The Two-dimensional Superscalar GAP Processor Architecture

Sascha Uhrig, Basher Shehan, Ralf Jahr, Theo Ungerer
Department of Computer Science
University of Augsburg
86159 Augsburg, Germany
{uhrig, shehan, jahr, ungerer}@informatik.uni-augsburg.de

***Abstract*—In this paper we evaluate the new Grid Alu Processor architecture that is optimized for the execution of sequential instruction streams. The Grid Alu Processor architecture comprises an in-order superscalar pipeline front-end enhanced by a configuration unit able to dynamically issue dependent and independent standard machine instructions simultaneously to the functional units, which are organized in a two-dimensional array. In contrast to well-known coarse-grained reconfigurable architectures no special synthesis tools are required and no configuration overhead occurs. Simulations of the Grid Alu Processor showed a maximum speedup of 2.56 measured in instructions per cycle compared to the results of a comparable configured SimpleScalar processor simulator. Further improvements introduced in this paper lead to a top speedup of about 4.65 for one benchmark. Our evaluations show that with restricted hardware resources, the performance is only slightly reduced. The gain of the proposed processor architecture is obtained by the asynchronous execution of most instructions, the possibility to issue multiple depending instructions at the same cycle and the acceleration of loops.**

***Keywords*-High Performance Processors, Reconfigurable Architecture, Instruction Level Parallelism**

I. Introduction

Current processor research focuses especially on multi-core architectures. Unfortunately, these architectures perform only well with parallel applications but they cannot increase the performance of sequential programs. The **G**rid **A**lu **P**rocessor (GAP) presented in [23] is designed to improve the execution of sequential instruction streams.

In general, control flow based applications contain small data-driven code sequences that can be accelerated by special hardware. Because of the differing demands of the applications, a general accelerator for all requirements cannot be found.

Application specific processors together with hardware/software codesign methodologies [17] contain at least one functional unit or functional block, which can be adapted to a concrete application. Mostly, these functional blocks are generated at the hardware-design phase i.e., the special demands of the application's software need to be known statically. Generally, the special functions consist of very simple operations like a single MAC operation or something similar.

Dynamically reconfigurable systems [6] comprise an array of functional units that can be reconfigured during runtime. However, the configuration of the array has special demands on the development tool chain and reconfiguration is time consuming. Hence, replacement of the operation should be well scheduled, otherwise the replacement overhead exceeds the gain in execution time.

Another issue is the global clock synchronous execution, which hampers the execution of simple instructions, because the working frequency depends on the most complex operation performed in any pipeline stage therefore decelerating simple and fast instructions. Our approach applies asynchronous timing between functional units, which allows the execution of simple operations in less than a cycle.

The GAP Architecture mixes the advantages of a superscalar processor and those of a coarse-grained dynamically reconfigurable system. A special configuration unit issues synchronously instructions to the processor back-end, consisting of an array of **F**unctional **U**nits (FUs), a branch unit, and several load/store units. Thereby even very small dataflow oriented instruction sequences benefit from the reconfigurable system-like architecture of the GAP. Instructions are executed within a combination of an asynchronous array of FUs and synchronous memory access and branch units. Due to the design of the array it is possible to issue independent as well as dependent instructions in the same clock cycle. Additionally to the improved issue and execution behavior, instructions issued to the array automatically form dataflow structures similar to the accelerator blocks within an application specific processor.

The contributions of this paper are a detailed description of the GAP architecture. Additionally, several optimizations are presented and evaluated that further improve the performance and, in parallel, decrease the hardware requirements of the GAP.

The next section provides a summary of coarse-grained reconfigurable systems, which are closest to the GAP architecture. An overview of the GAP is given in Section III together with an example of its functionality. Section IV describes the GAP architecture in detail and presents several optimizations of the basic architecture. A performance evaluation is shown in Section V. Section VI concludes the paper and identifies future work.

II. Related Work

Most reconfigurable architectures are attached to a master processor as an additional unit like a coprocessor. The master processor has to fulfill three tasks:

1) It has to provide configuration data to the reconfigurable system.
2) It has to take care about the activities of the reconfigurable system.
3) It executes the program parts that cannot be directed to the reconfigurable system.

Additionally, the presence and the architecture of the reconfigurable part must be taken into account at the software development phase. Example architectures with reconfigurable parts as coprocessor are the MOLEN [24], [25], [22], the GARP [10], and the MorphoSys [11] architectures. In contrast, the XTENSA chip from Tensilica, the CHIMAERA [9] project, and the PRISC processor [14] use reconfigurable functional units within the processor pipeline, see [6] for a more detailed overview.

The reconfigurable functional units within these processors are additional units besides the standard functional units of a superscalar processor. The additional units have to be accessed explicitly by the software. In contrast, the GAP features a homogeneous array of reconfigurable units that are transparent to the software.

A processor using a small reconfigurable architecture to accelerate statically determined parts of a program is presented by Clark et al. [4]. More flexible is the VEAL system [5] where the VEAL virtual machine is responsible to deal with the configuration of the loop accelerator online. Hence, an overhead for the dynamic compilation occurs. The PACT XPP architecture [13] contains several small VLIW processor cores located besides the reconfigurable array. These processors are responsible for the control driven parts of an application while the array offers high-throughput data operations. The WARP processor presented by Lysecky et al. [12] contains a FPGA-like accelerator, an on-chip profiler, and an on-chip CAD module (a separate CPU). The profiler identifies critical parts of the software and they are translated into a FPGA configuration online by the CAD module. Santos et al. [15] presented the 2D-VLIW processor. It comprises a pipelined two-dimensional array of functional units, which are configured by VLIW instructions containing as many operations as FUs are available. A similar approach is the ADRES reconfigurable processor [1], [26]. It comprises a VLIW front-end with a subordinated coarse grained reconfigurable array.

