Advances in Electrical and Computer Engineering **Volume 19, Number 3, 2019**

Compiler Optimization on Instruction Scheduling for a Specialized Real-Time Floating Point Soft-Core Processor

Michael KIRCHHOFF, Lothar WAGNER, Wolfgang FENGLER
Technische Universität Ilmenau, Group for Computer Architecture and Embedded Systems, Germany
michael.kirchhoff@tu-ilmenau.de

Abstract—**This paper presents the authors' research in the field of specialized optimizing assembly language compilers for embedded real-time soft-core processor systems on FPGAs. With this soft-core processor, we are targeting a highly specialized field of applications that require large floating-point precision and other unique characteristics. Therefore, a specialized optimizing assembly language compiler is necessary in order to provide the needed machine code and optimize it in a way that efficient usage of the internal parallelism mechanisms is possible, resulting in major performance benefits on single-core, multi-core and vector processors. One important key feature is the design-time analyzability to meet the hard real-time constraints of any given problem.**

Index Terms—**dynamic compiler, optimization methods, processor scheduling, scheduling algorithms, vector processor.**

I. INTRODUCTION

Embedded systems are becoming more and more common nowadays, largely driven by the enhanced capabilities of modern technology. As semiconductor complexity continues to rise in accordance to Moore's Law, the SOC (System on Chip) and FPGA (Field Programmable Gate Array) devices as part of SORC (System on Reprogrammable Chip) used in these products are getting more and more powerful. This enables the development of new products to solve problems that were too expensive for previous technologies. An important key point in using more and more complex technologies is the ability to minimize the cost of development of new products with those technologies. Here, SORCs and especially FPGAs provide a big advantage, the ability of reconfiguration, resulting in much shorter design cycles, tackling the challenge of the increasing system complexity and reducing the cost at the same time. To minimize the cost even further, it is necessary to reuse and to modularize as many components as possible. Therefore, the authors developed a complete tool-chain in order to (partially) automatize the development of SORC logic for very complex problems with a large quantity of requirements. One of the most important parts of the tool-chain will be introduced in this paper.

This is a good way to exploit high level programming languages in FPGA designs and to use a soft-core with accompanying software development tools [1-3].

A prime example of solvable problems with the help of new and faster SORCs is the real-time processing and evaluation of camera data where large streams of data have to be processed within strict timing limitations. As the capabilities of the hardware are ever increasing, the software and hardware-description development has to come up with modular, scalable and reusable solutions as well.

As the applications for these systems range from toys to safety critical and potentially dangerous devices there won't be any one-for-all solution for all these systems. Especially the class of applications with hard real-time and complex computation requirements will need a highly specialized, yet future-proof and reusable solution, which is targeting the priorities and limitations by design. This is especially true for real-time systems that can be safety-critical and violation of the timing constraints can lead to costly or dangerous accidents.

Having a scalable solution for this kind of application that can be deployed on modern and future devices, utilizing the ever growing complexity, would significantly speed up development times and thus not only decrease the time to market but also cut development costs.

In this paper, the authors suggest the use of a highly specialized soft-core processor for usage in FPGA designs that is specialized for hard real-time computing. Generic soft-core processors like the Nios II [4], the MicroBlaze [5] or the LEON3 [6] do not provide the necessary specialized logic in order to efficiently compute a complex hard real-time problem. A comparison of different available soft-core processors provided by commercial vendors and open source communities can be seen in [7]. All those soft-cores are not suitable for a highly specialized field of applications that require a large floating point precision and other unique characteristics like hard real-time ability or the possibility to use complex specialized operators [8-9].

Therefore, the authors developed a highly reusable and adaptable double precision floating point soft-core processor, called ViSARD [10], that tackles problems of this field of application, e.g. highly problem specialized operators or real-time guarantees.

The fully pipelined design of the ViSARD enables high processing speed while offering a deterministic, clock-exact timing in every execution. In order to specialize the soft-core to a specific task, the instruction set can be adjusted and entire operations can be added or removed from the design while processing speed of the individual instructions is adjustable as well. This process not only includes adding, removing and modifying the instructions units in the design phase but also swapping them out on-demand during runtime by partial reconfiguration of the FPGA. Special units can be loaded and unloaded as needed by the algorithm, changing the configuration of the soft-core as needed. The available instruction set not only include

This work has partly been funded by Thüringer Aufbaubank TAB and EFRE under grant 2016 VF 0023.

Digital Object Identifier 10.4316/AECE.2019.03007

elemental calculations but also offer extracting roots or exponential functions, providing fast and clock-accurate processing for sophisticated algorithms. In addition, the soft-core makes use of a 5-staged instruction pipeline and every operator is internally pipelined as well. By this way it is possible to maximize the throughput over parallelization approaches while maintaining the deterministic predictability and the real-time constraints. Theoretically, it is possible to start a new operation at every clock cycle, depending on memory specific dependencies of the operation input and output.

This way a compiler can make or break a soft-core-design as weak compilers can have a several-fold increase in computing times, potentially negating the features of the soft-core design. The challenge in designing a compiler for the suggested soft-core is taking into account the configuration and availability of instruction units as these are designed to be changed in any phase of the system's design. Optimizations have to be adjusted to the configured state of the soft-core and no hard-coded optimization logic can be used for this task. All this aims at maximizing the usage of all internal pipelines in order to minimize the computation time of any given problem.

The compiler presented in this paper is a consistent enhancement of a previous presented version [11] and features a high level of micro-parallelism utilization, drastically reducing execution time for given programs with respect to previous versions. Using a sequential source-code as the input, the compiler is able to generate machine code for an arbitrary number of processor-cores, automatically taking care of inter-core synchronizations and utilization of communication resources while the configuration of instruction units can be set for every individual core. As the multi-core feature is a new feature compared to the previous presented version, it is not the only enhancement. The biggest improvement is the replacement of fixed optimization algorithms in order to compile the source code. With the new approach, it is possible to set the optimization target by the programmer by weighting different parameters of the instructions, customizing the optimization for a given algorithm and soft-core configuration. This high level of flexibility and customizability combined with sophisticated algorithms to increase fine-grain parallelism and thus increased performance amongst a single or a number of codes make this compiler unique for use with the partially reconfigurable nature of the soft-core.

The multi-core feature in embedded systems has many potential approaches to solving the numerous multi-core-related problems like presented in [12-16].

In addition we present a cancellation of the classic variable to memory slot mapping which aims to provide more independent operations, resulting in a higher usage of internal pipelines. This is also improving the optimal memory usage since a fixed amount of resources is always reserved for memory in any FPGA design and can be effectively used with this approach independent of the actual number of used variables in any given problem.

The processor performance can be drastically increased by increasing the frequent usage of pipelines [17].

The achieved speed and pipelining benefits are always dependent on the program to be compiled and its internal dependencies as pipelining utilizes micro-level parallelism in order to keep processor utilization high. Examples can be artificially created for either extreme from no benefit to almost perfect utilization, which is why the authors have decided to stick to real-world algorithms for testing their results.

After pointing out major requirements for this objective and analyzing existing solutions, we develop a specifically designed algorithm combining new approaches. With the presented extensions of the compiler, it will be possible to distribute the instructions of a sequential program to multiple cores while also utilizing micro-parallelism on every core. We will extend the compiler with a mode to support asymmetric cores in a multi-core architecture. This will reduce the needed hardware resources for each soft-core without a significantly loss of performance.

As proof of concept, experiments will show the characteristics of the resulting machine code, e.g. level of parallelism and functional correctness. Those machine codes will be compared with existing solutions.

