Engineering an Optimized Instruction Set Architecture for AMIDAR Processors

Alexander Schwarz(✉) and Christian Hochberger

Technische Universität Darmstadt, Merckstr. 25, 64283 Darmstadt, Germany
{schwarz,hochberger}@rs.tu-darmstadt.de

Abstract. Newly developed instruction set architectures are nowadays typically based on the RISC principle. Yet, more abstract instruction sets also have their advantages. In the AMIDAR project Java Bytecode was used as the instruction set. Instructions are realized as compositions of *micro instructions* that are distributed to specialized functional units. An explicit timing of these micro instructions is not necessary in AMIDAR processors. This simplifies the conversion of compute intense instruction sequences into hardware structures while the system is running. The relatively high abstraction level of the Bytecode facilitates the analysis and synthesis remarkably. Yet, the native execution of the Bytecode comes with a number of drawbacks. In this contribution, we show a new instruction set architecture that preserves the high abstraction level of Bytecode while at the same time avoiding inefficient data transports. We show that on average the new instruction set reduces the number of clock cycles for our benchmark set by a factor of 3.

Keywords: Instruction set architecture · Microarchitecture · Self-timed · Java processor · Online synthesis

1 Introduction

Most new developments in the area of microprocessors use RISC instruction sets. The RISC nature of instruction sets eases decoding and creation of pipelines. On the down side, analyzing such instruction sequences can be very difficult. An instruction set with higher abstraction level will provide more specialized and targeted instructions. Thus, it will be easier to reverse engineer the intention of the programmer.

This is an essential property if we consider dynamic software/hardware migration. In AMIDAR processors, such online synthesis is one major factor for an efficient application execution. Existing AMIDAR processors use Java Bytecode as their instruction set. While we could demonstrate that it is very suitable for an online synthesis into HW structures, we also found that Bytecode makes excessive use of the stack and the local variable memory. Many of these data transports could be avoided.

The motivation to use Java Bytecode as instruction set is twofold: 1) Android based smartphones are programmed with languages that generate Java Bytecode

A. Brinkmann et al. (Eds.): ARCS 2020, LNCS 12155, pp. 124–137, 2020.
https://doi.org/10.1007/978-3-030-52794-5_10

which is then converted into Dalvik executables. Alternatively, a true Java Bytecode processor like an AMIDAR processor could be used in such platforms. 2) Java as programming language has gained a lot of attention for embedded systems due to its inherent safety features. Consequently, real HW implementations of Java Bytecode processors exist and are in commercial use.

In this contribution, we present a novel instruction set architecture (ISA) that preserves the high abstraction level of Java Bytecode, while at the same time reducing the amount of data transports to a minimum. On average, the resulting instruction set can be executed with less than one third of the original AMIDAR clock cycles.

The paper is structured as follows. Section 2 explains the AMIDAR principle. In Sect. 3 we explain the design of the new ISA (requirements, basic concept, code generation and binary format). Section 4 presents challenges together with our solutions for many detail problems with the new ISA. An evaluation of the new ISA is shown in Sect. 5. Finally, a conclusion and an outlook are given.

2 The AMIDAR Principle

AMIDAR [3] processors are composed of multiple functional units (FUs) which work independently of each other. Independence of FUs is a major strength of AMIDAR. It facilitates hardware design and provides opportunities for runtime reconfiguration. Figure 1 illustrates the structure of a Java processor. The Token Machine is a special FU which decodes program instructions into tokens and sends them to FUs using the token distribution network. Tokens contain the information which operations to execute and where to send the results to. Data can be exchanged between FUs using a data interconnect.

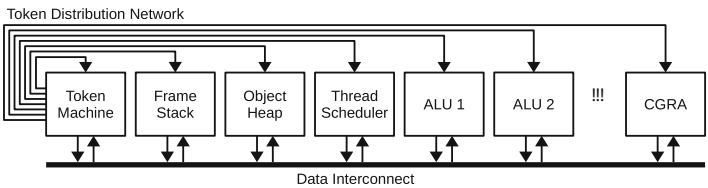

Fig. 1. Structure of the processor

In previous versions of AMIDAR processors, tags are used for synchronizing data with FU operations. Using this technique, every token and data packet contains a tag number. The receiving FU compares the tags of data packet and next token to execute. Only if both tags are equal, the data packet is accepted. Otherwise, the sender has to retry until data transmission is successful. Consequently, no assumptions about the timing of FUs are required and communication between FUs is self-timed.

