The Journal of Supercomputing, 32, 163–181, 2005

The AMIDAR Class of Reconfigurable Processors

STEPHAN GATZKA stephan.gatzka@inf.tu-dresden.de
CHRISTIAN HOCHBERGER christian.hochberger@inf.tu-dresden.de
Chair for Embedded Systems, TU Dresden

Abstract. In this contribution we present a novel general model for adaptive processors. We describe its basic principle of operation and introduce several formal characterizations. The adaptive operations that are possible with this model are thoroughly discussed. The model allows runtime variations of the type and number of functional units as well as variations of the communication structure. We introduce simple heuristics to achieve adaptivity of the architecture. Experimental results show that a processor implementing this model can adapt its architecture to the requirements of diverse applications.

Keywords: reconfigurable processors, adaptive computing, processor architecture

1. Introduction

Configurable Systems on a Chip (CSoC) are becoming more and more important in the embedded systems market. The main reasons for their growing popularity can be found in their cost effectiveness and flexibility. Mask costs will be very high in the future due to the required high resolution [9]. CSoCs will be cost effective under these circumstances since they offer the possibility to implement multi protocol/multi standard systems with a single chip and thus can be produced in much larger quantities. They will be flexible since the reconfigurable part can be used to adapt the hardware to future requirements that are unknown at the time of the initial development.

With CSoCs it will be possible to implement new IO functionality and peripherals in an existing system. It will also be possible to have peripherals to increase the performance (e.g. crypto accelerators), but the processor core itself cannot be changed or enhanced. To overcome this problem, we introduce the AMIDAR class of processor. It is a novel model and architecture of processors that can be adapted during runtime to the requirements of an application. AMIDAR stands for *A*daptive *Mi*croinstruction *D*riven *Ar*chitecture.

Although C as a programming language still dominates the development of embedded software, the growing tendency to use Java as a programming language for embedded systems makes this language an attractive object of study. Due to the code shipping abilities of Java [5], it is most likely, that especially systems programmed in Java will experience a shift in the requirements during their lifetime. Thus, we decided to evaluate our new model on the example of a Java bytecode processor.

1.1. Related work

Hardware implementations of Java bytecode processors are available in a large number. Yet, to our knowledge only the JEM-II processor can be customized for the application requirements [1]. But in this case only new bytecodes can be introduced as microcode sequences.

Also, recently some work has been conducted to build customized accelerators to speed up the execution of Java bytecode [6]. 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.

Other researchers have addressed (re)configurable processors in general. Some are just parameterized RISC cores [2], while others are truly reconfigurable. They typically depend on compile time analysis and generate a single datapath configuration for an application beforehand [3, 4, 7]. Very few processors are really reconfigured at runtime [8]. But even in this case, the configurations and the time of reconfiguration are defined at compile time.

To the best of our knowledge, there is no general model for an adaptive processor.

1.2. Paper outline

In the following section we will describe the general model of an adaptive processor in detail. In Section 3 we will then present the various adaptive operations that are possible with this model. Section 4 introduces some simple heuristics that can be used to adapt a specific architecture to applications. Experimental results for a Java bytecode processor are shown in Section 6. Finally, a conclusion and an outlook onto future work are given.

2. Model

In this section we will describe how our model of an adaptive processor works in general and we will also give a formal description of it.

It is useful to have a general model of an adaptive processor to identify and categorize the elements of the architecture which are subject to adaptivity. This model should achieve the following goals: The model must be general enough to express different architectures, it should expose as much parallelism as possible, it should be applicable for a simulation and it must be close to a hardware implementation.

2.1. Overview

Figure 1 shows the basic structure of an adaptive processor.

It consists of four main types of components: a token generator, functional units (FU), a token distribution network and a communication structure. The token generator is a specialized functional unit, which is always required. It controls the other components of the processor by means of tokens. These tokens are sent to the FUs over the token distribution network. The tokens tell the FUs what to do with input data and where to send the results. Functional units can have a very wide range of meanings: ALUs, register files, program and/or data memory, specialized address calculation units, etc.

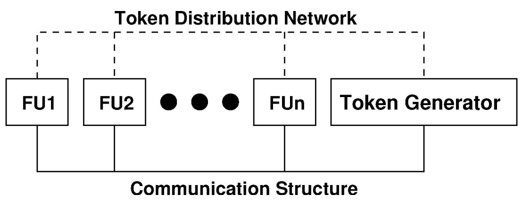

Figure 1. General model.

Data is passed between the FUs over the communication structure. This data can have various meanings: program information (instructions), address information, or application data.

2.2. *Definitions*

A functional unit is a piece of hardware that executes a specific task in the processor. Each FU has at most one output port [1] and an arbitrary number of input ports. A functional unit can be characterized by the values latency L, interval I and area A. The latency specifies the time needed by the FU to complete a single operation, whereas the interval specifies the time required between the start of two consecutive operations. The area specifies the required amount of chip resources for this unit. There can be different variations of FUs for the same task (latency optimized with minimal L, throughput optimized with minimal I or area optimized). Typically, latency optimal versions will not be throughput optimal and vice versa. Also, we assume that the consumed area is a monotonic function with respect to the values L and I. This means that a decrease of L or I always leads to an increased area. Other characterizations may be of interest like energy consumption or energy efficiency.

