasks and finally produce the output data. We chose `CAL` actor language [26] as the application development language because of its concurrent nature, compliance to streaming applications, and ease of use.

`CAL` [26] is a high-level dataflow programming language [64] that has been adopted by the MPEG Reconfigurable Video Coding (RVC) [65] working group as a part of their standardization efforts. A `CAL` program consists of actors, which are stateful operators. Actors consist of code blocks called actions that can transform input data streams into output data streams. The state of the actor usually changes while performing the transformation. When there is more than one actor in a program, these actors get connected with channels/ports to form a network. Each actor consumes the tokens on its input ports and produces new tokens on its output ports.

The execution order of the actions can be defined with an explicit schedule in the form of a state machine together with priority definitions for the actions. Additionally, CAL provides a network language to instantiate actors, define channels, and connect the actors via these channels.

Another advantage of using CAL is the opportunity to integrate application analysis and code generation (both software and hardware) in our Cal2Many framework.

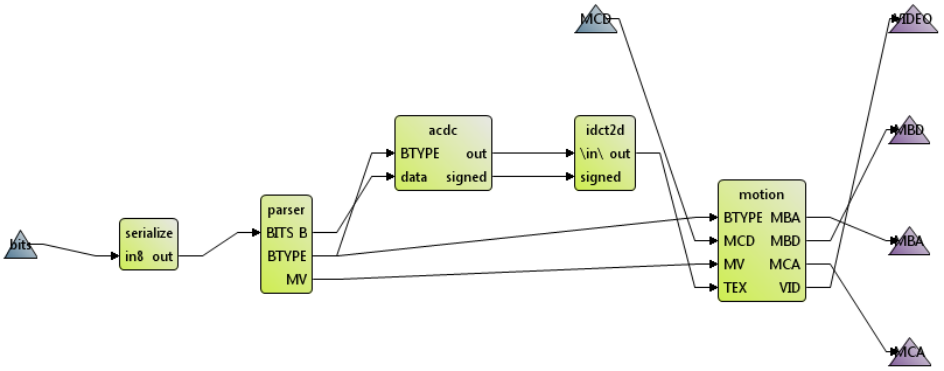

Figure 2. A streaming application example (MPEG-4 Simple Profile Decoder [66]).

3.2.2. Analysis

The analysis method and the tools employed depend on the programming language used for application development. For instance, for applications written in C, one can use Valgrind [67] and gnu profiler (gprof) [68] to obtain the total number of instruction fetches per function, the number of function calls, the total execution time of the application and the amount of time spent in each function. We used these tools for one of our Autofocus criterion calculation implementations, which is hand-written in C.

For the applications developed in CAL, we used TURNUS [69], which is a framework for profiling dataflow applications running on heterogeneous parallel systems. This framework implements a CAL dataflow profiler on top of the Open RVC-CAL Compiler (Orcc) [70]. TURNUS provides high level application and architecture modeling as well as tools for application profiling, system level performance estimation, and optimization.

TURNUS provides both static and dynamic analysis information relevant for both the software and hardware aspects, however our focus is on the software aspects. Some of the key analysis information provided by this tool are the firing rates of the actions, the number of executed operations, the input/output tokens consumed and produced, and communication buffer utilization. To obtain dynamic analysis data such as firing rates of the actions, the application was executed at least once. For each executed action, both the computational load and the data-transfers and storage load were evaluated. The computational load, given as weights in Figure 3, was measured in terms of executed operators and control statements (i.e., comparison, logical, arithmetic and data movement instructions). This information was used for hot-spot identification, while the communication information was used for configuring the network-on-chip.

This tool can provide performance estimation based on an architecture model that is an undirected graph where each node represents an operator (a processing element such as a CPU or an FPGA in the terms of [71]) or a communication element (e.g., bus and memories). However, the performance estimation is not critically necessary to identify the hot-spots of the applications. Moreover, we do not want to limit our hot-spot identification to any platform. Therefore, we do not focus on performance estimation.

There are other efforts to profile the applications at different levels such as basic block level in static single assignment (SSA) form to find out the most frequently executed instruction sequences [72,73]. However, we maintained the analysis at a higher level of abstraction to keep it simple and

architecture agnostic. CAL actions are meant to be small code blocks and yet they can be significantly compute-intensive by including heavy floating-point operations such as multiplication, division, and taking a square root. Therefore, analyzing the application at the action level is sufficient to identify hot-spots.

The case studies implemented in CAL were analyzed statically and dynamically via TURNUS. Firing rate of the actions together with the number of executed operations are sufficient to identify the hot-spots. The tool provides weights, which can be interpreted as a compute density, for each action based on their firing rate, number of operations and the number of operands used in the operations. This information can be used directly for hot-spot identification. Figure 3 shows an overview of the actions in the QRD case study and their weights in an actor. One can see that *calculate_boundary* action is the hot-spot of the given actor despite not being the most frequently executed action. This is due to the number of operations in the actions.

		Overall			
Actor	**Action**	**Firings**		**Weight**	
instance	calculate_boundary	256	8.60%	94,722.47	64.66%
instance	calculate_inner	1920	64.52%	46,080.00	31.46%
instance	read_x_in	256	8.60%	2560.00	1.75%
instance	read_x_in_not_done	256	8.60%	1792.00	1.22%
instance	col_done	256	8.60%	1280.00	0.87%
instance	row_done	16	0.54%	32.00	0.02%
instance	read_x_in_done	16	0.54%	16.00	0.01%

Figure 3. An overview of firing rates and weights of actions in an actor performing QR decomposition.

Some actors could have dynamic execution paths (different execution order and frequency for the actions) depending on the input tokens or the state of the actor. In such cases, the application can be executed a number of times to obtain an average value for the analysis data.

Gathering the analysis data and identifying the hot-spots are performed automatically by tools. Once the hot-spot actions are identified, they are tagged manually for the code generation.

3.2.3. Code Generation

To automate the code generation, we used our in-house developed Cal2Many framework, which is a source-to-source compiler [20]. Figure 4 gives an overview of the framework. It takes CAL application as input and transforms it into two consecutive intermediate representations, namely Actor Machine (AM) [74] and Action Execution Intermediate Representation (AEIR) [20]. After the transformations and utilization of necessary libraries, it finally generates native code in various target specific languages.

In CAL, actors execute by firing actions, which satisfy the conditions such as the state of the actor, the availability of input tokens, and their values. While generating the code, the framework has to take ordering of these conditions into account together with the number of the tests performed before firing an action. We adapted a simple actor model called *actor machine* to schedule the testing of the firing condition of an action. To get closer to the target imperative languages, we used the second intermediate representation (AEIR), which is a data structure that allows generating code in imperative languages such as `C`, `C++` and `java`. It has constructs for function declarations, expressions, statements, and functions calls. Translation of AM to AEIR consisted of two main passes. The first pass dealt with the translation of CAL constructs to imperative constructs including functions, statements, expressions, actions, and variable declaration. The pass took into account the translation of the AM to a sequential action scheduler.

We previously developed four different back-ends, i.e., sequential `C` for uni-processors, `aJava` and `aStruct` for Ambric architecture, native-`C` for Epiphany architecture, and target specific language

(Scala subset) for ePuma [75] and EIT [76] SIMD architectures, as shown in Figure 4. The `C` and `Chisel` hybrid back-end was developed as part of this work.

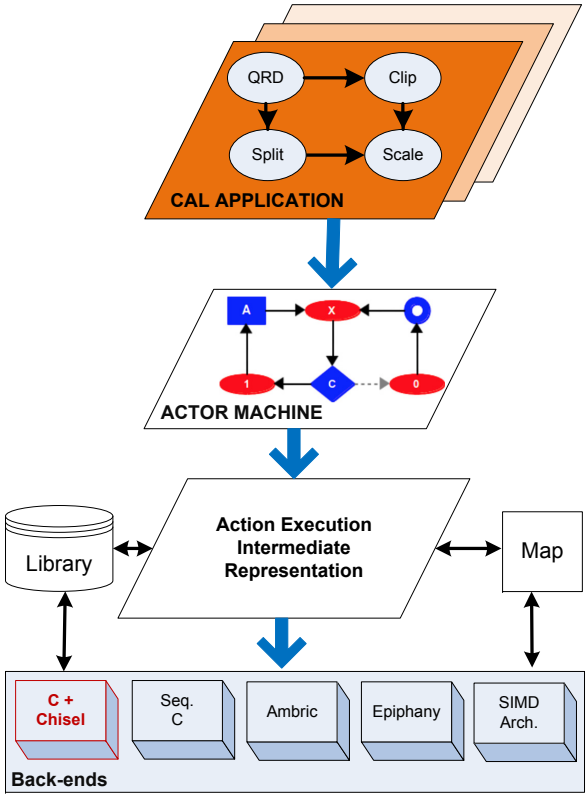

Figure 4. An overview of Cal2Many framework including the back-end for generating custom hardware blocks.

