Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2010, Article ID 953693, 15 pages
doi:10.1155/2010/953693

*Research Article*

# Low-Complexity Online Synthesis for AMIDAR Processors

**Stefan Döbrich and Christian Hochberger**

*Chair for Embedded Systems, University of Technology, Nöthnitzer Straße 46, 01187 Dresden, Germany*

Correspondence should be addressed to Stefan Döbrich, stefan.doebrich@tu-dresden.de

Received 4 March 2010; Revised 27 September 2010; Accepted 17 December 2010

Academic Editor: Lionel Torres



Future chip technologies will change the way we deal with hardware design. First of all, logic resources will be available in vast amount. Furthermore, engineering specialized designs for particular applications will no longer be the general approach as the nonrecurring expenses will grow tremendously. Reconfigurable logic has often been promoted as a solution to these problems. Today, it can be found in two varieties: field programmable gate arrays or coarse-grained reconfigurable arrays. Using this type of technology typically requires a lot of expert knowledge, which is not sufficiently available. Thus, we believe that online synthesis that takes place during the execution of an application is one way to broaden the applicability of reconfigurable architectures. In this paper, we show that even a relative simplistic synthesis approach with low computational complexity can have a strong impact on the performance of compute intensive applications.

## 1. Introduction

Following the road of Moore's law, the number of transistors on a chip doubles every 24 months. After being valid for more than 40 years, the end of Moore's law has been forecast many times now. Yet, technological advances have kept the progress intact.

Further shrinking of the feature size of traditionally manufactured chips will give us two challenges. Firstly, exponentially increased mask costs will make it prohibitively expensive to produce small quantities of chips for a particular design. Also, the question comes up: how to make use of the vast amounts of logic resources without building individual chip designs for each application?

MPSoCs currently receive a lot of attention in this area. They can be software programmed, and by the use of large amounts of processing cores, they can make (relatively) efficient use of large quantities of transistors. Yet, their big drawback is that all the software has to be restructured and reprogrammed in order to use many cores. Currently, this still seems to be a major problem.

Reconfigurable logic in different granularities has been proposed to solve both problems [1]. It allows us to build large quantities of chips and yet use them individually. Field programmable gate arrays (FPGAs) are in use for this purpose for more than two decades. Yet, it requires much expert knowledge to implement applications or part of them on an FPGA. Also, reconfiguring FPGAs takes a lot of time due to the large amount of configuration information.

Coarse-Grained Reconfigurable Arrays (CGRAs) try to solve this last problem by working on word level instead of bit level. The amount of configuration information is dramatically reduced, and also the programming of such architectures can be considered more *software style*. The problem with CGRAs is typically the tool situation. Currently available tools require an adaptation of the source code and typically have very high runtime so that they need to be run by experts and only for very few selected applications.

Our approach tries to make the reconfigurable resources available for all applications in the embedded systems domain, particularly mobile devices or networked devices where the requirements change. Thus, synthesis of accelerating circuits takes place during the applications execution as we do not have access to the applications beforehand. No hand crafted adaptation of the source code will be required, although it is clear that manual fine-tuning of the code can lead to better results. The intention of the AMIDAR concept is to provide a general purpose processor

that can automatically take advantage of reconfigurable resources.

In this contribution, we want to show that even a relatively simple approach to online circuit synthesis can achieve substantial application acceleration.

The remainder of this paper is organized as follows. In Section 2, we will give an overview of related work. In Section 3, we will present the model of our processor which allows an easy integration of synthesized functional units at runtime. In Section 4, we will detail how we figure out the performance sensitive parts of the application by means of profiling. Section 5 explains our online synthesis approach. Results for some benchmark applications are presented in Section 6. Finally, we give a short conclusion and an outlook onto future work.

## 2. Related Work

Fine-grained reconfigurable logic for application improvement has been used for more than two decades. Early examples are the CEPRA-1X which was developed to speed up cellular automata simulations. It gained a speedup of more than 1000 compared with state of the art workstations [2]. This level of speedup still persists for many application areas, for example, the BLAST algorithm [3]. Unfortunately, these speedups require highly specialized HW architectures and domain specific modelling languages.

Combining FPGAs with processor cores seems to be a natural idea. Compute intense parts can be realized in the FPGA and the control intense parts can be implemented in the CPU. GARP was one of the first approaches following this scheme [4]. It was accompanied by the synthesizing C compiler NIMBLE [5] that automatically partitions and maps the application.

Static transformation from high level languages like C into fine-grained reconfigurable logic is still the research focus of a number of academic and commercial research groups. Only very few of them support the full programming language [6].

Also, Java as a base language for mapping has been investigated in the past. Customized accelerators to speed up the execution of Java bytecode have been developed [7]. In this case, only a small part of the bytecode execution is implemented in hardware and the main execution is done on a conventional processor. Thus, the effect was very limited.

CGRAs have also been used to speed up applications. They typically depend on compile time analysis and generate a single-datapath configuration for an application beforehand: RaPiD [8], PipeRench [9], Kress-Arrays [10], or the PACT-XPP [11]. In most cases, specialized tool sets and special-purpose design languages had to be employed to gain substantial speedups. Whenever general-purpose languages could be used to program these architectures, the programmer had to restrict himself to a subset of the language and the speedup was very limited.

Efficient static transformation from high level languages into CGRAs is also investigated by several groups. The DRESC [12] tool chain targeting the ADRES [13, 14] architecture is one of the most advanced tools. Yet, it requires hand-written annotations to the source code and in some cases even some hand crafted rewriting of the source code. Also, the compilation times easily get into the range of days.

Several processor concepts provide special features to simplify the adaptation of the processor to the needs of particular applications. The academic approaches like XiRISC/PicoGA [15] or SpartanMC [16] register in this category as well as commercial products like the Stretch processor [17]. Typically, these processors are accompanied by a toolset that statically determines performance critical parts of the application and maps them to the configurable part of the device.

The RISPP architecture [18] lies between static and dynamic approaches. Here, a set of candidate instructions are evaluated at compile time. These candidates are implemented dynamically at runtime by varying sets of so called atoms. Thus, alternative design points are chosen depending on the actual execution characteristics.

Dynamic transformation from software to hardware has been investigated already by other researchers. Warp processors dynamically transform assembly instruction sequences into fine-grained reconfigurable logic [19]. This happens by synthesis of bitstreams for the targeted WARP-FPGA platform. Furthermore, dynamic synthesis of Java bytecode has been evaluated [20]. Nonetheless, this approach is only capable of synthesizing combinational hardware.

Dynamic binary translation can also be seen as a runtime mapping of applications to spatial hardware. This has been explored in the past, for example, in the Transmeta Crusoe processor [21], where x86 instructions were dynamically translated into VLIW code. The main purpose of this approach was to execute legacy x86 code with much better energy efficiency.

Newer approaches start from binary RISC code like MIPS [22] or SimpleScalar [23]. In both cases, sequences of machine instructions are mapped onto an array of ALU elements in order to parallelize the computations. Nonetheless, it must be mentioned that both approaches up to now offer only limited speed ups. Yet, the approach taken in [22] enables the usage of partially defect arrays which makes it an interesting approach for future unreliable devices.

The token distribution principle of AMIDAR processors has some similarities with Transport Triggered Architectures [24]. These processors are sometimes referred to as *move processors*, because they are programmed by specifying the transport (move) of data between different components of the processor. This approach allows a relatively simple introduction of new processor components for specialized applications. TTAs, are customized statically at design time. In AMIDAR processors, the tokens also control the movement of data between components. Yet, in TTAs an application is transformed directly into a set of tokens. Storing the full token set of an application leads to a very high memory overhead and makes an analysis of the executed code extremely difficult. In contrast to this approach, AMIDAR processors compute the token set dynamically.

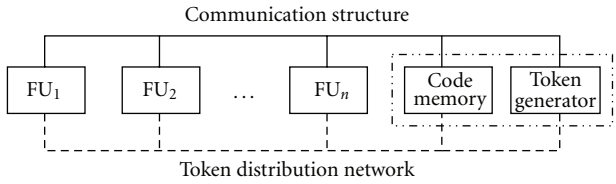


Figure 1: Abstract Model of an AMIDAR processor.