II. PROBLEM DEFINITION

When dealing with soft-core processors, specialized instruction sets and thus specialized machine code, a compiler is needed to efficiently produce the machine code. Creating a compiler for a specific architecture can incorporate specialized information about the target processor itself. This can include the availability of arithmetic operation units, as well as their processing latency and the number of cores when using multi-core processors. The ability to change these parameters in accordance to changes in hardware configuration is important, especially for soft-core processors, which can be customized for a specific task and where reconfiguration is a standard use-case (e.g. changing the core count and available instructions). Furthermore, the advent of partially reconfigurable FPGAs and thus partially reconfigurable soft-core processors emphasizes the need for agile and fast-changing optimization targets when compiling. While basic knowledge about the processors architecture is required when implementing the compiler, using the configuration of the soft-core as an input will allow better optimization of the machine code for the target while keeping its configuration changeable.

Compiling a sequential program for a multi-core processor not only includes distributing the instructions between the cores but also coordinating synchronization between them. This task involves keeping track of all values of each variable, making sure no core is working on an outdated copy of the variable.

For a more efficient design of multi-core processors, the approach of asymmetric cores can lead to less resource requirement but adds to the complexity of the compiler. When compiling sequential source-code for a multi-core architecture the available processing units have to be taken into account for each core individually when assigning the instructions to the cores.

This may lead to additional inter-core communication, thus emphasizing the need for an efficient handling of the limited communication resources by the compiler while maintaining the hard real-time requirements of the processor without wasteful usage of the limited FPGA resources.

The targeted work domain of the ViSARD soft-core is the real-time domain with computation expensive operations such as double precision floating point operations, e.g. computation of exponential functions. Because the target algorithms not only include the processing of camera data where large streams of data occur and have to be processed, but also relatively simple control algorithms with high precision, it is crucial to provide a maximum level of flexibility in order to achieve the best result for different problems within the targeted domain.

With the ViSARD soft-core it is possible to choose between e.g. the use a single- or multi-core architecture with single or double floating-point precision and to choose between SISD (single instruction single data), SIMD (single instruction multiple data), MISD (multiple instruction single data) or even MIMD (multiple instruction multiple data) realizations. All those cases have to be covered by the compiler.

Therefore, it is necessary to discard fixed optimization patterns in favor of an adjustable approach that is able to target the very specific optimization goal of each implemented problem.

III. RELATED WORK

Over the time, many different compiler scheduling optimization approaches emerged. Many of them consider the "on-the-fly" scheduling mechanisms, like [18-20] and are not suited for a real-time scheduling. In [20] the compiler uses a technique for scheduling threads to execute different regions of a program. Therefore, a control flow graph is determined that contains regions and directed edges between regions, with different execution priorities of each region. The directed edges indicate the direction of program control flow. This method is suited for SIMD architecture, but it is not possible to predict the exact runtime of a given algorithm before execution.

In [21] a compiler is presented together with the "Specialized Multi-Core Soft Microprocessor" it is programmed for. Here, design time instruction scheduling and core assignment is done as well as defining variable placement.

Two models are used, the "Program Memory Init model" and the "Data Memory Init model". Those models are used for organization of all program memories by providing memory images and realizing the initialization of data memory. In section VI we will use this compiler as one basis for comparison. One of the deficits of this approach is the strict value to variable mapping and the deriving code dependencies that reduces the parallelism and therefore increases the execution time.

An essential problem that was already discovered by [22] is the separate consideration and optimization of code selection, register allocation and instruction scheduling. It will occur that decisions made during any one phase place unnecessary constraints on the remaining phases.

Our approach will also combine those three phases and will optimize without any pre-fixed optimization algorithm but with a dynamically customizable optimization for any given problem.

IV. ViSARD TOOL-CHAIN

The assembly language compiler as part of the whole ViSARD tool-chain can be seen in Fig. 1.

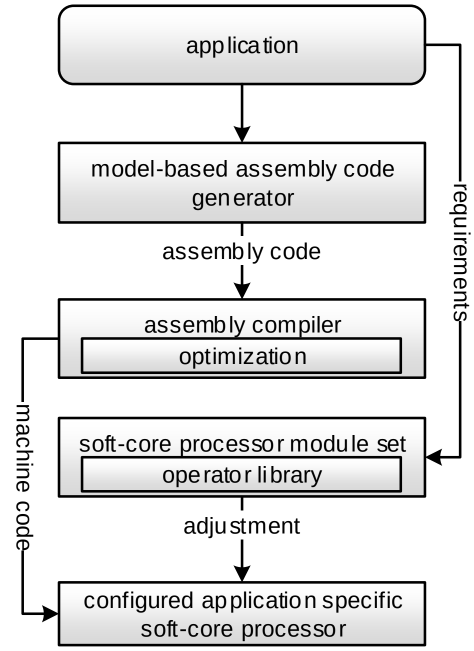
application
model-based assembly code generator
assembly code
assembly compiler
optimization
soft-core processor module set
operator library
adjustment
configured application specific soft-core processor
requirements
machine code

Figure 1. The complete ViSARD Tool-chain, Source: [11]

A. Model-based Assembly Code Generator

In every new project, requirements and constraints have to be defined. With this input it is possible to describe the target application that needs to execute an embedded algorithm with hard real-time characteristics and massive parallelism requirements. The application sets a scenario and the user can model the problem with the first part of the tool-chain, the model-based assembly code generator. This Matlab/Simulink based tool gives the user the ability to realize any given algorithm without the need of special knowledge of any programming language. The user simply drags and drops blocks that realize the needed functionality and connects them as desired. To get a better understanding of how this can look like, Fig. 2 shows a very simple example. This tool can be understood as a data-flow based model oriented way to effectively generate assembly code while maximizing the reuse aspect as it is possible to include previous generated models.

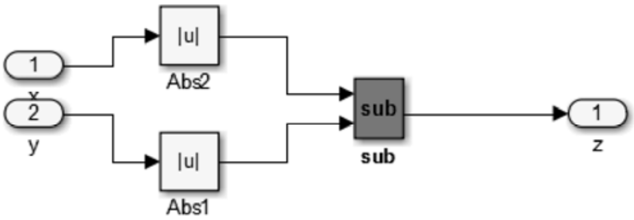

Figure 2. Model-based Assembly Code Generator, Source: [11]