We used the `C` back-end to generate the software code to be executed on the base RISC-V processor core. We modified the parallel `C` back-end to generate RISC-V compilable `C` code, including embedded custom instructions. However, code generation in our design flow consists of two passes. The first pass generates the architecture-specific software code, which is `C` code with embedded custom instructions provided by the RISC-V ISA. The second part is generating the hardware blocks in `Chisel`, corresponding to the hot-spots in the `CAL` application. To generate the hardware blocks, we developed a new back-end that generates a combination of hardware and software code, using the `Chisel` and `C` languages, respectively. Those actions, which are tagged as hot-spots during the analysis step, were converted into hardware blocks acting as accelerators in `Chisel`, whereas the rest of the application was converted into target-specific software, which at present is in `C`. The code generator makes use of a library of hardware blocks consisting of efficient implementations for floating-point division [77] and square root operations.

The bodies of the tagged actions are replaced with custom instruction calls to trigger the hardware accelerators. The necessary input data are forwarded to the accelerators and the results are read through these instructions. The bit fields of the instruction can be seen in Figure 5, where $xs1$ and $xs2$ indicate the usage of source registers, and xd indicates if the accelerator will write a result in the destination register. If the xd bit is set, the core will wait until the accelerator finishes the computation. The generated hardware is later integrated to a rocket core through the rocket chip generator.

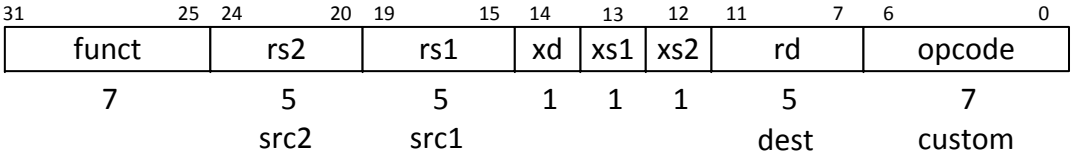

Figure 5. Custom instruction provided by the RISC-V ISA to call the custom hardware blocks.

Cal2Chisel Back-end

The common language used in the RISC-V environment is `Chisel` [21], which is an open-source hardware description language developed at UC Berkeley. It is embedded in the Scala language [78]. Some features of the `Chisel` language are parameterized generators, algebraic construction, abstract data types, interfaces, bulk connections, hierarchical, object oriented and functional constructions and multiple clock domains. The `Chisel` compiler, which is provided by UC Berkeley, can generate low level synthesizable `Verilog` code.

The new back-end makes use of the intermediate representations (AM and AEIR) provided by the Cal2Many framework to generate a combination of `C` (software) and `Chisel` (hardware) code. The `C` code for the untagged (non-hot-spot) actions is generated in the pass where the AEIR structure is visited once. However, when the tagged (hot-spot) actions are visited in the AEIR structure, the `Chisel` code and custom instruction calls in `C` are generated in two phases. In the first phase, the entire action structure is scanned to generate a static single assignment (SSA) form by generating new versions of variables for each assignment and placing phi functions in necessary places. Wires are defined for the first set of variables and the corresponding `Chisel` code is generated. Then, information about the usage of actor-wide global variables and input and output ports are gathered for identifying the inputs and the outputs of the accelerator. An instance of the `Chisel` code for inputs and outputs can be seen as follows:

```
val io = IO(new Bundle{
val r_in = Input(UInt(width = 32.W))
val x_in_in = Input(UInt(width = 32.W))
val r_out = Output(UInt(width = 32.W))
val c_out = Output(UInt(width = 32.W))
val s_out = Output(UInt(width = 32.W))
})
```

In the second phase, the custom instructions in C/assembly and the rest of the `Chisel` code is generated. The custom instructions are inlined as assembly code via macros defined in a header file. The inputs and outputs are identified, i.e., the global variables to be read are identified as inputs and the ones to be modified are identified as both inputs and outputs. All variables are defined (except the first set that are already defined in the first phase). While going through the statements, the phi functions are replaced by multiplexers and the common operations such as the floating-point operations and complex number operations are replaced with hardware blocks from a manually-developed library. Finally, the outputs are connected to the final versions of the corresponding variables. In the current implementation of the back-end, loops and arrays are not supported. If only one element of a global array is used in the tagged action, the index will be used out of the hardware block to access the array and the value will be an input to the hardware.

We have experienced that the critical path of the generated hardware in many cases becomes quite long and causes the maximum clock frequency to be significantly lowered. To alleviate this, it is necessary to introduce pipelining in the accelerators. Therefore, a pipelining feature was added to the back-end by using pipelined hardware blocks for arithmetic operations. Delay registers were inserted into the data paths where they need to be synchronized.

Figure 6 shows the generated `C` and `Chisel` code (without delay registers) for a tagged `CAL` action. In the `CAL` code, indices are used for accessing the array elements. However, to avoid transfer of the whole array to the accelerator, these indices are moved to the `C` code and only single array elements are transfered. The custom instruction is called twice with different *funct* fields (refer to Figure 5) to provide all the inputs to the accelerator. The *funct* field is used to let the interface know when all the inputs are provided so that they can be fed to the accelerator. The floating-point operations are performed by instantiating the pre-defined hardware blocks such as *FPAdd*, *FPMult* and *fpSqrt*, as seen in the generated `Chisel` code (Figure 6). The code also shows the connection between the modules and different variables.

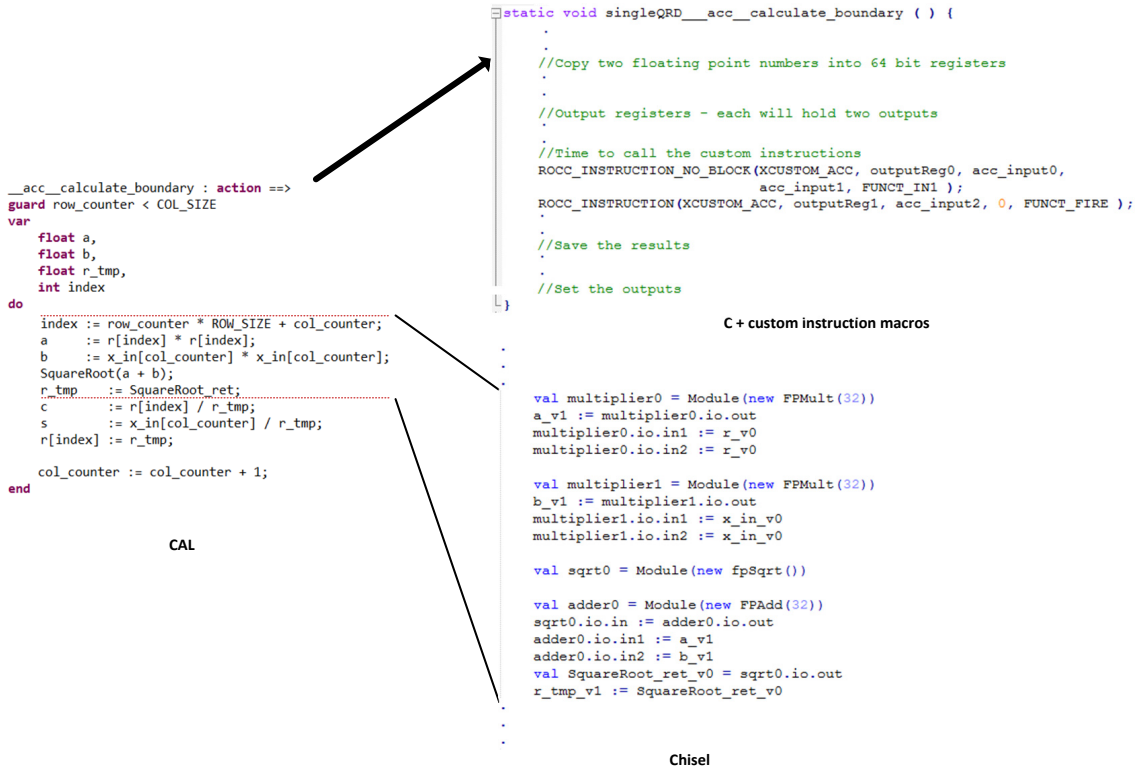

Figure 6. A tagged CAL action and the corresponding generated C and Chisel code.

3.2.4. Accelerator Integration

We have used the RISC-V environment for our integration step. RISC-V is an open instruction set architecture based on reduced instruction set computing (RISC) principles. It originated at University of California (UC), Berkeley. Several processing cores have been developed that implement this instruction set, such as the Berkeley Out-of-Order Machine (BOOM) [79], rocket core [27], Sodor CPU [80], picoRV32 [81] and scr1 [82]. We chose the rocket core [27] for our case studies as it has an interface to connect custom hardware blocks that enables us to create cores with application-specific augmentations. Additionally, the rocket chip generator produces emulator and synthesizable `Verilog` code for this core and its surrounding components.