The RAW microprocessor [18], [19] and the EDGE architecture [3] implemented by the TRIPS microprocessor seem to be comparable at first glance because of their tiled architectures. However, the EDGE architecture introduces a new type of instruction set architecture (ISA) as one of its core concepts. Instructions in this ISA also include information about how to map operations on the tiles. This placement has to be calculated statically by the tool-chain in advance. Hence, a special compiler is required and normal instruction streams cannot be executed. The RAW microprocessor focuses on scalability of the number of used tiles (and therefore the used number of transistors) and wire delays. But if more than one tile shall be used by a program, again a special compiler is needed to map the application. This architecture therefore has some properties of many-core systems but cannot speed up single threaded applications without a special compilation technique.

None of the above mentioned architectures is able to directly execute a complete conventional sequential instruction stream within a two-dimensional array of reconfigurable functional units without any additional tool. The presented GAP architecture [21] exactly provides this feature.

III. Basic Architecture of the GAP

This section describes the basic idea of the GAP architecture and demonstrates the execution of a sample code fragment. A more detailed description of the different parts of the GAP are given in the next section.

A. Architectural Overview

The basic aim of the GAP architecture is that execution of normal programs shall profit from reconfigurable architectures. The GAP architecture combines the two dimensional arrangement of functional units (FUs) from reconfigurable systems together with the instruction fetching and decoding logic of a general purpose processor leading to the architecture shown in figure 1. Together with some additional components for branch execution and memory accesses (at the west side and the east side of the array), the GAP is able to execute conventional program binaries within its two dimensional array of FUs. Small loops will be mapped into the array and profit from the special architecture. All other non-loop-based program sequences are executed also within the array, and thus, they can likewise profit from the asynchronous execution.

In contrast to conventional superscalar processors, the GAP is able to issue also data-dependent instructions to the corresponding FUs within the same clock cycle. As consequence, GAP as in-order issue processor, reaches an issuing throughput similar to that of an out-of-order processor but does not require any out-of-order logic like issuing buffers, reorder and commit stages, as well as register renaming. However, execution of instructions is more like data driven processing within a reconfigurable architecture. Additionally, instruction execution is not done in a simple clocked way, rather it is possible to execute several dependent operations in one machine clock cycle. Instruction execution and timing is described in Section IV-D.

The array is arranged in rows and columns of FUs (see Figure 2) that are described in Section IV-C in detail. At the basic GAP architecture each column is attached to an architectural register of the processor's instructions set. We chose the PISA (Portable Instruction Set Architecture) instruction set known from the SimpleScalar simulation tool set [2]. Hence, the basic array contains 32 columns of FUs for the general purpose registers and an additional special column

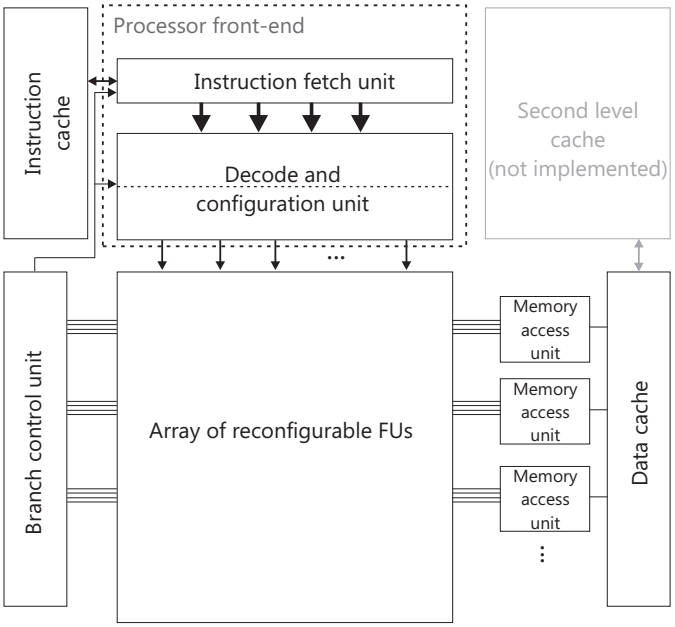

Fig. 1. Block diagram of the GAP core

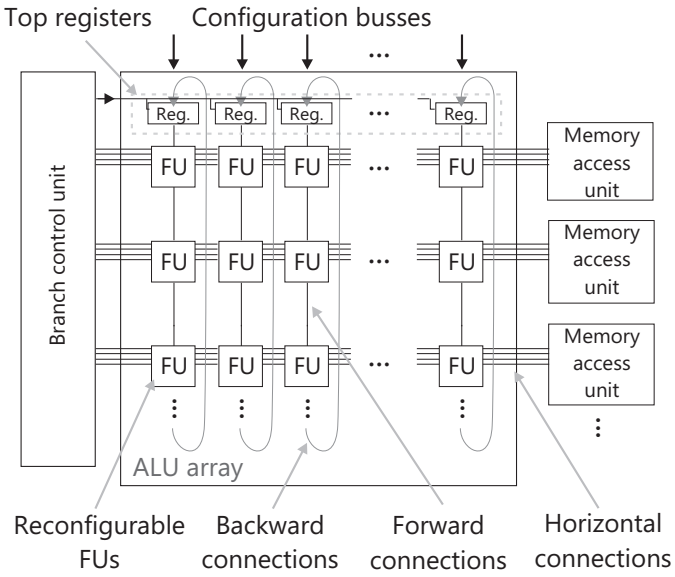

Fig. 2. General organization of the ALU array

for the multiply/divide registers (hi/lo) with the corresponding FUs (one multiply/divide unit per row). The number of rows depends on the maximum length of loop bodies that should be accelerated. An evaluation of the required number of rows is given in Section V.