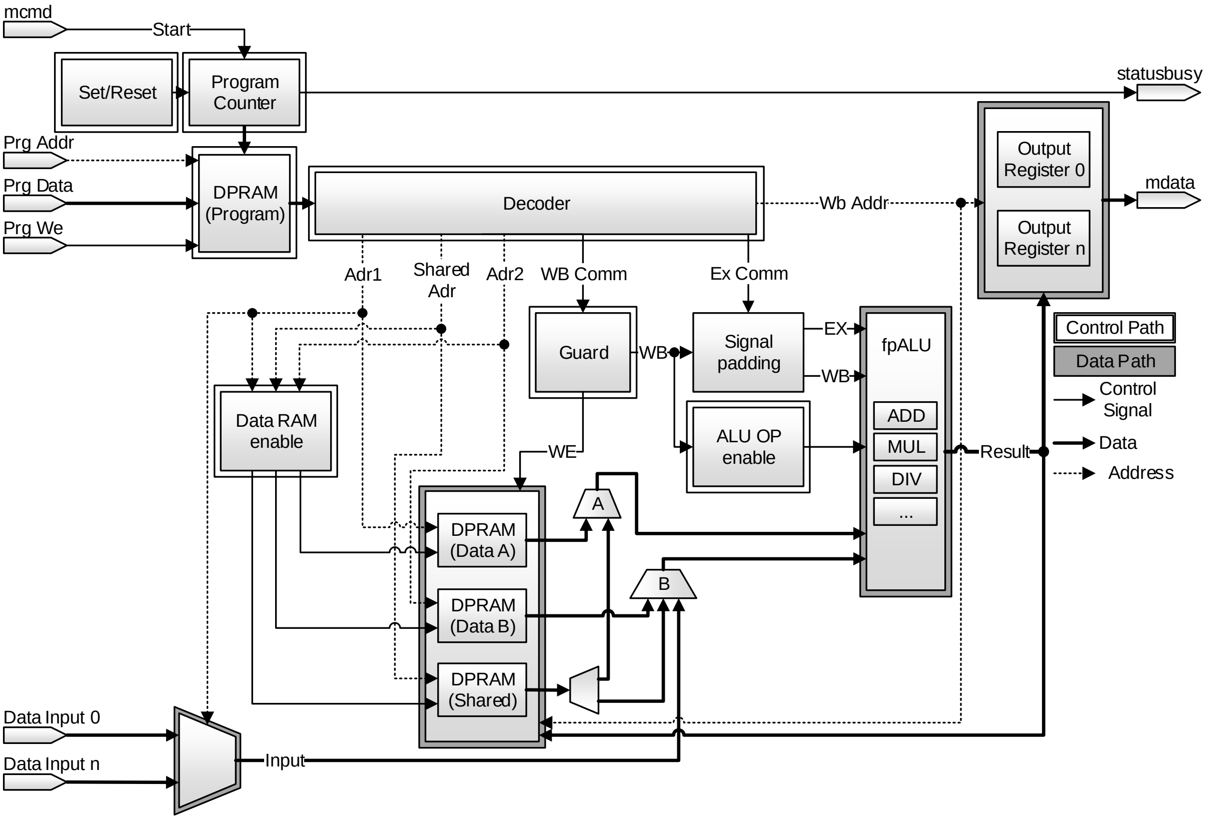

Figure 3. ViSARD Soft-Core Processor Schematic

As soon as the model-based algorithm is finished, it is possible to automatically generate the special assembly code needed to run the ViSARD soft-core.

In addition, different optimizations can be used to optimize the graph generated by the model (or even the model itself) by replacing blocks with faster equivalent logic, or remove dead parts in the model, resulting in a shorter assembly code. In addition, it is possible to minimize the usage of variables of the generated assembly code.

Of course it is also possible (even if not recommended) to skip the model-based assembly code generator and manually write the assembly code. After the assembly code is generated, it is then compiled with the tool presented in this paper as the second step of the tool-chain, which will be explained in detail in the later sections.

B. *ViSARD soft-core processor*

The third part of the tool-chain is a general soft-core processor module set. In this set are included many operators and other specializations that can be used. An example of those specializations is the possibility to run the ViSARD in single precision or, if needed, double precision [10].

The basic structure of the ViSARD soft-core processor can be seen in Fig. 3. It shows a single core in a (potentially) multi-core setup with the shared memory needed for communication with other cores. This architecture offers the possibility to be adjusted for any problem with a theoretical infinite number of cores. In this context, it is also possible to realize each core with a different set of operators in the floating-point arithmetic logic unit (ALU).

Even the adjustment of any ALU during runtime is possible, but the compiler needs to consider such a reconfiguration. This will be explained in detail in section VI.

As can be seen in Fig. 3, each core consists of an I/O-interface with data input and output, and different ports for controlling. Furthermore, it contains different cache blocks for storing the machine code as well as the needed values of the variables. As can be seen, there are three data caches. Data A and data B are needed to read up to two operands per cycle, so it is possible to start a new operation at every clock cycle. In addition, it is possible to change one operand per cycle with either an external data input or a value from a shared resource from another core. A shared memory architecture for the multi-core is used, comparable with the in [23] presented approach.

In every clock cycle both operands are read by the fully pipelined ALU. That means the ViSARD not only uses a 5-staged pipeline but also every operator (like addition or multiplication) inside the ALU is internally pipelined as well. After the computation of an operation is finished, the resulting value can either be stored inside data cache A and B only, or in the two local and the shared data caches, or can be put into the output register. All the mentioned parts are part of the data path, which is marked gray in the figure.

The control path, marked white-double-boxed in the figure, reads the next command from the program cache, decodes it and send the addresses, the write-back command and the execution command to the respective modules. The task of the "data ram enable" module is to activate the needed cache, or to deactivate it if it is not needed in the current clock cycle in order to reduce the power consumption. The "ALU op enable" module has the same task but for the operator modules inside the ALU. If a module is not needed for the current operation and there are no ongoing computations inside the internal operator pipeline, the operator module can be deactivated as well in order to further reduce the power consumption.

As can be seen in Fig. 1, there is a whole set instruction units for the ViSARD. This means the ViSARD is configurable for the special requirements that derive from a new application. It is possible to change, add or remove any operator in the ALU of each core. Furthermore, it is also possible to add or remove Cores, which means the ViSARD can be used as a single core or a multi-core processor, with no theoretical limitation on the number of cores. Also it is possible to use the ViSARD as a SISD (single instruction single data), SIMD or MIMD (multiple instruction multiple data) architecture. The SIMD version is called vector-ViSARD and provides benefits especially when computing hard real-time image processing algorithms.

It is also possible to reconfigure parts of the processor or even an entire processors as parts of SIMD or MIMD architectures during runtime. This can be interesting when the application needs different algorithms that change over time and have different constraints for the processing. Of course the compiler needs to consider such a case and this will be explained in detail in section VI.

According to the requirements derived from the application, the soft-core is adjusted and specialized. After this step, the soft-core gets the machine code that is needed to run the application on it. The machine code has two independent parts:

- Storage part
- Algorithm part

The storage part tells the soft-core processor exactly where to store variables. The algorithm part is the logical part that tells the soft-core processor how to manipulate those variables, e.g. do any kind of operation on them. The structure of the resulting machine code of the algorithm part can be seen in the following figure:

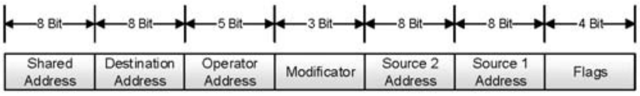

Figure 4. Machine Code Structure

There is a general address width of currently 8 Bit but adaption to specific problems is possible. The shared address defines the memory slot in the shared memory, in case a multi-core version of the ViSARD is used.

One of the novel approaches of the ViSARD architecture is the possibility to parallel write shared and (core-)local memory. With this approach it is possible to improve the CPI in case a multi-core configuration is used.

Currently it is possible to generate ViSARD processors with up to ten independent core modules but experiments have shown that more than four cores result in a decrease of overall processing speed because of the increasing multi-core overhead. This result is only a reflection of the algorithms tested by the authors and programs with less scheduling dependencies may scale better on a higher core count processor.

The destination address defines the local memory slot for the current operation result to be saved. The operator address and the modificator are defining the processing element (PE) that computes the current operation with the two variables defined as source address one and two.

The last 4 Bits are reserved for Flags that are manipulating the data path. If the application needs more space for storing variables, it can be adjusted. This would result in a larger bit representation of the respective address slots and can be handled by the compiler as well.

V. ASSEMBLY CODE DEPENDENCIES

The assembly language for the ViSARD architecture has been designed to realize sequential hard real-time algorithms, which can be verified for correctness without the complexity of multi-core communication. While the programmer does not need to take any form of parallelism into account while implementing their program, a fair amount of independent and thus parallelizable instructions might exist in the resulting source code. Analyzing the assembly code dependencies to parallelize the execution has been done before in order to utilize microparallelism on the ViSARD [11].

