Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2011, Article ID 601986, 22 pages
doi:10.1155/2011/601986

*Research Article*

# Exploring Online Synthesis for CGRAs with Specialized Operator Sets

**Stefan Döbrich and Christian Hochberger**

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

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

Received 10 August 2010; Revised 16 December 2010; Accepted 10 February 2011

Academic Editor: Michael Hübner



The design of energy-efficient systems has become a major challenge for engineers over the last decade. One way to save energy is to spread out computations in space rather than in time (as traditional processors do). Unfortunately, this requires to design specialized hardware for each application. Also, the nonrecurring expenses for the manufacturing of chips continuously grow. Implementing the computations on FPGAs and CGRAs solves this dilemma, as the non recurring expenses are shared between many different applications. We believe that online synthesis that takes place during the execution of an application is one way to broaden the applicability of reconfigurable architectures as no expert knowledge of synthesis and technologies is required. In this paper, we give a detailed analysis of the amount and specialization of resources in a CGRA that are required to grant a significant speedup of Java bytecode. In fact, we show that even a relatively small number of specialized reconfigurable resources is sufficient to speed up applications considerably. Particularly, we look at the number of dedicated multipliers and dividers. Also, we discuss the required number of concurrent memory access operations inside the CGRA. Again, it shows that two concurrent memory access operations are sufficient for almost all applications.

## 1. Introduction

Designers of almost all types of systems experience a continuously increased demand for performance and/or higher energy efficiency. Various options are intensely discussed to satisfy this demand.

The currently most often named technology is multicore processors. Using them to gain substantial performance improvements is a rather involved process, and, up till now, it is a technology which is typically mainly found in desktop and server systems. Only now dual or quad core systems are emerging in the area of embedded systems.

Popular technologies like general purpose graphics processors (GPGPU) consume vast amounts of energy and require very specialized programming environments (e.g., OpenCL or CUDA).

In the area of embedded systems, MPSoCs are a valid choice to cast the available transistors into usable computing power. However, existing application code has to be rewritten to distribute the application over the different cores and to synchronize the processing.

A more flexible way to use the available transistors is given by field programmable gate arrays (FPGAs). Here, the user can configure logic resources on a bit level (fine-grained logic) to build individual circuits that accomplish the required behaviour of the application. Unfortunately, this requires expert knowledge and thus is not an option for traditional software developers.

Some FPGA families even allow dynamic partial reconfiguration so that only a part of the implemented circuit is exchanged at runtime. This enables a dynamic adaptation of the implemented circuit to the characteristics of the application. The tool support which is required to handle this design style is enormous, and thus the corresponding configurations are created offline and outside of the target system. The high flexibility to configure circuits at bit level also comes with the big drawback of the large amount of configuration information.

Coarse-grain reconfigurable arrays (CGRAs) as a contrary technology to FPGAs, 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*.

In general, all of the above solutions require a major restructuring and/or rewriting of the application code. Often only a complete new development will reach the full potential of the underlying implementation technology.

The aim of our research is to provide a new processor paradigm the AMIDAR class of processors [1], which makes code optimization or architecture knowledge by developers needless for performance improvement/energy saving.

Certainly, it is clear that knowledge of the underlying architecture or paying attention to best practices can improve the resulting performance even more. But for all that, no special knowledge beyond software development skills is necessary.

The AMIDAR model itself is an abstract processor model which is capable of targeting different platforms—for instance, it may be implemented by full-custom design or as a soft-core targeting FGPAs. According to the actual implementation, the model provides built-in runtime adaptivity at different levels, even if not all technologies are capable of implementing all adaptive operation.

The set of adaptive operations covers reorganization of communication structures, evolution of functional units, and the synthesis of application-specific functional units at runtime. For the sake of readability, Section 3 gives an overview of the AMIDAR model, although similar descriptions have been published in other papers. Furthermore, it provides additional references to further reading about selected aspects of AMIDAR.

In order to prove the capabilities of our model, we have implemented a cycle accurate simulator, which allows the analysis of individual aspects regardless of the underlying technology. In previous papers, we have already discussed the effects of bus-level adaptivity and evolution of functional units [2].

Currently, our research targets the synthesis of new application-specific functional units at runtime. These functional units are mapped to a CGRA which is part of the processor itself. Choosing CGRAs as target platform over fine-grained structures reduces the amount of configuration information considerably.

The most promising target for the synthesis of new functional units are the runtime-intensive kernels of the actual application. In order to determine these code sequences, a continuous runtime profiling of the executed code is done. We have already proposed a hardware circuit that provides capability to handle this profiling and all associated information [3]. Nevertheless, we will give a short introduction to the actual mechanism in Section 4.

The synthesis is triggered in case the profiling mechanism detects a code sequence that consumes more execution time than a given threshold. As the synthesis takes place at runtime of the application, all of our algorithms are designed to consume as little runtime as possible. Hence, we are trading quality of the synthesis results for a better performance and a smaller memory footprint. We have proposed our synthesis algorithms in previous work [4, 5]. Anyhow, Section 5 details the synthesis as the algorithms are fundamental to our whole concept and the contributions of this article.

The main focus of this article lies on the implementation of improved resource constraints in the synthesis algorithms. We evaluated the influence of different CGRA characteristics on the runtime of synthesized functional units. These characteristics cover the actual size of the CGRA regarding the number of its operators, the set of operations implemented by each operator, as well as the effect of dual ported memory access within the CGRA. The evaluation of all benchmark applications is presented in Section 6.

A major insight that we gain from our benchmarks is that almost all application kernels we tested can be mapped to an array of four or at most eight operators. Thus, it is possible to implement several different kernels on a CGRA with sixteen operators in parallel. The more detailed conclusion in Section 7 additionally provides a comparison of AMIDARs performance with an Intel Pentium Core2 Duo processor.

We are aiming at a further performance and quality improvement of both, the synthesis algorithms as well as the generated functional units. Further details on the research work we are planning to accomplish in the future is given in Section 8.

## 2. Related Work

Fine grain 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 [6]. This level of speedup still persists for many application areas, for example, the BLAST algorithm [7]. 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 [8]. It was accompanied by the synthesizing C compiler NIMBLE [9] that automatically partitions and maps the application.

Static transformation from high-level languages like C into fine grain 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 [10, 11].

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 [12]. 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 [13], PipeRench [14], Kress-Arrays [15], or the PACT-XPP [16]. In most cases, specialized tool sets

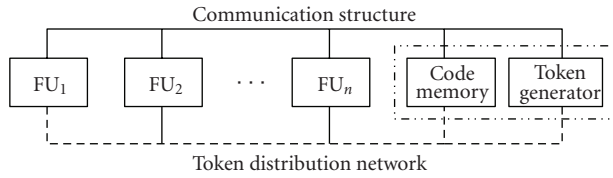


Figure 1: Abstract model of an AMIDAR processor.

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 [17] tool chain targeting the ADRES [18, 19] 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.

The RISPP architecture [20] 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 grain reconfigurable logic [21]. This happens by synthesis of bitstreams for the targeted WARP-FPGA platform. Furthermore, dynamic synthesis of Java bytecode has been evaluated [22]. Nonetheless, this approach is only capable of synthesizing combinational hardware.

The token distribution principle of AMIDAR processors has some similarities with transport triggered architectures [23]. Yet, in TTAs an application is transformed directly into a set of tokens. This leads to a very high memory overhead and makes an analysis of the executed code extremely difficult.

## 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 a minimum implementation of an AMIDAR-based Java machine is given. Finally, we discuss several mechanisms of the model that allow the processor to adopt 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 is 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 are 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 $T = \{UID, OP, TAG, DP, INC\}$. 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.

Table 1: Runtime of benchmarks on AMIDAR based Java processor and x86 Intel Core 2 Duo @ 2.66 GHz.

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

| Processor | Rijndael clock ticks | Twofish clock ticks | RC6 clock ticks | Serpent clock ticks |
|---|---|---|---|---|
| x86 | ≈8700 | ≈200000 | ≈28000 | ≈21000 |
| AMIDAR | 17760 | 525276 | 61723 | 44276 |

(b) Single block encryption of cryptographic cipher benchmarks

| Processor | Rijndael clock ticks | Twofish clock ticks | RC6 clock ticks | Serpent clock ticks |
|---|---|---|---|---|
| x86 | ≈8800 | ≈4000 | ≈5600 | ≈14000 |
| AMIDAR | 21389 | 12864 | 17371 | 34855 |

(c) Hash algorithms and message digests

| Processor | SHA-1 clock ticks | SHA-256 clock ticks | MD5 clock ticks |
|---|---|---|---|
| x86 | ≈9100 | ≈17200 | ≈4700 |
| AMIDAR | 23948 | 47471 | 11986 |

(d) Filter applications

| Processor | Sobel filter clock ticks | Grayscale filter clock ticks | Contrast filter clock ticks |
|---|---|---|---|
| x86 | ≈7900 | ≈200 | ≈370 |
| AMIDAR | 21124 | 236 | 608 |

(e) JPEG encoding and its application kernels

| Processor | JPEG encoder clock ticks | Color space transformation clock ticks | 2-D DCT clock ticks | Quantization clock ticks |
|---|---|---|---|---|
| x86 | ≈17300000 | ≈2700000 | ≈11200 | ≈4800 |
| AMIDAR | 17368663 | 3436078 | 23054 | 7454 |

A more detailed explanation of the model, its application to Java bytecode execution, and its specific features can be found in [1, 24].