The general data flow within the array is from the north to the south. A single 32 bit register is arranged at the top of each column and is called *top register*. This register contains the starting value of the corresponding architectural register of the column for the next execution phase. Each FU is able to read the output values of every FU from the previous row as inputs and its output is available for all FUs in the next row. So there is no data exchange within a single row and no data can be moved to the rows up. In general, each FU is configured by exactly one machine instruction or it is configured as route forward i.e., the FU bypasses the data from the previous FU in the same column to the next row.

Each array row is accompanied with a memory access unit that serves as communication interface to the data cache. The memory access units read the address from the previous row and forward data to a FU input of the same row in case of a load. A store receives the address as well as the store value from the previous row.

The single branch control unit on the west of the array has connections to all rows. Its task is to determine if conditional branches have to be taken. For this purpose, it checks the output value of a given FU against the branch condition. If the condition is true, the actual execution phase is stopped and the currently valid values in the corresponding row are copied from all columns to their top registers.

B. Code Execution Example

The placement of instructions into the array is demonstrated by the following simple code fragment. The pseudo machine instructions add fifteen numbers out of subsequent memory cells followed by negating the result.

```
1:    move R1,#15     ;15 values
2:    move R2,#addr   ;start address
3:    move R3,#0      ;Register for the sum loop:
4:    load R4,[R2]    ;load an element
5:    add R3,R3,R4    ;add
6:    add R2,R2,#4    ;inc address
7:    sub R1,R1,#1    ;dec counter
8:    jmpnz R1,loop   ;end of loop?
9:    sub R3,R1,R3    ;negate the result (R1=0)
```

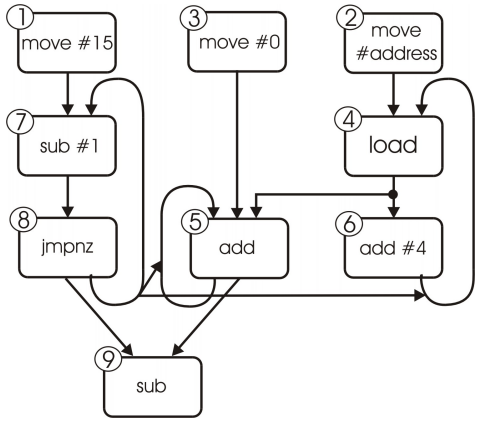

Fig. 3. Dependency graph of the example instructions.

Figure 3 shows the dependency graph of the 9 instructions, which can be recognized again at the placement of the instructions within the GAP back-end shown in Figure 4. The instructions 1 to 3 are placed within the first row of the array. Instruction 4 (`load`) depends on *R*2 that is written in the first row and therefore, it must be located in the second row. It reads the address as a result of instruction 2 and forwards the

data received from the memory into the column *R*4, which is the destination register of the load. The instructions 5 to 7 are placed in an analog way. Instruction 8 is a conditional branch that could change program flow. To sustain the hardware simplicity, conditional branches are placed below the lowest already arranged instruction. In this case, the branch has to be located after the third row. The last instruction is placed after the branch in the fourth row. Hence, if the branch is taken, the GAP takes a snapshot of the register contents at the corresponding row and copies the data into the top registers. In this case, instruction 9 is discarded. Otherwise, the *sub* is executed without any time loss.

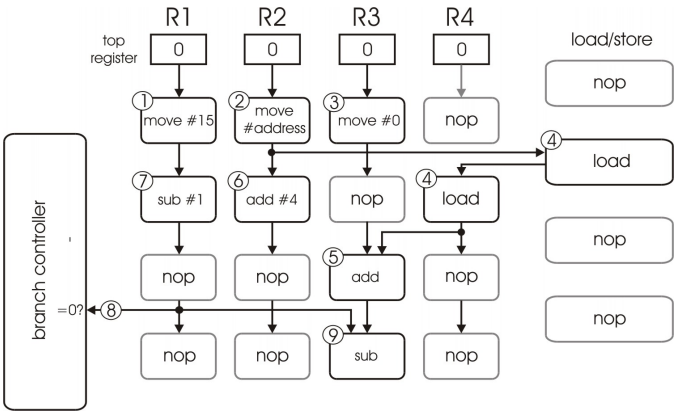

Fig. 4. Placement of the complete example fragment

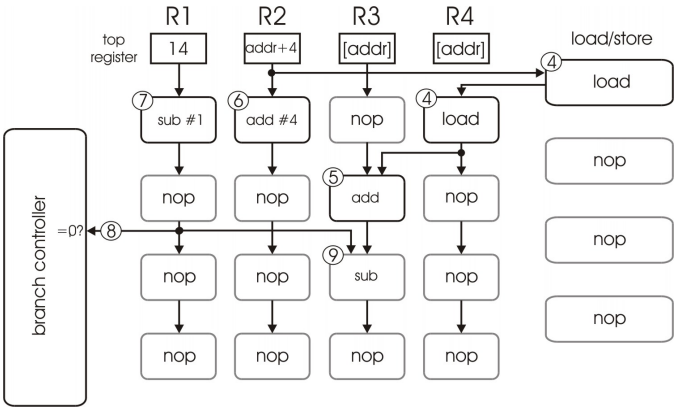

Fig. 5. Placement of the loop body (instructions 4 to 8) and the subsequent instruction (instruction 9)