Rocket core [27] is an in-order scalar processor based on the RISC-V ISA. It features a five-stage pipeline and has an integer ALU, an optional FPU and L1 data and instruction caches. This core supports up to four accelerators via an interface called rocket custom co-processor (RoCC) [83], as shown in Figure 7. We can see a tile consisting of the rocket core with L1 cache and an accelerator connected to the core and the cache via the RoCC interface. Custom instructions of the RISC-V ISA can be forwarded to the accelerator through this interface. Depending on a bit field in the custom instruction, the core might halt until it receives a response from the accelerator.

A rocket core can be generated via the rocket chip generator using a Scala program that invokes the `Chisel` compiler to produce RTL describing a complete system-on-chip (SoC). This generator allows the developers to configure the generated core by changing parameters such as cache sizes, FPU usage, number of cores, and accelerator usage. The Rocket chip environment also includes cycle-accurate Verilator [84] for simulations and a cycle-accurate C++ emulator.

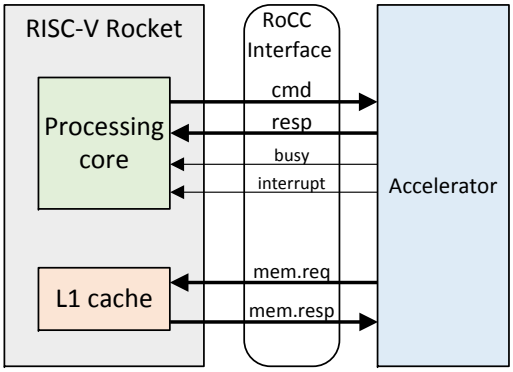

Figure 7. A simplified view of the RoCC interface.

The accelerator was integrated into the rocket core using the RoCC interface in several steps. First, the `Chisel` code for the accelerator was copied into the `Chisel` source folder of the rocket chip generator. Then, a `Chisel` class (which becomes a module in `Verilog`) was added to the rocket core source. This extends the RoCC interface, instantiates the accelerator, and connects the accelerator I/O wires to the RoCC interface I/O wires. This class needs to be instantiated in a configuration class within the configuration file of the rocket core, where the core components can be added, removed or modified. Custom instructions are bound to the accelerators in this configuration class. Each custom instruction can be bound to a different accelerator. Because four custom instructions are supported by RISC-V ISA, the number of accelerators that can be connected to the same core is limited to four. However, this can be increased using the *funct* field of the custom instruction as an identifier to determine the accelerator to use, while connecting the RoCC interface I/O wires to the accelerator I/O wires. The RISC-V ISA supports two 64-bit source registers in the custom instructions. If the accelerator requires more input data at once, the extended RoCC interface is used for storing the input data until the last inputs arrive and feed all the data to the accelerator at once. The accelerator returns the result through the same interface.

The rocket chip configuration file includes many different configurations to determine the components and their settings to be used in the generated architectures. A new configuration class is needed to instantiate the new rocket core integrated with the accelerator. Once the new configurations are added, the generator can produce a cycle-accurate emulator or synthesizable `Verilog` code for the new tile consisting of the base core, memory, and the accelerator.

This step of the design flow produces tiles containing different components including processor, accelerator, memory and the connections between these components. Theoretically, the types of the tiles can be:

- Processor Tile, consisting of a processing core, local memory and optionally an accelerator
- Memory Tile, consisting of only memory
- Accelerator Tile, consisting of only an accelerator

The tile types are illustrated in Figure 8. Thus far, the rocket chip generator allows us to generate processor tiles with and without the accelerators. However, we plan to extend the tile generation to cover all types of tiles.

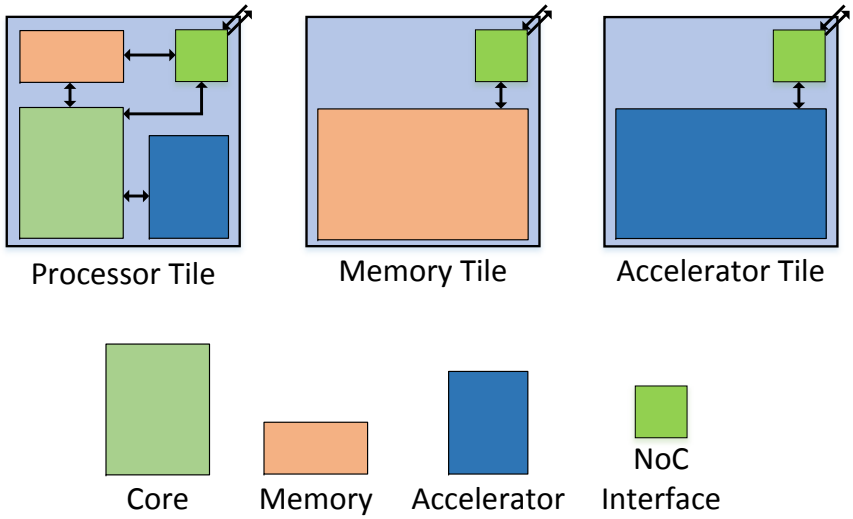

Figure 8. Types of tiles.

3.2.5. System Integration

The last step of the design flow is system-level integration of the tiles. This step was not realized in this study, hence we do not have a generated manycore architecture. The rocket chip generator has configurations to generate architectures with more than one core. However, it uses cache to connect the cores together. Because this is not as efficient as having a network-on-chip (to handle cases with intense core-to-core communication), we do not generate manycore architectures via the rocket chip generator. This step of the design method is being pursued in our ongoing work with the aim to add support for a network-on-chip to the rocket chip generator.

4. Case Studies and Implementations

This section provides the details of the implementation of the case studies Autofocus criterion calculation and QR decomposition.

The case studies are considered as a proof-of-concept for the design method, and to evaluate the hardware back-end of the Cal2Many framework. The first case study is QR decomposition, which is implemented in `CAL` actor language. The second case study is an Autofocus criterion calculation using cubic interpolation. Different versions of this application are implemented in sequential and dataflow fashions, in the `C` and `CAL` languages, respectively. The `C` version is implemented as a proof-of-concept, whereas the `CAL` version is implemented to evaluate the automatic code (`C` and `Chisel`) generation with a more complicated accelerator. The Cal2Many framework is used for automatically generating `C` and `Chisel` code for the `CAL` implementations of the case studies. Single tiles are produced at the end of each case study.

In the case studies, where the accelerators are automatically generated, hand-written accelerators are also developed, and used as reference implementations for evaluation. The accelerators are integrated into a rocket core through the RoCC interface. `Verilog` code for the accelerators is generated and synthesized using Xilinx tools. Additionally, rocket cores with and without the accelerators are synthesized individually. The accelerators are integrated to the rocket core in three steps:

- Connect the RoCC interface to the I/O of the accelerator.
- Adjust the core configuration and binding the accelerator to a custom instruction.
- Adjust the platform configuration to use the new core configuration which includes the accelerator.

The following sections provide details of the case studies and their corresponding implementations.

4.1. QR Decomposition

QR decomposition (QRD) or QR factorization is decomposition of a matrix into an orthogonal matrix Q and an upper triangular matrix R [85]. The decomposition equation for a square matrix A is simply $A = QR$. The matrix A does not necessarily need to be square. The equation for an $m \times n$ matrix, where $m \geq n$, is as follows:

$$A = QR = Q \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = Q_1 R_1.$$

QRD is used in numerous applications for replacing matrix inversions to avoid precision loss and reduce the number of operations. It is also a part of the solution to the linear least squares problem and the basis of an eigenvalue algorithm (the QR algorithm). There are several different methods to perform QRD, such as the Givens Rotations, Householder and Gram–Schmidt methods [22].

The method employed in this work for decomposition is Givens Rotations with a systolic array implementation [86]. The structure of the parallel implementation is given in Figure 9. There are 10 actors, consisting of two different types, namely boundary and inner actors. These actors can be mapped onto separate cores. However, in this case study, they are manually converted into actions to be combined in a single actor and executed on a single core. When these actors are combined, a few further control actions are required to help with communication and scheduling. All of the actions can be seen in Figure 3.

The following computations are performed in *calculate_boundary* and *calculate_inner* actions.

calculate_inner:

```
r = s * x  + c * r
x = c * x - s * r
```

calculate_boundary:

```
c = r / sqrt(r^2 + x^2)
s = x / sqrt(r^2 + x^2)
r = sqrt(r^2 + x^2)
```

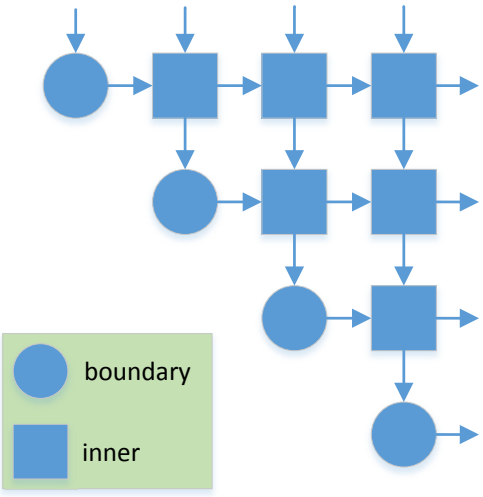

Figure 9. Systolic Array implementation of Givens Rotations (QRD). The arrows show the direction of the data movement.