## 3. The AMIDAR Processing Model

In this section, we will give an overview of the AMIDAR processor model. We describe the basic principles of operation. This includes the architecture of an AMIDAR processor in general, as well as specifics of its components. Furthermore, we discuss the applicability of the AMIDAR model to different instruction sets. Afterwards, an overview of an example implementation of an AMIDAR-based Java machine is given. Finally, we discuss several mechanisms of the model, that allow the processor to adapt to the requirements of a given application at runtime.

*3.1. Overview.* An AMIDAR processor consists of three main parts. A set of functional units, a token network, and a communication structure.

Two functional units, which are common to all AMIDAR implementations, are the code memory and the token generator. As its name tells, the code memory holds the applications code. The token generator controls the other components of the processor by means of tokens. Therefore, it translates each instruction into a set of tokens, which are distributed to the functional units over the token distribution network. The tokens tell the functional units what to do with input data and where to send the results. Specific AMIDAR implementations may allow the combination of the code memory and the token generator as a single functional unit. This would allow the utilization of several additional side effects, such as instruction folding. Functional units can have a very wide range of meanings: ALUs, register files, data memory, specialized address calculation units, and so forth. Data is passed between the functional units over the communication structure. This data can have various meanings: program information (instructions), address information, or application data. Figure 1 sketches the abstract structure of an AMIDAR processor.

*3.2. Principle of Operation.* Execution of instructions in AMIDAR processors differs from other execution schemes. Neither microprogramming nor explicit pipelining is used to execute instructions. Instead, instructions are broken down to a set of tokens which are distributed to a set of functional units. These tokens are 5 tuples, where a token is defined as $T = \{\text{UID}, \text{OP}, \text{TAG}, \text{DP}, \text{INC}\}$. It carries the information about the type of operation (OP) that will be executed by the functional unit with the specified id (UID). Furthermore, the version information of the input data (TAG) that will be processed and the destination port of the result (DP) are part of the token. Finally, every token contains a tag increment flag (INC). By default, the result of an operation is tagged equally to the input data. In case the TAG-flag is set, the output tag is increased by one.

The token generator can be built such that every functional unit which will receive a token is able to receive it in one clock cycle. A functional unit begins the execution of a specific token as soon as the data ports receive the data with the corresponding tag. Tokens which do not require input data can be executed immediately. Once the appropriately tagged data is available, the operation starts. Upon completion of an operation, the result is sent to the destination port that was denoted in the token. An instruction is completed, when all the corresponding tokens are executed. To keep the processor executing instructions, one of the tokens must be responsible for sending a new instruction to the token generator.

A more detailed explanation of the model, its application to Java bytecode execution, and its specific features can be found in [25, 26].

*3.3. Applicability.* In general, the presented model can be applied to any kind of instruction set. Therefore, a composition of microinstructions has to be defined for each instruction. Overlapping execution of instructions comes automatically with this model. Thus, it can best be applied if dependencies between consecutive instructions are minimal. The model does not produce good results, if there is a strict order of those microinstructions, since in this case no parallel execution of microinstructions can occur. The great advantage of this model is that the execution of an instruction depends on the token sequence, not on the timing of the functional units. Thus, functional units can be replaced at runtime with other versions of different characterizations. The same holds for the communication structure, which can be adapted to the requirements of the running application. Thus, this model allows us to optimize global goals like performance or energy consumption. Intermediate virtual assembly languages like Java bytecode, LLVM bitcode, or the .NET common intermediate language are good candidates for instruction sets. The range of functional unit implementations and communication structures is especially wide, if the instruction set has a very high abstraction level and/or basic operations are sufficiently complex. Finally, the data-driven approach makes it possible to easily integrate new functional units and create new instructions to use these functional units.

*3.4. Implementation of an AMIDAR-Based Java Processor.* The structure of a sample implementation of an AMIDAR-based Java host-processor is displayed in Figure 2. This section will give a brief description of the processors structure and the functionality of its contained functional units. The central units of the processor are the code memory and the token generator. The code memory holds the applications code. In case of a Java processor, this includes all class files and interfaces, as well as their corresponding constant pools and attributes. The Java runtime model separates local variables and the operand stack from each other. Thus,

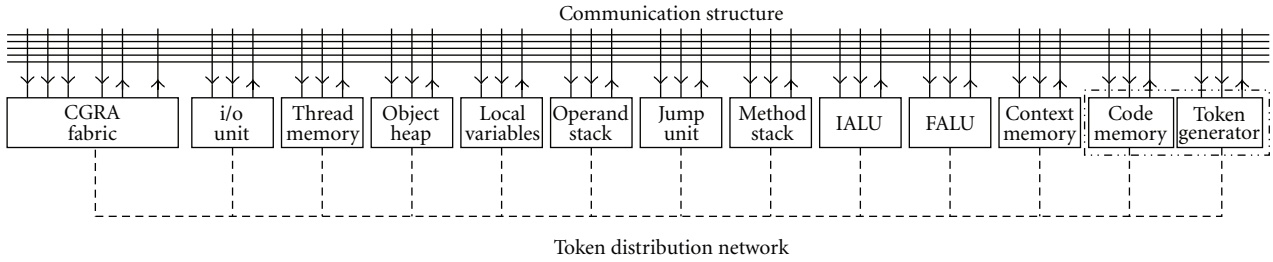


Figure 2: Model of a Java (non-)virtual machine on AMIDAR basis.

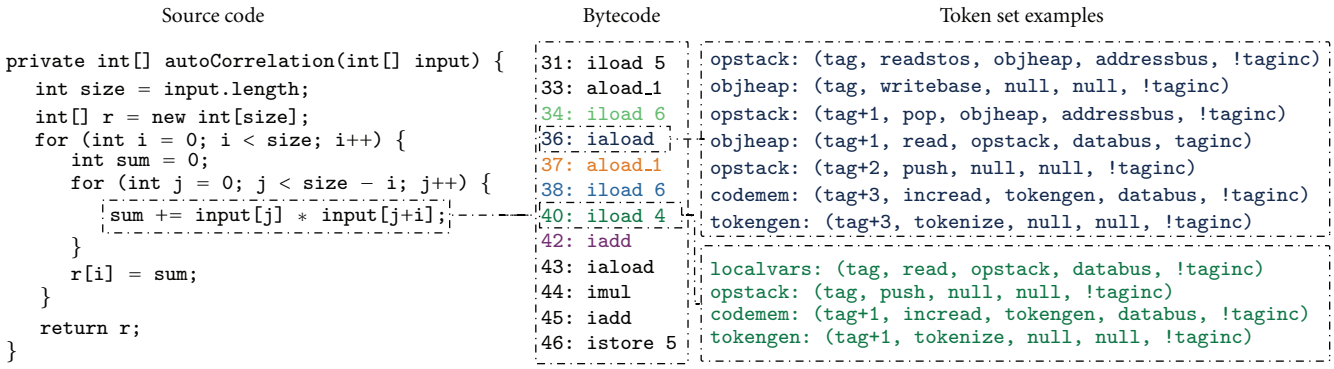


Figure 3: Example source code sequence and the resulting bytecode and exemplified token sequences.

a functional unit that provides the functionality of a stack memory represents the operand stack. Furthermore, an additional functional unit holds all local variables.

A local variable may be of three different types. It may be an array reference type or an object reference type, and furthermore, it may represent a native data type such as `int` or `float`. All native data types are stored directly in the local variable memory, while all reference types point to an object or array located on the heap memory. Thus, the processor contains another memory unit incorporating the so-called object heap. Additionally, the processor contains a method stack. This memory is used to store information about the current program counter and stack frame in case of a method invocation.