*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, and 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 an example implementation of an AMIDAR based Java 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. In case of a Java processor, the code memory holds 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, 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

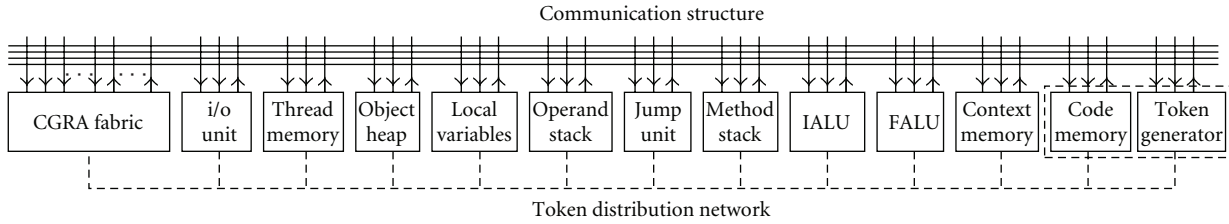


Figure 2: Model of a Java (non)Virtual Machine on AMIDAR basis.

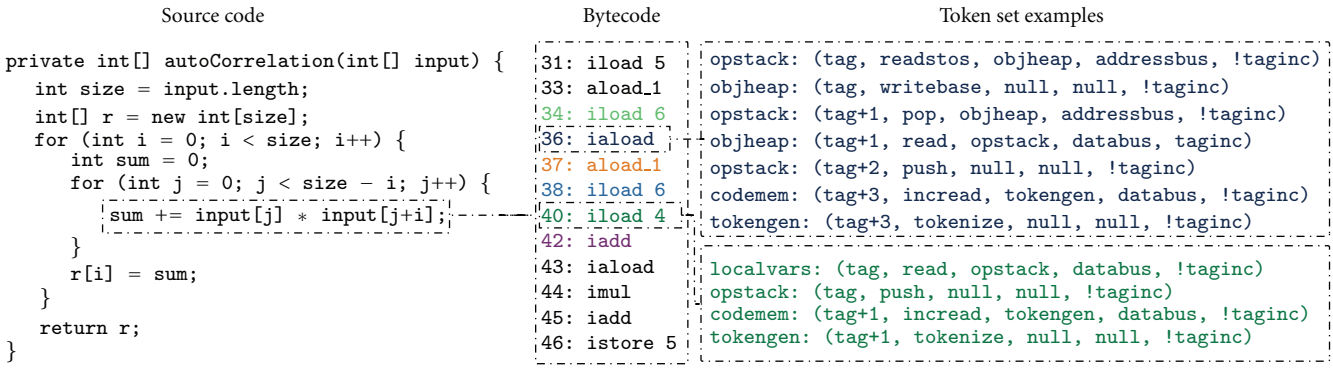


Figure 3: Example source code sequence and the resulting bytecode and exemplified token sequences.

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. The context of currently not running threads is stored in the context memory.

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.

Instructions and data are distributed over the communication network. In the presented case, this structure consists of four equal busses of 32 bit width. The busses are assigned to the functional units via round robbin.

*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 on 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 array's address on the heap and the offset of the actual value are positioned at the top of the stack. Firstly, the array's 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 and sent to the heap, and is 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 adopt 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 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 data transports can be minimized

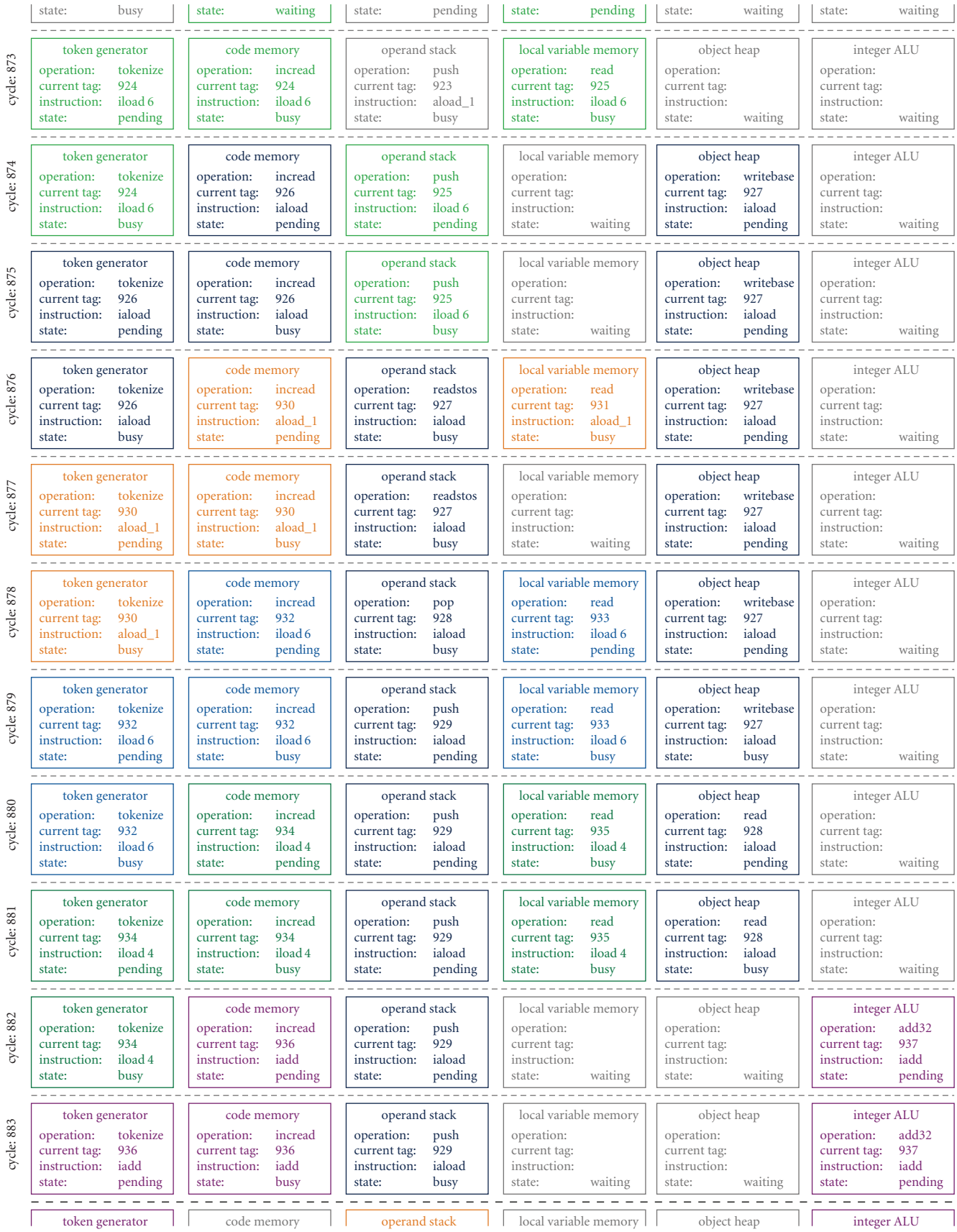


Figure 4: Visualized excerpt of an exeuction trace of the autocorrelation example.

Table 2: Runtime acceleration of benchmark applications.

(a) Round key generation

| Configuration | Rijndael | | Twofish | | RC6 | | Serpent | |
|---|---|---|---|---|---|---|---|---|
| | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup |
| Plain software | 17760 | — | 525276 | — | 61723 | — | 44276 | — |
| 4 operators | 4602 | 3.86 | 43224 | 12.15 | 3725 | 16.57 | 6335 | 6.99 |
| 8 operators | 4284 | 4.15 | 35130 | 14.95 | 3459 | 17.84 | 6245 | 7.09 |
| 12 operators | 4337 | 4.09 | 34280 | 15.32 | 3459 | 17.84 | 6230 | 7.11 |
| 16 operators | 4337 | 4.09 | 34112 | 15.40 | 3459 | 17.84 | 6230 | 7.11 |

(b) Single block encryption

| Configuration | Rijndael | | Twofish | | RC6 | | Serpent | |
|---|---|---|---|---|---|---|---|---|
| | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup |
| Plain software | 21389 | — | 12864 | — | 17371 | — | 34855 | — |
| 4 operators | 6230 | 3.43 | 8506 | 1.51 | 2852 | 6.09 | 3278 | 10.63 |
| 8 operators | 6181 | 3.46 | 8452 | 1.52 | 2810 | 6.18 | 3273 | 10.65 |
| 12 operators | 6167 | 3.47 | 8452 | 1.52 | 2768 | 6.28 | 3273 | 10.65 |
| 16 operators | 6167 | 3.47 | 8452 | 1.52 | 2768 | 6.28 | 3273 | 10.65 |

(c) Hash & digest algorithms

| Configuration | SHA-1 | | SHA-256 | | MD5 | |
|---|---|---|---|---|---|---|
| | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup |
| Plain software | 23948 | — | 47471 | — | 11986 | — |
| 4 operators | 4561 | 5.25 | 3619 | 13.12 | 1485 | 8.07 |
| 8 operators | 4561 | 5.25 | 3484 | 13.63 | 1485 | 8.07 |
| 12 operators | 4561 | 5.25 | 3484 | 13.63 | 1485 | 8.07 |
| 16 operators | 4561 | 5.25 | 3484 | 13.63 | 1485 | 8.07 |