According to the analysis tool, the *calculate_boundary* action is the hot-spot of the application, as shown in Figure 3. Therefore, `Chisel` code is generated for this action while `C` code is generated

for the remainder of the application. The hardware for the action is also implemented manually as a baseline for comparison.

RISC-V instructions support one destination register. However, the QRD accelerator returns at least three floating-point numbers, thus requiring at least two 64-bit registers. Therefore, the custom instruction is called twice for firing the hand-written accelerator: once for sending the input data to the accelerator and reading the first two results and once for reading the last result. These operations are distinguished via the *funct* bits of the custom instruction within the extended RoCC interface that stores the last result until the second instruction call. The generated `C + Chisel` accelerator requires three custom instruction calls because it has five floating-point inputs and three outputs. The custom instruction is called twice to send the inputs, and a further time to read the last output.

4.2. Autofocus Criterion Calculation in Synthetic Aperture Radar Systems

Synthetic-Aperture Radar (SAR) systems produce high resolution images of the ground. The back-projection integration technique has been applied in SAR systems, enabling processing of the image in the time domain, which makes it possible to compensate for non-linear flight tracks. However, the cost is typically a high computational burden. In reality, the flight path is not perfectly linear. This can, however, be compensated for in the processing. The compensations are typically based on positioning information from GPS. If this information is insufficient, or even missing, autofocus can be used. The autofocus calculations use the image data itself, and are performed before each subaperture merge. One autofocus method, which assumes a merge base of two, relies on finding the flight path compensation that results in the best possible match between the images of the contributing subapertures in a merge. Several flight path compensations are thus tested before a merge. The image match is checked according to a selected focus criterion. The criterion assumed in this study is maximization of correlation of image data. [24]. As the criterion calculations are carried out many times for each merge, it is important that these are done efficiently. The calculations include interpolations and correlation calculations.

The autofocus implementation takes 6×6 blocks of image pixels as input and applies complex cubic interpolations to these blocks. The hot-spot of the application is identified as the computation of the cubic interpolation. This computation is a combination of complex multiplication, complex subtraction and complex division. Each of these operations is implemented as a building block. Figure 10 presents the structure of the complex division block, consisting of floating-point operations. This block is used as one of the building blocks while developing the accelerator, and it is represented as *CDiv* in Figure 11 that shows the fully flattened version of the accelerator.

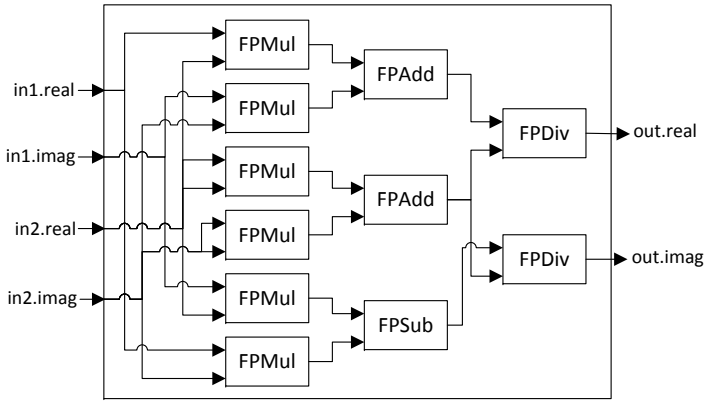

Figure 10. Building block that performs division on complex numbers.

The mathematical expressions of the operations performed during the cubic interpolation are as follows:

$$d_i = x_{int} - x_i \quad i \in \{0..3\}$$

$$p_{01} = \frac{(y_0 \times d_1) - (y_1 \times d_0)}{x_0 - x_1}$$

$$p_{12} = \frac{(y_1 \times d_2) - (y_2 \times d_1)}{x_1 - x_2}$$

$$p_{23} = \frac{(y_2 \times d_3) - (y_3 \times d_2)}{x_2 - x_3}$$

$$p_{02} = \frac{(p_{01} \times d_2) - (p_{12} \times d_0)}{x_0 - x_2}$$

$$p_{13} = \frac{(p_{12} \times d_3) - (p_{23} \times d_1)}{x_1 - x_3}$$

$$p_{03} = \frac{(p_{02} \times d_3) - (p_{13} \times d_0)}{x_0 - x_3}$$

where x_{int} is a constant, x and y are the positions and values of the input pixels, and p_{03} is the computed result. The Red highlighted box in Figure 11 performs the operations to calculate `p01` and the small red boxes represent delay registers.

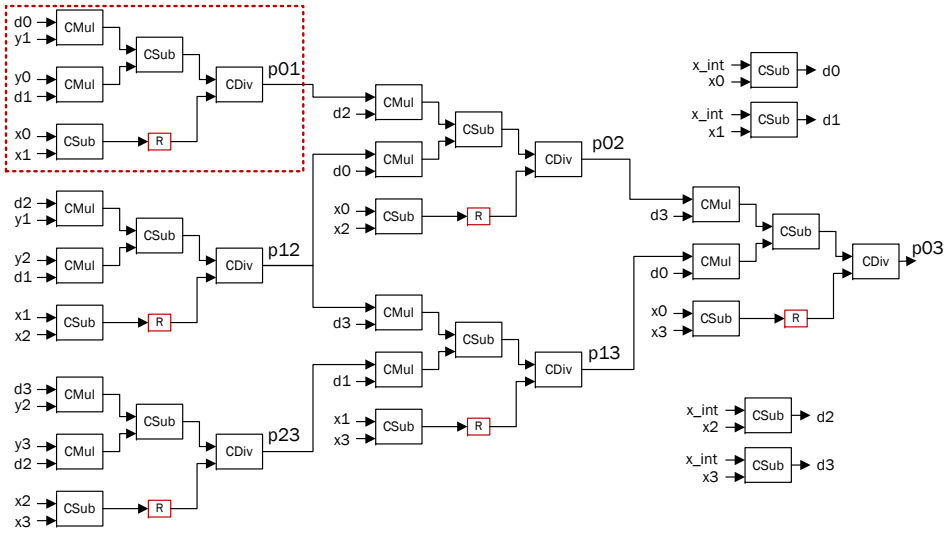

Figure 11. Structure of the flat cubic interpolation accelerator.

We implemented the autofocus criterion calculation in `CAL` actor language and generated the software and hardware through our tool-chain. Furthermore, we manually implemented the software and the hardware of the application in native `C` and `Chisel` languages, respectively. These implementations will be referred to as *Generated* and *Hand-written* in the rest of the paper.

The *Generated* and *Hand-written* implementations have different designs. Figure 12 presents the dataflow diagram of the `CAL` (*Generated*) implementation. One can see that the main function (*cubic*) is implemented as an actor and instantiated several times to exploit data-level and task-level parallelism. However, since our tool-chain supports composition of actors [87] and we are intending to generate code for single-core execution, therefore, all the actors of the autofocus criterion calculation (seen in Figure 12) are composed into a single actor to be mapped on a single core. This operation results in a more complex scheduler in the composition, and requires handling actor-to-actor communication. To analyze the `CAL` actor, the TURNUS framework is used, and cubic interpolation is identified as the hot-spot.

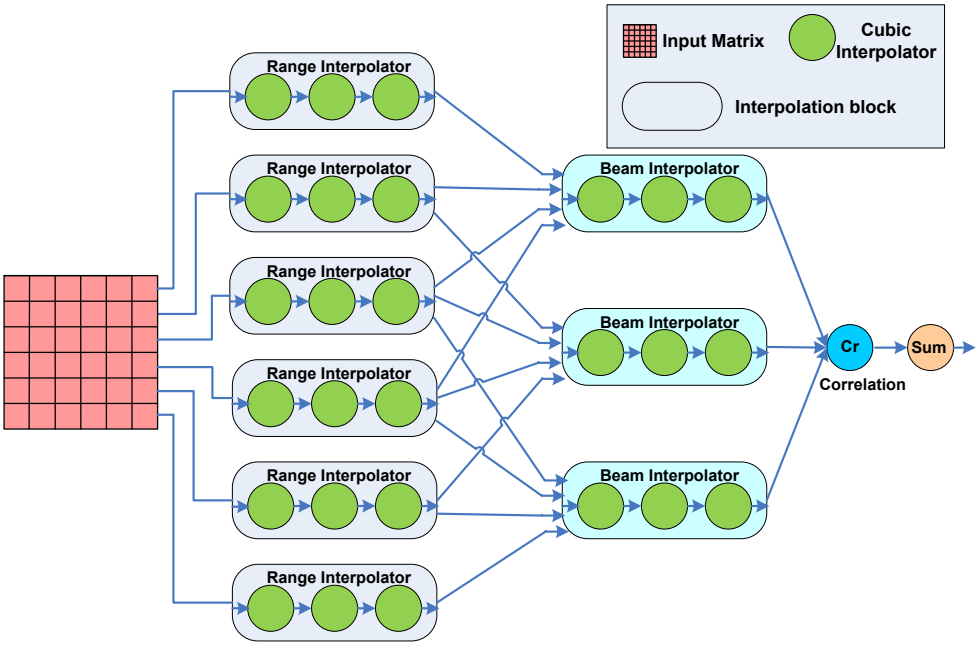

Figure 12. Dataflow implementation of autofocus criterion calculation.