In order to process arithmetic operations, the processor will contain at least one ALU functional unit. Nonetheless, it is possible to separate integer and floating point operations into two disjoint functional units, which improves the throughput. Furthermore, the processor contains a jump unit which processes all conditional jumps. Therefore, the condition is evaluated and the resulting jump offset is transferred to the code memory. Additionally, the context memory contains the states of all sleeping threads in the system.

*3.5. Example Token Sequence and Execution Trace.* In order to give a more detailed picture of an actual applications execution on an AMIDAR processor, we have chosen an autocorrelation function as an example. The source code of the autocorrelation function, its resulting bytecode, and sample token sequences for two of its bytecodes are displayed in Figure 3. The `iaload` instruction at program counter 36 is focussed in the further descriptions.

The `iaload` bytecode loads an integer value from an array at the heap and pushes it onto the operand stack. Initially, the arrays address on the heap and the offset of the actual value are positioned at the top of the stack. Firstly, the arrays address is read from the second position of the stack and is sent to the heap where it is written to the base address register. Afterwards, the actual offset is popped of the stack, sent to the heap, and used as address for a read operation. The read value is sent back to the operand stack and pushed on top of the stack.

Figure 4 shows an excerpt of the execution of the autocorrelation function. Each line of the diagram represents the internal state of the displayed functional units in the corresponding clock cycle. Furthermore, all operations that belong to the same instruction are colored identically, which visualizes the overlapping execution of instructions.

*3.6. Adaptivity in the AMIDAR Model.* The AMIDAR model exposes different types of adaptivity. All adaptive operations covered by the model are intended to dynamically respond to the running applications behavior. Therefore, we identified adaptive operations that adapt the communication structure to the actual interaction scheme between functional units. Furthermore, a functional unit may be the bottleneck of the processor. Hence, we included similar adaptive operations for functional units. The following subsections will give an overview of the adaptive operations provided by the

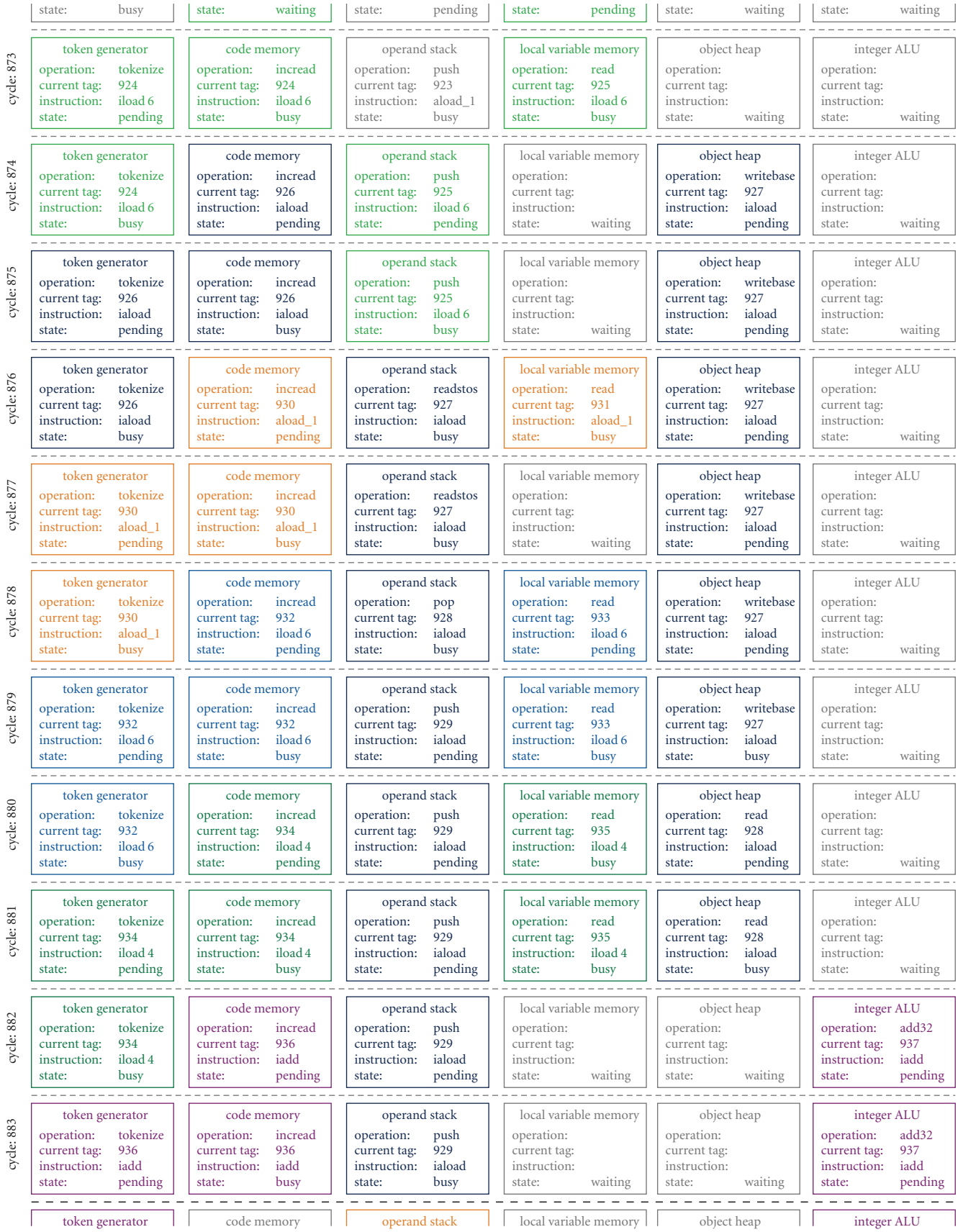


Figure 4: Visualized excerpt of an execution trace of the autocorrelation example.

AMIDAR model. Most of the currently available reconfigurable devices do not fully support the described adaptive operations (e.g., addition or removal of bus structures). Yet, the model itself contains these possibilities, and so may benefit from future hardware designs.

*3.7. Adaptive Communication Structures.* The bus conflicts that occur during the data transports between functional units can be minimized by adapting the communication structure. Therefore, we designed a set of several adaptive operations that may be applied to it.

In order to exchange data between two functional units, both units have to be connected to the same bus structure. Thus, it is possible to connect a functional unit to a bus in case it will send data to/receive data from another functional unit. This may happen if the two functional units do not have a connection yet. Furthermore, the two units may have an interconnection, but the bus arbiter assigned the related bus structure to another sending functional unit. In this case, a new interconnection could be created as well.

As functional units may be connected to a bus structure, they may also be disconnected. For example, this may happen in case many arbitration collisions occur on a specific bus. As a result, one connection may be transferred to another bus structure by disconnecting the participants from one bus and connecting them to a bus structure with sparse capacity.

In case the whole communication structure is heavily utilized and many arbitration collisions occur, it is possible to split a bus structure. Therefore, a new bus structure is added to the processor. One of the connections participating in many collisions is migrated to the new bus. This reduces collisions and improves the applications runtime and the processors throughput. Vice versa, it is possible to fold two bus structures in case they are used rarely. As a special case, a bus may be removed completely from the processor. This operation has a lower complexity than the folding operation, and thus may be used in special cases.

In [27], we have shown how to identify the conflicting bus taps and we have also shown a heuristics to modify the bus structure to minimize the conflicts.

*3.8. Adaptive Functional Units.* In addition to the adaptive operations regarding the communication structure, there are three different categories of adaptive operations that may be applied to functional units.

Firstly, variations of a specific functional unit may be available. This means, for example, that optimized versions regarding chip size, latency, and throughput are available for a functional unit. The most appropriate implementation is chosen dynamically at runtime and may change throughout the lifetime of the application. The AMIDAR model allows the processor to adapt to the actual workload by substitution of two versions of a functional unit at runtime. In [26], we have shown that the characteristics of the functional units can be changed to optimally suit the needs of the running application.

Secondly, the number of instances of a specific functional unit may be increased or decreased dynamically. In case a functional unit is heavily utilized, but cannot be replaced by a specialized version with a higher throughput or shorter latency, it may be duplicated. The distribution of tokens has to be adapted to this new situation, as the token generator has to balance the workload between identical functional units.