(d) Image Processing

| Configuration | Sobel filter | | Grayscale filter | | Contrast filter | |
|---|---|---|---|---|---|---|
| | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup |
| Plain software | 21124 | — | 236 | — | 608 | — |
| 4 operators | 2246 | 9.41 | 59 | 4.00 | 90 | 6.76 |
| 8 operators | 2246 | 9.41 | 59 | 4.00 | 90 | 6.76 |
| 12 operators | 2246 | 9.41 | 59 | 4.00 | 90 | 6.76 |
| 16 operators | 2246 | 9.41 | 59 | 4.00 | 90 | 6.76 |

(e) JPEG encoding

| Configuration | JPEGEncoder | | Color Space Transformation | | 2-D Forward DCT | | Quantization | |
|---|---|---|---|---|---|---|---|---|
| | Clock Ticks | Speedup | Clock Ticks | Speedup | Clock Ticks | Speedup | Clock Ticks | Speedup |
| Plain software | 17368663 | — | 3436078 | — | 23054 | — | 7454 | — |
| 4 operators | 4737944 | 3.67 | 323805 | 10.61 | 2743 | 8.40 | 1816 | 4.10 |
| 8 operators | 4645468 | 3.74 | 292889 | 11.73 | 2572 | 8.96 | 1816 | 4.10 |
| 12 operators | 4620290 | 3.76 | 277431 | 12.39 | 2545 | 9.06 | 1816 | 4.10 |
| 16 operators | 4612561 | 3.77 | 269702 | 12.74 | 2545 | 9.06 | 1816 | 4.10 |

by adapting the communication structure. Therefore, we designed a set of several adaptive operations that may be applied to it. In [2], we have shown how to identify the conflicting bus taps and we have also shown a heuristic to modify the bus structure to minimize the conflicts.

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.

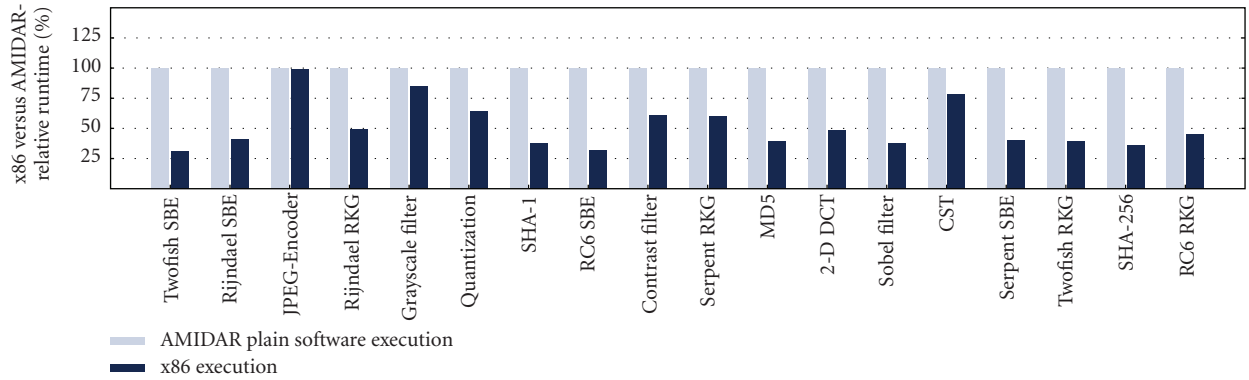


Figure 5: Runtime comparison of AMIDAR based Java processor and x86 Intel Core 2 Duo @ 2.66GHz.

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 transfered 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.

All of the described adaptive bus operations have been evaluated at topology level in the already mentioned paper. A hardware technique which allows the actual execution of these operations has not yet been part of our research.

*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 adopt to the actual workload by substitution of two versions of a functional unit at runtime. In [1], 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. In contrary to the preceding and succeeding technique, this one has not been evaluated yet. Although the model itself offers this type of adaptivity, it should be noted that we do not further investigate it in this contribution.

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 [3], 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.

*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 a large amount of configuration data to be computed and also since the fine grain structure is neither required nor helpful for the implementation of most code sequences, we focus on CGRAs for the inclusion into AMIDAR processors. Successfully employing CGRAs in reconfigurable computing is shown in [25].

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 case there is no spare time concurrently to the executed task, 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 case there is enough spare time, the synthesis process did not slow down the application any way and there are no objections against this type of adaptation.

*3.11. AMIDAR Performance Evaluation.* We compared the AMIDAR based Java processor to an Intel Core 2 Duo in order to obtain an impression of its runtime performance. Therefore, we compiled a set of benchmarks to native code. The benchmarks that were used are described in detail in Section 6.1. The runtime of the benchmarks is displayed in Figure 5 and Table 1. Figure 5 depicts the relation between the two different benchmarks for all applications. The AMIDAR execution is used as baseline while the runtime of the x86 execution is displayed proportionately.

It can be seen, that the runtime on the basic AMIDAR processor is is up to three-times higher. Furthermore, the JPEG Encoder benchmarks as a whole application does not fall behind x86 execution, because of its high amount of memory accesses. Overall, it can be said that the execution of a program on an AMIDAR processor takes twice the time as the execution on an x86 processor. Standard interpreter Java Virtual Machines do not achieve such a good relative performance compared to natively compiled code.

## 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 [3], we have shown a profiling algorithm and corresponding hardware implementation which generates 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.

Profiling is based on the fact that the last instruction of loops is always branch with negative offset in Java bytecode. Also, negative branch offsets are only used for this purpose and do not occurat other places of the code. The value of an instructions counter is added to an associated loop register (one for each loop or loop nesting level). These loop registers are realized by a fully associative memory. The size of this memory depends on the maximum number of loops and loop nesting levels in a method. It is usually very small for real live applications (<16). The assiciative memory has to be saved during method calls and returns. This requires typically less time than the housekeeping of the method call itself. Thus, profiling does not introduce any runtime overhead.

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 code sequences containing the following instruction types, 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.

From this group, only access to multidimensional arrays and method invocations are important from a performance aspect.

Multidimensional arrays do actually occur in compute kernels. Access operations on these arrays are possible in principle in the AMIDAR model. Yet, multidimensional arrays are organized as arrays of arrays in Java. Thus, access operations need to be broken down into a set of stages (one for each dimension), which is not yet supported by our synthesis algorithm. Nevertheless, a manual rewrite of the code is possible to map multidimensional arrays to one dimension. Reorganizing memory access patterns during the synthesis process could certainly improve the performance here, but the required dependency analysis is far too complex to be carried out online.

Similarly, method inlining can be used to enable the synthesis of code sequences that contain method invocations. Techniques for the method inlining are known from JIT compilers that preserve the polymorphism of the called method. Yet, these techniques require the abortion of the execution of the HW under some conditions, which is not yet supported by our synthesis algorithm.

## 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, and runs as a low-priority process concurrently to the application. Thus, it only occurs if spare computing time remains in the system, and also cannot interfere with the running application.

*5.1. Synthesis Algorithm.* An overview of the synthesis steps is given in Figure 6. The parts of the figure drawn in grey are not yet implemented.

Firstly, an instruction graph of the given sequence is created. In this graph, every instruction is represented by a node. The predecessor/successor relations are represented by the graphs edges. 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.

In the next step, the graph is annotated with a virtual stack. This stack does not contain specific data, but contians the information about the producing instruction that would have created it. This allows the designation of connection structures between the different instructions as the 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 its execution. Therefore, a list of data that has to be loaded and a list of data that has to be stored are 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 transfered to 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 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 potential applications. In general, SFUs 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 loop iteration.

Arrays and objects may be accessed from different branches that are executed in parallel. 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 additionally we implemented 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. Thus, placement and routing for the CGRA are not required. We use a cycle accurate simulation of 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 processor. If marker instructions of previously synthesized FUs were found, the original instruction sequence has to be restored. Furthermore, the affected SFUs have to be unregistered from the processor, and the hardware used by them has to be released.

The synthesis process is depicted in Figure 7. It shows the initial bytecode sequence, 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 consists of three major steps: (1) a token set has to be generated which allows the token generator to use the SFU. (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.

The token set consists of three parts: (1) the tokens that transport input data to the SFU, these tokens are sent to the appropriate data sources (e.g., object heap), (2) the tokens that control the operation of the SFU, that is, that start the operation (which happens once the input data is available) and emit the results, and (3) the token set that stores the results of the SFU operation in the corresponding memory.