This approach of analyzing sequential source code can also be expanded to support multi-core parallelism (while maintaining micorparallelism usage on each core). Offloading the complexity of multi-core parallelism into the compiler not only has benefits but also presents a number of difficulties. Such a compiler enables a given (sequential) program to be re-compiled for an arbitrary number of cores and thus providing easier scalability, especially in late design phases. However, the quality of the result has to be critically evaluated because the compiler has no information about the high-level design of the algorithm, which could be used by a programmer to parallelize streams of instructions. In order to archive acceptable parallelism, sophisticated scheduling techniques are needed that might need to be adjusted according to a specific (soft- or hardware) design.

Traditionally directional graphs are used to analyze these dependencies [11].

To get a better understanding, a short explanation of the used assembly code style is necessary. Every command is built up according to the following scheme:

TABLE I. MNEMONIC TABLE WITH EXAMPLES

Operand modificator	**DPRAM B or IN-MUX**	**DPRAM A**	**Result Address**
Mnemonic	**Op1**	**Op2**	**Op3**
In	0	?	Var1
Out	Var1	?	0
Add	Var1	Var2	Var3
Mul	Var1	Var1	Var1
Sqrt	Var1	?	Var2

As can be seen in Table 1, every command consists of 4 parts: a modification operand, telling to the ALU what operator is needed, the name of the first operand or the address of the data input, an optional second operand and the address of the variable where the result will be stored. The "?" is a placeholder in case no second operand is needed.

When using a multi-core architecture it is possible to load/store values from a shared DPRAM but it is not possible for the developer to explicitly define the access to the shared cache since the compiler determines if and when a value needs to be stored/loaded from the shared cache.

One of the main problems that a compiler has to solve is to maximize the micro parallelism, increasing performance by utilizing the pipelining possibilities as much as possible. The example in Table 1 shows that every operation depends on the "In" operation. So in this example it would not be possible to schedule any operation before the "In" operation is finished. This case can occur very often in large algorithms. In this paper, we will provide a solution on maximizing this micro parallelism by minimizing or even eliminating this problem.

VI. THE OPTIMIZING ASSEMBLY COMPILER

The compiler discussed is based on the theoretical results of [11] and elaborates on the object-oriented approach especially with respect to variable referencing and handling while adding support for a partially reconfigurable multi-core architecture. In addition, this new approach advances the fixed optimization approaches by replacing them with a flexible optimization approach and aims to maximize the pre-reserved memory space utilization.

A. Features and characteristics

The compiler is designed and implemented to address the problems and needs discussed in earlier chapters. The compiler offers a configuration file to set the available cores and the available instructions on each individual core. The performance of an operation can also be set on a per core basis so one core may have a fast multiplication unit while other cores have slower ones or do not support multiplications at all. These information's will be used when compiling and optimizing a given program without the need for hard-coded knowledge about the cores or instructions. Thus any given program can be re-optimized to a new soft-core configuration by changing the configuration file and recompiling, not only enabling better parallelization of hardware design and software development but also supporting the partially reconfigurable design of the soft-core, offering a wide range of different hardware configurations to be available.

One of the main differences between the ViSARD and general soft-core processors is the hard real-time capability.

In order to ensure this characteristic, some adjustments were needed both in the assembly code and in the architecture. The adjustments made to the assembly code directly affects the compiler and will therefore be discussed:

The use of (conditional) jump instructions in the assembly code is prohibited and thus not supported by the processor. So also the compiler will not allow any conditional jump instructions as inputs. However, this limitation is an insignificant disadvantage in the addressed domain, since the tasks realized in this domain are fixed at design time and mechanisms are implemented, to replace the corresponding jumps without showing a negative effect.

Because all algorithms are fixed at design time, it is possible to unroll any loop that otherwise would have to be defined by a (conditional) jump instruction in the assembly code.

With this adjustment, it is possible to fully analyze the resulting machine code and with it the clock accurate behavior of the processor during design time and therefore guarantee that the hard real-time constraints are met.

In order to prevent too long machine code, resulting by unrolling every loop, it is possible to realize a hardware-controlled loop. This hardware loops are controlled by the hardware (the Set/Reset Module, see Fig.3). Basically it is possible to simply not unroll the loop and give the processor the information's where (line of code) the loop starts, how much clock cycles it takes and how many iterations of the loop are needed. During run-time the hardware will then use those information and realize a loop, the so called hardware loop.

The compiler will ensure that those parts of the machine code that will be used as hardware loops are encapsulated from the rest of the machine code to prevent unpredictable behaviour.

The optimization of the compiler will automatically parallelize given source-code on multiple processor cores and handle all synchronizations and data exchange between the cores.

Using complex variable optimizations, enabled by the object oriented approach of the compiler, code dependencies can be resolved at the cost of a slight increase of memory usage, efficiently shortening execution times on single- and multi-core processors. This will not only decrease the size of the programs binary but also reduce the time to load the execution binary into a processor core after a runtime-reconfiguration which is critical for reducing the time needed for the reconfiguration process.

By using a user-configurable "penalty expression" for determining the dynamic scheduling priorities, no hard-coded scheduling algorithms are needed. All available metrics for optimization can be weighted and combined by the user as needed, in order to optimize the scheduling to a given program and hardware design.

B. Implementation overview

Using an object-oriented programming language to implement the compiler enables easy implementation of traditionally complex tasks. By having each instruction represented by an instance of an "Instruction-Object" the logic for testing execute-ability can be handled by each instruction internally, knowing its own dependencies and requirements. This not only includes the variables needed as inputs to the instruction but also the required execution unit that has to be present on a core to be able to execute this instruction.

Fig. 5 illustrates the execute-ability check of an instruction. The instruction object will call the functions of the variables to check their presence and the core to check for the required execution unit in order to combine these information and return the result to the scheduler.

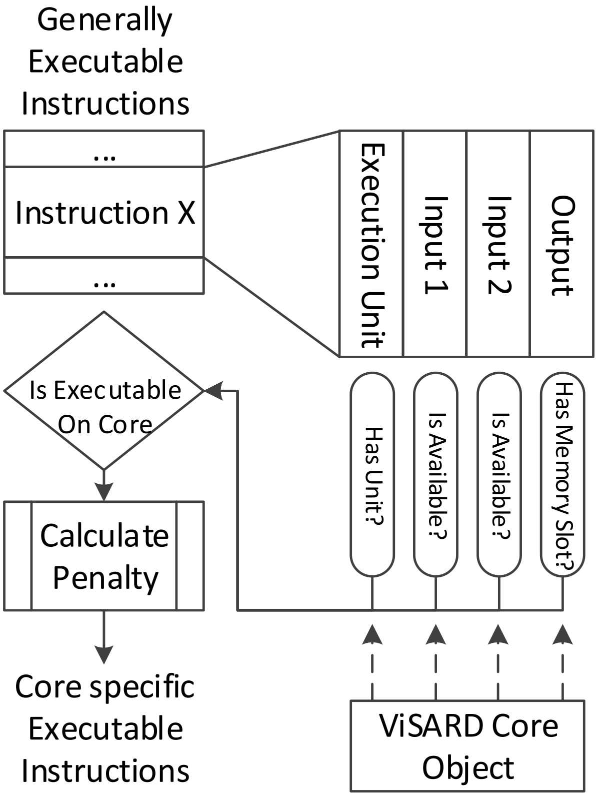

Figure 5. Implementation Overview

As can be seen, it will be checked if the specific core has the needed execution unit for executing the current instruction. If yes, it will check for all needed input operands and for a free slot on the memory of that core. If all conditions are true, it will schedule the instruction on that core if the calculated penalty has the best result on this core.

With this realization, it is also possible to realize the support for the partially reconfigurable multi-core architecture. The compiler need the information at which clock cycle the partial reconfiguration begins, how long does it take to reconfigure the defined part of the soft-core (e.g. the exact core of the multi-core architecture to be reconfigured) and what execution units are exchanged during the reconfiguration.