Apart from its role as instruction decoder, the Token Machine executes all operations which change control flow like branches and method invocations. Furthermore, it provides constants contained in the code. The Frame Stack FU stores for each thread a stack of method frames. Every frame comprises a section for local variables and an operand stack. The Object Heap FU stores objects, arrays and static variables. The Thread Scheduler FU decides which thread to execute and provides thread synchronization using monitors. Several ALUs exist for integer and floating point arithmetic. A coarse grained reconfigurable array (CGRA) is used as flexible hardware accelerator [10].

3 Design of the New ISA

3.1 Motivation

The Frame Stack FU has been identified as bottleneck in previous versions of AMIDAR processors, which use Java Bytecode as their instruction set. Most instructions access the operand stack for reading or writing data. This results in many transfers from and to the Frame Stack FU. Consequently, reducing these data transfers is the main motivation for developing a new ISA.

3.2 Requirements

The processor should run Java programs on a high level of abstraction, like the previous AMIDAR implementation. However, these programs should be executed with higher performance by eliminating unnecessary data transfers. Reconfiguration features like dynamic software/hardware migration should still be supported. This leads to the following requirements.

- Data should be transmitted directly between FUs whenever possible without using intermediate storage. Thereby, execution time and energy consumption are reduced (see Sect. 5).
- Hardware requirements should be moderate. Reducing hardware requirements to an absolute minimum is not the goal. Complex FU operations should still be supported to provide fast execution of programs. On the other hand, complex and energy-consuming techniques for dynamic scheduling and data synchronization should be avoided (see Sects. 4.3 and 4.4).
- Instruction encoding should be compact in order to avoid a bottleneck between code memory and instruction decoder. However, an increased code size in comparison to Java Bytecode can hardly be avoided because small code size is a major strength of Bytecode due to the stack principle (see Sect. 5).
- Arbitrary complex control flow which is expressible in high level languages should be supported (see Sect. 3.3).
- No assumptions about FU timing behavior should be required, neither during code generation nor during token generation (See Sect. 4.3).
- The token generator should have the freedom to assign operations to different FUs as another means for runtime reconfiguration. This assignment should not be fixed by the programmer or code generator (see Sect. 4.5).

3.3 Basic Concept

The basic idea of the new ISA is to specify data flow between instructions explicitly instead of using an operand stack. Each instruction which produces a result specifies another instruction which will receive this result. The four components of an instruction are shown in Fig. 2. Every instruction specifies the operation to execute. Some operations require an additional constant. The *result reference* specifies the instruction which will receive the result. It consists of an instruction offset and a port. The offset is relative to the current instruction in order of execution. A value of 0 references the instruction which is executed next. It is important to note that the static position in the code is not relevant for this offset. Many operations require more than one operand. Therefore, a port number is used in the result reference to specify which of these operands is sent.

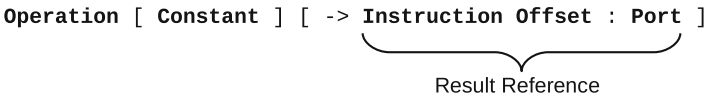

Fig. 2. Assembler representation of one instruction

An example of the resulting code is given in Fig. 3 together with an illustration of control and data flow. The first instruction sends the constant 10 to port 0 of either the `add` or the `sub` instruction. The `read` instruction in line 2 obtains a value from scratch pad memory address 3 and sends it to port 1 of the `brg` instruction in line 4. The next instruction in line 3 sends a value from scratch pad memory to port 1 of either the `add` or the `sub` instruction. The branch instruction in line 4 determines which of both is executed by comparing the received value with zero. Both the `add` and the `sub` instructions send their result to port 0 of the `mul` instruction in line 11. As only one of both is executed, the `mul` instruction receives exactly one value at its port 0. This value is multiplied with the value read from scratch pad memory address 4. The result is written back to the same address.