In a next step, it is necessary to make the SFU accessible to the other processor components. This requires to register

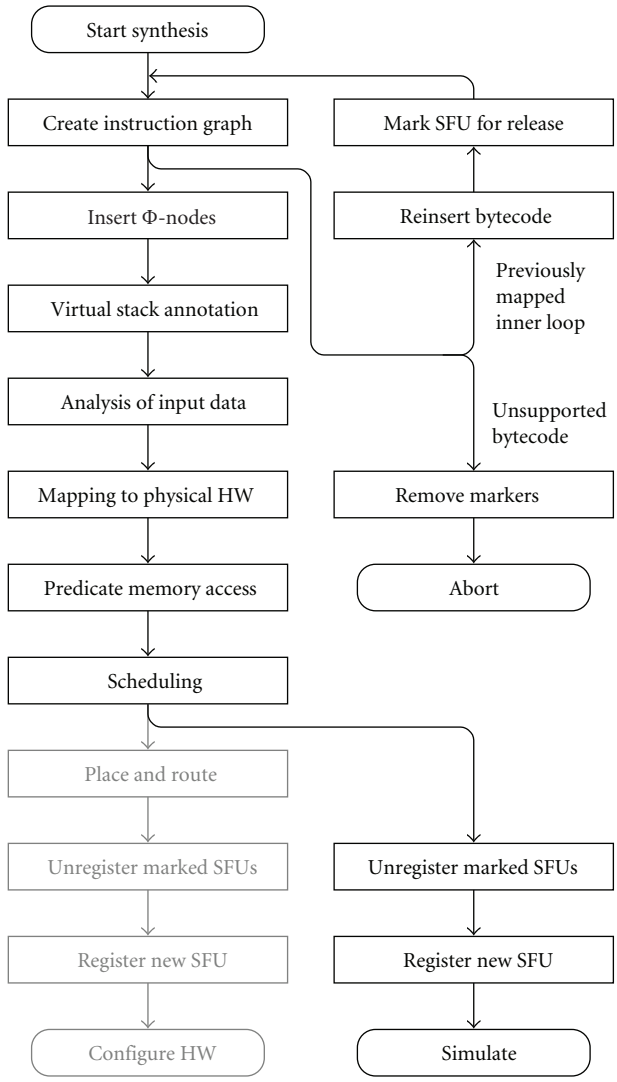


Figure 6: Overview of synthesis steps.

it in the bus arbiter and to update the token generator with the computed token sets. The token set will be triggered by a reserved bytecode instruction.

Finally, the original bytecode sequence has to be replaced by the reserved bytecode instruction. To allow multiple SFUs to coexist, the reserved bytecode carries the ID of the targeted SFU. Patching of the bytecode sequence is done in such a way that the token generator can continue the execution at the first instruction after the transformed bytecode sequence. Also, it must be possible to restore the original sequence in case an embracing loop nesting level will be synthesized.

Now, the sequence is not processed in software anymore but by a hardware SFU. Thus, it is necessary to adjust the profiling data of the affected code sequence.

In [26], we have given further information and a more detailed description of the integration process.

## 6. Evaluation

In previous research [27], we have evaluated the potential speedup of a simplistic online synthesis with unlimited resources. This is an unrealistic assumption. Thus, we are targeting an architecture based on a CGRA with a limited number of processing elements, and a single shared memory for all arrays and objects [5]. The scheduling of all operations is calculated by longest path list scheduling. The following dataset shows the characteristics of every benchmark and the influence of online synthesis at an applications runtime behavior:

(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,

(iv) the number of complex operations within those contexts.

The reference value for all measurements is the plain software execution of the benchmarks without synthesized functional units. 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.

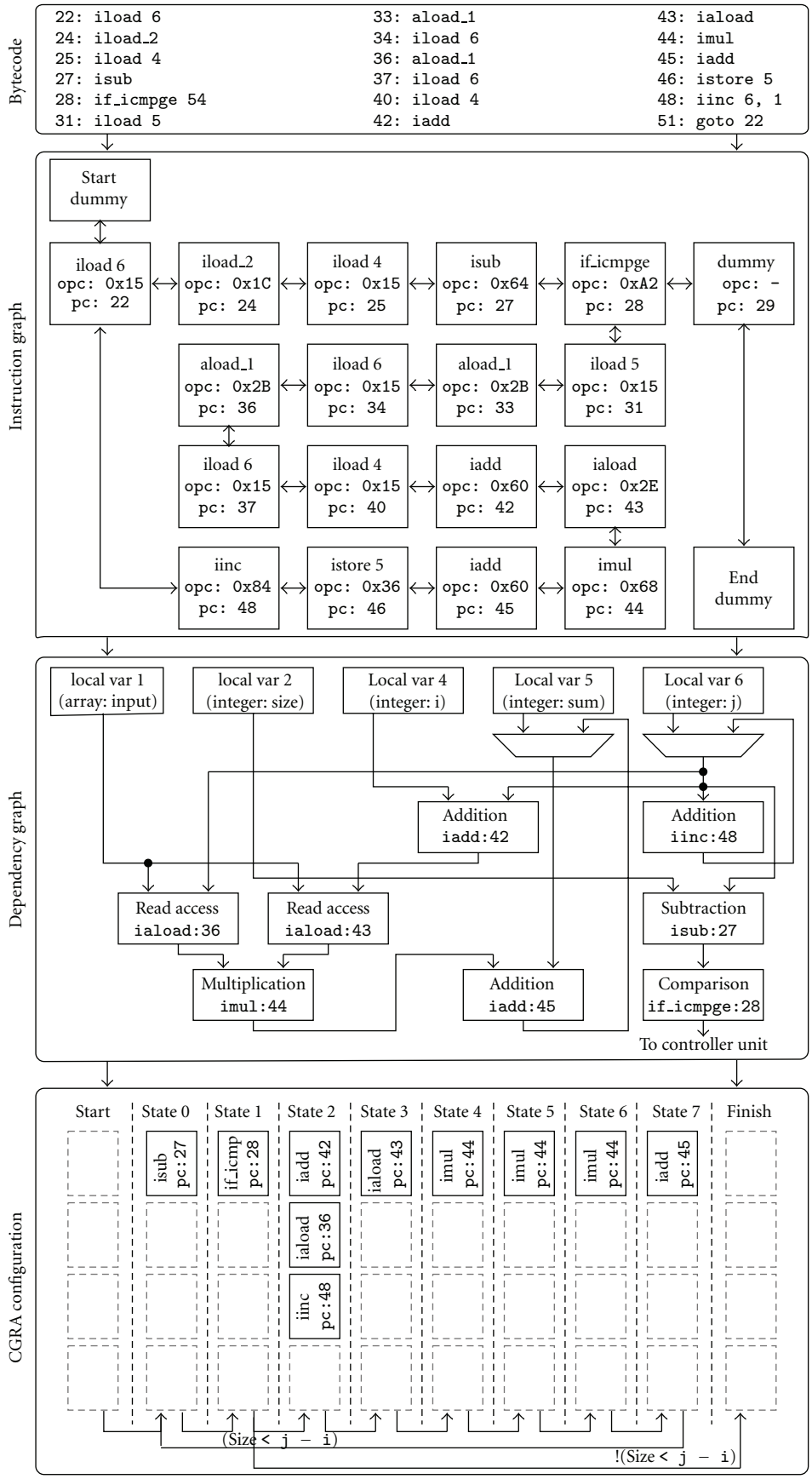


Figure 7: Example of intermediate synthesis results.

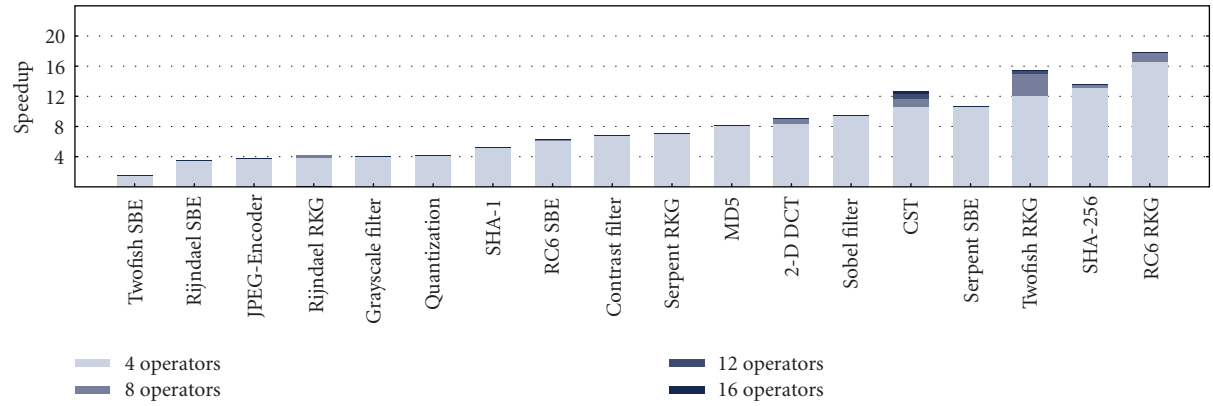


Figure 8: Diagram of runtime acceleration of benchmark applications.

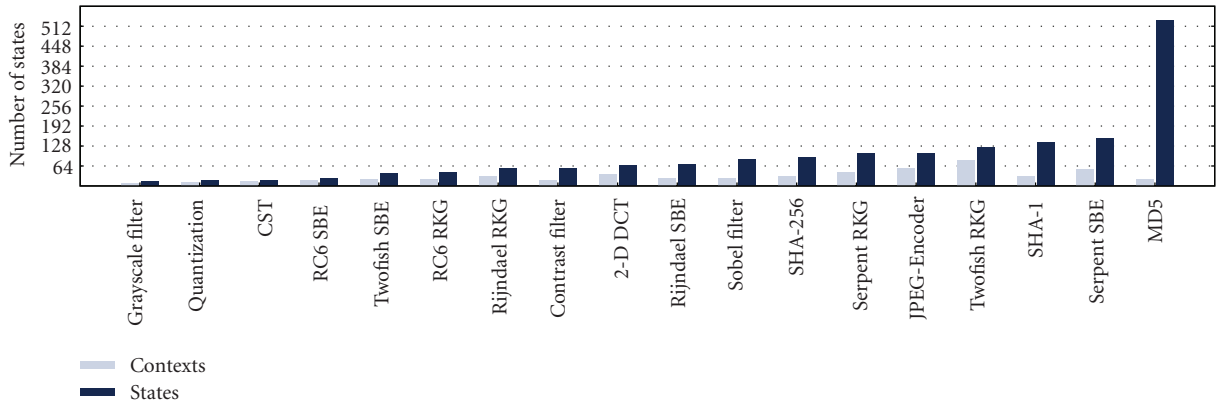


Figure 9: Diagram of complexity of the schedules of benchmark applications.