Because of the hard real-time constraint, the full analyzability at design time is given and furthermore all mentioned needed information's are available to the compiler. So the compiler can set the availability of all execution units of the affected core(s) to false at this time frame, resulting in no computation on this core during the reconfiguration period.

C. Handling of variables

In a fixed variables-handling approach, each variable is bound to a specific memory address for the entire runtime of the program. This simple approach leads to various problems in regards to efficiency and scalability. For example a specific variable might be used only once or twice within the program. This leads to a mostly unused memory cell and results in an inefficient memory usage. Another scenario is that multiple operations are referring to the same variable. The emerging dependencies are preventing the compiler from increasing the processing speed by reducing pipelining possibilities.

The concept of representing instructions as instances of an object can be applied to variables as well with each variable being represented as an instance of a "Variable-Object". Furthermore, this approach can be expanded to each individual value of a variable being represented as an individual instance of a "Variable-Value-Object". For this approach, the Variables will become Factory-Objects, in charge of producing the instances for the individual Values.

An example of the variable value handling can be seen in Fig. 6. In this example, three variables are used in different operations. The summation uses *A* and *B*, the division uses the result of the summation, stored in *B* and the original value of *A*, and the multiplication uses the original value of *B* and *C*. As can be seen there is a problematic dependency chain between those three operations as with the traditional approach, the multiplication has to be started before the summation overwrites the original value of variable *B*, and the division relies on the result of the summation before it can be started.

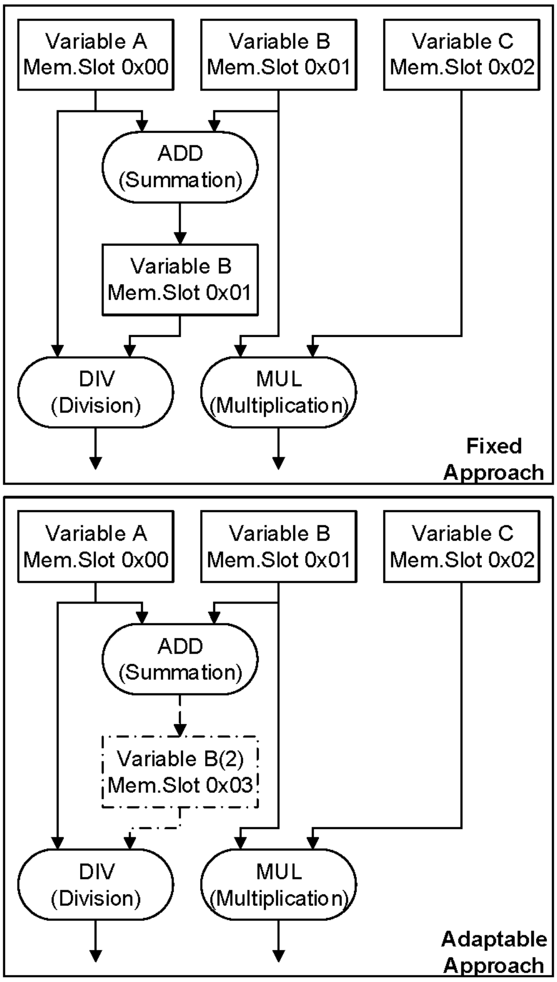

Figure 6. Variable value example

This can lead to an inefficient pipeline usage as it forces the compiler to find a slot where both operations *ADD* and *MUL* can be started right after one another or to even force the compiler to start the multiplication before the summation. This would be inefficient since the summation would be started later and therefore finish later, resulting in an increased waiting time for the division to be started and increase the overall computation time of the given algorithm.

With this adaptable approach, it is possible to simply store the new value of *B* in another memory slot (in this example slot *0x03*). Now both operations (*ADD* and *MUL*) are completely independent of each other, resulting in the freedom of scheduling the operations to whatever time-slot fits best for maximizing the pipeline utilization and minimizing the overall computation time. This approach removes pseudo data-dependencies like write after read and writes after write dependencies, and realizes a register renaming approach. [24]

Every time an operation uses a variable as an input, the resulting read access will be registered in the value-object. Likewise, the write accesses are registered in every value-object. This way each value keeps track of when it was produced (written) and consumed (read). The information includes which core produced the value and if it has been written to the shared cache or synchronized to another core's local cache.

While this adds a layer of complexity in the compiler's handling of the variables, it will provide more accurate handling of the dependencies of the instructions and improve parallelism with the instructions referencing a specific instance of the value of a variable.

With each value of each variable being individually tracked and registering when this value is being produced and when it is being consumed for the last time, it is possible to use an individual cache-address for each of these values.

As a result, the different values of a variable are not tied to the same memory address; it is even possible to overwrite old variables with other variables in order to maximize the memory utilization. This leads to two possible benefits, depending on the nature of the input of the compiler.

If a programmer declares a large number of variables with each variable being used only a few times in local contexts, the different variables may be moved to the same physical cache address to be able to keep the cache size small. The compiler will ensure the usage of the cache cells will not be overlapping and variable integrity is kept.

Tracking individual values of the variable across different cache cells (and different cores) is important for multi-core processors to be able to minimize synchronization between the cores but can also be beneficial for a single-core processor. By allowing for multiple values of the same variable to be stored in different cache cells and thus being available at the same time, parallelization of programs with highly dependent instructions can be greatly improved. Having different values of the same variable on different cache cells will inevitably increase the required number of cache cells. This is not a disadvantage since the soft-core itself reserves a minimum number of FPGA resources for storing variables and therefore it only leads to a better utilization of anyway allocated FPGA resources.

D. Scheduling by using configurable penalties

The result of the automatic parallelization will greatly depend on the scheduling-algorithm used, as it will determine the time and the core used to execute every individual instruction. For multi-core processors the limited communication resources between the cores have to be taken into account when scheduling an instruction. In the special case of asymmetric multi-cores, another layer of complexity is added, as not every core is physically able to execute every instruction.

The approach of [11] so far was to statically measure the relation between the time and distance (time distance ratio *TDR*) from an operation to its successor operations as shortly described below. Therefore, the operations were internally described as nodes of a directed dependency graph where the *TDR* of every operation *i* was computed as follows:

$$TDR_i = \sum_{j=level_i}^{level_k} (averageDelay_j) - delay_i \tag{1}$$

where $level_k$ represents the level of the successor node of *i*. The $averageDelay_j$ matches the equation:

$$averageDelay_j = \frac{\sum_{m=0}^{cmdNumber} levelDelay_m}{nodesPerLevel_i} \tag{2}$$

where

$$levelDelay_m = \begin{cases} delay_i \mid m \in level_i, \\ 0 \mid m \notin level_i. \end{cases} \tag{3}$$

where $nodesPerLevel_i$ represents the number of nodes that $level_i$ have and *cmdNumber* is the total number of all nodes in the graph.

As can be seen this approach is very complex in understanding and not very flexible, as it cannot be customized for a specific problem with special characteristics.

To be able to provide the best-suited scheduling algorithm for each algorithm and hardware-architecture combination, a combination of multiple metrics is now used for scheduling. In contrast to a fixed scheduling algorithm of the previous version of the compiler presented in [11], the combination and weight of the individual metrics can be optimized for a specific problem. The benefit of using an “object-oriented”-like approach for the compiler is key to implementing this functionality. The user-defined scheduling penalty-expression can be parsed and stored in a penalty calculation object at the beginning of the compilation which is then passed down to the instructions by property injection.