After the first execution of the loop body, the backward branch to the label *loop* has to be performed. The branch target is already mapped into the array but it is placed among other instructions that must not be executed again. To keep the hardware simple, the GAP is only able to copy data into the top registers of the first row. The decoding starts again with the new address. The newly started decoding maps only the loop and the following instructions into the array (see Figure 5). Now, the loop target is the first instruction within the array, and hence, all subsequent loop iterations can be executed directly within the array without any additional fetch, decoding and configuration phases. Due to the placement of instructions following the loop body, the GAP does not suffer from any misprediction penalty at loop termination as other processors using branch prediction would do.

IV. Detailed GAP Description

A detailed description of the superscalar pipeline front-end, the back-end architecture, the execution and timing of the operations, the memory hierarchy, and several optimizations are presented in this section.

A. Superscalar Processor Front-end

Instructions are fetched from the instruction fetch unit out of a normal program binary generated by a standard compiler e.g., the GCC. A program counter determines the position of the next instructions to fetch. The current implementation fetches four instructions in parallel. During decoding, each instruction is treated in one of the following ways:

- **Unconditional direct branches:** These instructions are executed directly within the decode stage i.e., the program counter is set to the given value and the decoding of the subsequent instructions is cancelled.
- **Arithmetic-logic instructions:** All arithmetic and logic instructions are register-register instructions (a load/store instruction set architecture like PISA is assumed). All decoded arithmetic/logic instructions are placed into the array of FUs. The column in which an operation is placed is determined by the output register of the instruction and the row depends on the availability of the input data and the last read access to the output register. Input data is available after the last write access to the corresponding register i.e., the lowermost instruction placed within the column. Because instructions are only placed and not executed, it is possible to place independent as well as dependent instructions into the array at the same cycle.
- **Conditional branches and register indirect jumps:** The branch control unit is able to handle one branch or jump instruction per row of the array. Because both types of control flow changes (conditional branches and register indirect jumps) depend on the content of at least one register, branches/jumps have to be placed below the row in which the value of the corresponding register is calculated. Additionally, it is necessary for the loop acceleration to catch all register values out of a single row (after one loop iteration) and copy them into the top registers for the next iteration. Hence, branches/jumps are placed below the lowest operation of the loop body within the array.
- **Load/store instructions:** Memory instructions are executed by the memory access units. Each unit is able to execute one load or one store instruction. A load instruction is placed as high in the array as possible i.e., below the last write access to the address register and the last read access to the destination column. Store instructions additionally depend on preceding branches. All memory operations are equipped with an ordinal

number that prevents load instructions to be executed before a foregoing store instruction.

The front-end is active as long as the array is not filled up to the last row. If the array is full, the back-end is busy with executing the instructions already placed in the array. This could happen if some long latency operations like load instructions have to be executed or if a loop is executed. In both cases, the front-end can switch into sleep mode to save energy.

When the execution of the operations inside the array reaches the last row or when the branch controller signals a taken branch, the front-end resumes to fetch and decode instructions. If needed, the program counter is set to the new value given by the branch controller beforehand.

The processor front-end is connected to the back-end by several configuration lines. Because four instructions are decoded in a single cycle, also up to four operations could be issued to a column in the array. Hence, the maximum of four configuration busses per column, including the branch controller and the memory access units, are required. But, due to our evaluations we find out that only two busses per column are sufficient. If more than two instructions (out of four) must be issued to the same column, an additional cycle is required. The configuration unit is aware of this circumstance.

The only feedback signals from the back-end to the front-end are the signals from the branch controller to the fetch unit (a new program counter and a valid signal) and a *finished* signal. The *finished* signal indicates that the execution has reached the end of the array.

B. Timing Analysis at the Front-end

Besides the placement of the operations into the array, the front-end is also responsible to determine the timing. Because of the asynchronous execution of the operations, the time at which a valid data word arrives at a particular location is not known. To synchronize the synchronous units (see Section IV-C) with the asynchronous execution within the array, the configuration unit is aware of the timing of each operation. Therefore, the runtime of each operation is known by the configuration unit in terms of so-called *pico-cycles*. We have chosen one pico-cycle as $\frac{1}{4}$ of a machine clock cycle. At each configuration step i.e., the assignment of one operation, the configuration unit calculates the time at which the FU output is valid. Inside the configuration unit one pico-cycle counter is responsible for each architectural register i.e., each column. The number of pico-cycles required for the current operation is added to the longest path of the source values. For the evaluations, we assumed the execution times for the operations as shown in Table I. These assumptions are based on the fact that some operations require a carry chain (*add, sub, setlt, setgt*), require a shifter (*shl, shr*), or are flat logic (*and, or, xor, not*).