Finally, dynamically synthesized functional units may be added to the processors datapath. It is possible to identify heavily utilized instruction sequences of an application at runtime. A large share of applications for embedded systems rely on runtime intensive computation kernels. These kernels are typically wrapped by loop structures and iterate over a given array or stream of input data. Both cases are mostly identical, as every stream can be wrapped by a buffer, which leads back to the handling of arrays by the computation itself. In [28], we have shown a hardware circuit that is capable of profiling an applications loop structures at runtime. The profiles gained by this circuit can be used to identify candidate sequences for online synthesis of functional units. These functional units would replace the software execution of the related code.

It should be mentioned that in this work only dynamically synthesizing funcational units is taken into account as adaptive operation.

*3.9. Synthesizing Functional Units in AMIDAR.* AMIDAR processors need to include some reconfigurable fabric in order to allow the dynamic synthesis and inclusion of functional units. Since fine-grained logic (like FPGAs) requires large amount of configuration data to be computed and also since the fine-grained structure is neither required nor helpful for the implementation of most code sequences, we focus on CGRAs for the inclusion into AMIDAR processors.

The model includes many features to support the integration of newly synthesized functional units into the running application. It allows bulk data transfers from and to data memories, it allows the token generator to synchronize with functional unit operations that take multiple clock cycles, and finally it allows synthesized functional units to inject tokens in order to influence the data transport required for the computation of a code sequence.

*3.10. Latency of Runtime Adaptivity.* Currently, we cannot fully determine the latencies regarding the runtime behavior of the adaptive features of the AMIDAR model. The feature which is currently examined in our studies is the runtime synthesis of new functional units. Right now, the synthesis process itself is not executed as a separate Java thread within our processor, but only as part of the running simulator. Thus, the process of creating new functional units is transparent to the processor. Hence, a runtime prediction is not possible yet. It should be mentioned that the code currently used for synthesis could be run on the target processor as it is written in Java.

Nonetheless, the usefulness of synthesizing new functional units can be determined in two ways. In one case, there is no spare time concurrently to the executed task. Then, the runtime of the synthesis process for new functional units slows down the current operation, but after finishing

the synthesis, the functional units execute much faster. Thus, eventually, the runtime lost to the synthesis process will be gained back. In the other case, there is enough spare time, the synthesis process did not slow down the application anyway and there are no objections against this type of adaptation.

## 4. Runtime Application Profiling

A major task in synthesizing hardware functional units for AMIDAR processors is runtime application profiling. This allows the identification of candidate instruction sequences for hardware acceleration. Plausible candidates are the runtime critical parts of the current application.

In previous work [28], we have shown a profiling algorithm and corresponding hardware implementation which generate detailed information about every executed loop structure. Those profiles contain the total number of executed instructions inside the affected loop, the loops start program counter, its end program counter, and the total number of executions of this loop. The profiling circuitry is also capable to profile nested loops, not only simple ones. The whole process of profiling is carried out in hardware and does not need a software intervention. As the exact method to evaluate the profiles is not of further interest for the following considerations, we will not explain it in detail here. The interested reader is referred to [28] for a full account of the resulting hardware structures.

A profiled loop structure becomes a synthesis candidate in case its number of executed instructions surmounts a given threshold. The size of this threshold can be configured dynamically for each application.

Furthermore, an instruction sequence has to match specific constraints in order to be synthesized. Currently, we are not capable of synthesizing every given instruction sequence. Bytecode containing the following instruction types cannot be handled as our synthesis algorithm has not evolved to this point yet

(i) memory allocation operations,

(ii) exception handling,

(iii) thread synchronization,

(iv) some special instructions, for example, `lookupswitch`

(v) access operations to multidimensional arrays,

(vi) method invocation operations.

Of these bytecodes memory allocation operations, exception handling, thread synchronization and the special instructions do not matter much as they typically do not occur in compute intensive instruction sequences.

Access to multidimensional arrays does indeed occur in compute kernels and would be important for a broad applicability of the synthesis. The AMIDAR model itself does not prevent this type of array access from synthesized functional units. Unfortunately, our synthesis algorithm does not handle the multistage access up to now. Multidimensional arrays are constructed as arrays of arrays in Java. Thus, such an array may not be allocated as a single-block and the detection of its actual address at runtime is not possible with a simplistic approach. Yet, a manual rewrite of the code is possible to map multidimensional arrays to one-dimensional arrays.

Finally, method invocations in general are not mappable to synthesized functional units. Nevertheless, object-oriented languages often contain getter/setter methods in objects which would break the synthesis of methods. Thus, method inlining should be applied to improve the applicability of the synthesis algorithm. However, care must be taken to deal with polymorphism appropriately. Here, techniques which are usually applied in JIT compilation can be used as well for the synthesized functional unit. In general, the approach is to assume a particular class for the called method. At runtime, the synthesized functional unit has to check whether the given object really belongs to this class. Hardware execution may only occur if this condition is fulfilled. Otherwise, the execution of the synthesized functional unit must be aborted and the original bytecode sequence must be executed instead. Currently, we do not support this special handling in the synthesis algorithm, but we plan to do so in the future.

## 5. Online Synthesis of Application-Specific Functional Units

The captured data of the profiling unit is evaluated periodically. In case, an instruction sequence exceeds the given runtime threshold, the synthesis is triggered. The synthesis runs as a low-priority process concurrently to the application. Thus, it can only occur if spare computing time remains in the system. Also, the synthesis process cannot interfere with the running application.

*5.1. Synthesis Algorithm.* An overview of the synthesis steps is given in Figure 5. The parts of the figure drawn in grey are not yet implemented.

Firstly, an instruction graph of the given sequence is created. In case an unsupported instruction is detected, the synthesis is aborted. Furthermore, a marker of a previously synthesized functional unit may be found. If this is the case, it is necessary to restore the original instruction information and then proceed with the synthesis. This may happen if an inner loop has been mapped to hardware before, and then the wrapping loop will be synthesized as well.

Afterwards, all nodes of the graph are scanned for their number of predecessors. In case a node has more than one predecessor, it is necessary to introduce specific Φ-nodes to the graph. These structures occur at the entry of loops or in typical if-else-structures. Furthermore, the graph is annotated with branching information. This will allow the identification of the actually executed branch and the selection of the valid data when merging two or more branches by multiplexers. For if-else-structures, this approach reflects a speculative execution of the alternative branches. The condition of the if-statement is used to control the selection of one set of result values. Loop entry points are treated differently, as no overlapping or software pipelining of loop kernels is employed up to now.

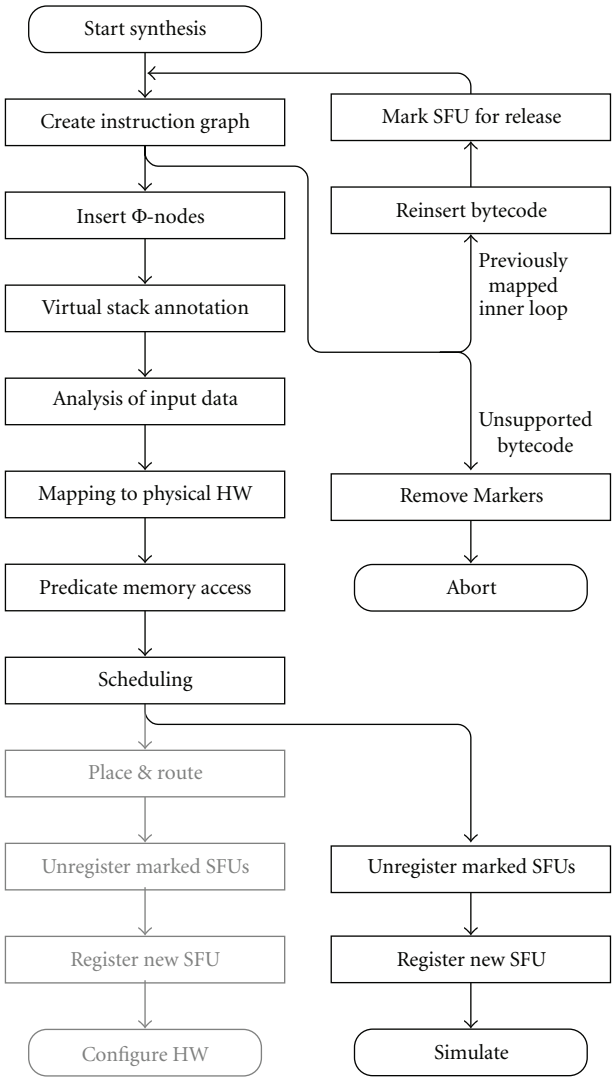


Figure 5: Overview of synthesis steps.