This example shows some important features of this kind of data flow description. Every value which is produced by an instruction must have exactly one receiver on every possible path of the program. Furthermore, every instruction must receive exactly one value on each of its ports on every possible path of the program. Every port of an instruction can behave like a ϕ function as known from static single assignment (SSA) forms in compiler engineering. Port 0 of the `mul` instruction is an example for this. Either the result of the addition or of the subtraction is received depending on the previously executed program path.

3.4 Code Generation

Code for the new ISA can be generated from two types of sources. The first type is assembler code. As the processor operates on a similar level of abstraction as

```
1    imm  10 -> 3:0
2    read  3 -> 1:1
3    read  2 -> 1:1
4    brg  L1
5    add     -> 2:0
6    goto L2
7  L1:
8    sub     -> 1:0
9  L2:
10   read  4 -> 0:1
11   mul     -> 0:0
12   write 4
```

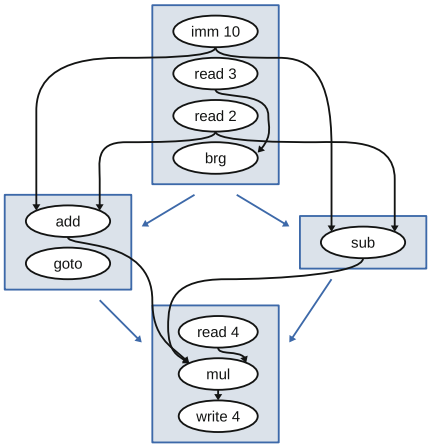

Fig. 3. Basic code example (black: data flow, blue: control flow) (Color figure online)

Java Bytecode, meta-information like class structures is part of this code. Bodies of methods are filled with instructions in assembler representation as defined in Fig. 2. An assembler has been engineered which converts a set of assembler files to a single binary named *New AMIDAR Executable* (NAX). This binary contains all information which is required to execute a program on a hardware implementation of the processor.

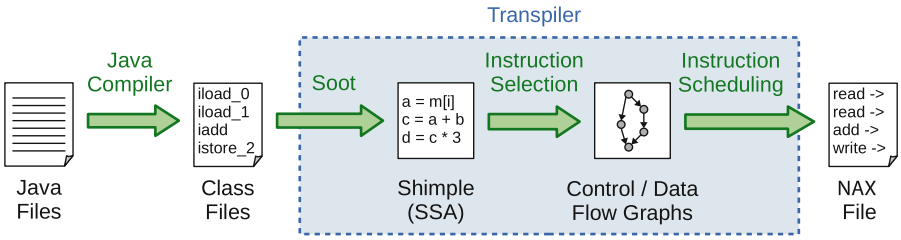

Fig. 4. Code generation from Java source code

The second much more useful type of source code is Java code. The corresponding tool flow is depicted in Fig. 4. A standard Java compiler produces class files from Java source code. A newly developed transpiler converts a set of class files to a NAX file. Figure 4 also shows a simplified version of this transpilation process. The Java analysis and optimization framework Soot is used to convert Bytecode from class files to an SSA form called Shimple [6]. Instruction selection creates a control flow graph for each method of the Shimple representation. Each node of such a control flow graph in turn points to a directed acyclic graph (DAG) defining data dependencies between instructions in the corresponding

block. Instruction scheduling orders the instructions in each block to respect dependencies implied by the DAG on the one hand and hardware restrictions on the other hand.

3.5 Binary Format

A binary format for the instructions has already been defined as shown in Fig. 5. Every instruction has a width of 24 bits. This is the smallest multiple of one byte which can store all relevant information and leaves small room for extensions. Code is stored in an external DRAM which is accessed using a 32 bit AXI interface. A sequence of 32 bit words is converted to a sequence of 24 bit instructions in the instruction fetch stage of the Token Machine.

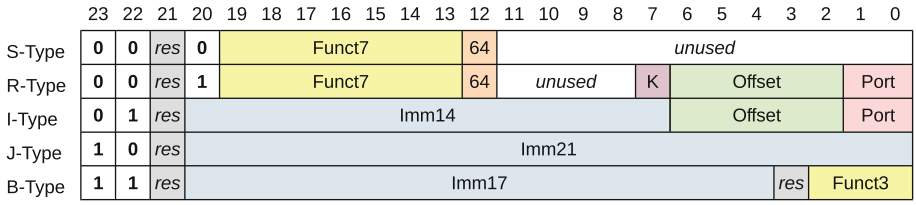

Fig. 5. Binary format of instructions

Five types of instructions exist. The type is encoded in the highest bits. Bit 21 is reserved for future extensions.