If the resulting pico-cycle counter exceeds one machine clock cycle, a so-called *timing token register* inside the FU is activated (see Section IV-D) and the counter is decreased by four. Otherwise, the timing token register of the FU is bypassed i.e., the timing token passes immediately. The pico-cycle counter is implemented as a two bit counter and the overflow indicates that the token register has to be activated. Due to the overflow, decreasing the counter is not required explicitly.

Operation	Pico-cycles
Add, Sub	3
Setlt, Setgt	3
And, Or, Xor, Not	1
Shl, Shr	2

TABLE I
PICO-CYLE TIMES ASSUMED FOR THE EVALUATION

In a real implementation of GAP the longest signal path depends on the technology and the placement. Hence, the pico-cycles required for an instruction class are not known at design time (before synthesis). To address this problem, they can be stored in an internal look-up table (RAM) that is initialized with the maximum pico-cycle value (four pico-cycles in this case) at system reset. A test software determines the actual value per instruction class and sets the entries in the table accordingly.

C. Back-end Architecture

The GAP back-end comprises the FU array, the branch controller, and the memory access units (see Figure 2). The branch controller and the memory access units are designed as synchronous units while the FU array is asynchronous (except for the configuration part and the timing token register). The three parts are described below.

- **Functional unit array:** The array consists of a two-dimensional arrangement of identical FUs. Each FU contains an ALU that is able to perform the following operations: *+, -, shl, shr, and, or, xor, not, neg, setlt, setgt*. Besides each ALU, a configuration register including a constant value, two input multiplexers, and the token logic for the timing is present. Moreover, a bypass multiplexer is available that allows the route-forward operation i.e., the output value of the previous FU in the same column is routed to the next row. The block diagram of a complete FU is shown in Figure 6. The input multiplexers are able to select the output of each FU in the row above or the top registers, respectively. Because of the asynchronous execution inside the array, no registers are located in the data path of the FU. In contrast, the configuration network is fully synchronous and the register for the timing token can be enabled by configuration.
- **Branch controller:** The branch controller comprises a configuration register, two input selectors, and a compare-to-zero comparator per row of the array. Additionally, the branch controller offers a bus for the new program counter (for taken branches) to the processor front-end together with a *valid*-signal. Besides controlling branches, this unit enables the top registers to store the values delivered by the feedback network. It is also aware of the time when all operations in the array are finished i.e., when all timing

tokens arrived at the last row. Furthermore, the branch controller manages the loop acceleration i.e., if the branch target is the first instruction that is already configured in the array, the configuration is left unmodified. Only the new data is copied into the top registers in this case and the timing tokens are activated at the top row.

- **Memory access units:** The memory access units are synchronous units. They are responsible for read and write operations of the array to the main memory. A read or write operation occurs if the software executes a load or store instruction, respectively. To reduce the access time, small local caches containing a single cache line are present in each memory access unit (see Section IV-E). Each memory access unit contains a configuration register, two input selectors, and an output bus. The input selectors can choose any of the FUs' output of the upper row as address and write data input. The output bus directs the read data to any FU of the same row. An operation of the memory access unit is triggered at the time the required timing tokens are available. The output token is synchronously generated if the operation is performed.

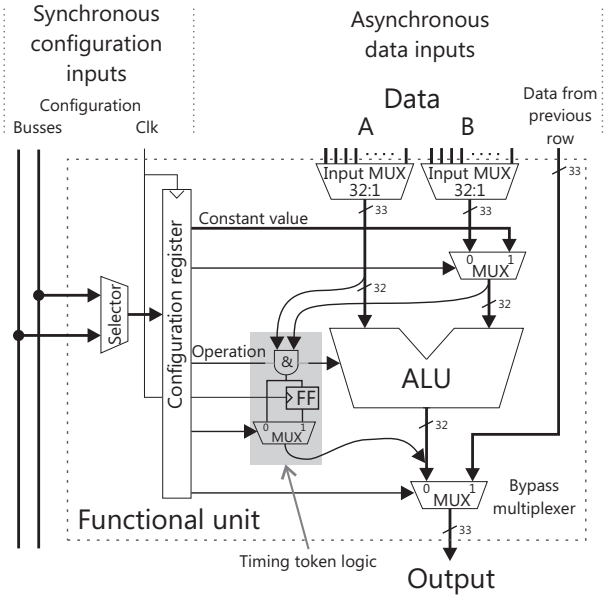

Fig. 6. Block diagram of a functional unit comprising an ALU, input selectors, bypass network, and a configuration register set by one of the two configuration busses

The units of the back-end are configured by different synchronous configuration busses. These busses range from the configuration unit of the front-end to the last row of the array. Because of a high number of rows, these busses could be comparatively long and, hence, have a negative impact on the maximum clock rate. To overcome this problem, a register can be integrated to pipeline the transportation of the configuration data and to shorten the maximum delay.

A much more interesting issue is the wire length of the horizontal connections. If the value of register zero (*R*0) is required at the memory access unit or the value of *R*31 is required at the branch controller, these values have to pass from the leftmost FU to the right side of the array or vice-versa, respectively. To get rid of these long wires together with the delay, we evaluated different array extensions described in Section IV-F. For our evaluations, we assumed that the maximum delay is taken into account by the pico-cycles required for an operation.

D. Instruction Execution and Timing

The front-end units of the GAP i.e., the instruction fetch, decode and configuration units, work in a synchronous way as well as the configuration paths within the array of FUs. The branch controller and the memory access units are also synchronous to a global clock signal. In contrast to these units, execution of the operations within the FUs is performed asynchronously.