In a second step, the graph is annotated with a virtual stack. This stack does not contain specific data, but the information about the producing instruction that would have created it. This allows the designation of connection structures between the different instructions as the immediate predecessor of an instruction may not be the producer of its input.

Afterwards an analysis of access operations to local variables, arrays and objects takes place. This aims at loading data into the functional unit and storing it back to its appropriate memory after the functional units execution. Therefore, a list of data that has to be loaded and a list of data that has to be stored is created.

The next step transforms the instruction graph into a hardware circuit. This representation fits precisely into our simulation. All arithmetic or logic operations are transformed into their abstract hardware equivalent. The introduced Φ-nodes are transformed into multiplexer structures. The annotated branching information helps to connect the different branches correctly and to determine the appropriate control signal. Furthermore, registers and memory structures are introduced. Registers are used to hold values at the beginning and the end of branches in order to synchronize different branches. Localization of memory accesses is an important measure to improve the performance of many potential applications. In general, synthesized functional units could also access the heap to read or write array elements, but this access would incur an overhead of several clocks. The memory structures are connected to the consumer/producer components of their corresponding arrays or objects. A datapath equivalent to the instruction sequence is the result of this step.

Execution of consecutive loop kernels is strictly separated. Thus, all variables and object fields altered in the loop kernel are stored in registers at the beginning of each iteration of the loop.

Arrays and objects may be accessed from different branches that are executed in parallel branches. Thus, it is necessary to synchronize access to the affected memory regions. Furthermore, only valid results may be stored into arrays or objects. This is realized by special enable signals for all write operations. The access synchronization is realized through a controller synthesis. This step takes the created datapath and all information about timing and dependency of array and object access operations as input. The synthesis algorithm has a generic interface which allows to work with different scheduling algorithms.

Currently, we have implemented a modified ASAP scheduling which can handle resource constraints and list scheduling. The result of this step is a finite state machine (FSM) which controls the datapath and synchronizes all array and object access operations. Also the FSM takes care of the appropriate execution of simple and nested loops.

As mentioned above, we do not have a full hardware implementation yet, but we use a cycle accurate simulation. Hence, placement and routing for the CGRA structure are not required, as we can simulate the abstract datapath created in the previous steps.

In case the synthesis has been successful, the new functional unit needs to be integrated into the existing processor. If one or more marker instructions of previously synthesized functional units where found, the original instruction sequence has to be restored. Furthermore, the affected synthesized functional units have to be unregistered from the processor and the hardware used by them has to be released for further use.

The synthesis process is depicted in Figure 6. It shows the initial bytecode sequence and the resulting instruction graph, as well as data dependencies between the instructions and the final configuration of the reconfigurable fabric. The autocorrelation function achieves a speedup of 12.42 on an array with four operators and an input vector of 32 integer values.

*5.2. Functional Unit Integration.* The integration of the synthesized functional unit (SFU) into the running application consist of three major steps. (1) A token set has to be generated which allows the token generator to use the SFU.

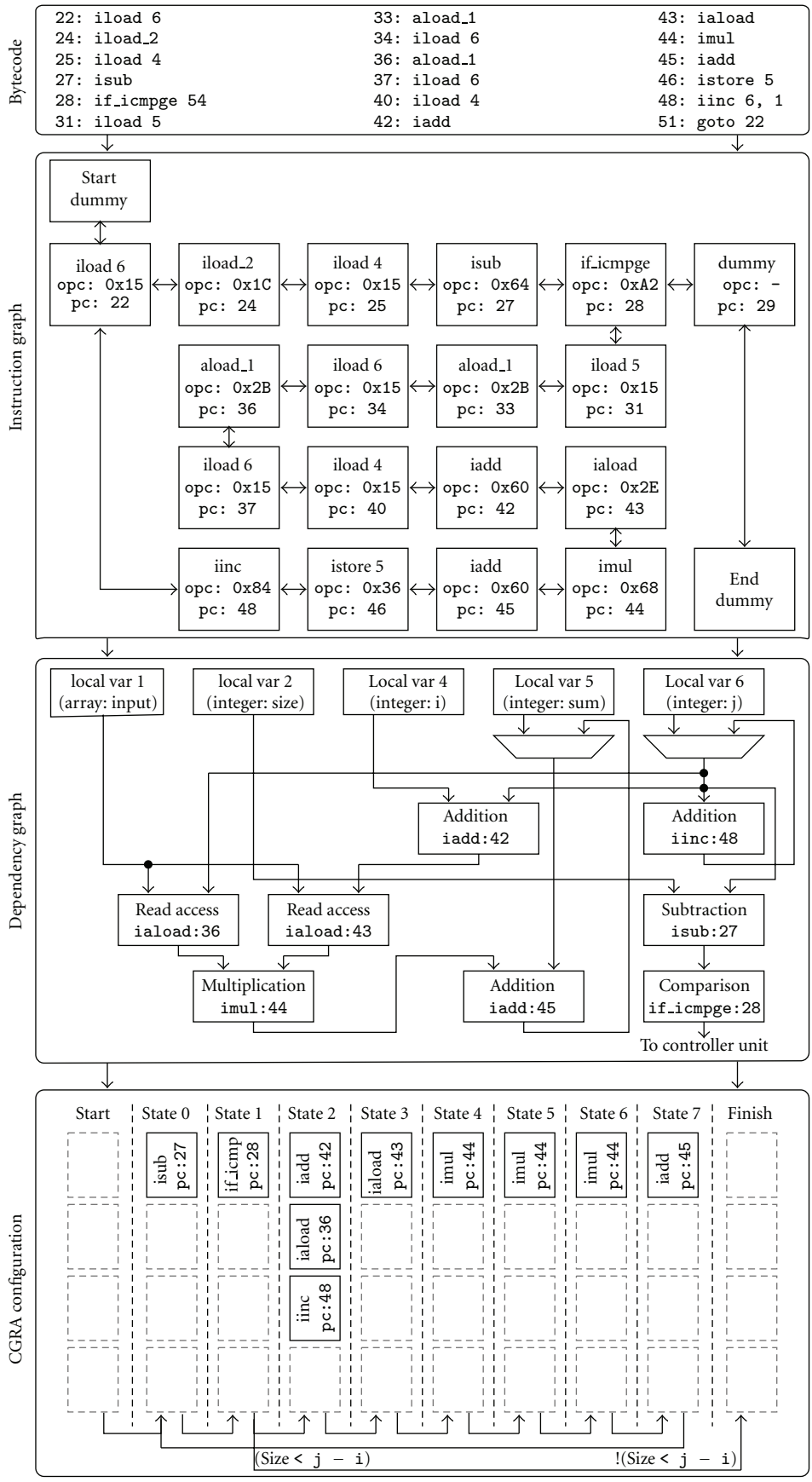


Figure 6: Example of intermediate synthesis results.