The cubic interpolation of autofocus criterion calculation is implemented in `CAL` as a action that performs only a single step of the interpolation (calculation of a single p value). This helps to keep the action simple and decreases the number of global variables that provide the input values to the interpolation kernel. The action is executed six times in a row to complete the interpolation of 4 pixels and produce a single pixel result.

The *Generated* accelerator makes use of complex number arithmetic blocks implemented manually with pipelines. The number of used complex and floating-point blocks of the accelerators are given in Tables 1 and 2. The generated accelerator results are equal to the results given for the *Folded Accelerator* in the tables. The generated accelerator requires 4 complex numbers (each of 64-bits) as inputs to be fired. Each complex number has two floating-point values representing the real part and the imaginary part. The total number of inputs to the accelerator is 8 floating-point numbers, represented in IEEE-754 single precision (binary32) format [88]. The RoCC interface supports transfer of two 64-bit values from core to the accelerator with each custom instruction. Therefore, we combine real and imaginary values of each complex number in a 64-bit register and send 4 floating-point numbers to the accelerator with each instruction. Hence, two custom instruction calls are required to transfer four complex numbers to the accelerator. The accelerator is pipelined and has 20 stages. We have extended the RoCC interface to handle synchronization of the input data between the instruction calls for firing. The extended interface stores the input data until it receives the instruction with the last data required. Then, all of the input data are fed to the accelerator together with an enable signal that fires the accelerator.

Two different versions of the *Hand-written* accelerator were developed. The first version of the cubic interpolation accelerator, which is flat and presented in Figure 11, consists of six identical structures—one of them is highlighted in Figure 11. This hardware accelerator is identical to the cubic interpolation function of the hand-written `C` implementation, and requires 4 pixels as input, with each pixel consisting of two complex numbers (value and position). These inputs are named as x and y in Figure 11. It takes four instructions to send all the input data.

The second version of the hand-written accelerator is optimized (folded) for resource usage. It uses only one of these six identical structures in a software controlled loop and the four individual *CSub* blocks to perform the cubic interpolation, as shown on the right hand side of the Figure 11.

Similar to the generated accelerator, this accelerator is fired six times by the software to calculate one result. To be fired, this accelerator requires four complex numbers, which takes two instructions.

The flat accelerator is designed with a fully pipelined structure and can produce one result (a complex number) per cycle. However, due to the limitations in the data transfer, it can be fired at every fourth cycle (ignoring the cache misses). The folded accelerator also has a pipelined structure and can be fired at every cycle. However, input limitations mean that it can only be fired every other cycle. Moreover, with the folded accelerator, the first three iterations of the loop for calculating one final result can be executed two cycles apart (without waiting for the result of previous iterations). However, synchronization problems may occur with the results returned from the accelerator if the core does not halt until the arrival of the result.

Table 1 presents the number of (blocks) complex operations executed by the accelerators, whereas Table 2 shows the number of floating-point operations performed by each block and the accelerators. The generated accelerator and the folded hand-written accelerator have the same results and are presented in *Folded Accelerator* column. In total, 140 floating-point operations are performed for each set of inputs (4 pixels). Floating-point multiplication and division operations employ integer multiplications. To perform these integer multiplications on-chip, DSP blocks are utilized.

Table 1. Number of complex operations in the accelerators.

	Flat Accelerator	Folded Accelerator
CMul	12	2
CDiv	6	1
CSub	16	6

Table 2. Number of floating-point operations in blocks and the accelerators.

	Flat Accelerator	Folded Accelerator	CMul	CSub	CDiv
FPMul	84	14	4	-	6
FPDiv	12	2	-	-	2
FPSub	50	15	1	2	1
FPAdd	24	4	1	-	2

5. Results

This section provides the performance, resource utilization and timing results for the case studies considered in this work. The performance results are obtained through executing the case studies on the cycle-accurate emulators generated with the rocket chip generator. Hardware counters are used to obtain the cycle counts. Resource usage results are provided by Xilinx synthesis tools. The target platform is the Xilinx VCU108 evaluation kit, including the Virtex UltraScale XCVU095 FPGA.

5.1. QR Decomposition

The QRD case study takes a matrix of 16×16 elements as input data. Table 3 present the performance results of different implementations of the QRD in terms of achieved clock frequency and cycle count. The first implementation results are for software generated by the Cal2Many framework and executed on the rocket core without using any custom hardware. The second row of results correspond to the rocket core and accelerator where both the software for the core and hardware for the accelerator are generated by the Cal2Many tool-chain. Finally, the last set of results are for the generated software code executing on rocket core along with a hand-written custom hardware implementation.

Table 3. Performance results for QRD in terms of cycle count and clock frequency.

	Cycle Count	Clock Freq.
Rocket Core	366 k	58 MHz
Rocket Core + Generated Accelerator	92 k	56 MHz
Rocket Core + Hand-written Accelerator	89 k	56 Mhz

It is apparent from the results that could achieve similar clock frequencies for all three cases, primarily because of the pipelining incorporated in the Cal2Chisel back-end that results in generating pipelined accelerators. The combination of accelerator and the rocket core outperforms the single-core execution by a factor of 4. However, there is a small drop of 4% in the performance of the generated accelerator with respect to the hand-written accelerator. The main reason for this difference the different the number of inputs. During code generation, use of the global array indices is moved out of the accelerator. However, they are still passed to the accelerator as inputs. The hand-written accelerator takes in two floating-point inputs requiring a single custom instruction call, whereas the generated accelerator takes in five floating-point inputs, requiring at least two custom instruction calls. Copying the input variables to the source registers adds extra cost. This difference can be reduced further by incorporating optimizations in the hardware code generation.

The resource usage and clock frequency results for the rocket core and the accelerators can be seen in Table 4. The main reason for the difference in look-up tables and flip-flops in the two accelerator implementations is the lack of support for a mixture of integer/floating-point operations. The code generation tool treats integer operations similarly to floating-point operations, and utilizes floating point operation blocks. Specifically, there is an integer addition operation that is converted into a floating-point operation in the generated accelerator, which causes an increase in resource usage. Thus, there is potential for further improvement in the resource usage of the generated hardware. The maximum clock frequency of the generated accelerator is 20 MHz without the pipelining feature. However, by inserting the pipeline stages, the maximum clock frequency increases to 104 MHz, equivalent to the clock frequency of the hand-written accelerator. The accelerators use the same hardware library for the arithmetic operations, which plays a significant role in determining the clock frequency.

Table 4. Resource usage and timing results of the rocket core and QRD accelerators

	LUT	FF	BRAM	DSP	Clock Freq.
Rocket Core	39,843	16,512	12	24	58 MHz
Generated Accelerator	1165	914	2	25	104 MHz
Hand-written Accelerator	999	812	2	25	104 MHz

In addition, the accelerator is fully pipelined and parallelized, meaning it can produce one set of results per cycle. However, due to limitations in the data movement between the core and the accelerator, it is not possible to make use of this feature. The standalone latency of the accelerator is 14 cycles and the total latency (including data movement) is approximately 27 cycles.

To summarize, we are able to achieve an improvement of 4× in performance with respect to rocket core software execution, and the overhead of generated accelerator vs hand-written accelerator is minimal.

5.2. Autofocus Criterion Calculation

The the autofocus criterion calculation application, consisting of integrated cubic interpolation accelerators, is tested using 12 kernels, each consisting of 6×6 image pixels. Cubic interpolation is applied 324 times within the autofocus criterion calculation application.

Table 5 shows the performance results for the autofocus criterion calculation in terms of cycle count and achieved clock frequencies. All of these implementations use hand-optimized software. There are two implementations of the hand-written accelerators, namely: flat and folded. The flat accelerator is fully parallelized and has the best performance in terms of cycle count. On the other hand, the folded hand-written implementation, which applies the same cubic implementation block in a loop, has similar a performance to the generated accelerator. Clearly, all of the implementations with integrated accelerators outperform the software execution on rocket core by a factor of 2.8–4.8.

Table 5. Performance results for the autofocus criterion calculation in terms of cycle counts and achieved clock frequencies.