So, the data stored within the top registers move to all FUs in the first row that are configured to use data out of at least one of these registers. The output of each FU in the first row moves to the inputs of the FUs in the second row corresponding to their configuration and so on.

To align synchronous parts with data from asynchronous execution, we introduced so-called *timing tokens*. These tokens propagate in parallel to each data word through the array. If a FU is configured for an operation with two registers as source operands, the FU sends a token if both source tokens are available. To generate the delay of the tokens, the FUs are equipped with an additional synchronous single bit register, which allows to bypass the token signal depending on its configuration. In general, these registers are in bypass mode. The idea is that a token moves ahead of the corresponding data till it reaches an active timing token register. Here, it has to wait until the next rising clock edge i.e., until the data at the associated FU is valid. The token registers are activated by the configuration unit if the longest data path from the current FU back to the FU with the last active token register or the top of the array is longer than one machine clock cycle. Hence, if a synchronous unit receives a token, the data at the input will be valid not later than the next rising edge of the clock signal.

Accordingly, the synchronous branch controller as well as the memory access units use the timing token of the incoming data as *data valid* signal. If it is set at the rising edge of the clock signal, the corresponding action must be performed. Additionally, an internal *done* state must be set to ensure that the action is done only once. Because of the nature of the timing token, it is not reset during a single execution iteration of one configuration. If a configuration is executed multiple times like a loop body or if the array is reconfigured, the *done* state must be reset.

E. Memory Hierarchy

Reaching a high memory throughput is a main requirement to get a high processor performance. The GAP back-end comprises one memory access unit per row. Hence, assuming

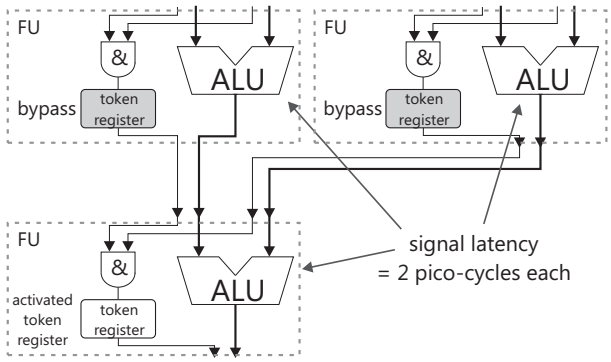

Fig. 7. Three instructions, each requires 2 pico-cycles. The token register of the upper FUs are bypassed and the one of the lower FU is activated

n rows also *n* memory accesses can occur simultaneously. To reduce the concurrency at the memory interface, each memory access unit contains its own small private cache. Because of hardware simplicity, the current evaluations assume only a single cache line within each memory access unit. These caches serve only for read accesses, writes go directly to the next memory hierarchy i.e., the data cache. For cache consistency, writes are also directed to all other memory access units, which compare the received write address to their own cache and update it, if necessary.

The memory write sequence is guaranteed by the back-end by allowing only a single write access in one cycle. This technique enables the memory access units to distribute writes through a shared write bus to the other memory access units and the data cache in parallel. Additionally, writes are executed in the original program order and take care of preceding taken branches. Hence, no unintended writes can occur. Between two consecutive writes all loads that are in the program order in between these writes can be executed simultaneously.

Memory throughput is increased by the data cache, which is a non-blocking cache [16]. All read accesses from the memory access units go to the data cache. If one request cannot be satisfied it is redirected to the next level cache (which is not implemented in our current simulator). Meanwhile, requests from other memory access units can be served by the data cache.

F. Architectural Optimizations

A problem of GAP arises from branches that can be resolved only at the execution time inside the array. These branches lead to a flush and reconfiguration of the array if they do not form a loop. Another shortcoming of the basic GAP architecture is the huge amount of chip area required for the 32 columns for 32 architectural registers. Additionally, the wire delay is a very important issue in modern chip design and the long horizontal wires have a big impact on the operating performance (not the clock frequency because the concerned parts are asynchronous). The following optimizations of the GAP architecture consider these drawbacks.

1) Branch Prediction: Like most other processors the GAP can profit from branch prediction. A taken conditional branch inside the array that does not implement a loop leads to a flush of the array and a new configuration phase. For this purpose, a branch prediction tries to increase the length of the dataflow graph mapped into the array by predicting the direction of conditional branches. A misprediction results in an array flush and starts a new configuration phase. Hence, the number of speculation levels of the GAP predictor is only limited by the number of rows because we allow only one branch per row.

In contrast to conventional speculative superscalar processors, the GAP does not require any commit logic that is aware of speculative execution. Instead, the register values are taken out of the row of the mispredicted branch and are copied into the top registers. Now, they are available for the further correct execution of the program.

2) Predicated Execution: Another possibility to reduce the number of flushes is to get rid of some conditional branches. Therefore, the optimized GAP is able to use predicated execution. In the case of a predicated instruction, it will be placed in the array like a normal instruction. An additional logic inside the FUs allows to use predication for all instructions. A flag is set in the configuration register of the corresponding FU, which indicates that the operation is predicated. If the predicate is true, the operation is executed but if it is false, the FU switches to route-forward mode. The predicate itself is determined by the branch control unit i.e., the original branch preceding the predicated instruction is not executed as a branch but it sets the corresponding predication signal appropriately. This technique is only useful for forward branches with very small jump distances because the predicated instructions have to be taken into account completely during the calculation of timing tokens.