This way each instruction can calculate its own penalty for execution on a given core, keeping information local to the objects they belong to. These metrics include simple expressions such as the execution time of an instruction, which can be used for example for an implementation of an SJF (Shortest Job First) or LJF (Longest Job First) Algorithm.

This is done by either putting a positive or a negative penalty on the execution time of the instruction.

Another example is illustrated in Fig. 7. This expression would result in an optimization algorithm that prioritizes all instructions with results that are required often, meaning that instructions that are used very frequently in the given assembly code would be scheduled as soon as possible.

```
<Penalty>
 <Name>RROmax</Name>
 <Expression>Instruction.Delay
 −100∗Operand3.ReadOperationsTotal</Expression>
</Penalty>
```

Figure 7. Example Penalty PROmax

The availability of the input variable values can also be used as a metric for scheduling. These values can either be available in the local cache of a core, available in the shared cache or available on another core's local cache. The latter case is the least preferable as it would require not only a read but also a write access to the shared cache, potentially blocking the communication resources for other cores.

Another example for a scheduling-metric is the amount of instructions that depend on an instruction. This enables the priorization of calculation of values that are required by a large number of upcoming operations.

The compiler will compute a penalty for every theoretically startable assembly instruction at every clock cycle. The scheduling now depends completely on those metrics and the penalty computed with it. This penalty calculation replaces the previous complex and not adaptable optimization algorithms.

In the following, all usable expressions are explained. The values of the expressions define their weighting in the penalty, e.g. zero means that the expression is ignored:

- Instruction.Delay: (Value 0..n) set importance of computation time of the instructions.
- Operand.Read: (1..n) set importance of the number of instructions that needs this specific operand as input. Can be adjusted for both input operands separately.
- Operand3.ReadOperationsTotal: (0..n) set importance of the number of following instructions, that need the result of current instruction as input
- Operand.Is.Variable: (0..1) checks if Operand is a variable (also adjustable for both operands separately) .
- Operand.Is.Constant: (0..1) checks if Operand is a constant (also adjustable for both operands separately) .
- Operand.Bypass: (0..1) checks if input of the current instruction is available in the bypass (also adjustable for both operands separately).
- Operand.Local: (0..1) checks if input of the current instruction is available on the local core (also adjustable for both operands separately). Only useable for multi-core architectures.
- Operand.Shared: (0..1) checks if input of the current instruction is available on the shared memory (also adjustable for both operands separately). Only useable for multi-core architectures.
- Operand.Requestable: (0..1) checks if input of the current instruction is available only on other cores (also adjustable for both operands separately). Only useable for multi-core architectures.

Overall, there are 16 penalty expressions that the programmer can choose from, combine them in every thinkable relation to each other, and weight each expression exactly with the value best suited for the special problem. In addition, the authors offer pre-defined penalty sets as example sets to use without the need for every programmer to work into every possible penalty expression.

However, if needed, the programmer can create combinations of any length of all possible penalty expressions, customizing any thinkable optimization algorithm for any given scheduling problem. The ability to have scheduling that is customizable by the user allows for a great flexibility in optimization targets and eliminates possible negative effects of fixed optimization approaches for specific problems with special characteristics. By weighting the available metrics in any desired combination and being able to (re-) configure the soft-core to change its feature and performance set, the development of the system remains flexible even after the source code for its algorithm has been written and verified.

VII. EXPERIMENTAL RESULTS

To test the new features and show the difference in quality compared to the previous published optimization approaches, multiple experiments were executed. For once, a benchmarking tool generated thousands of assembly codes with different characteristics, e.g. with a set level of dependence in between the lines of code with rising code lengths from 1000 up to 10.000 lines of code, in steps of 250. To get meaningful results, the test characteristics went from zero data dependency up to data dependencies of 100 (in steps of 10), with 100 test runs for each dependency degree and each different code length, resulting in a total amount of 40,700 automatically tested (and pseudo random generated) assembly codes.

However, there was also various real world data processing algorithms tested and for this paper we picked one to explain in more detail and another four to show as a short summary. The chosen real world algorithm that will be explained in detail is a data processing algorithm that analyzes huge amount of image data streams in real-time. Those real world experiments are important as artificial testing programs can be created to meet either of the extremes from perfect pipelining improvements to no benefits from the newly introduced optimizations.

The use of a pre-existing algorithm that is being used in real applications will produce results that will set expectations for more real world usage.

It should be mentioned that this compiler is designed to solve large and complex problems. For very short algorithms it would be possible to just brute-force an optimal solution and therefore it would be no need for a compiler with such complex optimization approaches as presented.

The one real world problem presented here in detail is the "white light interferometry" (WLI) algorithm. WLI is a method for obtaining 3D-topology information’s by capturing multiple images of the same surface while increasing the distance to the surface. This creates an image stack from which 3D-information will be computed.

As can be seen in Fig. 8, the algorithm shows a tilted sample surface with correlograms emerging in z-direction for three selected surface points. According to the elevation difference of the surface points also the corresponding correlograms will manifest in different z intervals for each surface point.

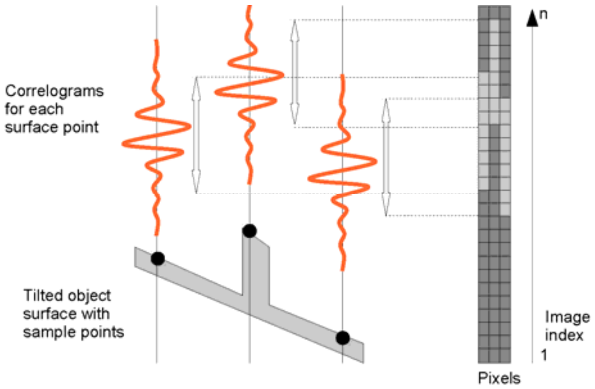
Correlograms for each surface point
Tilted object surface with sample points
n
Image index
1
Pixels

Figure 8. WLI Correlograms in the Captured Image Stack, Source: [25]

The amount of data streams that must be handled are images of 1024x1024 pixels with a 8-bit gray scale per image, with 500 fps, resulting in a data stream of nearly 525 MiB per second and a total image stack of up to 70,000 images for each calculation.

As the image capturing processes, fixed intervals with maximum changes of intensity will occur. With this, the topology of the surface can be derived by computing the center of the correlogram in relation to the image index. The relevant information cannot be computed from one pixel frame, but needs a pixel volume. [25]

The post processing of the WLI problem will determine the surface topology from the pixel stacks and produce a high-resolution surface map. The task is to realize an assembly program that solves the complete post-processing under hard real-time conditions. The ViSARD is able to solve this problem, if the converted assembly code provides enough parallelism.

In order to test the machine code that was produced by the compiler from each real world experiment, we used an experimental setup. Here the Xilinx Zynq 7020 (XC7Z020) [26] FPGA was used via a hardware-in-the-loop setup. On this FPGA we realized a single core ViSARD soft-core processor that runs all the machine code with pre-defined input values and returns the results to the computer. There, the results were checked for any computation errors to ensure that the compiler produced correct working machine code.

As mentioned in section IV B. the compiler always computes two output files: the file with the resulting schedule of the algorithm and a separate file with the storage instructions for the variables. The storage file will be used to compare the achieved memory utilization with a (for this test) fixed reserved memory space with a (current) maximum of 255 variables. The scheduling file will be used to compare the actual computation time the soft-core would need for the execution of every compiled assembly code.

The experiments have shown that 255 variables are enough for even very complex algorithms like WLI.

Because of reasons of clarity, we will only show the results of the best optimization algorithm of the previous compiler as base for comparison with the adaptable approaches.