*6.1. Benchmark Applications.* We chose 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, and furthermore evaluated the runtime behavior of image processing kernels. All of these benchmark applications are pure computation kernels. Regularly, they are part of a surrounding application. Thus, we selected the encoding of a bitmap image into a JPEG image as a benchmark application. This application contains several computation kernels, such as color space transformation, 2-D 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 cipher benchmarks contains the four block ciphers Rijndael, Twofish, Serpent, and RC6, which all were part of the Advanced Encryption Standard (AES) evaluation process.

We analyzed the runtime behavior of 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 selected 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, 2-D 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 of the mentioned processing steps had been executed 120 times for each of

Table 3: Complexity of the schedules of benchmark applications.

(a) Round key generation

| Configuration | Rijndael | | Twofish | | RC6 | | Serpent | |
|---|---|---|---|---|---|---|---|---|
| | States | Contexts | States | Contexts | States | Contexts | States | Contexts |
| 4 operators | 57 | 42 | 230 | 110 | 48 | 21 | 124 | 37 |
| 8 operators | 55 | 31 | 148 | 113 | 44 | 20 | 106 | 43 |
| 12 operators | 55 | 31 | 130 | 91 | 44 | 20 | 103 | 42 |
| 16 operators | 55 | 31 | 122 | 83 | 44 | 20 | 103 | 42 |

(b) Single block encryption

| Configuration | Rijndael | | Twofish | | RC6 | | Serpent | |
|---|---|---|---|---|---|---|---|---|
| | States | Contexts | States | Contexts | States | Contexts | States | Contexts |
| 4 operators | 78 | 37 | 46 | 33 | 25 | 17 | 153 | 54 |
| 8 operators | 71 | 31 | 40 | 26 | 23 | 20 | 152 | 54 |
| 12 operators | 69 | 26 | 40 | 22 | 23 | 19 | 152 | 54 |
| 16 operators | 69 | 23 | 40 | 20 | 23 | 19 | 152 | 54 |

(c) Hash & digest algorithms

| Configuration | SHA-1 | | SHA-256 | | MD5 | |
|---|---|---|---|---|---|---|
| | States | Contexts | States | Contexts | States | Contexts |
| 4 operators | 138 | 29 | 107 | 28 | 531 | 20 |
| 8 operators | 138 | 29 | 92 | 31 | 531 | 20 |
| 12 operators | 138 | 29 | 92 | 31 | 531 | 20 |
| 16 operators | 138 | 29 | 92 | 30 | 531 | 20 |

(d) Image processing

| Configuration | Sobel filter | | Grayscale filter | | Contrast filter | |
|---|---|---|---|---|---|---|
| | States | Contexts | States | Contexts | States | Contexts |
| 4 operators | 86 | 23 | 13 | 9 | 56 | 18 |
| 8 operators | 86 | 23 | 13 | 9 | 56 | 18 |
| 12 operators | 86 | 23 | 13 | 9 | 56 | 18 |
| 16 operators | 86 | 23 | 13 | 9 | 56 | 18 |

(e) JPEG encoding

| Configuration | JPEG-encoder | | Color space transformation | | 2-D forward DCT | | Quantization | |
|---|---|---|---|---|---|---|---|---|
| | States | Contexts | States | Contexts | States | Contexts | States | Contexts |
| 4 operators | 132 | 64 | 22 | 17 | 89 | 43 | 16 | 11 |
| 8 operators | 109 | 61 | 18 | 15 | 70 | 42 | 16 | 11 |
| 12 operators | 110 | 60 | 17 | 15 | 67 | 39 | 16 | 11 |
| 16 operators | 105 | 55 | 17 | 14 | 67 | 36 | 16 | 11 |

the three color components, which resulted in a total of 360 processed input blocks.

*6.2. Runtime Acceleration.* Except for the contrast and grayscale filter, all applications contained either method invocations or access to multidimensional arrays. As we mentioned above, the synthesis does not support these instruction types 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. Speedups between 3.5 and 12.5 were achieved for most kernels. Nonetheless, several applications, for example, SHA-256, gained better results originating from a benefiting communication/computation ratio. The JPEG-encoding application as a whole has gained a speedup of 3.77, which fits into the overall picture.

The encryption of the Twofish cipher is an outlier, being caused by a large communication overhead. This overhead can be reduced by caching objects and arrays inside the CGRA.

In case the cached values did not change since the last usage of a synthesized functional unit, they do not have to be transfered to the reconfigurable fabric again. We evaluated the usefulness of such a caching algorithm [28], but have not extended our synthesis to make use of it yet.

Table 4: Overall utilization of complex processing elements in synthesized functional units.

| Configuration | Contexts | 0 | | ≤1 | | ≤2 | | >2 | |
|---|---|---|---|---|---|---|---|---|---|
| 4 operators | 1913 | 1269 | 66% | 1616 | 84% | 1751 | 92% | 162 | 8% |
| 8 operators | 1748 | 1210 | 69% | 1496 | 86% | 1589 | 91% | 159 | 9% |
| 12 operators | 1725 | 1215 | 70% | 1499 | 87% | 1583 | 92% | 142 | 8% |
| 16 operators | 1710 | 1223 | 71% | 1501 | 88% | 1582 | 92% | 128 | 8% |

Table 5: Largest number of equal operation types executed within a single state on an array with 16 operators.

(a) Round key generation

| Operation Type | Rijndael | Twofish | RC6 | Serpent |
|---|---|---|---|---|
| Combinational | 7 | 13 | 7 | 16 |
| Multiplication | 1 | 15 | 0 | 0 |
| Type conversion | 0 | 1 | 2 | 0 |
| Division | 0 | 0 | 0 | 0 |

(b) Single block encryption

| Operation Type | Rijndael | Twofish | RC6 | Serpent |
|---|---|---|---|---|
| Combinational | 16 | 14 | 9 | 15 |
| Multiplication | 0 | 8 | 4 | 0 |
| Type conversion | 0 | 0 | 0 | 0 |
| Division | 0 | 0 | 0 | 0 |

(c) Hash/digest algorithms

| Operation Type | SHA-1 | SHA-256 | MD5 |
|---|---|---|---|
| Combinational | 12 | 16 | 4 |
| Multiplication | 0 | 0 | 0 |
| Type Conversion | 0 | 0 | 0 |
| Division | 0 | 0 | 0 |

(d) Image processing

| Operation Type | Sobel filter | Grayscale filter | Contrast filter |
|---|---|---|---|
| Combinational | 3 | 5 | 5 |
| Multiplication | 1 | 3 | 3 |
| Type Conversion | 1 | 0 | 3 |
| Division | 1 | 0 | 3 |

(e) JPEG encoding

| Operation type | JPEG-encoder | Color space transformation | 2-D Forward DCT | Quantization |
|---|---|---|---|---|
| Combinational | 7 | 13 | 7 | 16 |
| Multiplication | 1 | 15 | 0 | 0 |
| Type conversion | 0 | 1 | 2 | 0 |
| Division | 0 | 0 | 0 | 0 |

The runtime results for all benchmarks are shown in Figure 8 while the corresponding measurement values are given in Table 2.

*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 a configuration profile for every context, which reflects every operation that is executed within the related state. Accordingly, we removed all duplicates from the set of configurations. The number of remaining elements 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

Table 6: Influence of a specialized operator set with 1 multiplication and 1 division operator on benchmark applications.

(a) Round key generation

| Configuration | Rijndael | | Twofish | | RC6 | | Serpent | |
|---|---|---|---|---|---|---|---|---|
| | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup |
| Plain software | 17760 | — | 525276 | — | 61723 | — | 44276 | — |
| 4 operators | 4913 | 3.61 | 48509 | 10.83 | 3979 | 15.51 | 6429 | 6.89 |
| 8 operators | 4480 | 3.96 | 48236 | 10.89 | 3580 | 17.24 | 6234 | 7.10 |
| 12 operators | 4427 | 4.01 | 48236 | 10.89 | 3580 | 17.24 | 6234 | 7.10 |
| 16 operators | 4427 | 4.01 | 48236 | 10.89 | 3580 | 17.24 | 6234 | 7.10 |

(b) Single block encryption

| Configuration | Rijndael | | Twofish | | RC6 | | Serpent | |
|---|---|---|---|---|---|---|---|---|
| | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup |
| Plain software | 21389 | — | 12864 | — | 17371 | — | 34855 | — |
| 4 operators | 6303 | 3.39 | 8683 | 1.48 | 2995 | 5.80 | 3599 | 9.68 |
| 8 operators | 6002 | 3.56 | 8629 | 1.49 | 2870 | 6.05 | 3219 | 10.83 |
| 12 operators | 6002 | 3.56 | 8620 | 1.49 | 2870 | 6.05 | 3219 | 10.83 |
| 16 operators | 6002 | 3.56 | 8620 | 1.49 | 2870 | 6.05 | 3219 | 10.83 |

(c) Hash & digest algorithms

| Configuration | SHA-1 | | SHA-256 | | MD5 | |
|---|---|---|---|---|---|---|
| | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup |
| Plain software | 23948 | — | 47471 | — | 11986 | — |
| 4 operators | 4676 | 5.12 | 4396 | 10.80 | 1485 | 8.07 |
| 8 operators | 4561 | 5.25 | 3484 | 13.63 | 1485 | 8.07 |
| 12 operators | 4561 | 5.25 | 3484 | 13.63 | 1485 | 8.07 |
| 16 operators | 4561 | 5.25 | 3484 | 13.63 | 1485 | 8.07 |