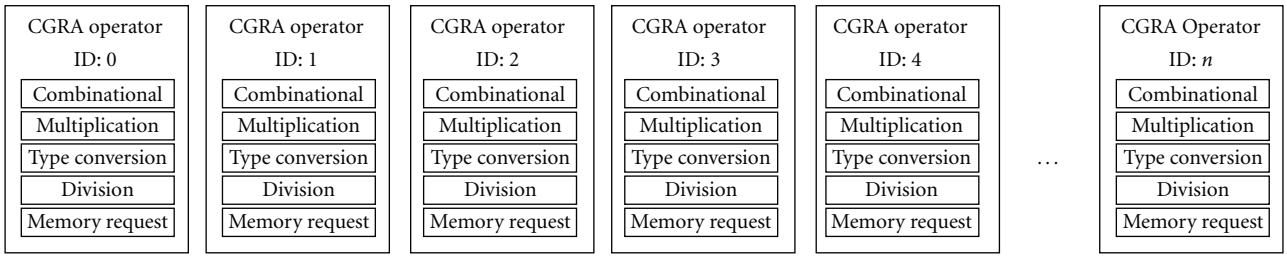


Figure 7: Configuration of the homogeneous coarse grain reconfigurable array.

(2) The SFU has to be integrated into the existing circuit and (3) The synthesized code sequence has to be patched in order to access the SFU. This last step comes with an adjustment of the profiling data which led to the decision of synthesizing an FU.

The generated set must contain all tokens and constant values that are necessary to transfer input data to the SFU, process it, and write back possible output data. All tokens are stored in first-in-first-out structures. This handling assures that tokens and constants are distributed and processed in their determined order. The lists of input and output local variables now can be used to feed the token generation algorithm.

In a second step, it is necessary to create tokens for the controlling of the SFU itself. One token has to be created in order to trigger the SFUs processing when all input data has arrived. A second token triggers the transfer of output data to its determined receiver.

In a last step, the output token sequence for the SFU is created. This set of tokens will be distributed to the functional units which will consume the generated output. The sequence must specify all operations that have to be processed in order to write the functional units output data to its receivers corresponding memory position.

In a next step, it is necessary to make the SFU known to the circuit components. Firstly, is has to be registered to the bus-arbiter. This allows the assignment of bus structures to the SFU. Furthermore, the SFU must be accessible by the token generator. The token generator works on a table which holds a token set for each instruction. Thus, the token generator needs to know a set of tokens which are necessary to control the SFU. This information is passed to the token generator in special data structures which are characterized by the ID of the SFU. This ID is necessary to distribute the correct set of tokens once the SFU is triggered by the new instruction.

The SFU needs to be accessed through a bytecode sequence in order to use it. Therefore, it is necessary to patch the affected sequence. The first instruction of the loop is patched to a specific newly introduced bytecode which signals the use of an SFU. The next byte represents the global identification number of the new SFU. It is followed by two bytes representing the offset to the successor instruction which is used by the token generator to skip the remaining instructions of the loop. In order to be able to revoke the synthesis, it is necessary to store the patched four bytes with the token data. Patching the bytecode must not be done as long as the program counter points to one of the patched locations.

Now, the sequence is not processed in software anymore but by a hardware SFU. Thus, it is necessary to adjust the profiling data which led to the decision of synthesizing a functional unit for a specific code sequence. The profile related to this sequence now has to be deleted, as the sequence itself does no longer exist as a software bytecode fragment.

In [29], we have given further information and a more detailed description of the integration process.

## 6. Evaluation

In order to evaluate the potential of our synthesis approach, we evaluated a set of benchmark applications. In previous research [30], we have evaluated the potential speedup of a simplistic online synthesis with unlimited resources. To be more realistic, we assumed a CGRA with 16 operators. Every operator is capable of executing the same set of basic functionality, as shown in Figure 7. Furthermore, the CGRA contained a single shared memory for all arrays and objects. As a result, all memory access operations must be synchronized by the scheduling algorithm. The scheduling itself has been calculated by a longest path list scheduling.

The following data-set was gained for every benchmark:

(i) its runtime, and therewith the gained speedup,

(ii) the number of states of the controlling state machine,

(iii) the number of different contexts regarding the CGRA.

The reference value for all measurements is the plain software execution of the benchmarks. Note: the mean execution time of a bytecode in our processor is ~4 clock cycles. This is in the same order as JIT-compiled code on IA32 machines.

*6.1. Benchmark Applications.* We used applications of four different domains to test our synthesis algorithm. Firstly, we benchmarked several cryptographic ciphers as the importance of security in embedded systems increases steadily. Additionally, we chose hash algorithms and message digests as a second group of appropriate applications. Thirdly, we evaluated the runtime behavior of image processing kernels.

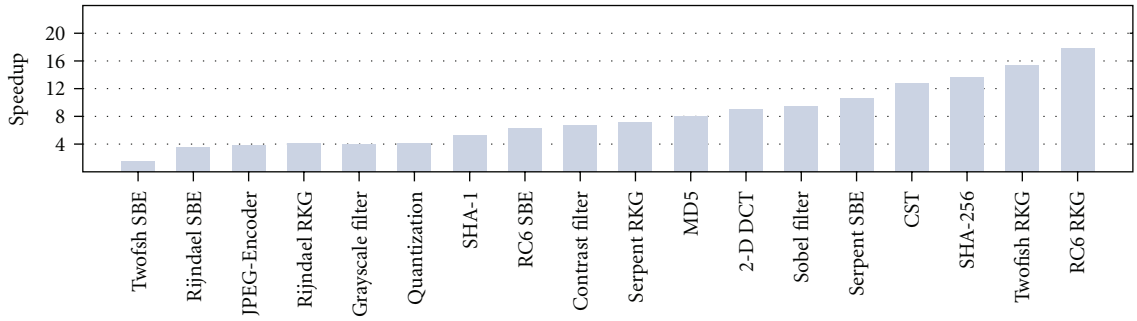


Figure 8: Diagram of runtime acceleration of benchmark applications.

All of these benchmark applications are pure computation kernels. Regularly, they are part of a surrounding application. Thus, we chose the encoding of a bitmap-image into a JPEG-image as our last benchmark application. This application contains several computation kernels, such as color space transformation, 2D forward DCT, and quantization. Nonetheless, it also contains a substantial amount of code that utilizes those kernels, in order to encode a whole image.

The group of cryptographic benchmarks contains four block ciphers. Rijndael, Twofish, Serpent, and RC6 all were part of the Advanced Encryption Standard (AES) evaluation process.

We evaluated the round key generation out of a 256-bit master key, as this is the largest common key length of those ciphers. Furthermore, we reviewed the encryption of a 16-byte data block, which is the standard block size for all of them. We did not examine the decryption of data, as it is basically an inverted implementation of the encryption. Thus, its runtime behavior is mostly identical.

Another typical group of algorithms used in the security domain are hash algorithms and message digests. We chose the Message Digest 5 (MD5) and two versions of the Secure Hash Algorithm (SHA-1 and SHA-256) as representatives. For instance, these digests are heavily utilized during TLS/SSL encrypted communication.

We measured the processing of sixteen 32 bit words, which is the standard input size for those three algorithms.

Thirdly, we rated the effects of our synthesis algorithm onto image processing kernels. Therefore, we chose a discrete differentiation that uses the Sobel convolution operator as one of those tests. This filter is used for edge detection in images. Furthermore, a grayscale filter and a contrast filter have been evaluated. As its name tells, the grayscale filter transforms a colored image into a grayscale image. The contrast filter changes the contrast of an image regarding given parameters for contrast and brightness.

These three filters operate on a dedicated pixel of an image, or on a pixel and its neighbours. Thus, we measured the appliance of every filter onto a single pixel.

Finally, as we mentioned before, we encoded a given bitmap image into a JPEG image. The computation kernels of this application are the color space transformation, 2D forward DCT and quantization. We did not downsample the chroma parts of the image. The input image we have chosen has a size of $160 \times 48$ pixels, which results in $20 \times 6$ basic blocks of $8 \times 8$ pixels. Thus, every one of the mentioned processing steps had been executed 120 times for each of the three color components, which results in a total of 360 processed input blocks.

*6.2. Runtime Acceleration.* First benchmarks with "out-of-the-box" code have shown no improvement in speed through the synthesis for all applications, except from the contrast and grayscale filter. Analysis of these runs have pointed to a failed synthesis due to unsupported bytecodes. The crucial parts of those applications contained either method invocations or access to multidimensional arrays. As we mentioned above, these instruction types are not supported by our synthesis algorithm yet. In order to show the potential of our algorithm, we inlined the affected methods and flattened the multidimensional arrays to one dimension.