TABLE II. AVERAGED RESULTS

	Execution Time	**Memory Utilization**	**Processor Utilization**
Compiler from [21]	24.18 %	22.75 %	23.74 %
Previous Version of the Compiler with *TDR* [11]	42.3 %	22.75 %	13.57 %
No Optimization	100 %	22.75 %	5.74 %
Pipeline Optimization Only	32.82 %	22.75 %	17.49 %
Pipeline & Variable Optimization	8.91 %	60.39 %	64.43 %

The comparison value of the execution time is always the result of the compiler with every optimization disabled set to 100%. The memory utilization is the difference between the actual needed memory and the theoretical available memory when reserving enough space for 255 variables. As can be seen, with only the pipeline optimization with the help of adjustable penalty-algorithms, it was already possible to reduce the needed execution time by 9.48%, compared to the previous version of the compiler. Still, the compiler from [21] achieved a better result because of the used bypassing of operation results that enables a faster access to the new values of the variables. However, with the help of the adaptable approach, the variable optimization approach and the penalty optimization, it was possible to reduce the overall execution time to 8.91% and achieve a 15.27% better result, compared to the compiler from [21] and a 33.39% better result, compared to the old version of the compiler. As can be seen, the memory utilization of this optimization approach is 60.39%, which means nearly 40% of the memory is still unused, but the overall utilization of reserved memory is far better than any other result. All other compiler and optimization approaches have the assembly code specific pre-defined data dependencies that prevent a better reduction of the execution time. Because of this, all the memory utilization values are 22.75%; this is the averaged number of variables that the assembly codes used.

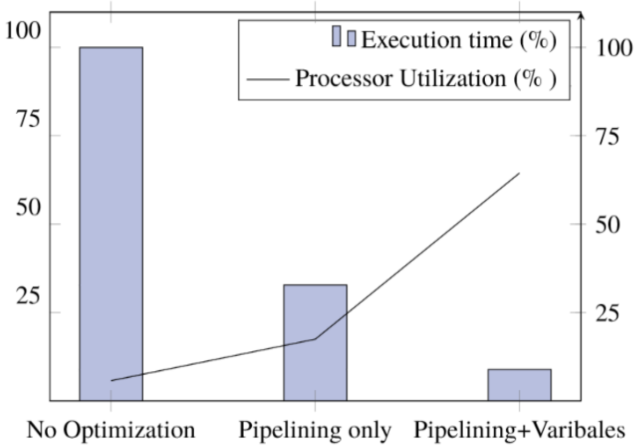

Figure 9. Averaged Results of the new Compiler

Fig. 9 shows the experiment results from the new compiler presented in this paper. As the algorithms used for testing are generated in a memory-efficient way, the memory optimization did not find options to reduce memory utilization by combining multiple variables in a cache cell while on the other hand more cache cells are utilized for resolving code dependencies. Using the newly presented memory and pipeline optimizations execution time could be cut by 91.09%, resulting in a processor utilization of 64.43%.

That means only 35.57% of the processor clock cycles are not used for starting new operations and only this time is spent waiting for the execution units' latencies.

To be able to evaluate the quality of the realized compiler even more, more real world problems were implemented and experiments were carried out. A short summary of some of the further implemented algorithms as well as the experimental results will be given now:

- “Ellipse” is an algorithm from our research projects that performs ellipse-shaped regression and computational correction with signals from incremental sensors (see [27]). It includes regression using the ‘recursive least squares’ method, parameter calculation, and signal correction.
- “FIR64” is an eight order finite impulse response filter with 64 coefficients (see [28]).
- “Kalman” is a modified Kalman filter with four inputs for use as a state estimator in a non-trivial closed-loop control algorithm. The time consuming matrix operations have been optimized with respect to weakly occupied matrices (e.g. triangular matrices), see [29].
- “6-Axes” is a closed-loop control algorithm for three-dimensional motion control in a mechanical system, featuring three translative axes and three rotational axes. This algorithm includes six PID controllers, three Kalman filters as mentioned above (state estimators for translative axes), and further calculations (see [30], [31]).

For those four experiments, only the results of the current version of the compiler will be given, because the results do not vary from the previous experiments. Those four algorithms were created using the “model-based assembly code generator” (see Fig. 1).

Table III. Results (Ellipse, FIR64, Kalman, 6-Axes)

	Execution Time	**Memory Utilization**	**Processor Utilization**
Ellipse (no Optimization)	100 %	29.80 %	9.02 %
Ellipse (Pipeline only)	39.02 %	29.80 %	23.11 %
Ellipse (Pipeline & Variable)	31.06 %	37.65 %	29.03 %
FIR64 (no Optimization)	100 %	28.63 %	17.48 %
FIR64 (Pipeline only)	71.33 %	28.63 %	25.51 %
FIR64 (Pipeline & Variable)	35.43 %	28.24 %	49.34 %
Kalman (no Optimization)	100 %	29.80 %	7.29 %
Kalman (Pipeline only)	20.42 %	29.80 %	35.71 %
Kalman (Pipeline & Variable)	12.22 %	26.63 %	59.65 %
6-Axes (no Optimization)	100 %	43.53 %	9.09 %
6-Axes (Pipeline only)	21.85 %	43.53 %	41.61 %
6-Axes (Pipeline & Variable)	10.12 %	43.53 %	89.81 %

As can be seen in Table III., the experiments confirm the previous results: the overall processor utilization can be greatly increased by using the penalty based optimization approach presented in this paper. An additional improvement can be achieved when activating the variable (memory) optimization. For example the FIR-filter, with both optimizations enabled, a processor utilization gain from 7.29 % over 25.51 % (for pipeline optimization only) up to 49.34 % can be achieved.

In contrasts to the previous results, the overall memory utilization does not necessary increase when enabling the memory optimization algorithm. An example for this is the Kalman filter. Here, the total memory utilization even decreases from 29.80 % to 26.63 % with simultaneous increasing in processor utilization from 35.71 % to 59.65 % as the variable optimization is enabled. This is because this optimization will re-use memory slots from variables as soon as they won’t be read (and therefore needed) any more by any (following) assembly instruction in addition to have (potentially) multiple instances of one variable at the same time. So if many variables are rarely used it is even possible to reduce the memory consumption while maximizing the processor utilization.

VIII. Conclusion and Future Work

As presented in this paper, it was possible to develop a new optimizing assembly code compiler that offers a nearly unlimited amount of scheduling optimization algorithms, customizable to every specific field of application, with the help of 16 arbitrarily combinable and factorizable penalty expressions. In addition, a new variable to memory cell relationship was introduced. The assumption that a variable is always present on the same cache cell is no longer appropriate, as the availability in several cores has to be taken into account. The adaptable approach allows the compiler automatically to unite multiple variables to one memory slot, or spit a variable to multiple memory slots, dependent on the current data dependency situation. With this approach, it is possible to maximize the processor utilization of the soft-core processor and therefore to minimize the needed execution time of any given problem. Furthermore, the compiler is able to compile any assembly code for a theoretical infinite number of cores in a multi-core application scenario. Various tests have shown the quality improvement of the generated machine code, at it is possible to reduce the needed execution time by over 90%, compared to a not optimized machine code.

In the future, it is planned to expand both the compiler and the soft-core processor with a bypassing functionality, allowing the access to any computation result without the current delay of saving it into a memory slot.

Another possible enhancement would be way to automatically test any given assembly code with different combinations of penalty expressions and automatically compare the scheduling results to find the best-suited optimization algorithm for any given problem.

References

[1] O. Esko, P. Jaaskelainen, P. Huerta, S. Carlos, J. Takala, J.I. Martinez, “Customized exposed datapath soft-core design flow with compiler support”, 2010 International Conference on Field Programmable Logic and Applications (FPL), pp. 217-222, doi:10.1109/FPL.2010.51.