(d) Image processing

| Configuration | Sobel Filter | | Grayscale Filter | | Contrast Filter | |
|---|---|---|---|---|---|---|
| | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup |
| Plain software | 21124 | — | 236 | — | 608 | — |
| 4 operators | 2370 | 8.91 | 79 | 2.99 | 116 | 5.24 |
| 8 operators | 2246 | 9.41 | 77 | 3.06 | 112 | 5.43 |
| 12 operators | 2246 | 9.41 | 77 | 3.06 | 112 | 5.43 |
| 16 operators | 2246 | 9.41 | 77 | 3.06 | 112 | 5.43 |

(e) JPEG encoding

| Configuration | JPEG-Encoder | | Color space transformation | | 2-D forward DCT | | Quantization | |
|---|---|---|---|---|---|---|---|---|
| | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup |
| Plain software | 17368663 | — | 3436078 | — | 23054 | — | 7454 | — |
| 4 operators | 4868492 | 3.57 | 362450 | 9.48 | 2996 | 7.69 | 1886 | 3.95 |
| 8 operators | 4847803 | 3.58 | 354721 | 9.69 | 2960 | 7.79 | 1816 | 4.10 |
| 12 operators | 4847803 | 3.58 | 354721 | 9.69 | 2960 | 7.79 | 1816 | 4.10 |
| 16 operators | 4847803 | 3.58 | 354721 | 9.69 | 2960 | 7.79 | 1816 | 4.10 |

those two bounds, as it depends on the place-and-route results of the affected operations.

The context informations, for the benchmarks are presented in Table 3 while a graphical representation is given in Figure 9. It shows the size of the controlling finite state machine (States), and the number of actually different contexts (Contexts) for every one of our benchmarks. It shows, that only three of eighteen state machines on an array with 16 processing elements 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.

Table 7: Influence of a specialized operator set with 3 multiplication operators and 1 division operator on benchmark applications.

(a) Round key generation

| Configuration | Rijndael | | Twofish | | RC6 | | Serpent | |
|---|---|---|---|---|---|---|---|---|
| | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup |
| Plain software | 17760 | — | 525276 | — | 61723 | — | 44276 | — |
| 8 operators | 4480 | 3.96 | 37939 | 13.85 | 3580 | 17.24 | 6234 | 7.10 |
| 12 operators | 4427 | 4.01 | 37582 | 14.15 | 3580 | 17.24 | 6234 | 7.10 |
| 16 operators | 4427 | 4.01 | 37582 | 14.15 | 3580 | 17.24 | 6234 | 7.10 |

(b) Single block encryption

| Configuration | Rijndael | | Twofish | | RC6 | | Serpent | |
|---|---|---|---|---|---|---|---|---|
| | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup |
| Plain software | 21389 | — | 12864 | — | 17371 | — | 34855 | — |
| 8 operators | 6002 | 3.56 | 8413 | 1.53 | 2828 | 6.14 | 3224 | 10.81 |
| 12 operators | 6002 | 3.56 | 8404 | 1.53 | 2807 | 6.19 | 3219 | 10.83 |
| 16 operators | 6002 | 3.56 | 8404 | 1.53 | 2807 | 6.19 | 3219 | 10.83 |

(c) Hash & digest algorithms

| Configuration | SHA-1 | | SHA-256 | | MD5 | |
|---|---|---|---|---|---|---|
| | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup |
| Plain software | 23948 | — | 47471 | — | 11986 | — |
| 8 operators | 4561 | 5.25 | 3601 | 13.18 | 1485 | 8.07 |
| 12 operators | 4561 | 5.25 | 3502 | 13.56 | 1485 | 8.07 |
| 16 operators | 4561 | 5.25 | 3502 | 13.56 | 1485 | 8.07 |

(d) Image processing

| Configuration | Sobel filter | | Grayscale filter | | Contrast filter | |
|---|---|---|---|---|---|---|
| | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup |
| Plain software | 21124 | — | 236 | — | 608 | — |
| 8 operators | 2246 | 9.41 | 75 | 3.15 | 112 | 5.43 |
| 12 operators | 2246 | 9.41 | 75 | 3.15 | 112 | 5.43 |
| 16 operators | 2246 | 9.41 | 75 | 3.15 | 112 | 5.43 |

(e) JPEG Encoding

| Configuration | JPEG-Encoder | | Color space transformation | | 2-D forward DCT | | Quantization | |
|---|---|---|---|---|---|---|---|---|
| | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup | Clock ticks | Speedup |
| Plain software | 17368663 | — | 3436078 | — | 23054 | — | 7454 | — |
| 8 operators | 4650780 | 3.73 | 300618 | 11.43 | 2563 | 8.99 | 1816 | 4.10 |
| 12 operators | 4650780 | 3.73 | 300618 | 11.43 | 2563 | 8.99 | 1816 | 4.10 |
| 16 operators | 4650780 | 3.73 | 300618 | 11.43 | 2563 | 8.99 | 1816 | 4.10 |

*6.4. Resource Utilization.* Another characteristic of the synthesized control units, is the distribution of multicycle operations like multiplication, type conversion, or division (complex operations) within the created contexts.

Table 4 shows the aggregate distribution of complex operations within the schedules. It shows a total number of 1913 contexts for all of our benchmarks, as we scheduled them for a reconfigurable array with four operators. Furthermore, it can be seen that a large set of 1269 contexts did not contain any complex operation. Furthermore, the bigger part of the remaining contexts utilized only one or two complex operations, which sums up to 1751 contexts containing two or less complex operations. Hence, only 162 contexts used more than two complex operators.

Entirely, it can be seen that the 1-quantile covers more than 84% of all contexts, regardless of the reconfigurable arrays size. Furthermore, the 2-quantile contains more than 91% of the contexts. Thus, it is reasonable to reduce the complexity of the reconfigurable array, as a full-fledged homogeneous array structure may not be necessary. Hence, the chip-size of the array would shrink. Nonetheless, this would also decrease the gained speedup. The following subsection shows the influence of such a limitation on the runtime and speedup, with the help of small modifications to the constraints of our measurements.

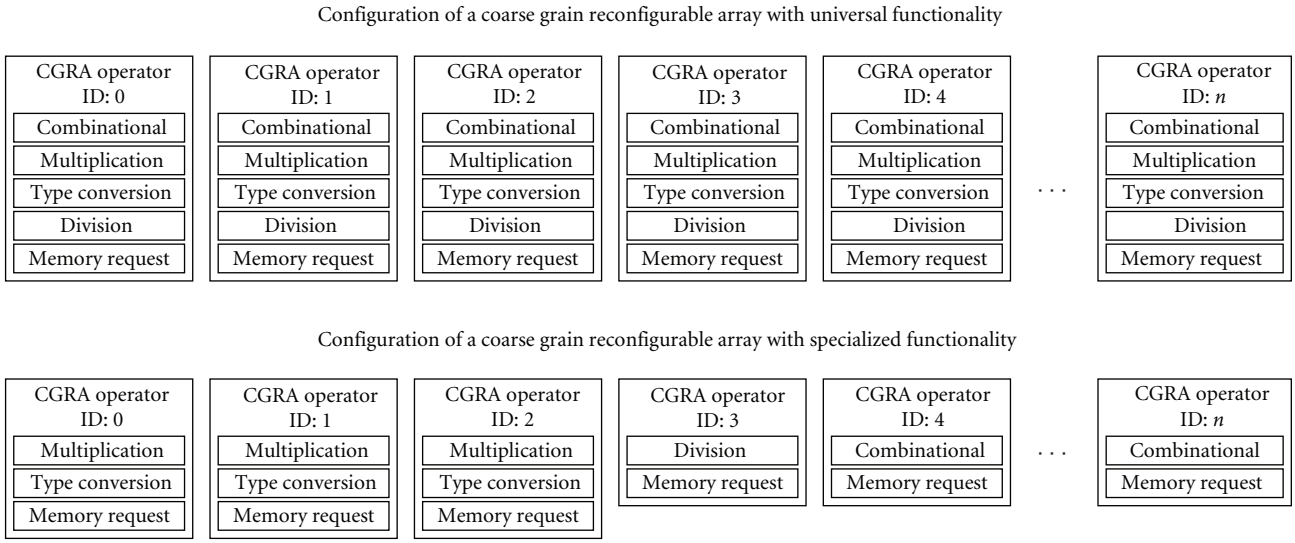


Figure 10: Specialization of an array with 3 multiplication operators and 1 division operator.