A token is a 5-tuple: $T = \{UID, OP, TAG, DP, INC\}$. UID identifies the functional unit to which this token belongs. OP specifies, which operation this FU has to carry out on the tagged data. The TAG is used to distinguish different data that is sent to the same FU (explained in more detail below). *DP* is the destination address, where the result of the operation is to be sent. It consists of a UID and a port number, since functional units can have more than one input port. *INC* is a boolean value, that controls the creation of the tag that is sent out with the result data. If it is *false*, the tag is passed unchanged. Otherwise the tag value is incremented. $UID(T_i)$ denotes the functional unit to which the token T_i is to be sent. $OP(T_i)$ denotes the operation that has to be carried out by token T_i. The meaning of $TAG(T_i)$, $DP(T_I)$ and $INC(T_i)$ is analogous.

An instruction is a composition of an arbitrary number of tokens: $Ins = \{T_1, T_2, \ldots, T_N\}$. The model itself doesn't require an order of the tokens, but specific implementations may impose restrictions on the order in which tokens are distributed to the functional units.

The communication structure consists of an arbitrary number of busses. A bus is a pair of sets of ports $B = (S, D)$ where S is the set of source ports driving the bus and D is the set of destination ports which read data from the bus. $S(B)$ denotes the set of source ports driving bus B and $D(B)$ denotes the set of destination ports of Bus B. Since input ports and output port of FUs are always disjoint, busses are unidirectional.

2.3. Principle of operation

Program information (i.e. the instructions) is sent to the token generator. Now, the token generator creates a set of tokens for this instruction and distributes them concurrently to the functional units. A functional unit begins the execution of a specific token as soon as the data ports have the data with the corresponding tag. Upon completion of an operation $OP(T_i)$ the result is sent to the destination $DP(T_i)$. The tag that is sent together with the result depends on $INC(T_i)$. $TAG(T_i)$ is used if it is false, otherwise $TAG(T_i) + 1$ is used. 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.

This data driven approach has a number of advantages:

- It implies a maximum of parallelism, which is only limited by data dependencies between consecutive instructions. It should be noted, that these dependencies can only originate from application data (like user register values or stack data).
- It does not rely on a particular timing of the FUs. Execution of an instruction will work, no matter how long a single FU needs to complete its token. Also, it is not necessary to know the structure of the communication network beforehand. Thus, it can be changed during runtime without the need to reconfigure the token generator.
- It allows overlapping execution of instructions, since the token generator can start distributing tokens for more than one instruction. It only has to increment the tag field of the token for each instruction to separate data belonging to different instructions.
- It allows the introduction of new FUs and instructions using these FUs. For this purpose a small part of the token generator must be reconfigurable, to store the token set for new instructions and to attach new FUs to the token generator (which are not addressed by normal instructions).

Although this model looks like a dataflow machine, it has to be noted that it actually processes instructions by this scheme and not application data.

2.4. Tag increment

Figure 2 shows an example architecture of a simple processor using the given model. Input ports of the FUs are shown on the top border of the FUs, whereas the output ports are shown on the bottom border. The token distribution network is shown with dashed lines.

Let us consider the execution of the following instruction `MAC R1, R2, R3`. It means, that the product of R2 and R3 shall be accumulated in R1. It is useful to write the instruction as an equation to evaluate the sequence of tokens that have to be executed to complete this instruction. Tokens have to be generated for almost any data transfer and for all processing steps. Figure 3 shows the equation and all the corresponding tokens.

The meaning of the tokens is as follows:

- T_1 is sent to the register file. It triggers the transfer of the value of `R3` to port 1 of the ALU. The register address is part of the instruction and thus, the token generator sends it to the register file.

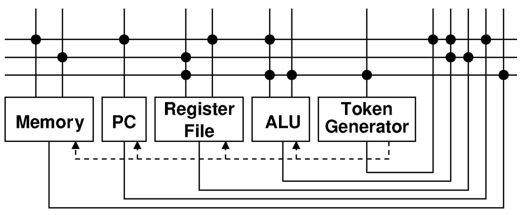

Figure 2. Simple example architecture.

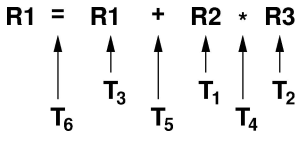

Figure 3. Equation and tokens for a MAC instruction.

- T_2 is also sent to the register file and triggers the transfer of the value of `R2` to port 2 of the ALU.
- T_3 is likewise sent to the register file and triggers the transfer of the value of `R1` to port 1 of the ALU. T_3 must have a different tag number than T_1 and T_2, since the ALU must be able to distinguish this data from the previous two.
- T_4 is sent to the ALU and instructs the ALU to compute the product of the data with the appropriate tag. The result is sent to port 2 of the ALU. In this case the tag must be incremented, to match the tag of T_3, which delivers the value of `R1` to the ALU.
- T_5 is also sent to the ALU. It results in the computation of the sum of `R1` and the product `R2` $*$ `R3`. This result is sent to the register file.
- T_6 is sent to the register file and stores the value at the data port (which comes from the ALU) to the given address (which is again given by the token generator).

Two more tokens must be executed, to keep the machine running:

- T_7 is sent to the PC unit. The PC must send its current value to the memory to address the next instruction. Also the PC must increment its value afterwards.
- T_8 is sent to the memory. The memory sends the addressed instruction to the token generator.

Tag increment occurs in this example in token T_4. More formally, the following condition must be fulfilled for any instruction $Ins = \{T_1, T_2, \ldots, T_N\}$:

$$\begin{aligned}\forall_{i,j\in[1,N],i\neq j} &: UID(T_i) = UID(T_j)\\ &\Rightarrow TAG(T_i) \neq TAG(T_j)\end{aligned} \tag{1}$$

This means that the same FU is used repeatedly in the execution of an instruction. Let us assume, that T_j depends on the result of T_i. In this case it is mandatory, that one of the tokens on the dependence graph from T_i to T_j increments the tag, since otherwise the FU could not distinguish between the two values.