- S-type is used for normal instructions which do not produce results. This is typically the case for memory store operations. The *Funct7* field holds the operation. Bit 12 distinguishes between 32 bit and 64 bit operations.
- R-type is used for normal instructions which produce results. This type contains the same fields as S-type plus instruction offset and port for specifying the result reference. Bit 7 is set to 1 if the result should be kept in the output queue as explained in Sect. 4.1.
- I-type is used for sending constant values. Constants up to 14 bits can be stored in the *Imm14* field. Larger constants must either be computed or stored in the constant pool. Special operations exist for loading these constants from the pool.
- J-type is used for unconditional jumps. The *Imm21* field holds the address of the jump target relative to current position in the code.
- B-type is used for conditional branches. The *Imm17* field again holds the relative address of the target. The comparison which decides whether the branch is taken or not is encoded in field *Funct3*. Bit 3 is reserved for future extensions.

4 Challenges

Realization of this ISA has been started by implementing an assembler and a software simulator. Afterwards, the transpilation process has been developed to be able to write programs in Java. All design choices have been taken with possible hardware implementations in mind. This section depicts some of the challenges which have been encountered on this way and their solutions.

4.1 Duplicating Data

As already mentioned in Sect. 3.3 each result must have exactly one receiving instruction. However, one value might be required as operand for multiple operations. Two mechanisms are provided to solve this problem. The first one is a small scratch pad memory, which is implemented as additional functional unit. Values can be written to it and can be read multiple times using addresses. Nevertheless, this contradicts the original idea of transferring data directly between FUs without intermediate storage. Using the second mechanism, instructions can specify that their result should not be removed from the output queue of the sending FU. Afterwards, a special `send_again` instruction can be used to send this value again to another receiver. If instructions are close together and only few copies are required, the last mechanism is preferred. Otherwise, scratch pad memory is used. The generic structure of an FU is shown in Fig. 6 and is explained in Sect. 4.3.

4.2 Discarding Data

Conversely, it is beneficial in some situations to discard data explicitly. For example, if control flow branches and a value is only required in one branch, the processor must be instructed to discard this value. In register based architectures this is done implicitly by overwriting registers. In this ISA the `nop` instruction can be used for this purpose. When a result is targeted to such an instruction during execution, the sending FU is informed to remove the value from its output queue without sending it.

4.3 Data Synchronization

The tag mechanism used for data synchronization in the previous AMIDAR implementation has several disadvantages. Firstly, data must be resent frequently in some situations, which results in lost bus cycles. Secondly, concurrency is limited because only data packets are accepted which match the next token to execute. Thirdly, depending on size and topology of the interconnect, tag comparators can be part of a long combinatorial path starting from the sending FU via the comparator of the receiving FU back to the acknowledgment signal of the sending FU.

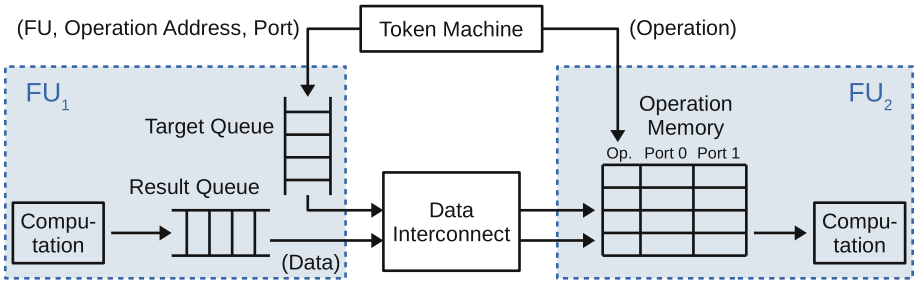

Fig. 6. Hardware components for synchronizing data

Consequently, a new synchronization mechanism has been invented. It uses explicit *operation addresses* to match data and operations. The important hardware components for this mechanism are depicted in Fig. 6. The first thing to note is that tokens are transferred to an FU in two parts. The operation code is sent as soon as the instruction has been decoded. The target information is sent afterwards when the receiving instruction has been decoded.