	Cycle Count	Clock Freq.
Rocket Core	306 k cycles	58 MHz
Rocket Core + Hand-written Flat Accelerator	64 k cycles	56 MHz
Rocket Core + Hand-written Folded Accelerator	108 k cycles	58 MHz
Rocket Core + Generated Accelerator	108 k cycles	58 MHz

The hand-written flat accelerator implementing cubic interpolation produces a final result in 52 cycles. Due to the pipelined structure, the accelerator can produce one result every clock cycle, provided all the required inputs are loaded. Since the RoCC interface can provide at most four floating-point inputs per-cycle, the accelerator can produce four result per four cyclez. Furthermore, the rocket core waits until the accelerator produces its result. Hence, the accelerator can compute one result in 52 cycles. When the accelerator is integrated into the core, it takes eight cycles to copy the data into the source registers and fire the accelerator. Additionally, it takes two cycles to read the result. Therefore, in total, it takes 62 cycles to compute a result. While measuring these results, the caches are pre-filled, hence there are no cache misses.

The hand-written folded accelerator and the generated accelerator, both of which perform cubic interpolation, are similar in their design. They both compute one of the six identical functions shown in Figure 11 and produce an intermediate result in 20 cycles. If the data movement overhead is ignored, this accelerator can calculate the final result in six iterations, taking 120 cycles. However, for each firing of the accelerator, data movement costs six additional cycles (four for filling the source register and two for reading the destination register). Thus, each iteration takes 26 cycles, meaning that the whole autofocus criterion computation takes 156 cycles (ignoring any cache misses).

Table 6 provides synthesis results for individual rocket core, individual accelerator implementations and integrated (core + accelerator) designs. The results for the integrated designs include the resource usage of the extended RoCC interfaces. The hand-written folded and the generated accelerators use only one of the six main blocks, highlighted with dashed lines in Figure 11. Therefore, one would expect a 6× reduction in resource usage compared to the hand-written flat implementation. However, the `CSub` components, which produce the `d` values, are used in the folded version as well. Hence, LUT and FF usage do not decrease exactly by six times. However, one can see that the DSP and BRAM usages actually drop to 1/6, as they are only used in the main blocks.

Table 6. Resource usage and timing results for the rocket core and the integrated designs including hand-written and generated accelerators.

	LUT	FF	DSP	BRAM	Clock Freq.
Rocket Core	39,843	16,512	24	12	58 MHz
Hand-written Flat Accelerator	24,707	14,471	252	3	126 MHz
Rocket + Hand-written Flat Accelerator	68,860	36,722	276	16	56 MHz
Hand-written Folded Accelerator	5187	3732	42	0.5	131 MHz
Rocket + Hand-written Folded Accelerator	46,124	21,518	66	14	58 MHz
Generated Accelerator	5239	3220	42	0.5	117 MHz
Rocket + Generated Accelerator	44,767	20,113	58	14	58 MHz

The increase in the size of the integrated design causes the critical path of the processor to increase insignificantly. When integrated, the flat accelerator causes the max clock frequency of the processor to decrease from 58 MHz to 56 MHz, whereas the folded and generated accelerators do not result in any reduction of the clock frequency. When synthesized individually, hand-written flat, hand-written folded, and generated accelerators achieve 126 MHz, 131 MHz and 117 MHz clock frequencies, respectively.

We have also automatically generated software code from the `CAL` program and executed it separately on the rocket core and on the integrated design (rocket core + accelerator). The software execution on the rocket core, take 607 k cycles. The use of the generated accelerator reduces the execution time to 196 k cycles. The main reason is that the hand-written (`C`) implementation fires the accelerator six times in a row (in the same function). However, the generated (`CAL`) implementation fires the accelerator once with each function call and returns to the scheduler, which is a state machine implemented as `if-else` statements. The variables copied into the source registers are global variables and it takes four instructions to move them to the source registers in the generated implementation, whereas in the hand-written implementation the variables copied into the source registers are sent as arguments/parameters to the function and it takes two instructions to move them to the source registers. The `CAL` implementation consists of many actors and would benefit from being mapped onto different cores. The automated composition of the actors cause a significant performance reduction, which will be our next goal for improving software generation.

To summarize, the hand-written flat accelerator increases the performance of the rocket core by a factor of 4.8 at the cost of a significantly larger resource usage. However, the hand-written folded and the generated accelerators increase the performance by 2.8× with a very modest increase in the resource usage.

6. Conclusions

Manycore architectures are emerging to meet the performance demands of today's complex applications. These architectures usually consist of identical or generic cores. However, applications usually consist of a variety of tasks. To achieve the highest efficiency, there is a need to specialize the cores for individual tasks and introduce heterogeneity. However, designing such an architecture is a challenging task.

In this paper, we propose an approach for designing domain-specific heterogeneous tiles based on instruction augmentation through integrating custom hardware to simple cores. The main purpose of generating these tiles is to use them to build future manycore architectures. The design approach aims to build the architecture based on the requirements of the applications within a domain. We developed a tool to automatically generate the custom hardware blocks from `CAL` applications. We generated custom hardware that is then integrated into a core that executes the RISC-V instruction set. We evaluated our approach by implementing QR decomposition and an autofocus criterion calculation case studies in `CAL` actor language. The case studies revealed that RISC-V custom instructions can be used for integrating specialized custom hardware to boost performance. The processing cores with custom

hardware outperform the cores without custom hardware by a factor of 2.8–4.8. Additionally, it is shown that automated hardware generation increases the area of the hardware by approximately 0–12% and the performance drop (in terms of cycle count) is 0–9%, which can be decreased even further by optimizing the code generation tool. The pipelining stages implemented during hardware generation play a significant role in improving the maximum clock frequency for the accelerators. However, when integrated with the rocket core, the frequency is saturated to the maximum clock frequency of the rocket core i.e., 56 or 58 MHz.

The significant advantage of our design approach is in terms of productivity, as the resulting accelerator can be produced from the same dataflow program without committing any extra effort into developing the hardware. Manycore architectures are slowly evolving in the direction of specialized cores, and we believe that the proposed approach will facilitate the design of new architectures based on specialized cores.

In the future, we plan to generate a manycore architecture by integrating many tiles with a network-on-chip. While doing this, we will use several case studies to generate the tiles to support a set of applications. Later, we aim to extend the hardware block library and fully automate every design step, including system integration.

Acknowledgments: This research was performed within the HiPEC (High Performance Embedded Computing) and NGES (Towards Next, Generation Embedded Systems: Utilizing Parallelism and Reconfigurability) projects, which are funded by the Swedish Foundation for Strategic Research (SSF) and a VINNOVA Strategic Innovation grant, respectively. The funds for the projects cover the costs of publishing this research in open access journals. We would like to thank Struan Gray for his contributions with the language of the paper and to Schuyler Eldridge for his help with the accelerator integration.

Author Contributions: Suleyman Savas developed the design method and fine tuned it with feedback from the other authors. The experiments were conceived by all authors. Suleyman Savas designed and performed the experiments. The results were analyzed by all authors. Suleyman Savas led the paper writing with contributions from the other authors.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Bohnenstiehl, B.; Stillmaker, A.; Pimentel, J.J.; Andreas, T.; Liu, B.; Tran, A.T.; Adeagbo, E.; Baas, B.M. KiloCore: A 32-nm 1000-processor computational array. *IEEE J. Solid-State Circuits* **2017**, *52*, 891–902.[CrossRef]
2. Behling, S.; Bell, R.; Farrell, P.; Holthoff, H.; O'Connell, F.; Weir, W. *The POWER4 Processor Introduction and Tuning Guide*; IBM Redbooks: New York, NY, USA, 2001.
3. Intel Corporation, Intel Pentium Processors. Available online: https://www.intel.com/content/www/us/en/products/processors/pentium.html (accessed on 20 December 2017).
4. Rusek, F.; Persson, D.; Lau, B.K.; Larsson, E.G.; Marzetta, T.L.; Edfors, O.; Tufvesson, F. Scaling up MIMO: Opportunities and challenges with very large arrays. *Signal Process. Mag. IEEE* **2013**, *30*, 40–60.[CrossRef]
5. Larsson, E.; Edfors, O.; Tufvesson, F.; Marzetta, T. Massive MIMO for next generation wireless systems. *Commun. Mag. IEEE* **2014**, *52*, 186–195.[CrossRef]
6. Barham, P.; Dragovic, B.; Fraser, K.; Hand, S.; Harris, T.; Ho, A.; Neugebauer, R.; Pratt, I.; Warfield, A. Xen and the art of virtualization. In Proceedings of the ACM SIGOPS Operating Systems Review, Bolton Landing, NY, USA, 19–22 October 2003; Volume 37, pp. 164–177.
7. Dua, R.; Raja, A.R.; Kakadia, D. Virtualization vs containerization to support paas. In Proceedings of the International Conference on Cloud Engineering (IC2E), Boston, MA, USA, 10–14 March 2014; pp. 610–614.
8. Koufaty, D.; Marr, D.T. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro* **2003**, *23*, 56–65.[CrossRef]
9. Kumar, R.; Farkas, K.I.; Jouppi, N.P.; Ranganathan, P.; Tullsen, D.M. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-36, San Diego, CA, USA, 3–5 December 2003; pp. 81–92.
10. Brodtkorb, A.R.; Dyken, C.; Hagen, T.R.; Hjelmervik, J.M.; Storaasli, O.O. State-of-the-art in heterogeneous computing. *Sci. Progr.* **2010**, *18*, 1–33.[CrossRef]