2.5. *Propositions*

It is useful to give some formal characterizations of the communication structure to discuss implementation aspects.

Let us consider a set of N busses $B_1, B_2, \dots, B_N$. We can now describe three special conditions:

(1) A bus B_i is called a single source bus, if

$$|S(B_I)| = 1 \tag{2}$$

In this case, the outport of the driving FU can be connected directly to the bus (omitting any driver circuitry).

(2) The set of busses, which are connected to a given output port SP is given by equation (3).

$$DB_{SP} = \{i \in [1, N] : SP \in S(B_i)\} \tag{3}$$

Routing information at the output port is only required if $|DB_{SP}| > 1$.

(3) The set SB of busses, which are connected to a given input port DP is given by equation (4).

$$SB_{DP} = \{i \in [1, N] : DP \in D(B_i)\} \tag{4}$$

An input port is called dedicated, if $|SB_{DP}| = 1$.

It means, that this input port is connected only to one bus. Thus the bus can be connected directly to the input port of the FU. No driver circuits are required.

Real redundancy is only present, if an output port is connected to at least two busses and these busses are also connected to the same input port of an FU.

2.6. *Special precautions for dynamically synthesized FUs*

It may be necessary to establish a synchronization mechanism between the FU and the token generator for dynamically synthesized FUs. An example are very data intensive FUs. Such functional units often are deeply pipelined to achieve a high throughput, but the latency on the other hand is also high. Other functional units waiting for the results may be blocked resulting in critical deadlock states. For this reason it is desirable to block the token generator from delivering new tokens until the FUs completed their calculations. For this task we introduce a SYN packet which will be sent to the specialized functional unit. The token generator now stops to deliver new tokens. The functional unit receiving the SYN packet must now send a corresponding (to the tag inside the SYN packet) data packet to the token generator. Now the token generator will continue to issue tokens. The data sent to the token generator may be used only to inform the token generator to continue its work, but may also be used to signal special states of the functional unit, for instance an exception. The token generator now can react on this special situation. It should be noted, that the usage of SYN packets eliminates the parallelism inside the processor. The synchronization mechanism is certainly only required for dynamically synthesized FUs. The performance gain provided by such FUs should always be high enough to neglect the effects of the synchronization packets.

Until now single data transfers between different FUs were introduced. Dynamically synthesized FUs often require high data volumes for high utilization. Single data transfer will not provide the required data bandwidth for such FUs. To overcome this problem AMIDAR-class processors may include support for burst and bulk transfers. These transfers are not required for a correct function of the model and therefore will be described in Section 5.3.

2.7. Applicability

In general, the presented model can be applied to any kind of instruction processing, where a single instruction is composed of microinstructions. Obviously, the model doesn't produce good results, if there is a strict order of those microinstructions, since in this case no parallel execution of microinstructions can occur.

Overlapping execution of instructions comes automatically with this model. Thus, it can best be applied if dependencies between consecutive instructions are minimized.

The great advantage of this model is that the execution of an instruction is not dependent on the exact timing of FUs. Thus, FUs 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 applications. Thus, this model allows us to optimize global goals like performance or energy consumption.

As previously mentioned, intermediate virtual assembly languages like Java bytecode or the .NET code seem to be good candidates for instruction sets. The range of FU implementations and communication structures is especially wide, if the instruction set has a very high abstraction level and basic operations are sufficiently complex.

Finally, the data driven approach makes it possible to easily integrate new FUs and create new instructions to use these FUs.

3. Adaptive operations

Adaptivity in this model can be seen on two hierarchical levels. On the top level the available chipsize is partitioned into an area for communication infrastructure and an area for

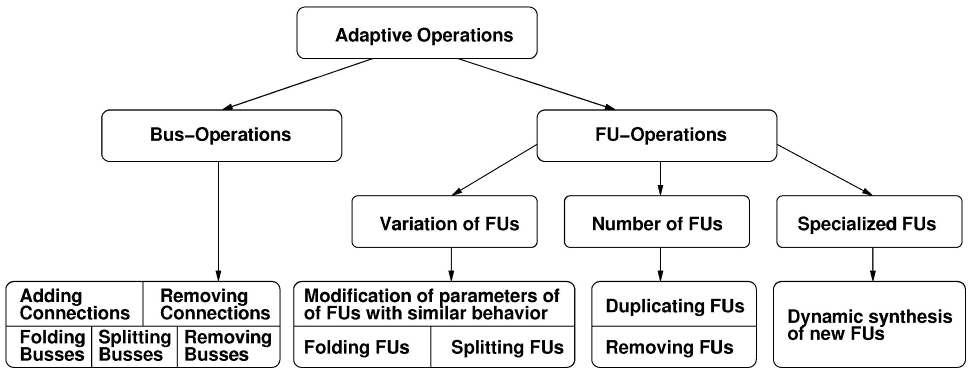

Figure 4. Adaptive operations.

functional units. Most of the currently available reconfigurable devices will not fully support this type of adaptivity, since resources for communication may not be suitable for functional units and vice versa. Yet, the model should be general enough to capture these possibilities. On the lower hierarchical level we have adaptive operations that reconfigure each of the two main areas. Figure 4 illustrates the different adaptive operations of our model.