[2] Altera Corporation, “Nios II Processor Overview”, https://www.altera.com/products/processors/overview.html, accessed: Jan 22, 2018.

[3] Andreas Erik Hindborg, Pascal Schleuniger, Nicklas Bo Jense, Maxwell Walter, Laust Brock-Nannestad, Lars Bonnichsen, Christian W. Probst, Sven Karlsson, “Automatic generation of application specific FPGA multicore accelerators”, Signals Systems and Computers 2014 48th Asilomar Conference on, pp. 1440-1444, 2014, doi:10.1109/ACSSC.2014.7094700.

[4] Danko Ivošević, Vlado Sruk, “Unified flow of custom processor design and FPGA implementation”, EUROCON 2013 IEEE, pp. 1721-1727, 2013, doi:10.1109/EUROCON.2013.6625209.

[5] Xilinx Inc., “MicroBlaze Soft Processor Core”, https://www.xilinx.com/products/design-tools/microblaze.html, accessed: Jan 22, 2018.

[6] Gaisler Research, “LEON3 Processor”, http://www.gaisler.com/index.php/ products/processors/leon3, accessed: Jan 22, 2018.

[7] J.G. Tong, I.D. Anderson, M.A. Khalid, “Soft-core processors for embedded systems”, International Conference on Microelectronics, 2006, ICM'06, pp. 170-173, doi:10.1109/ICM.2006.373294.

[8] Andreas Erik Hindborg, Pascal Schleuniger, Nicklas Bo Jensen, Sven Karlsson, “Hardware realization of an FPGA processor — Operating system call offload and experiences”, Design and Architectures for Signal and Image Processing (DASIP) 2014 Conference on, pp. 1-8, 2014, doi:10.1109/DASIP.2014.7115604 .

[9] Oliver Stecklina, Michael Methfessel, “A Tiny Scale VLIW Processor for Real-Time Constrained Embedded Control Tasks”, Digital System Design (DSD) 2014 17th Euromicro Conference on, pp. 559-566, 2014, doi:10.1109/DSD.2014.31.

[10] M. Kirchhoff, W. Fengler, “Realization of an embedded hard realtime softcore processor”, 2014 7th GI Workshop on Autonomous Systems, pp. 33-42.

[11] M. Kirchhoff, N. Kaptsova, D. Streitpferdt, W. Fengler, “Optimizing compiler for a specialized real-time floating point softcore processor”, 2017 8th Annual Conference of Industrial Automation and Electromechanical Engineering, IEMECON, pp. 181-188, doi:10.1109/IEMECON.2017.8079585.

[12] M. Levy, T.M. Conte, “Embedded multicore processors and systems”, IEEE micro, 29(3), pp. 7-9, doi:10.1109/MM.2009.41.

[13] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S.W. Liao, E. Bugnion, M.S. Lam, “Maximizing multiprocessor performance with the SUIF compiler”, Computer, 29(12), pp. 84-89, 1996, doi:10.1109/2.546613.

[14] Jiayin Li, Meikang Qiu, Jianwei Niu, Meiqin Liu, Bin Wang, Jingtong Hu, “Impacts of Inaccurate Information on Resource Allocation for Multi-Core Embedded Systems”, Computer and Information Technology (CIT) 2010 IEEE 10th International Conference on, pp. 2692-2697, 2010, doi:10.1109/CIT.2010.452.

[15] Roman Atachiants, Gavin Doherty, David Gregg, “Parallel Performance Problems on Shared-Memory Multicore Systems: Taxonomy and Observation”, Software Engineering IEEE Transactions on, vol. 42, no. 8, pp. 764-785, 2016, doi:10.1109/TSE.2016.2519346.

[16] Sungju Lee, Eunji Lee, Yongwha Chung, Hyeonjoong Cho, Byoungki Min, “Energy-efficient protection of video surveillance data using multicore-based video sensors”, Digital Content Multimedia Technology and its Applications (IDC) 2010 6th International Conference on, pp. 327-330, 2010.

[17] E. Sprangle, D. Carmean, “Increasing processor performance by implementing deeper pipelines”, ACM SIGARCH Computer Architecture News, Vol. 30, No. 2, pp. 25-34, IEEE Computer Society, doi:10.1109/ISCA.2002.1003559.

[18] T. Hagras, J. Janeček. “A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems”, Parallel Computing, 31(7), pp. 653-670, 2005, doi:10.1109/IPDPS.2004.1303056.

[19] Y. Yan, R. Zheng, “Code Generating Method, Compiler, Scheduling Method, Scheduling Apparatus and Scheduling System”, U.S. Patent Application No. 15,058,610, 2016.

[20] G. Diamos, M Mehrara, “Compiler-controlled region scheduling for SIMD execution of threads”, U.S. Patent No. 9,424,038. Washington, DC: U.S. Patent and Trademark Office, 2016.

[21] B. Däne, A. Pacholik, S. Zschäck, W. Fengler, C. Ament, T. Braune, “Designing a Control Application by Using a Specialized Multi-Core Soft Microprocessor”, IFAC Proceedings Volumes, 46(28), pp. 221-226, 2013, doi:10.3182/20130925-3-CZ-3023.00034.

[22] S. Novack, A. Nicolau, “Mutation scheduling: A unified approach to compiling for fine-grain parallelism”, International Workshop on Languages and Compilers for Parallel Computing, pp. 16-30, Springer Verlag, Berlin, Heidelberg, 1994.

[23] Z. Yu, K. You, R. Xiao, H. Quan, P. Ou, Y. Ying, X. Zeng, “An 800 MHz 320 mW 16-core processor with message-passing and shared-memory inter-core communication mechanisms”, 2012 IEEE International Solid-State Circuits Conference, ISSCC, pp. 64-66, doi:10.1109/ISSCC.2012.6176931.

[24] S. Dezso, “The design space of register renaming rechniques” 2000 IEEE micro 20 (5), pp. 70-83, doi:10.1109/40.877952.

[25] M. Müller, T. Machleidt, W. Fengler, “SoC Design for Complex Standalone Optical Measurement Devices”, 2014 7th GI Workshop on Autonomous Systems, pp. 66-75.

[26] Xilinx Inc., “Zynq-7000 SoC Data Sheet: Overview. DS190 v1.11.1”, https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf, accessed: July 22 2019.

[27] T. Hausotte, B. Percle, U. Gerhardt, D. Dontsov, E. Manske, G. Jäger, “Interference signal demodulation for nanopositioning and nanomeasuring machines”, Measurement Science and Technology, 23(7):074004, 2012, doi:10.1088/0957-0233/23/7/074004.

[28] B. Shenoi, “Introduction to digital signal processing and filter design”, John Wiley & Sons, 2005.

[29] A. Pacholik, J. Klöckner, M.Müller, I. Gushchina, W.Fengler, “LiSARD: LabVIEW integrated softcore architecture for reconfigurable devices”, 2011 International Conference on Reconfigurable Computing and FPGAs (ReConFig '11), pp. 442-447, Cancun, Mexico, 2011, IEEE Computer Society CPS, doi:10.1109/ReConFig.2011.56.

[30] A. Amthor, S. Zschäck, C. Ament, “Position control on nanometer scale based on an adaptive friction compensation scheme”, 2008 34th Annual Conference of IEEE Industrial Electronics, pp. 2568-2573, IEEE, 2008, doi:10.1109/IECON.2008.4758361.

[31] S. Zschäck, J. Klöckner, I. Gushchina, A. Amthor, W.Fengler, “Control of nanopositioning and nanomeasuring machines with a modular FPGA based data processing system”, Mechatronics, 23(3):257–263, 2013, doi:10.1016/j.mechatronics.2012.12.003.