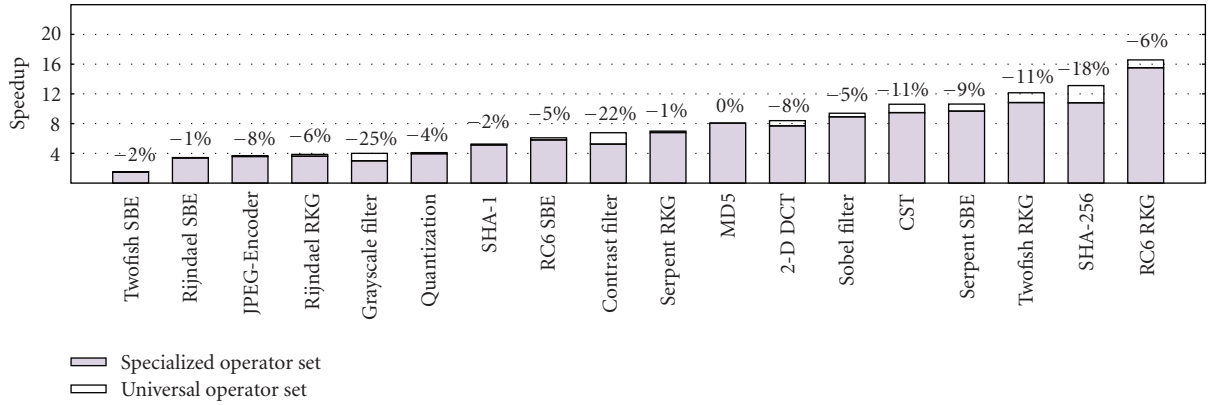


Figure 11: Speedup of benchmark applications on a specialized 4 operator array with 1 multiplication and 1 division operator.

*6.5. Passing from Universal to Specialized Operator Sets.* The results in the preceding subsections suggest the use of a heterogeneous array, as more than 90% of the contexts that were created by our synthesis algorithm used two or less complex operators. Single operators of this array would not provide the full functionality from the preceding measurements, but a specific subset. Thus, the functionality would be distributed all over the array while reducing the operators chip size and resource consumption significantly.

A well-informed decision about the structure of such specialized operators should be based on an analysis of the distribution of operations within the states of the controlling units. Table 5 shows the largest number of equally typed operations that are executed within a single state for all of our benchmarks. It can be seen that most benchmarks very seldom contain type conversion or division operations. Furthermore, a large subset of benchmarks does not utilize more than three multiplication operators in parallel.

This distribution of operations within the created schedules suggests the use of only a single dedicated division operator inside the array. The number of multiplication operations may be confined to one, as only six benchmarks utilize more than one operation at a time. Nonetheless, type conversion operations have to be executed as well. In case a combined operator for those two operation types is established, the number of its instances may be increased up to three.

All other operators up to the arrays size implement combinational operations, which are the most common instruction type inside the schedules. In addition to their specialized functions, all operators are able to generate a memory read/write request. The exemplified structure of a specialized array with a single division operator, three multiplication/type conversion elements, and one up to $n$ combinational operators is sketched in Figure 10.

*6.6. Runtime Impact of Specialized Operator Sets.* In order to analyze the effects of the aforementioned specialization, we reconfigured our array to meet the given constraints. Firstly, we measured the runtime of our benchmarks on an array with a single division operator and a dedicated multiplication/type conversion operator.

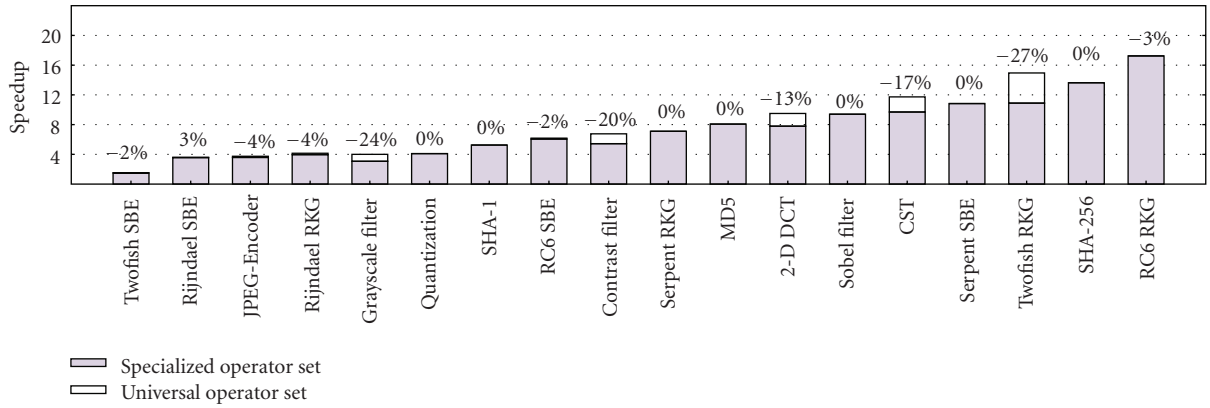


Figure 12: Speedup of benchmark applications on a specialized 8 operator array with 1 multiplication and 1 division operator.

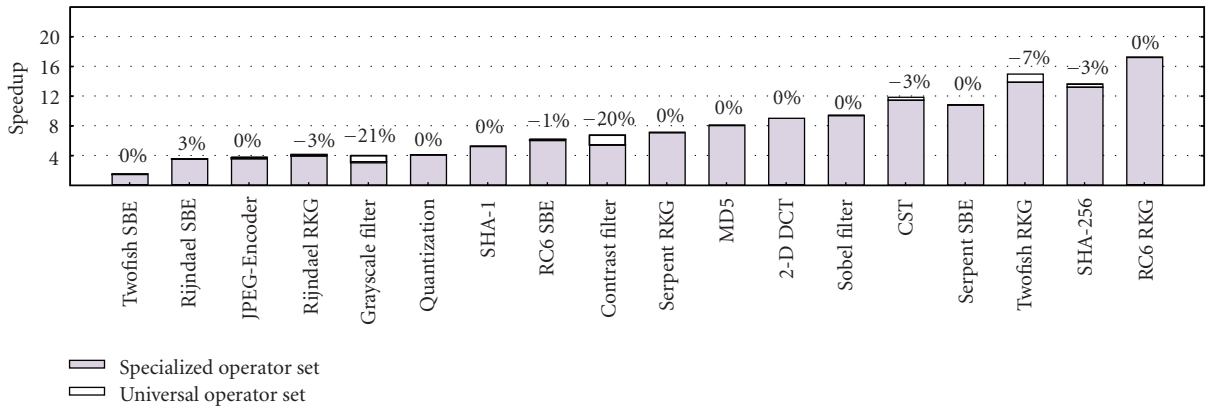


Figure 13: Speedup of benchmark applications on a specialized 8 operator array with 3 multiplication operators and 1 division operator.

The results of the corresponding measurements are shown in Table 6. Just as the measurements regarding an unconstrained reconfigurable array, it can be seen that an array size beyond eight operators does not provide further runtime improvement.

A comparison of the achieved speedups with the results regarding an unconstrained array are displayed in Figures 11 and 12. The annotated percentage numbers display the change of the speedup for a specific benchmark in comparison to its execution at the full-fledged array with the *corresponding size*.

It can be seen that most benchmarks slowed down slightly while only a small number of benchmarks (grayscale filter, contrast filter, color space transformation and twofish round key generation) took speedup losses of two-digit percentage numbers. Furthermore, only two benchmarks on an array with four operators and seven benchmarks on an array with eight operators did not slow down. Due to the heuristic character of the list scheduling, the Rijndael single block encryption slightly improved its runtime.

In a second evaluation iteration, we increased the number of multiplication/type conversion operators to three. Considering the resulting number of four non-combinational operators in this specific setup, it is not possible to evaluate an array of size four, as it does not contain any combinational operators. Thus, none of the benchmarks can be scheduled successfully.

The resulting measurements are displayed in Table 7. Expectedly, there has not been any major runtime improvement beyond an array size of eight operators. A comparison of the achieved speedups with the results regarding an unconstrained array are display in Figure 13. It can be seen that only two benchmarks lose more than 10% of their speedup. Furthermore, the number of lossless benchmark increased to ten out of eighteen.

*6.7. Widening the Memory Bottle Neck.* The previously shown characteristics of the benchmark applications have shown that most operations are executed parallel to others. As many of our benchmarks rely on array operations, it seems reasonable to allow more than one operation at a time to access the object/array memory. This can be achieved by using a dual ported memory inside the reconfigurable array.

We measured the effects of such an improved memory infrastructure on basis of an eight operator array with three multiplication/type conversion operators and a solely division operator. The achieved speedups in comparison to the execution on a similar array with a single ported memory are displayed in Figure 14, while the corresponding measurements are shown in Table 8. It can be seen that

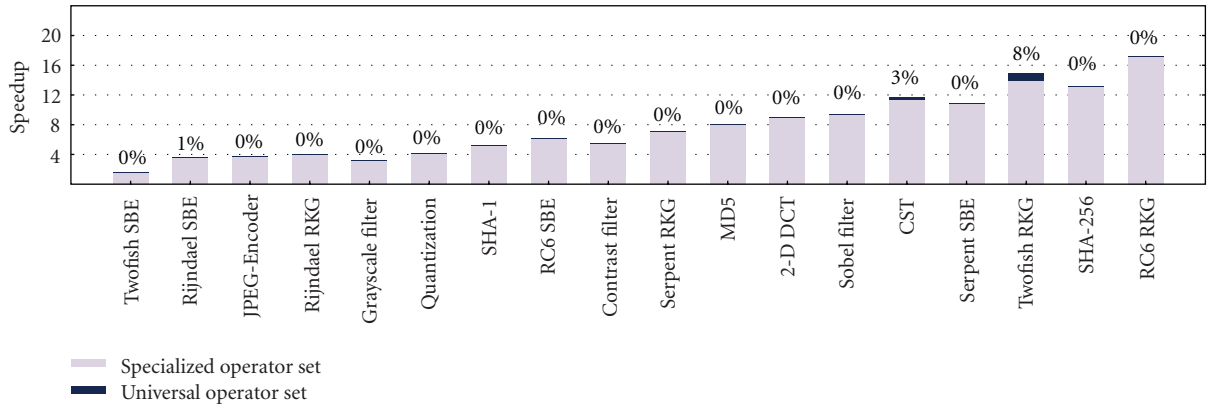


Figure 14: Speedup of Benchmark Applications on a Specialized 8 Operator Array With 3 Multiplication Operators and 1 Division Operator and Dual Ported Memory Acccess.