Within the communication area several adaptive operations are possible:

- *Adding and removing connections.* If a functional unit has to send a data packet to another FU but is not connected to it, it is necessary to create a new connection.
- *Folding busses.* Two busses may be merged to a new bus, if there are only few collisions on both busses. A possible hardware implementation is described in Section 5.
- *Splitting busses.* Busses with a high utilization and many delays can be split into two busses. The decision, which bus has connections to which FU is done by the heuristics as described in Section 4.
- *Removing busses.* Although this operation is already implied by folding of busses, it is useful as a separate operation, because folding of busses has a higher complexity than a simple remove.

The adaptivity of the communication structure is comparable with the dynamic reconfiguration of the forwarding unit of a superscalar microprocessor.

Within the functional unit area three different categories of adaptive operations can be applied:

- *Variation of FUs.* In this case variations of a certain FU may be available. From these variations the heuristics (see Section 4) choose the most appropriate. This operation is fairly simple because it does not affect the token generator.

  Moreover, it is possible to split an FU into more specialized ones. For example, an ALU may be used for address calculations. This ALU may be split into an address calculation unit and a normal ALU. Folding of FUs into one FU is the complementary operation to FU splitting.

- *Increase and decrease the number of instances of an FU.* If the interval I of an FU can not be decreased by a more specialized version, it is possible to duplicate this FU. Token distribution must be adapted to this new situation which must also care about an equal utilization of the new FUs. Therefore, this adaptive operation is not as easy to implement as a simple FU exchange. A possible implementation is given in Section 5.
- *Addition of newly synthesized FUs.* It is also possible to identify heavily used instruction sequences and synthesize a new FU for such sequences. The instruction sequence is replaced by a new instruction and the token generator is updated with a token sequence for this new instruction. It may be applicable to synthesize complete methods or functions. The calling function can easily replace the calling code to access the new hardware. This is the most complicated adaptive operation, but promises the highest performance gain.

4. Heuristics

According to the adaptive operations described in Section 3 we need heuristics to decide which adaptive operations lead to the best results regarding the requirements of the running application. The heuristics have to decide, how the available chip area should be divided. There must be a global heuristics that calculates the ratio between chipsize for FUs and chipsize for busses. Furthermore, we need local heuristics to assign the resources within the different areas of adaptivity (communication and FUs). Figure 5 shows the interaction between global and local heuristics.

4.1. Statistical data

All heuristics are based on stalls and utilization of functional units and busses. The stalls are divided into *input stalls* and *wait stalls*, so every component requires two counters to store them. An input stall occurs, if a component (FU or bus) could not accept data because it is currently working. Input stalls can occur in busses and functional units. This is in indicator, that a component is apparently not efficient enough to handle incoming data. More precisely, an input stall in an FU indicates that the interval I of the FU is not low enough.

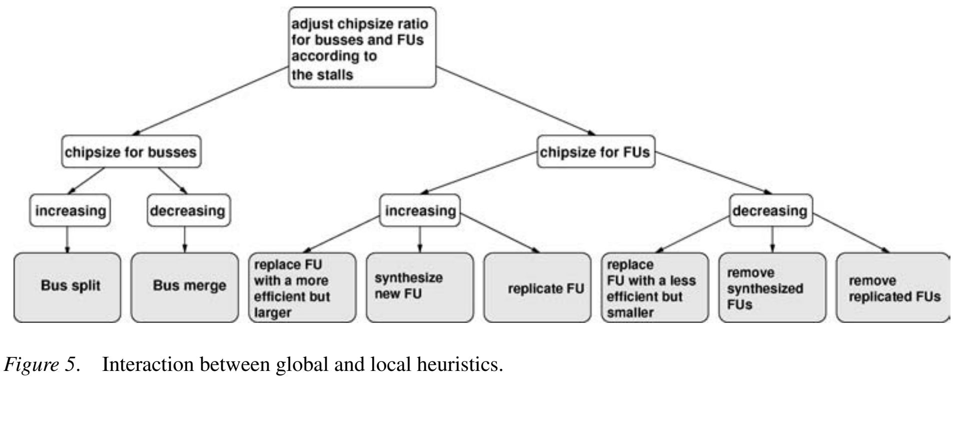

Figure 5. Interaction between global and local heuristics.

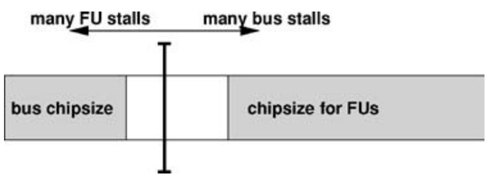

Figure 6. Division of chipsize.

Wait stalls occur if a component is ready to work, but has to wait for input data. Wait stalls are saved in the sender component, because the sender component is not efficient enough to deliver data in time, i.e., the latency L is to high.

A description, how these statistical data can be gathered, is given in Section 5.

4.2. Global heuristics

The collected statistical data is the input for the heuristics. The global heuristics is very straightforward. The main idea of the global heuristics is a slider, which divides the overall chipsize into chipsize for communication and chipsize for functional units. Figure 6 illustrates this idea.

The criterion for shifting the ratio of chip size for communication and chip size for FUs is based on stalls inside the communication structure and stalls inside the FUs. The stalls inside the FUs are accumulated to ST_{FU} and the stalls inside the communication structure are accumulated to ST_{bus}. A_{FU} is the area for FUs and A_{bus} is the area for the communication structure. The shift of costs between the communication structure and the FUs is calculated as follows:

$$\begin{aligned} inc^{t+1} &= inc^{t} + \left(ST^{t}_{FU} - ST^{t}_{\text{bus}}\right) * c \\ A^{t+1}_{FU} &= A^{t}_{FU} + ceil(inc^{t}) \\ A^{t+1}_{\text{bus}} &= A^{t}_{\text{bus}} - ceil(inc^{t}) \end{aligned}$$

The attenuation factor c ensures that only long-termed requirements cause a reconfiguration. Thus, after a large number of execution steps the global heuristics can shift one cost point between the communication structure and the FUs.