It is assumed, that an instruction IN_1 has already been decoded which results in operation OP_1 to be executed on FU_1. The result of OP_1 has already been computed and stored in the *result queue* of FU_1. IN_1 references instruction IN_2 as receiver for its result.

Now, the Token Machine decodes IN_2 and sends the corresponding operation OP_2 to the *operation memory* of FU_2. A line of this memory consists of an operation and one data word for each port of FU_2. An operation is stored together with its operands in one line. The address of a line is named *operation address*. Operations are written and read cyclically. Before the next operation can be written to a line, this line must be read and sent to execution. Hence, operation storage has FIFO semantics. Instructions which are mapped to the same FU are executed in the order they are decoded. In contrast, operands can be stored to the memory in any order using operation address and port. An operation can only be sent to execution when all its operands have been stored to the memory.

The Token Machine has an operation counter for FU_2 which is in sync with the operation write address of the operation memory. Therefore, the token machine knows the operation address of OP_2. It sends FU address and operation address of OP_2 together with the result port specified in IN_1 to the *target queue* of FU_1. The token machine sends operations and corresponding target information in the same order. As a consequence, the entries at the front of result queue and target queue belong to each other. They are removed simultaneously and sent via data interconnect to FU_2 where the data word is written to the memory location given by operation address and port.

Decoding is blocked when the target queue is full or no free operation address is available. As an operation is always sent before the target information pointing to this operation, it can be guaranteed that free space is always available in the

operation memory when sending a data word. Consequently, no acknowledgment signal is required from the receiver to the sender.

Sizes of result queues and numbers of lines in the operation memories are free parameters which still have to be optimized. These parameters must be known for instruction scheduling. The values assumed during code generation may be lower than those provided by hardware.

4.4 Target Resolution

After the Token Machine has decoded an instruction and has assigned it to an FU, it must resolve the result reference and send this target information to the FU. The required hardware components are illustrated in Fig. 7.

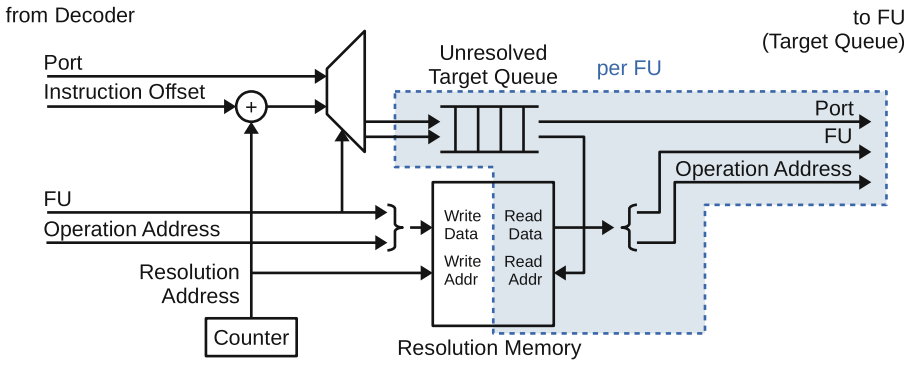

Fig. 7. Hardware components for resolving result references

The main component is the *resolution memory*. It stores FU and operation addresses of the instructions which have been decoded last. The number of addresses in this memory is a free parameter and limits the distance of result references between instructions. A counter generates the *resolution address* for each decoded instruction. It serves as write address for the resolution memory.

Now assume instruction IN_1 has just been decoded and assigned to FU_1 with resolution address RES_1. Its instruction offset points to instruction IN_2, which will be executed by FU_2 with resolution address RES_2. Port and offset are directly extracted from the instruction. Adding RES_1 to the offset yields RES_2, which is stored in the *unresolved target queue* of FU_1 together with the port.

When IN_2 is decoded, its FU and operation addresses are stored to the resolution memory at address RES_2. At the same time, RES_2 is located at the front of the unresolved target queue. As a consequence, the resolution memory is read from this address. The circuit detects when FU and operation addresses of IN_2 are available and sends this information together with the port to the target queue of FU_2. The most significant bit of the resolution address is not

used for addressing the memory but as tag for the memory contents. This allows to detect when new information has been written.