The subsequent evaluations have shown sophisticated results. The gained speedup ranges from 1.52 to 17.84. The best results of the cryptographic cipher benchmarks were achieved by the round key generation of RC6 and Twofish, and furthermore the encryption of Serpent. All of them could gain speedups larger than 10. The smallest improvement of factor 1.5 was obtained by the Twofish encryption. This is clearly an outlier, which comes due to a very large communication overhead of the synthesized functional unit.

The hash algorithms we chose for our test achieved speedups from 5.25 to 13.63. The best result has been achieved by SHA256, while SHA1 has gained the weakest speedup. The MD5 could be sped up by a factor of eight.

The speedups of the image processing kernels range from 4.00 to 9.41. As the application of a grayscale filter to a pixel is a simple operation, it could only be sped up by the factor of four. The contrast filter could be accelerated by a factor of 6.76, while the Sobel convolution achieved the best result with a speedup of 9.41.

The JPEG encoding application as a whole has gained a speedup of 3.77. The computation kernels themselves achieved better results. The quantization has been sped up by a factor of 4.10, while the 2D forward DCT improved by a factor of 9.06. The largest speedup of 12.74 has been gained by the color space transformation.

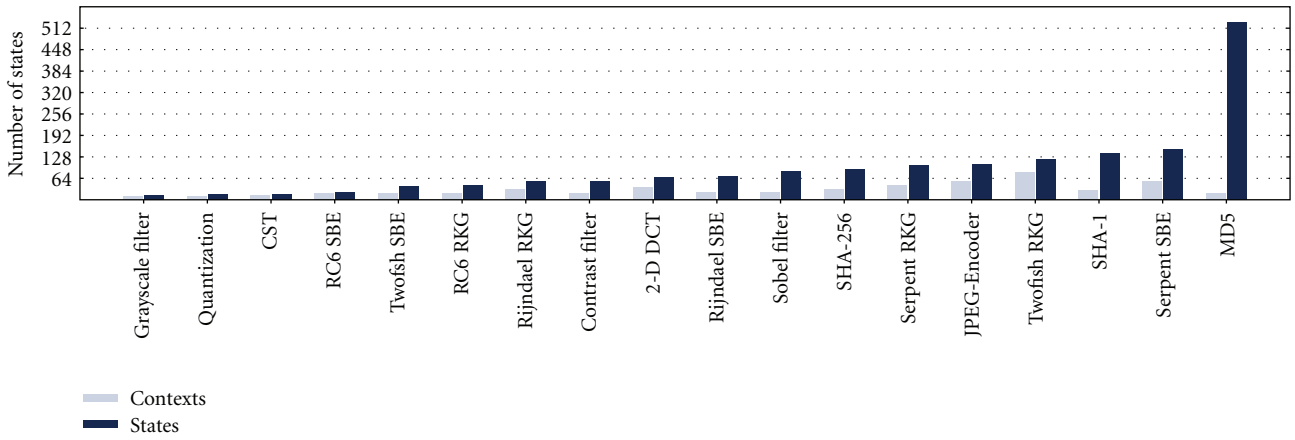


Figure 9: Diagram of complexity of the schedules of benchmark applications.

Table 1: Runtime acceleration of benchmark applications.

(a) Round key generation of cryptographic cipher benchmarks

| Configuration | Rijndael | | Twofish | | RC6 | | Serpent | |
|---|---|---|---|---|---|---|---|---|
| | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup |
| Plain software | 17760 | — | 525276 | — | 61723 | — | 44276 | — |
| Synthesis enabled | 4337 | 4.09 | 34112 | 15.40 | 3459 | 17.84 | 6230 | 7.11 |

(b) Single block encryption of cryptographic cipher benchmarks

| Configuration | Rijndael | | Twofish | | RC6 | | Serpent | |
|---|---|---|---|---|---|---|---|---|
| | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup |
| Plain software | 21389 | — | 12864 | — | 17371 | — | 34855 | — |
| Synthesis enabled | 6167 | 3.47 | 8452 | 1.52 | 2768 | 6.28 | 3273 | 10.65 |

(c) Hash algorithms and message digests

| Configuration | SHA-1 | | SHA-256 | | MD5 | |
|---|---|---|---|---|---|---|
| | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup |
| Plain software | 23948 | — | 47471 | — | 11986 | — |
| Synthesis enabled | 4561 | 5.25 | 3484 | 13.63 | 1485 | 8.07 |

(d) Image processing kernels

| Configuration | Sobel convolution | | Grayscale filter | | Contrast filter | |
|---|---|---|---|---|---|---|
| | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup |
| Plain software | 21124 | — | 236 | — | 608 | — |
| Synthesis enabled | 2246 | 9.41 | 59 | 4.00 | 90 | 6.76 |

(e) JPEG-encoding and its application kernels

| Configuration | JPEG-encoder | | Color space transformation | | Forward DCT | | Quantization | |
|---|---|---|---|---|---|---|---|---|
| | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup |
| Plain software | 17368663 | — | 3436078 | — | 23054 | — | 7454 | — |
| Synthesis enabled | 4612561 | 3.77 | 269702 | 12.74 | 2545 | 9.06 | 1816 | 4.10 |

Table 2: Complexity of the schedules of benchmark applications.

(a) Round key generation of encryption of cryptographic cipher benchmarks

| Rijndael | | Twofish | | RC6 | | Serpent | |
|---|---|---|---|---|---|---|---|
| States | Contexts | States | Contexts | States | Contexts | States | Contexts |
| 55 | 31 | 122 | 83 | 44 | 20 | 103 | 42 |

(b) Single-block encryption of encryption of cryptographic cipher benchmarks

| Rijndael | | Twofish | | RC6 | | Serpent | |
|---|---|---|---|---|---|---|---|
| States | Contexts | States | Contexts | States | Contexts | States | Contexts |
| 69 | 23 | 40 | 20 | 23 | 19 | 152 | 54 |

(c) Hash algorithms and message digests

| SHA-1 | | SHA-256 | | MD5 | |
|---|---|---|---|---|---|
| States | Contexts | States | Contexts | States | Contexts |
| 138 | 29 | 92 | 30 | 531 | 20 |

(d) Image processing kernels

| Sobel convolution | | Grayscale filter | | Contrast filter | |
|---|---|---|---|---|---|
| States | Contexts | States | Contexts | States | Contexts |
| 86 | 23 | 13 | 9 | 56 | 18 |

(e) JPEG-encoding and its application kernels

| JPEG-encoder | | Color space transformation | | Forward DCT | | Quantization | |
|---|---|---|---|---|---|---|---|
| States | Contexts | States | Contexts | States | Contexts | States | Contexts |
| 105 | 55 | 17 | 14 | 67 | 36 | 16 | 11 |

The runtime results for all of the benchmarks are displayed in Figure 8, while the corresponding measurements are shown in Table 1.

*6.3. Schedule Complexity.* In a next step, we evaluated the complexity of the controlling units that were created by the synthesis. Therefore, we measured the size of the finite state machines, that are controlling every synthesized functional unit. Every state is related to a specific configuration of the reconfigurable array. In the worst case, all of those contexts would be different. Thus, the size of a controlling state machine is the upper bound for the number of different contexts.

Afterwards, we created an execution profile for every context. This profile contains a key for every operation that is executed within the related state. Accordingly, we removed all duplicates from the set of configurations. The number of elements in this resulting set is a lower bound for the number of contexts that are necessary to drive the functional unit. The effective number of necessary configurations lies between those two bounds, as it depends on the place-and-route results of the affected operations.

The context information for the benchmarks is displayed in Figure 9, while the actual number of states and contexts is given in Table 2. It shows the size of the controlling finite state machine (States), and the number of actually different contexts (Contexts) for every of our benchmarks. It can be seen, that only three of eighteen state machines consist of more than 128 states. Furthermore, the bigger part of the state machines contains a significant number of identical states regarding the executed operations. Thus, the actual number of contexts is well below the number of states.