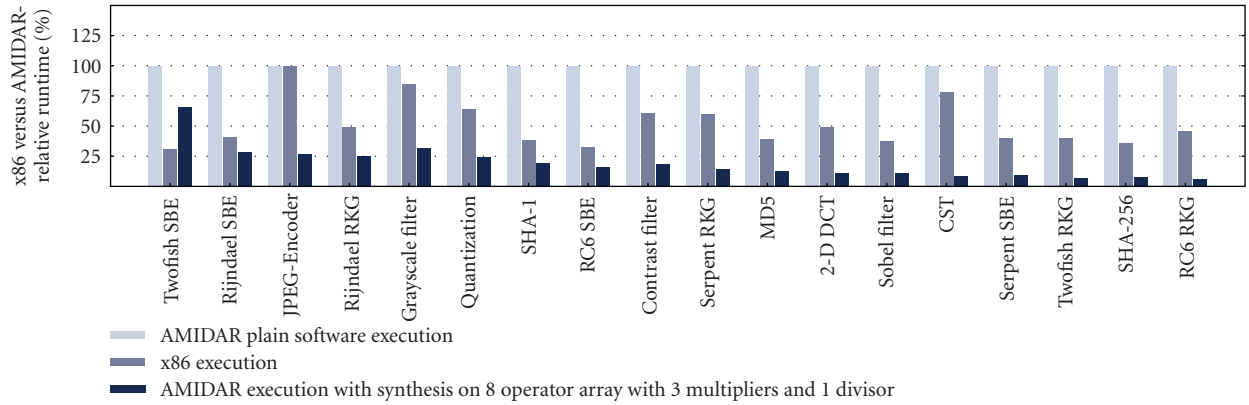


Figure 15: Comparison of AMIDAR plain software execution, x86 execution and AMIDAR execution with enabled synthesis.

only a small number of three benchmarks benefit from the additional memory access operation.

## 7. Conclusion

In this article, we have shown an online-synthesis algorithm for AMIDAR processors. The displayed approach targets maximum simplicity and runtime efficiency of all used algorithms.

It is capable of synthesizing functional units fully automated at runtime regarding given resource constraints. The target technology for our algorithm is a coarse-grain reconfigurable array. Initially, we assumed a reconfigurable fabric with homogeneously formed processing elements and one single shared memory for all objects and arrays. Furthermore, we used list scheduling as scheduling algorithm.

We evaluated our algorithm by examining four groups of benchmark applications. On average across all benchmarks, a speedup of 7.95 was achieved.

Comparing the runtime of the benchmarks, regarding the underlying reconfigurable fabrics size, shows notable results. An array of eight processing elements delivers the maximum speedup for most benchmarks. The improvements gained through the use of a larger array are negligible. Thus, the saturation of the speedup was achieved with a surprisingly moderate hardware effort.

Furthermore, we displayed the complexity of the synthesized finite state machines. This evaluation showed that most of our benchmarks could be driven by less than 128 states and that more than 90% of these corresponding contexts contained two or less complex operations.

Regarding this distribution of non-combinational operations, we proposed to scale down the full-fledged functionality of the reconfigurable array to a set of specialized operators. These operators are capable of executing a dedicated subset of operations.

Subsequently, we have shown the impact of a specialized operator set onto our benchmarks on four and eight operator arrays. Firstly, we reconfigured the array to contain a single multiplication/type conversion operator and a single division operator while all other operators were combinational.

This configuration resulted in speedup losses for nearly all benchmarks on a four-operator array and ranged up to 25%. Additionally, more than half of the benchmarks on an eight operator array were slowed down to 27%.

As a result of these measurements, we increased the number of multiplication/type conversion operators to three. Only two benchmarks have been affected significantly when

Table 8: Effects of dual ported memory access on an 8 operator array with 3 multiplication operators and 1 division operator.

(a) Round key generation

| Benchmark | Clock ticks | Speedup |
|---|---|---|
| Rijndael | 4480 | 3.96 |
| Twofish | 35264 | 14.90 |
| RC6 | 3580 | 17.24 |
| Sepent | 6234 | 7.10 |

(b) Single block encryption

| Benchmark | Clock ticks | Speedup |
|---|---|---|
| Rijndael | 5981 | 3.58 |
| Twofish | 8413 | 1.53 |
| RC6 | 2828 | 6.14 |
| Sepent | 3224 | 10.81 |

(c) Hashes & digests

| Benchmark | Clock ticks | Speedup |
|---|---|---|
| SHA-1 | 4561 | 5.25 |
| SHA-256 | 3601 | 13.18 |
| MD5 | 1485 | 8.07 |

(d) Image processing

| Benchmark | Clock ticks | Speedup |
|---|---|---|
| Sobel filter | 2246 | 9.41 |
| Contrast filter | 112 | 5.43 |
| Grayscale filter | 75 | 3.15 |

(e) JPEG encoding

| Benchmark | Clock ticks | Speedup |
|---|---|---|
| JPEG–Encoder | 4653315 | 3.73 |
| Color Sp. Trans. | 292889 | 11.73 |
| 2-D DCT | 2563 | 8.99 |
| Quantization | 1816 | 4.10 |

using this configuration while more than half of the benchmarks did not sustain any losses.

In a final test series, we assumed a dual ported memory inside the reconfigurable array, instead of a single ported memory. This allows an improved scheduling of memory access operations and is supposed to improve the benchmarks runtime. Nonetheless, this approach delivered negligible runtime improvements on only three of the eighteen benchmarks. All other applications have not been affected.

From the presented evaluation results, an array with eight specialized operators (three dedicated multipliers/type converters, one dedicated divider and five combinatorial operations) seems to be the best compromise between speedup and area. Only few applications seem to benefit from two concurrent memory access operations, so a single memory operation should be sufficient.

Furthermore, diagram shows the compared runtime of the AMIDAR plain software execution, the already mentioned benchmark execution on a x86 architecture, and, finally, the execution of an AMIDAR processor with the proposed CGRA extension. It can be seen that most benchmarks outperform the x86 execution. Regarding the achieved speedup of 7.95 across all applications and the plain software execution time, the AMIDAR execution with an enabled synthesis is approximately four times faster than the execution on an x86 processor.

## 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 contains the implementation of access to multidimensional arrays and automatic inlining of invoked methods at synthesis time. Additionally, we are going to explore the effects of instruction chaining in synthesized functional units, as well as the overlapping of a data transfer to a synthesized functional unit and its execution.

Larger numbers of processing elements within the CGRA currently do not seem to have a substantial effect. We hope to improve the usefulness of larger arrays by employing a simplified version of software pipelining.

Also, the interaction of simplified place & route tools and the underlying routing architecture of the CGRA will be an important field of research.

Currently, we are able to simulate AMIDAR processors based on different instruction set architectures, such as LLVM-Bitcode, .NET Common-Intermediate-Language, Dalvik-Executables, and Java Bytecode. In the future, we are planning to investigate the differences in execution of those instruction sets in AMIDAR-processors.

## References

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

[2] S. Gatzka and C. Hochberger, "The organic features of the AMIDAR class of processors," in *Proceedings of the International Conference on Automation, Robotics and Control Systems (ARCS '05)*, pp. 154–166, 2005.

[3] S. Gatzka and C. Hochberger, "Hardware based online profiling in AMIDAR processors," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS '05)*, p. 144b, April 2005.

[4] S. Döbrich and C. Hochberger, "Low-complexity online synthesis for amidar processors," *International Journal of Reconfigurable Computing*, vol. 2010, Article ID 953693, 15 pages, 2010.

[5] S. Döbrich and C. Hochberger, "Practical resource constraints for online synthesis," in *Proceedings of the 5th International Workshop on Reconfigurable Communication-Centric Systems on Chip (ReCoSoC '10)*, pp. 51–58, 2010.

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

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

[8] 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.

[9] 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.

[10] N. Kasprzyk and A. Koch, "Advances in compiler construction foradaptive computers," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 26–29, Las Vegas, Nev, USA, 2001.

[11] 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.

[12] 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.

[13] 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.

[14] Y. Chou, P. Pillai, H. Schmit, and J. P. Shen, "PipeRench implementation of the instruction path coprocessor," in *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '00)*, pp. 147–158, Monterey, Calif, USA, December 2000.

[15] 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.

[16] 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.

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

[18] F. Bouwens, M. Berekovic, B. De Sutter, and G. Gaydadjiev, "Architecture enhancements for the ADRES coarse-grained reconfigurable array," in *Proceedings of the 3rd International Conference on High-Performance Embedded Architectures and Compilers (HiPEAC '08)*, pp. 66–81, Springer, Berlin, Germany, 2008.

[19] 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.

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

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

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

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

[24] 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.

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

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

[27] 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.

[28] S. Döbrich and C. Hochberger, "Predicting hardware acceleration through object caching in AMIDAR processors," in *Proceedings of the International Conference on Automation, Robotics and Control Systems (ARCS '10)*, pp. 162–171, 2006.