4.3. Local heuristics

The local heuristics decide autonomously how to deal with the globally assigned chip area. There are local heuristics for the communication structure and for the functional units.

If the local heuristics for busses can use more chipsize, the heuristics tries to split the bus with most stalls. The source and destination information for delayed data packages are used to make new connections between functional units. If the local heuristics for communication structures has to save chip area, it will first try to merge two busses. These busses are chosen by looking at time conflicts within the last clock ticks. Only busses can be merged that have time conflicts in less than a given percentage over the last n ticks. If no busses can be merged, connections of busses that were very seldom in use within the last n clock ticks would be removed.

The cost of busses are proportional to the number of functional units that are connected to this bus. The cost model is influenced by the idea of a connection matrix where every connection between a functional unit and a bus can be implemented as a tristate buffer with constant cost.

The heuristics for functional units has more different operations. If the heuristics can use more chipsize, it is possible to replace an FU by a more efficient one, duplicate an FU and synthesize a new FU. First of all, the heuristics tries to exchange the FU with most stalls by a more efficient one. The different types of stalls control, which variation of an FU is used to replace the current FU. Figure 7 illustrates the idea.

If an FU has many input stalls, it cannot accept data because it is currently working. The FU is replaced by a throughput optimized one. If the FU has many wait stalls, this indicates, that another FU was waiting for data. So we need a latency optimized version of the FU. A more generalized heuristics may be based on a vector space, where the number of stalls is the length of a vector in the corresponding dimension. The angle of the resulting vector to the reference vectors of different FU variations is the criterion, which FU is used.

If a functional unit could not be improved anymore, the heuristics may consider to duplicate this FU. This sounds to be a simple process, but requires an adaption of token distribution which has to care about load balancing between similar FUs.

If the heuristics for functional units has to save chipsize, it first removes duplicate FUs. Again, removing duplicate FUs is possible but not trivial due to the adaption of token distribution. If the heuristics has to save more chipsize, FUs with the fewest stalls and lowest utilization are replaced by slower but smaller versions.

A heuristics that triggers a direct synthesis of a specialized FU cannot be based on stalls inside an FU. The token generator has to track which opcode sequences or methods/functions

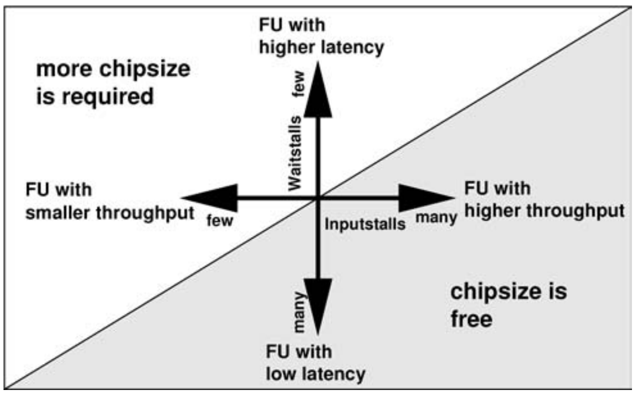

Figure 7. Exchange of functional units.

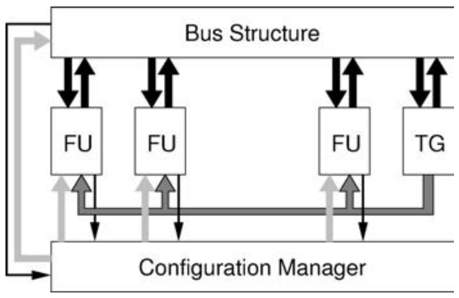

Figure 8. Model with configuration manager.

are often used. If these sequences are often slowed down by stalls in the FUs, it may be feasible to trigger a synthesis process for this opcode sequence.

5. Implementation issues

5.1. *Configuration management*

Up to now, the presented model does not include any component to manage the replacement of FUs or reconfiguration of the bus structure. Figure 8 shows a possible extension of the model in order to manage the configuration of the processor.

The configuration manager gathers statistical data from the functional units and the bus structure. It applies the presented heuristics to this data and decides, what actions to take: replacement of functional units or reconfiguration of the bus structure.

5.2. *Specific problems*

Figure 9 shows all places, where an implementation of the model will probably experience some problems.

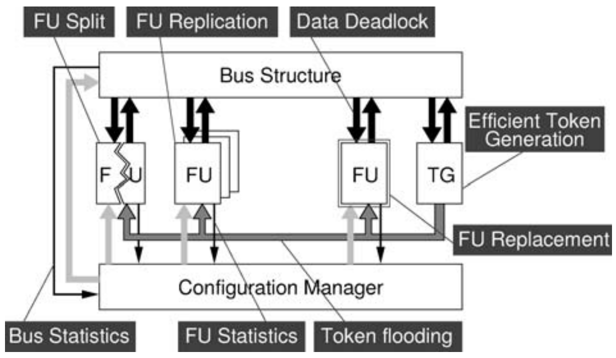

Figure 9. Problematic parts of the implementation.

Splitting of FUs and FU replication can cause problems concerning the token distribution to the separate FUs. Reception of data at the inputs of the FUs can cause data deadlocks if the tags of the received data and the tag associated with the tokens don't match. Efficient token generation can be a problem, since varying amounts of tokens have to be provided for different instructions. Another problem can arise from flooding FUs with tokens, which must be prevented. Finally, gathering statistical data from the bus structure and the FUs can be implemented in different ways.