4.5 Instruction Scheduling

Instruction scheduling is a more complex task in comparison to register based architectures. Several constraints beyond data and control dependencies between instructions must be considered to produce executable code.

- Instruction offsets are limited by the binary instruction format and the size of the resolution memory in the Token Machine.
- When a result is sent over a branch to different (exclusive) instructions, these instructions must have the same distance to the sender because the sender can only specify one instruction offset. This can be seen in Fig. 3. If a `nop` would be inserted before the `sub` instruction, the operands of the subtraction would not be received.
- The code must be free of deadlocks. If no care is taken, deadlocks are easily produced, which cause the processor to stop. As this constraint is the most difficult to handle, it is explained in more detail.

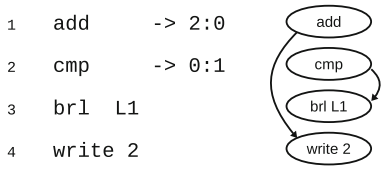

Fig. 8. Deadlock example

Figure 8 shows an example for a deadlock. Inputs of addition and comparison are not shown because they are not relevant for the deadlock. Both addition and comparison are executed on the same FU. Consequently, the result of the addition is placed at the front of the output queue, the result of the comparison behind it. However, the receiver of the addition result cannot be resolved because the branch has not been evaluated yet. Therefore, this result cannot be removed from the output queue. The branch in turn is waiting for the result of the comparison, which is blocked by the result of the addition. A cycle of dependencies is produced, which causes the processor to stop. A simple solution for this deadlock is to change the order of addition and comparison.

There are many more constellations causing deadlocks. They can be statically detected by building dependency graphs and searching for cycles in these graphs. Theoretically, a dependency graph must be constructed for each possible execution path in a program. Calling convention ensures that no deadlocks can appear across method boundaries. Consequently, methods can be analyzed separately.

Loops still produce an infinite number of paths. However, result references are limited to the current or the next loop iteration. Therefore, no additional deadlocks can appear after analyzing two loop iterations. The number of paths can still grow exponentially. In practice, this problem is solved using a sliding window algorithm. The window slides along the control flow of the method. Whenever the next instruction is added to the window, cycles are searched and removed. Afterwards, instructions which can be proven not to cause new deadlocks are removed from the window. When the algorithm detects that a window position has already been encountered, analysis of this path can be finished. While the problem still has exponential complexity, this algorithm finds all deadlocks in reasonable time even in methods with very complex control flow.

Different actions for removing deadlocks have been implemented. A suitable action is chosen depending on the deadlock constellation. In contrast to finding deadlocks, the problem of removing deadlocks has not been fully solved yet. In some situations, the scheduler fails to produce code free of deadlocks and informs the user about it. Current research investigates different approaches for systematically resolving all deadlocks.

A special `forward` operation is available to facilitate instruction scheduling. It just forwards the received input to another instruction. In hardware, forwarding is done by a separate FU, which helps to fulfill the mentioned constraints.

No new dependencies are introduced if two instructions are executed on distinct FUs instead of on a single FU. Hence, this cannot cause new deadlocks. Consequently, exact assignment of operations to FUs is not required for deadlock analysis. It must only be guaranteed that certain categories of instructions will not be executed on the same FU.

5 Evaluation

The benchmark set used for evaluation comprises 9 encryption algorithms, 7 hash algorithms, and 4 image filters. Additionally, ADPCM encoding/decoding, JPEG encoding, and regular expression matching have been evaluated. Execution times have been determined using simulators which imitate hardware behavior. FU timings of the existing hardware implementation are applied. Each benchmark has been executed once in the simulator for the Bytecode based AMIDAR processor and twice in the simulator for the new ISA. In the last case, benchmarks have been executed with 1 and 2 instructions decoded in parallel. Afterwards, the speedup has been calculated. For the new ISA, the following parameters have been chosen, which seem to be minimal values for reasonable execution.