3) Horizontal Array Dimension: The horizontal extension of the basic functional unit array is as high as the number of architectural registers. Using a standard RISC instruction set architecture like ARM, PISA, or openRISC (ORBIS32) requires 16 or 32 columns. The high number of columns leads to a high wire delay between the leftmost and the rightmost FUs of the GAP. Additionally, the input selectors of each FU must be able to select each of the outputs of the previous row resulting in 33-to-1 33 bit width multiplexers (32 registers plus one input for the memory access unit * 32 data bits plus the timing token).

Reducing the number of columns has been reached by decoupling the architectural registers from the array columns. The processor front-end supports a technique like the well-known register renaming [20]. But, in contrast to the standard register renaming that maps less architectural registers to a higher number of physical registers, GAP does it vice-versa: a high number of architectural registers is mapped to a lower number of physical registers i.e., columns.

Mapping is done by a mapping table with one entry per architectural register and a column counter. At an array flush, each entry in the mapping table is set to invalid and the counter is set to zero. Every time the result of an instruction should be written into an architectural register, the column is determined by the value in the corresponding entry of the mapping table.

If it is invalid, the content of the counter is written into the mapping table and the counter is increased. If the counter reaches the maximum value (the number of columns) no instructions writing to further architectural registers can be placed inside the array. The configuration stops at that time.

As a result of the lower number of columns, some configurations have to be split into multiple configurations for execution. In Section V-C we have evaluated how many configurations can be satisfied by a smaller array dimension.

V. Evaluation

The evaluation of the GAP architecture concerns the array size with different optimization steps. In all evaluations, the GAP features branch prediction unless otherwise noted. Besides the pure performance, we also evaluated the utilization of the FU array and the memory access units.

A. Evaluation Methodology

To evaluate the performance of the basic GAP architecture, we compared it to a superscalar processor simulated by the SimpleScalar (SS) tool set with the parameters given in Table II. Therefore, the GAP architecture is implemented as a cycle accurate simulator written in C++. The pipeline stages, the FU array, the branch controller, the memory access units and the cache are modelled in the GAP simulator. Both simulators execute the identical binary files.

Parameter	SimpleScalar	Basic GAP
L1 I-Cache	128:64:1	128:64:1
L1 D-Cache	1024:32:1	1024:32:1
Fetch, decode, and issue width	8-way	4-way
Multipliers	1	1 per row
Integer ALUs	8	31 per row
Branch prediction	bimod	bimod
Return address stack	8	none
Branch target buffer	512*4	none
Load/store queue	8 entries	one L/S unit per row
Memory latency	24	24

TABLE II
Parameters of both simulators

At first glance, it seems not to be fair to compare the SimpleScalar architecture with 8 ALUs to the GAP with up to 992 ALUs (31 columns x 32 rows, *R0* is fixed to 0). But we first evaluated the IPC rates of the SimpleScalar configured by the given parameters and with different numbers of ALUs to identify its peek performance. Therefore, we used the same benchmarks as for the evaluation of the GAP. The results presented in figure 8 show only a marginal increase in average IPC using more than 4 ALUs. The reason is that the issue unit is not able to find enough independent instructions to execute in the same cycle. To overcome this bottleneck, the fetch width and the decode width have to be increased together with the size of the Register Update Unit (RUU). But increasing the issue complexity would influence the maximum frequency dramatically as stated by Cotofana et al. [7]. To be conservative, we used twice the fetch and decode width for the SimpleScalar than for the GAP and an RUU size of 128 for the SimpleScalar while the GAP does not need any issue queue.

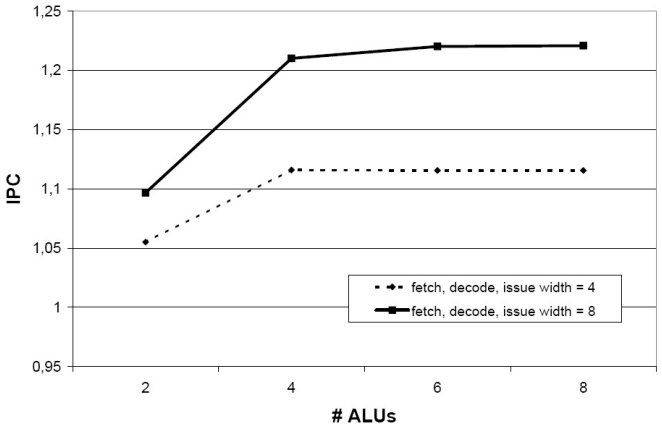

Fig. 8. Average IPC rates of all benchmarks using the SimpleScalar with different number of ALUs and a fetch/decode width of 4 and 8, respectively.

Several benchmarks out of the MiBench [8] benchmark suite without floating point operations are selected to determine the GAP's performance. As input data, the *small* data sets are used. The *stringsearch* benchmark is an adapted version of the original benchmark reduced to the central algorithm without any output. Unless otherwise stated, the identical binary files are used for the GAP simulation as well as for the SimpleScalar.

B. Performance of the GAP Architecture With Branch Prediction and Predicated Execution

The first performance evaluation concerns two different optimization steps of the GAP: the basic architecture with integrated branch prediction and the basic architecture with branch prediction and additional predicated execution. The latter requires a slight modification of the execution binary because of the predication bits. This modification is done only by an analysis of the executable and not by compiler options. As GAP array size we have chosen 4, 8, 16, and 32 rows and always 31 columns of FUs, one multiplier and one memory access unit per row.