In the following we will sketch solutions for all these problems.

(1) *FU statistics*: Counting stalls of FUs and busses can be implemented in a likewise fashion. For each stall type (wait or input) there must be one counter. Counting input stalls can be achieved in the following way. Every time an FU cannot accept data, it sends a reject signal to the sending FU and increases its number of input stalls. The reject signal can be implemented as a simple open-collector wire for every bus because every transmission belongs to one sender FU and one receiver FU.

The wait stall counter increments its value, if a token already arrived and one port of the FU has its corresponding input data but the FU has to wait for more data for further processing. The receiving FU counts these clock cycles and after a threshold value the FU signals this condition to the sending FU which was responsible for the delay. This signal also can be implemented as an open collector wire.

(2) *Bus statistics*: According to Section 4.1 busses have to count input stalls. This can easily be achieved as described in the previous paragraph. For busses it is also necessary to record source and destination of delayed data packets in a small table. This information is required for the adaptive operation *bus split* (see Section 3) to determine, which bus will be connected to which FU.

The measurement of the utilization of busses can also easily be implemented in hardware. A shift register of n bits stores the clock cycles in which the bus was in use. A 1 is shifted in, if the bus was working in this clock cycle, otherwise a 0 is shifted in. Now, the configuration manager can apply the following simple rule for bus folding: Folding of two busses is useful, if a bitwise conjunction of the corresponding registers leads to a small number of 1's and a threshold is exceeded.

(3) *Token flooding*: Another problem is, that the token generator may flood the functional units with tokens, which must also be prevented. Each FU has an input buffer for tokens. The depth of the buffers can be individually adjusted regarding to the maximum number of overlapping tasks for each FU. Each functional unit has the ability to signal the token generator that the input queue is filled. This can be done with a simple open-collector wire for all FUs. If the token queue for one FU is filled, the FU can set the signal to low and the token generator has to stop the generation of tokens.

(4) *FU replacement*: The replacement of an existing FU is a fairly simple operation. The new FU has the same ports and behaves just like the old one. It is more complicated to determine the moment, when it is possible to replace the FU. Of course it is possible to replace an FU which has no pending data packets and tokens. But in a running application this is often not the case. A solution is the separation of the FU and the

token buffer. Only the FU will be replaced, the token buffer remains unmodified. If an FU should be replaced, this state will be signaled to the token buffer which further accepts tokens to prevent deadlocks. But the buffer will not deliver these tokens to the FU until the configuration of the FU was done.

(5) *Deadlocks*: A problem of the model is, that input data may arrive in an unpredictable order at the functional units, which does not match the sequence of tokens in the FUs. This could potentially result in a deadlock situation. Two solutions for this problem are possible:

    (1) We can introduce buffers at each input port to accumulate input data until all data with the tag of the pending token is available. This requires a buffer depth corresponding to the maximum number of concurrent tags in the system. The tag information of the pending tokens must be compared to the tag information in the input buffers. This solution would imply a severe hardware overhead.
    (2) We can serialize the tokens inside a functional unit. This allows us to eliminate the input buffers, since the FU only accepts data that matches the tag of the currently pending token. For this purpose we establish a protocol (like an nACK signal, which also may be an open-collector signal) that allows the receiver to reject data that currently can't be processed (with regard to the required and presented tag). This solution is much more suitable for a real hardware implementation.

    It is important to note, that there is always a partial order of instructions, that no deadlock will occur. The token generator is responsible to create the tokens in a correct order.

(6) *FU split*: Splitting an FU is more complicated to accomplish. The new FUs must be supplied with new tokens and with data packets. One option is that the token generator has to be informed about the new FUs and has to issue new tokens for it. This solution imposes severe changes inside the token generator for each FU split. This should be avoided. Another solution is to define a kind of meta component, which encapsulates the original FU and, after the FU split operation, both new FUs. Now, the meta component is responsible for delivering the correct data packets and tokens to the individual FUs, the token generator does not need to be informed about the FU split operation. The main advantage of this solution is, that changes in the meta component are much more locally and therefore much better to control.

(7) *FU replication*: The replication of FUs imposes similar difficulties like the FU split. Either the token generator must be informed about the new FU and generates suitable tokens or a meta component distributes the tasks locally. The meta component does simple load balancing using the utilization information inside the FUs.

(8) *Efficient token generation*: It should also be noted, that the token generator itself can become a bottleneck of the architecture. Our experience gives rise to the claim, that this will not happen if the complexity of the instruction is sufficiently high, since then the frequency of instruction execution is lower than with more simple instruction sets. The Java bytecode certainly fulfills this requirements.

5.3. Extensive data transfers

Data transfers between FUs will typically only involve single word transfers. This situation changes if we have a closer look at synthesized FUs. In this case synthesized FUs will often need to transfer whole data sets from one of the memories to a synthesized FU or vice versa. Obviously, the token generator could distribute a single token for each of the required data transfers, but this seems to be very inefficient. In order to better support this situation we propose two specialized memory access tokens:

- Burst Data Transfer. In this case the memory needs two words of information to commence the operation: the starting address and the number of words to transfer. In case of a read request the memory starts to deliver data from consecutive addresses to the targeted FU upon reception of this information. In case of a write request the memory waits for the data to arrive and writes each incoming word to consecutive addresses.
- Bulk Data Transfer. It may occur that the synthesized FU needs to access the memory in irregular address patterns. In this case the FU needs to supply an address word for each transfer. The addresses are transfered to the memory over existing bus connections. This type of access is not as fast as a burst transfer, but is still much more efficient, than distributing a set of tokens for each transfer (at least one for the address and one for the data).