- Resolution Memory Size: 16
- Operation Memory Size (all FUs): 8
- Output Queue Size (all FUs): 4

Figure 9 illustrates the speedups for all benchmarks. An average speedup of 3.69 is achieved in the single issue case and 4.64 in the dual issue case. Hence, a

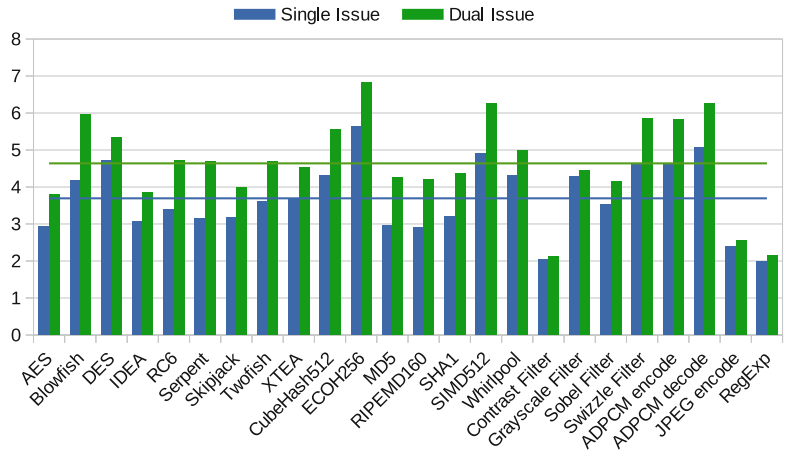

Fig. 9. Speedups achieved in comparison to Bytecode (simulated)

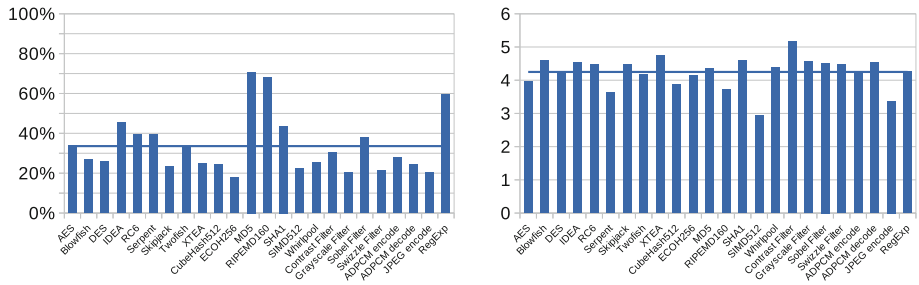

Fig. 10. Data transfers between FUs (left) and code size (right) in comparison to Bytecode

significant speedup in comparison to Java Bytecode can be noted. Furthermore, dual issue is clearly advantageous for this ISA.

However, some benchmarks differ from the average. The first exception is *Contrast Filter*, which uses floating point operations. As these operations are time consuming, speedup achieved by an improved ISA is lower. *JPEG encode* is a complex, data dominated algorithm, which is split across multiple methods. This lowers the speedup to 2.55. *RegExp* is an example for control flow dominated algorithms with many method invocations. It shows a speedup of 2.14.

The main reason for high speedups is the reduced number of data transfers between FUs. They are decreased by a factor of 2.98 in average as shown in Fig. 10. This has been a major design goal as defined in Sect. 3.2. On the other hand, code size grows by a factor of 4.25 in average, which is caused by an increased size and number of instructions.

All benchmarks require less than 10 s for code creation from class files, with the exception of *SIMD512* (40 s) and *RIPEMD160* (15 s). In this benchmark

set a substantial number of methods from the Java standard library has to be included in the binary file. Measurement has been carried out on an Intel Core i7-6700 CPU with 16 GB RAM and a Java 1.8 HotSpot JVM on Ubuntu 16.04.

6 Related Work

AMIDAR processors use principles from dataflow machines [4]. Thus, often a comparison is made with such processors. In contrast to such machines, AMIDAR avoids the known issues with typical dataflow machines [7]:

- Broadcasting of tokens is done only for a handful of FUs. Thus, handling of tokens is not a problem.
- In a dataflow machine, the availability of input data must be checked for a huge set of operations concurrently. This is often done using costly content addressable memories. In AMIDAR, the availability of input data needs to be checked only localy inside of a functional unit. Thus, it can be implemented much more efficiently.
- Dataflow machines can suffer from deadlocks, if the program is not composed in a proper way. Such situations are not easy to detect and thus greatly complicate the compiler.