11. Liu, S.; Ro, W.W.; Liu, C.; Salas, A.C.; Cérin, C.; Han, J.J.; Gaudiot, J.L. EHA: The extremely heterogeneous architecture. In Proceedings of the 12th International Symposium on Pervasive Systems, Algorithms and Networks (ISPAN), San Marcos, TX, USA, 13–15 December 2012; pp. 30–36.
12. Mittal, S. A survey of techniques for architecting and managing asymmetric multicore processors. *ACM Comput. Surv. (CSUR)* **2016**, *48*, 45.[CrossRef]
13. Zarrin, J.; Aguiar, R.L.; Barraca, J.P. Manycore simulation for peta-scale system design: Motivation, tools, challenges and prospects. *Simul. Model. Pract. Theory* **2017**, *72*, 168–201.[CrossRef]
14. Binkert, N.; Beckmann, B.; Black, G.; Reinhardt, S.K.; Saidi, A.; Basu, A.; Hestness, J.; Hower, D.R.; Krishna, T.; Sardashti, S.; et al. The gem5 simulator. *ACM Sigarch Comput. Archit. News* **2011**, *39*, 1–7, [CrossRef]
15. Sanchez, D.; Kozyrakis, C. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. *ACM Sigarch Comput. Archit. News* **2013**, *41*, 475–486.[CrossRef]
16. Miller, J.E.; Kasture, H.; Kurian, G.; Gruenwald, C.; Beckmann, N.; Celio, C.; Eastep, J.; Agarwal, A. Graphite: A distributed parallel simulator for multicores. In Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA), Bangalore, India, 9–14 January 2010; pp. 1–12.
17. Carlson, T.E.; Heirmant, W.; Eeckhout, L. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC), Seatle, WA, USA, 12–18 November 2011; pp. 1–12.
18. Fu, Y.; Wentzlaff, D. PriME: A parallel and distributed simulator for thousand-core chips. In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Monterey, CA, USA, 23–25 March 2014; pp. 116–125.
19. Hardavellas, N.; Somogyi, S.; Wenisch, T.F.; Wunderlich, R.E.; Chen, S.; Kim, J.; Falsafi, B.; Hoe, J.C.; Nowatzyk, A.G. Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *ACM Sigmetr. Perform. Eval. Rev.* **2004**, *31*, 31–34.[CrossRef]
20. Gebrewahid, E.; Yang, M.; Cedersjo, G.; Ul-Abdin, Z.; Gaspes, V.; Janneck, J.W.; Svensson, B. Realizing efficient execution of dataflow actors on manycores. In Proceedings of the 12th IEEE International Conference on Embedded and Ubiquitous Computing (EUC), Milano, Italy, 26–28 August 2014; pp. 321–328.
21. Bachrach, J.; Vo, H.; Richards, B.; Lee, Y.; Waterman, A.; Avižienis, R.; Wawrzynek, J.; Asanović, K. Chisel: Constructing hardware in a scala embedded language. In Proceedings of the 49th Annual Design Automation Conference, San Francisco, CA, USA, 3–7 June 2012; pp. 1216–1225.
22. Savas, S.; Raase, S.; Gebrewahid, E.; Ul-Abdin, Z.; Nordström, T. Dataflow implementation of qr decomposition on a manycore. In Proceedings of the Fourth ACM International Workshop on Many-core Embedded Systems, Seoul, South Korea, 18–22 June 2016; pp. 26–30.
23. Paulraj, A.J.; Gore, D.A.; Nabar, R.U.; Bolcskei, H. An overview of MIMO communications-a key to gigabit wireless. *Proc. IEEE* **2004**, *92*, 198–218.[CrossRef]
24. Ul-Abdin, Z.; Ahlander, A.; Svensson, B. Energy-Efficient Synthetic-Aperture Radar Processing on a Manycore Architecture. In Proceedings of the 2013 42nd International Conference on Parallel Processing, Lyon, France, 1–4 October 2013; pp. 330–338.
25. Neville, E.H. *Iterative Interpolation*; St. Joseph's IS Press: Trivandrum, India, 1934.
26. Eker, J.; Janneck, J.W. Dataflow programming in CAL—balancing expressiveness, analyzability, and implementability. In Proceedings of the Conference Record of the Forty Sixth Asilomar Conference on Signals, Systems and Computers (ASILOMAR), Pacific Grove, CA, USA, 4–7 November 2012; pp. 1120–1124.
27. Rocket Core Overview. Available online: http://www.lowrisc.org/docs/tagged-memory-v0.1/rocket-core/ (accessed on 4 April 2018).
28. Waterman, A.; Lee, Y.; Patterson, D.A.; Asanovic, K. The risc-v instruction set manual, volume I: Base user-level isa. *EECS Dep.* **2011**, *7*, 475.
29. Asanovic, K.; Avizienis, R.; Bachrach, J.; Beamer, S.; Biancolin, D.; Celio, C.; Cook, H.; Dabbelt, D.; Hauser, J.; Izraelevitz, A.; et al. *The Rocket Chip Generator*; EECS Department, University of California: Berkeley, CA, USA, 2016.
30. Sano, K.; Hatsuda, Y.; Yamamoto, S. Scalable streaming-array of simple soft-processors for stencil computations with constant memory-bandwidth. In Proceedings of the 19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Salt Lake City, UT, USA, 1–3 May 2011; pp. 234–241.

31. Tanabe, S.; Nagashima, T.; Yamaguchi, Y. A study of an FPGA based flexible SIMD processor. *ACM Sigarch Comput. Archit. News* **2011**, *39*, 86–89.[CrossRef]
32. Schurz, F.; Fey, D. A programmable parallel processor architecture in FPGAs for image processing sensors. In Proceedings of the integrated design and process technology (IDPT'07) Conference, Antalya, Turkey, 3–8 June 2007; pp. 30–35.
33. Ajayi, T.; Al-Hawaj, K.; Amarnath, A.; Dai, S.; Davidson, S.; Gao, P.; Liu, G.; Lotfi, A.; Puscar, J.; Rao, A.; et al. Celerity: An Open-Source RISC-V Tiered Accelerator Fabric. In Proceedings of the Symposium on High Performance Chips (Hot Chips), Cupertino, CA, USA, 20–22 August 2017.
34. Svensson, B. A study of design efficiency with a high-level language for FPGAs. In Proceedings of the International conference on Parallel and Distributed Processing Symposium (IPDPS), Long Beach, California, USA, 26–30 March 2007; pp. 1–7.
35. Bjesse, P.; Claessen, K.; Sheeran, M.; Singh, S. Lava: Hardware design in Haskell. In Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming, Baltimore, MD, USA, 26–29 September 1998; Volume 34, pp. 174–184.
36. Baaij, C.; Kooijman, M.; Kuper, J.; Boeijink, A.; Gerards, M. Clash: Structural descriptions of synchronous hardware using haskell. In Proceedings of the 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD), Lille, France, 1–3 September 2010; pp. 714–721.
37. CLASH, from Haskell to Hardware. Available online: http://www.clash-lang.org/ (accessed on 21 April 2018).
38. Vivado Design Suite, Xilinx Inc. Available online: https://www.xilinx.com/products/design-tools/vivado.html/ (accessed on 21 April 2018).
39. Catapult C Synthesis, Calypto Design Systems. Available online: http://calypto.agranderdesign.com/catapult_c_synthesis.php/ (accessed on 21 April 2018).
40. Impulse CoDeveloper, Impulse Accelerated Technologies. Available online: http://www.impulseaccelerated.com/products.htm (accessed on 21 April 2018).
41. EXCite, Y Explorations Inc. Available online: http://www.yxi.com/products.php (accessed on 21 April 2018).
42. Stratus High-Level Synthesis, Cadence Design Systems. Available online: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html/ (accessed on 21 April 2018).
43. Synphony C Compiler, Synopsis Inc. Available online: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/synphony-c-compiler.html (accessed on 21 April 2018).
44. Catapult High-Level Synthesis Platform, Mentor Graphics. Available online: https://www.mentor.com/hls-lp/catapult-high-level-synthesis/ (accessed on 21 April 2018).
45. CyberWorkBench, NEC Corporation. Available online: http://www.nec.com/en/global/prod/cwb/index.html? (accessed on 21 April 2017).
46. Trajkovic, J.; Abdi, S.; Nicolescu, G.; Gajski, D.D. Automated generation of custom processor core from c code. *J. Elect. Comput. Eng.* **2012**, *2012*, 7.[CrossRef]
47. Goodwin, D.; Petkov, D. Automatic generation of application specific processors. In Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, San Jose, CA, USA, October 30–November 1 2003; pp. 137–147.
48. Clark, N.T.; Zhong, H.; Mahlke, S.A. Automated custom instruction generation for domain-specific processor acceleration. *IEEE Trans. Comput.* **2005**, *54*, 1258–1270.[CrossRef]
49. Koeplinger, D.; Delimitrou, C.; Prabhakar, R.; Kozyrakis, C.; Zhang, Y.; Olukotun, K. Automatic generation of efficient accelerators for reconfigurable hardware. In Proceedings of the 43rd International Symposium on Computer Architecture, Seoul, South Korea, 18–22 June 2016; pp. 115–127.
50. Kathail, V.; Hwang, J.; Sun, W.; Chobe, Y.; Shui, T.; Carrillo, J. SDSoC: A Higher-level Programming Environment for Zynq SoC and Ultrascale+ MPSoC. In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 21–23 February 2016; p. 4.
51. Janneck, J.W.; Miller, I.D.; Parlour, D.B.; Roquier, G.; Wipliez, M.; Raulet, M. Synthesizing hardware from dataflow programs. *J. Signal Process. Syst.* **2011**, *63*, 241–249.[CrossRef]
52. Siret, N.; Wipliez, M.; Nezan, J.F.; Rhatay, A. Hardware code generation from dataflow programs. In Proceedings of the Conference on Design and Architectures for Signal and Image Processing (DASIP), Edinburgh, UK, 26–28 October 2010; pp. 113–120.