It should be noted, that in both cases it is required, that each data transfer is carried out with an incremented tag. Otherwise, it would not be possible to match data and addresses in the FU and in the memory. Also, the token generator will typically continue to distribute tokens for the following instructions. The token generator should avoid to reuse tag numbers that have already been used by the bulk or burst data transfers. Furthermore, in order to avoid data hazards it may sometimes be necessary to include a SYN token for the synthesized FU.

6. Experimental results

To test the applicability of our model we derived the architecture of a Java bytecode processor from it. A simulator implements this model to verify it and to prove that adaptivity leads to a measurable performance gain. Figure 10 shows the architecture of the Java bytecode processor.

Currently, the simulator adapts the communication structure to the requirements of the application (adding and removing of connections to busses, splitting and merging busses) and is able to exchange functional units. The implemented heuristics and cost model are the same as described in Section 4.

6.1. Simulation environment

The simulator is programmed entirely in Java. Especially, the object oriented view on components of a processor is a big advantage concerning the dynamic adaptation. Every

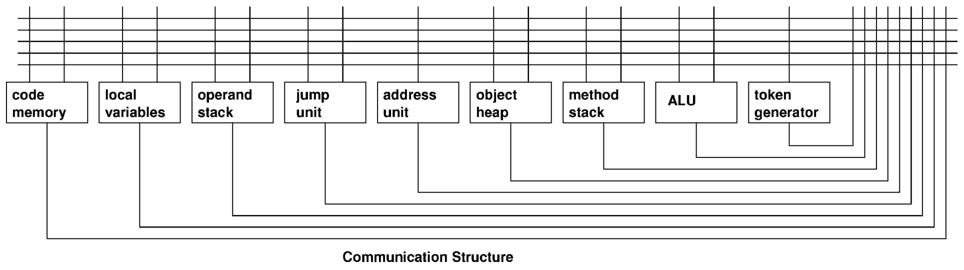

Figure 10. Architecture of a Java bytecode processor.

component is a Java object with specific properties that may be exchanged with a different Java object. An adaptive operation is done just by the exchange of two objects. Inheritance allows us to reuse methods in classes for related components. We dismissed the usage of VHDL, because adaptivity could not easily be expressed.

The simulator is generic enough to implement different types of processors within. The Java bytecode processor is just an example architecture using different Java-specific functional units. Other processors which obey to the AMIDAR operating principle are of course implementable.

The simulator implements an enhanced unit delay model. The delay of a functional unit depends on the executed operation. A multiplication inside an ALU may last longer than an addition. Currently, there is no dependence between the delay of an FU and the input data. The functional units report the delay to the simulator. Hence it is possible to implement a more complex delay model inside the FUs.

The simulator is cycle precise. The smallest time unit is a cycle. No component can implement a delay less then one cycle. The raising edge of the clock triggers all functional units to begin their work with the input data, the falling edge triggers the transmission of data from the busses to the receiving functional units.

All components (busses, functional units) are implemented just like real hardware works. Especially, heuristics and collecting statistical data are implemented with real hardware circuits in mind (see Section 5). This allows very precise predictions about runtime and improvements of the dynamically adapted circuit.

6.2. Test applications

To prove the effectiveness of adaptivity in the Java bytecode processor, two test applications with different characteristics had to be selected that are expected to lead to different structures. Therefore, a data dominated application (signal convolution) and a control dominated application (calculation of Ackermann's function) were chosen.

A Java program that calculates the convolution of two signals was used as the data dominated application. The control dominated application calculates Ackermann's function ack(3,2), which mainly results in recursive method calls and if-then-else structures. We expected that the resulting structures look different because of the very different usage of some functional units in the test applications. The convolution program heavily uses the object heap and the ALU, whereas the Ackermann program will use the jump unit and the method stack. Operand stack and local variable memory will be used in both applications in the same manner. The different characteristics are reflected in the appearance of different bytecodes. The convolution program consists basically of array and ALU operations. In contrast, the Ackermann program uses if-bytecodes, invokes, returns and only few ALU operations.

6.3. Results

To evaluate the speedup of the dynamic bus adaption and FU exchange in comparison to a static architecture both test programs were run in the simulator in different modes. Firstly, we measured the minimal cost (worst case performance), which results in one single bus for all components and slowest FUs. Then we measured the best case performance. Faster FUs were used if they lead to a performance improvement. The results are shown in Table 1. It turns out, that the maximum speedup is between 23% and 29%.

Secondly, we ran the applications in adaptivity mode with a cost limit. According to Table 1 the adaptive circuit for the convolution program performs 6% slower than the best case architecture, and the circuit for the Ackermann program is less than 1% slower than the best case circuit. The cost limit in both cases is set to 83% of the best case circuit. Figure 11 shows more clearly, that the adaptive circuit is nearly equally fast as the best case but only requires half the cost increase compared to the cost increase of the best case. Table 1 also shows, which functional units were replaced during the simulation. In the best case circuit, of course, all memory components (stacks, local variables) and the ALU are throughput optimized variants. It can be seen, that in the adaptive case the different applications cause the usage of different FUs. Interestingly, running the Ackermann program replaces the normal ALU with a slower one. This is definitely a good decision, because the Ackermann program does not perform many calculations in comparison to method calls and memory fetches.