## 7. Conclusion

In this article, we have shown a simplistic online-synthesis algorithm for AMIDAR processors. It is capable of synthesizing functional units fully automated at runtime regarding given resource constraints. The target technology for our algorithm is a coarse-grained reconfigurable array. We assumed a reconfigurable fabric with homogeneously formed processing elements and one shared memory for all objects and arrays.

The displayed synthesis approach targets maximum simplicity and runtime efficiency of all used algorithms. Therefore, we used list scheduling as scheduling algorithm and passed on more complex methods like software pipelining. Furthermore, we have not run optimization algorithms on the finite state machines that were created by our synthesis.

In order to demonstrate the capabilities of our algorithm, we have chosen four groups of benchmark applications. Those applications were cryptographic ciphers, message digests, graphic filters, and the JPEG encoding process. All benchmarks could be accelerated by the synthesis, and the mean speedup that has been achieved is 7.95. Furthermore, we displayed the complexity of the gained synthesis results. This evaluation showed that most of our benchmarks are to be driven by less than 128 contexts.

## 8. Future Work

The full potential of online synthesis in AMIDAR processors has not been reached yet. Future work will concentrate on improving our existing synthesis algorithm in multiple ways. This includes the implementation of access to multidimensional arrays and inlining of invoked methods at synthesis time. Additionally, we will explore the effects of instruction chaining in synthesized functional units. Furthermore, we are planning to overlap the transfer of data to a synthesized functional unit and its execution. We are planning to introduce an abstract description layer to our synthesis. This will allow easier optimization of the algorithm itself and will open up the synthesis for a larger number of instruction sets. Currently, we are able to simulate AMIDAR processors based on different instruction sets, such as LLVM-Bitcode, .NET Common-Intermediate-Language, and Dalvik-Executables. In the future, we are planning to investigate the differences in execution of those instruction sets in AMIDAR-processors.

## References


[1] S. Vassiliadis and D. Soudris, *Fine-and Coarse-Grain Reconfigurable Computing*, Springer, New York, NY, USA, 2007.

[2] C. Hochberger, R. Hoffmann, K.-P. Völkmann, and S. Waldschmidt, "The cellular processor architecture CEPRA-1X and its conguration by CDL," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '00)*, pp. 898–905, 2000.

[3] E. Sotiriades and A. Dollas, "A general reconfigurable architecture for the BLAST algorithm," *Journal of VLSI Signal Processing*, vol. 48, no. 3, pp. 189–208, 2007.

[4] J. R. Hauser and J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor," in *Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '97)*, pp. 12–21, April 1997.

[5] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood, "Hardware-software co-design of embedded reconfigurable architectures," in *Proceedings of the 37th Design Automation Conference (DAC '00)*, pp. 507–512, June 2000.

[6] A. Koch and N. Kasprzyk, "High-level-language compilation for reconfigurable computers," in *Proceedings of the International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC '05)*, pp. 1–8, 2005.

[7] Y. Ha, R. Hipik, S. Vernalde et al., "Adding hardware support to the hotSpot virtual machine for domain specific applications," in *Proceedings of the International Conference on Field Programmable Logic (FPL '02)*, pp. 1135–1138, 2002.

[8] C. Ebeling, D. C. Cronquist, and P. Franklin, "RaPiD—reconfigurable pipelined datapath," in *Proceedings of the International Conference on Field Programmable Logic (FPL '96)*, pp. 126–135, 1996.

[9] Y. Chou, P. Pillai, H. Schmit, and H. P. Shen, "PipeRench implementation of the instruction path coprocessor," in *Proceedings of the Annual International Symposium on Microarchitecture (MICRO '00)*, pp. 147–158, 2000.

[10] R. W. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, "Mapping applications onto reconfigurable Kress Arrays," in *Proceedings of the International Conference on Field Programmable Logic (FPL '99)*, pp. 385–390, 1999.

[11] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "PACT XPP—a self-reconfigurable data processing architecture," *Journal of Supercomputing*, vol. 26, no. 2, pp. 167–184, 2003.

[12] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," in *Proceedings of the Design, Automation and Test in Europe (DATE '03)*, pp. 10296–10301, 2003.

[13] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "ADRES: an architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *Proceedings of the International Conference on Field Programmable Logic (FPL '03)*, pp. 61–70, 2003.

[14] W. Kehuai, A. Kanstein, J. Madsen, and M. Berekovic, "MT-ADRES: multithreading on coarse-grained reconfigurable architecture," in *Proceedings of the International Workshop on Applied Reconfigurable Computing (ARC '07)*, pp. 26–38, 2007.

[15] A. Lodi, M. Toma, F. Campi, A. Cappelli, R. Canegallo, and R. Guerrieri, "A VLIW processor with reconfigurable instruction set for embedded applications," *IEEE Journal of Solid-State Circuits*, vol. 38, no. 11, pp. 1876–1886, 2003.

[16] G. Hempel, C. Hochberger, and A. Koch, "A comparison of hardware acceleration interfaces in a customizable soft core processor," in *Proceedings of the International Conference on Field Programmable Logic (FPL '10)*, pp. 469–474, 2010.

[17] C. Rupp, Multi-scale programmable array (US patent 6633181), October 2003.

[18] L. Bauer, M. Shafique, S. Kramer, and J. Henkel, "RISPP: rotating instruction set processing platform," in *Proceedings of the 44th ACM/IEEE Design Automation Conference (DAC '07)*, pp. 791–796, June 2007.

[19] R. L. Lysecky and F. Vahid, "Design and implementation of a microblaze-based WARP processor," *ACM Transactions on Embedded Computing Systems*, vol. 8, no. 3, pp. 1–22, 2009.

[20] A .C .S. Beck and L. Carro, "Dynamic reconfiguration with binary translation: breaking the ILP barrier with software compatibility," in *Proeedings of the Design Automation Conference (DAC '05)*, pp. 732–737, September 2005.

[21] J. C. Dehnert, B. K. Grant, J. P. Banning et al., "The transmeta code morphing™ software: using speculation,recovery, and adaptive retranslation to address real-life challenges," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO '03)*, pp. 15–24, 2003.

[22] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro, "Transparent reconfigurable acceleration for heterogeneous embedded applications," in *Proceedings of the Design, Automation and Test in Europe (DATE '08)*, pp. 1208–1213, 2008.

[23] S. Uhrig, B. Shehan, R. Jahr, and T. Ungerer, "A Two-dimensional Superscalar processor architecture," in *Proceedings of the Computation World: Future Computing, Service Computation, Adaptive, Content, Cognitive, Patterns (ComputationWorld '09)*, pp. 608–611, 2009.

[24] H. Corporaal, *Microprocessor Architectures: From VLIWto TTA*, John Wiley & Sons, New York, NY, USA, 1997.

[25] S. Gatzka and C. Hochberger, "A new general model for adaptive processors," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '04)*, pp. 52–60, June 2004.

[26] S. Gatzka and C. Hochberger, "The AMIDAR class of reconfigurable processors," *Journal of Supercomputing*, vol. 32, no. 2, pp. 163–181, 2005.

[27] S. Gatzka and C. Hochberger, "The organic features of the AMIDAR class of processors," in *Proceedings of the International Workshop on Applied Reconfigurable Computing (ARC '05)*, pp. 154–166, 2005.

[28] S. Gatzka and C. Hochberger, "Hardware based online profiling in AMIDAR processors," in *Proceedings of the19th IEEE International Parallel and Distributed Processing Symposium (IPDPS '05)*, April 2005.

[29] S. Döbrich and C. Hochberger, "Towards dynamic software/hardware transformation in AMIDAR processors," *Information Technology*, vol. 50, no. 5, pp. 311–316, 2008.

[30] S. Döbrich and C. Hochberger, "Effects of simplistic online synthesis for AMIDAR processors," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '09)*, pp. 433–438, December 2009.