53. Bezati, E.; Mattavelli, M.; Janneck, J.W. High-level synthesis of dataflow programs for signal processing systems. In Proceedings of the 8th International Symposium on Image and Signal Processing and Analysis (ISPA), Trieste, Italy, 4–6 September 2013; pp. 750–754.
54. Zarrin, J.; Aguiar, R.L.; Barraca, J.P. Hard: Hybrid adaptive resource discovery for jungle computing. *J. Netw. Comput. Appl.* **2017**, *90*, 42–73.[CrossRef]
55. Michalska, M.; Boutellier, J.; Mattavelli, M. A methodology for profiling and partitioning stream programs on many-core architectures. *Procedia Comput. Sci.* **2015**, *51*, 2962–2966.[CrossRef]
56. Wang, C.; Li, X.; Zhang, H.; Wang, A.; Zhou, X. Hot spots profiling and dataflow analysis in custom dataflow computing SoftProcessors. *J. Syst. Softw.* **2017**, *125*, 427–438.[CrossRef]
57. Janneck, J.W.; Miller, I.D.; Parlour, D.B. Profiling dataflow programs. In Proceedings of the International Conference on Multimedia and Expo, Hannover, Germany, 23–26 April 2008; pp. 1065–1068.
58. Savas, S.; Gebrewahid, E.; Ul-Abdin, Z.; Nordström, T.; Yang, M. An evaluation of code generation of dataflow languages on manycore architectures. In Proceedings of the 20th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Chongqing, China, 20–22 August 2014; pp. 1–9.
59. Olofsson, A.; Nordström, T.; Ul-Abdin, Z. Kickstarting high-performance energy-efficient manycore architectures with epiphany. In Proceedings of the 48th Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA, USA, 2–5 November 2014; pp. 1719–1726.
60. Salminen, E.; Kulmala, A.; Hamalainen, T.D. Survey of network-on-chip proposals. *White Pap. OCP-IP* **2008**, *1*, 13.
61. Fernandez-Alonso, E.; Castells-Rufas, D.; Joven, J.; Carrabina, J. Survey of NoC and programming models proposals for MPSoC. *Int. J. Comput. Sci. Issues* **2012**, *9*, 22–32.
62. Agarwal, A.; Iskander, C.; Shankar, R. Survey of network on chip (noc) architectures & contributions. *J. Eng. Comput. Archit.* **2009**, *3*, 21–27.
63. Najjar, W.A.; Lee, E.A.; Gao, G.R. Advances in the dataflow computational model. *Parallel Comput.* **1999**, *25*, 1907–1929.[CrossRef]
64. Horowitz, E. Data Flow Programming Languages. In *Fundamentals of Programming Languages*; Springer: Berlin/Heidelberg, Germany, 1984; pp. 373–393.
65. Bhattacharyya, S.S.; Eker, J.; Janneck, J.W.; Lucarz, C.; Mattavelli, M.; Raulet, M. Overview of the MPEG Reconfigurable Video Coding Framework. *J. Signal Process. Syst.* **2011**, *63*, 251–263.[CrossRef]
66. Savas, S. Implementation and Evaluation of Mpeg-4 Simple Profile Decoder on a Massively Parallel Processor Array. Master's Thesis, Halmstad University, Halmstad, Sweden, 2011.
67. Nethercote, N.; Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, San Diego, CA, USA, 11–13 June 2007; Volume 42, pp. 89–100.
68. Fenlason, J.; Stallman, R. GNU Gprof. GNU Binutils. Available online: http://www.gnu.org/software/binutils (accessed on 21 April 2018).
69. Casale-Brunet, S.; Alberti, C.; Mattavelli, M.; Janneck, J.W. Turnus: A unified dataflow design space exploration framework for heterogeneous parallel systems. In Proceedings of the Conference on Design and Architectures for Signal and Image Processing (DASIP), Cagliari, Italy, 8–10 October 2013; pp. 47–54.
70. Yviquel, H.; Lorence, A.; Jerbi, K.; Cocherel, G.; Sanchez, A.; Raulet, M. Orcc: Multimedia Development Made Easy. In Proceedings of the 21st ACM International Conference on Multimedia, Barcelona, Spain, 21–25 October 2013; pp. 863–866.
71. Pelcat, M.; Nezan, J.F.; Piat, J.; Croizer, J.; Aridhi, S. A system-level architecture model for rapid prototyping of heterogeneous multicore embedded systems. In Proceedings of the Conference on Design and Architectures for Signal and Image Processing (DASIP), Sophia Antipolis, France, 22–24 October 2009; p. 8.
72. Atasu, K.; Dimond, R.G.; Mencer, O.; Luk, W.; Özturan, C.; Dündar, G. Optimizing instruction-set extensible processors under data bandwidth constraints. In Proceedings of the Conference on Design, Automation and Test in Europe, EDA Consortium, Nice, France, 16–20 April 2007; pp. 588–593.
73. Haaß, M.; Bauer, L.; Henkel, J. Automatic custom instruction identification in memory streaming algorithms. In Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, New Delhi, India, 12–17 October 2014; p. 6.

74. Janneck, J. A machine model for dataflow actors and its applications. In Proceedings of the Conference Record of the 45th Asilomar Conference on Signals, Systems and Computers (ASILOMAR), Pacific Grove, CA, USA, 6–9 November 2011; pp. 756–760.
75. Karlsson, A.; Sohl, J.; Liu, D. Epuma: A processor architecture for future dsp. In Proceedings of the International Conference on Digital Signal Processing (DSP), Singapore, 21–24 July 2015; pp. 253–257.
76. Zhang, C. Dynamically Reconfigurable Architectures for Real-Time Baseband Processing. Ph.D. Dissertation, Lund University, Lund, Sweden, 2014.
77. Savas, S.; Hertz, E.; Nordström, T.; Ul-Abdin, Z. Efficient Single-Precision Floating-Point Division Using Harmonized Parabolic Synthesis. In Proceedings of the Annual Symposium on VLSI (ISVLSI), Bochum, Germany, 3–5 July 2017; pp. 110–115.
78. Odersky, M. *The Scala Language Specification*, Version 2.9; Programming Methods Laboratory, EPFL: Lausanne, Switzerland, 2014
79. Asanovic, K.; Patterson, D.A.; Celio, C. *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor*; Technical Report; University of California at Berkeley: Berkeley, CA, USA, 2015.
80. Celio, C. RISC-V Sodor CPU. Available online: https://github.com/ucb-bar/riscv-sodor (accessed on 3 November 2017).
81. Clifford, W. PicoRV32—A Size-Optimized RISC-V CPU. Available online: https://github.com/cliffordwolf/picorv32 (accessed on 3 November 2017).
82. Syntacore. SCR1—An Open-Source RISC-V Compatible MCU Core. Available online: https://github.com/syntacore/scr1 (accessed on 3 November 2017).
83. Yarp, C. An Introduction to the Rocket Custom Coprocessor Interface. Available online: http://c199.eecs.berkeley.edu/~cs250/sp16/disc/Disc02.pdf (accessed on 10 November 2017).
84. Snyder, W.; Galbi, D.; Wasson, P. Verilator, Verilog HDL Simulator. Available online: https://www.veripool.org/wiki/verilator (accessed on 10 November 2017).
85. Goodall, C.R. 13 Computation using the QR decomposition. *Handb. Stat.* **1993**, *9*, 467–508.
86. Savas, S. Linear Algebra for Array Signal Processing on a Massively Parallel Dataflow Architecture. Bachelor's Thesis, School of Information Technology, Halmstad University, Halmstad, Sweden, 2009.
87. Gebrewahid, E. Tools to Compile Dataflow Programs for Manycores. Ph.D. Thesis, Halmstad University Press, Halmstad, Sweden, 2017.
88. IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*; IEEE Press: Piscataway, NJ, USA, 2008; p. 58.