Even if dedicated dataflow processors are not longer researched due to the mentioned problems, dataflow is still used in scientific computing approaches. Maxeler uses a Java-like language to generate dataflow graphs and a compiler maps those graphs onto a set of field programmable gate arrays [2]. The big drawback of this approach is its inability to execute regular code. It is only efficient in high-throughput computing.

In AMIDAR, FUs synchronize with each other by the exchange of data. In a similar manor, Transport Triggered Architectures (TTA) [1] use the transport of data to start new operations. Nevertheless, AMIDAR provides more elasticity, since it allows arbitrary execution time for an FU without the need to adjust the microinstructions. In contrast, TTAs require exact knowledge of the FU timing, since the result of an operation must be moved to its destination at the proper time. Even worse is the problem of the huge code memory of TTAs. In order to provide a high degree of parallelism, TTAs must be able to control as many independent data transports as possible. This results in very wide instructions which in turn need a large code memory. Unfortunately, the majority of the code uses only few of the possible transport slots. Approaches have been published that reduce this memory size by means of compression [5]. AMIDAR avoids the huge code memory in a different way by generating the token sets on the fly from a more abstract instruction set.

Finally, one could think about other instruction set architectures than Java Bytecode. Candidates could be Low Level Bit Code [8] from the LLVM framework, Common Intermediate Language [9] from the .NET framework. They share approximately the same abstraction level. Yet, it turns out that both come with severe drawbacks compared to the Java Bytecode. Compute instructions in CIL

and LLVM Bit Code do not contain type information. Thus, the required type of operation (int, float, double) has to be reconstructed from the sources of the data. In the worst case, they need to be combined with type conversions at runtime.

7 Conclusion and Future Work

In this work, a promising novel ISA for AMIDAR processors has been presented. It borrows ideas from data flow architectures and in simulation shows significant speedups compared to Java Bytecode as ISA. Through thorough engineering we were able to fulfill almost all requirements that were defined. Only code size leaves room for improvement. However, we are willing to pay this cost in favor of the provided advantages.

A hardware implementation is already existing for a number of components for this new ISA. The remaining components are currently in progress. The full implementation will then be validated against the simulation. An adaptation of the synthesis process to the new ISA is also currently in progress.

We believe that our transpiler still has some room for improvement. In order to support general purpose applicability of the processor, we will need to add support for multi-threading and for debugging (which both already exist for the Bytecode based AMIDAR processor).

References

1. Corporaal, H.: Microprocessor Architectures: From VLIW to TTA. Wiley, Hoboken (1997)
2. Gan, L., et al.: A highly-efficient and green data flow engine for solving Euler atmospheric equations. In: 2014 24th International Conference on Field Programmable Logic and Applications (FPL), pp. 1–6 (2014)
3. Gatzka, S., Hochberger, C.: The AMIDAR class of reconfigurable processors. J. Supercomput. **32**(2), 163–181 (2005). https://doi.org/10.1007/s11227-005-0290-3
4. Gurd, J.R., Kirkham, C.C., Watson, I.: The Manchester prototype dataflow computer. Commun. ACM **28**(1), 34–52 (1985)
5. Heikkinen, J., Cilio, A., Takala, J., Corporaal, H.: Dictionary-based program compression on transport triggered architectures. In: IEEE International Symposium on Circuits and Systems (ISCAS 2005), pp. 1122–1125 (2005)
6. Lam, P., Bodden, E., Lhotak, O., Hendren, L.: The Soot framework for java program analysis: a retrospective. In: Cetus Users and Compiler Infrastructure Workshop (CETUS 2011), October 2011
7. Lee, B., Hurson, A.: Issues in dataflow computing. In: Yovits, M.C. (ed.) Advances in Computers, vol. 37, pp. 285–333. Elsevier, Amsterdam (1993)
8. LLVM Project: LLVM bitcode file format. https://llvm.org/docs/BitCodeFormat.html
9. Various: Standard ECMA-335 Common Language Infrastructure (CLI). ECMA International, Geneva, Switzerland (2012)
10. Wolf, D.L., Jung, L.J., Ruschke, T., Li, C., Hochberger, C.: AMIDAR project: lessons learned in 15 years of researching adaptive processors. In: 2018 13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), pp. 1–8, July 2018