Figure 12 shows the speedup in relation to the cost limit. The non-monotonic characteristics of the runtime is due to coarse grained cost of the functional units. Thus, it can happen, that cost points are transfered to the FU area which results in a degradation of the communication. Probably, this behavior can be eliminated with a more elaborate heuristics.

Table 1. Measurements for the Ackermann and Convolution application

			Used FUs		
Circuit	Costs	Clocks	Stacks	Loc. vars	ALU
Ackermann					
Best	113	22408	Piped	Piped	piped
Worst	69	31480	Normal	Normal	slow
Adaptive	94	22509	Piped	Piped	slow
Convolution					
Best	113	19143	Piped	Piped	Piped
Worst	69	24903	Normal	Normal	Slow
Adaptive	94	20435	Piped opstack	Piped	Piped

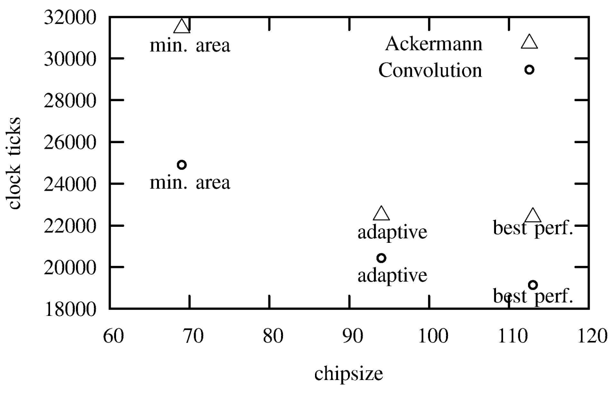

Figure 11. Comparison of adaptive, best case and worst case circuits.

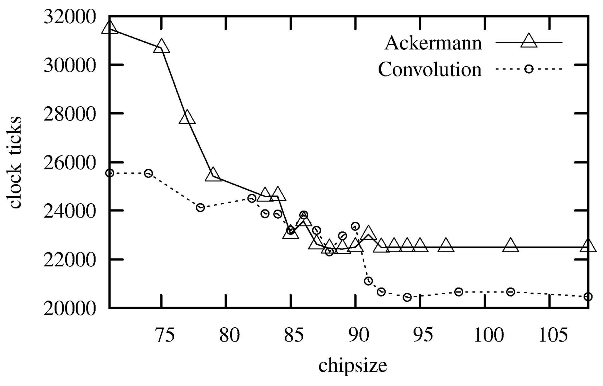

Figure 12. Cost-speedup relation for the Ackermann and Convolution application.

7. Conclusion

We presented a novel model of an adaptive processor. The execution model allows any component of the processor to be replaced at runtime. We simulated the implementation of a Java bytecode processor using this model. Although, we do not exploit the full amount of adaptivity in this simulation, it already shows the benefits of this approach.

The architectures that were derived through adaptation performed very close to the optimal solution.

8. Future work

The experiments that we have carried out so far were focused on performance optimizations. We will conduct further experiments to prove the feasibility of the model for other optimization goals like energy consumption or energy efficiency.

Currently, we don't generate application specific functional units. Especially, the Java bytecode presents a very good basis for the synthesis of functional units for code fragments. This allows us to transform only those parts of code into hardware that have a strong impact on the runtime (e.g. inner parts of loops).

Note

1. In fact, the token generator seems to be the only FU that potentially doesn't need an output port.

References

1. ajile Systems. aj–100 Datasheet, 2000. http://www.ajile.com/.
2. F. Campi, R. Canegallo, and R. Guerrieri. IP-reusable 32-bit VLIW Risc core. In *European Solid State Circuits Conference (ESSCIRC)*, Sept. 2001, pp. 456–459.
3. Y. Chou, P. Pillai, H. Schmit, and H. P. Shen. Piperench implementation of the instruction path coprocessor. In *Proceedings of the 33th Annual International Symposium on Microarchitecture*, pp. 147–158, Monterey, Dec. 2000.
4. C. Ebeling, D. C. Cronquist, and P. Franklin. Rapid—Reconfigurable pipelined datapath. In R. W. Hartenstein and M. Glesner, eds., *Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, pp. 126–135, Berlin, Springer Verlag, 1996.
5. S. Gatzka, C. Hochberger, and H. Kopp. Deployment of middleware in resource constrained embedded systems. In *Tagungsband der GI/OCG-Jahrestagung 'Informatik 2001'*, pp. 223–231, Wien (Österreich), September 2001. Österreichische Computer Gesellschaft.
6. Y. Ha, R. Hipik, S. Vernalde, V. Diederik, M. Engels, R. Lauwereins, and H. De Man. Adding hardware support to the HotSpot virtual machine for domain specific applications. In M. Glesner, P. Zipf, and R. Michel, eds., *Field-Programmable Logic and Applications. Reconfigurable Computing Is Going Mainstream (LNCS 2438)*, pp. 1135–1138, Berlin, Heidelberg, Springer, 2002.
7. R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger. Using the KressArray for reconfigurable computing. In J. Schewel, ed., *Configurable Computing: Technology and Applications, Proc. SPIE 3526*, pp. 150–161, Bellingham, WA, 1998. SPIE—The International Society for Optical Engineering.
8. K. V. Palem, S. Talla, and P. W. Devaney. Adaptive explicitly parallel instruction computing. In J. Morris, ed., *Proceedings of the 4th Australasian Computer Architecture Conference*, Singapore, Springer Verlag, 1999.
9. SIA—Semiconductor Industry Association. The international technology roadmap for semiconductors. http://www.itrs.net/, 2001.