Figure 9 shows the IPC rates of the benchmarks using the GAP compared to the SimpleScalar. Using 16 or 32 rows, the GAP outperforms the SS in all cases, except for *bitcount*. The benchmarks *adpcm*, *dijkstra*, and *stringsearch* result in a higher IPC even with only 4 rows. On average the GAP with 4 rows reaches a slightly higher IPC than the SS. The top performance is reached by the *SHA* benchmark with 32 rows: The IPC is 2.56 times the IPC of the SimpleScalar.

After adapting the executable binary files with predication bits, we simulated the same benchmarks with enabled predicated execution. Figure 10 shows the results of these measurements. The predicated execution achieves a slightly better IPC on average with 32 rows. But, in particular the *adpcm* obtains a significantly higher performance in all row configurations

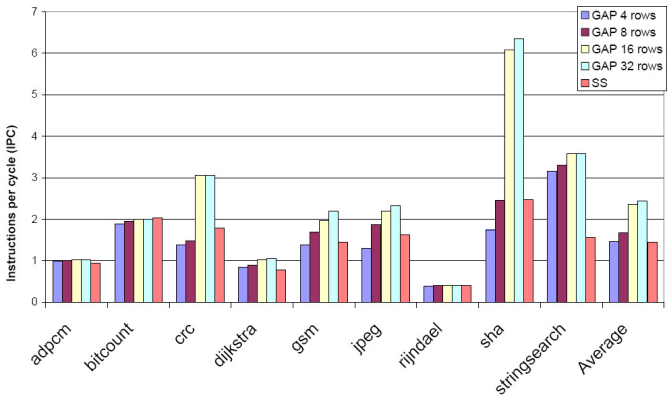

Fig. 9. IPC rate of the GAP with branch prediction compared to the SimpleScalar.

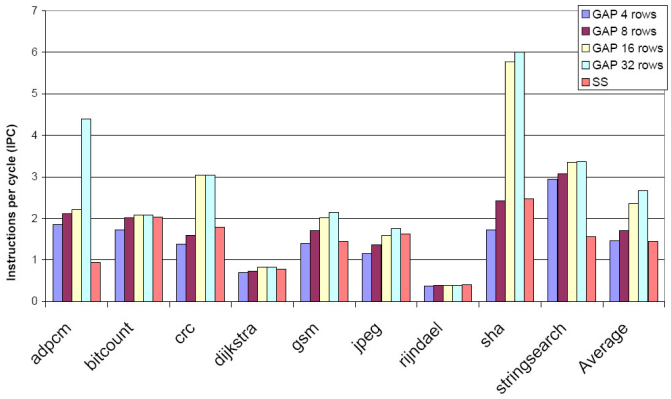

Fig. 10. IPC rate of the GAP with branch prediction and predicated execution compared to the SimpleScalar.

(nearly 2-fold with 4 rows up to 4.65-fold with 32 rows) while *dijkstra* and *rijndael* show a performance loss. The *bitcount* demonstrates an unexpected behavior: The performance of the 4-row version is decreased while the configurations with 8, 16, and 32 rows show a performance gain. All other benchmarks reach at least the same performance with enabled predicated execution than without.

The performance loss of some configurations/benchmarks can be explained by the timing of the execution inside the array. If a short forward branch, which is mostly taken, is substituted by predicated execution, the predicated instructions increase the longest path inside the array without being executed. Hence, the time to execute a complete configuration is also increased, which in turn reduces the IPC. In the case of the dramatic performance increase of the *adpcm*, which reaches with 32 rows an IPC of 4.65 times the IPC of the SS, the predicated execution shows its positive aspects: branches that are resolved by predicated execution are hard to predict, which leads to a better performance by using predicated execution. The following evaluations are performed only with branch prediction without predicated execution.

C. Utilization of the GAP Back-end

The basic architecture of the GAP consists of one column per architectural register and one memory access unit per row. At the maximum configuration evaluated in Section V-B this results in 992 FUs, 32 multiplier/dividers, and 32 memory access units. The second evaluation step concerns the number of FUs and memory access units that are really required to reach the performance mentioned above. Because of the direct relationship of the destination register as well as the dependencies of an instruction to its placement inside the array, we expect a very low utilization of the FUs.

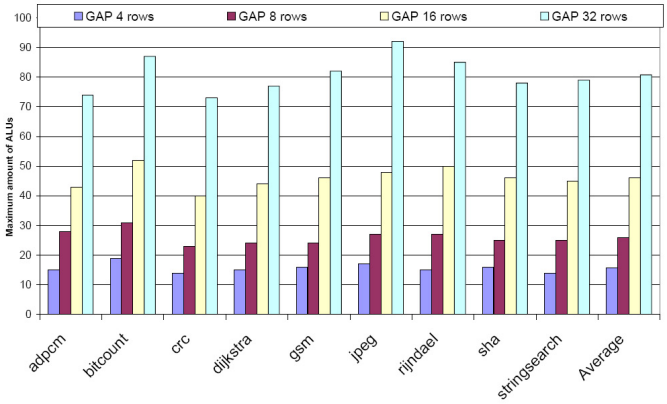

Fig. 11. Maximum number of used FUs per configuration with branch prediction.

Figure 11 shows the amount of FUs, which are used for real operations (not as route forward) with enabled branch prediction. The average overall utilization using 32 rows is about 81 FUs. That means, the utilization with 32 rows ranges from 6.7% up to 8.2%. Using only four rows, the GAP needs just 15 FUs on average i.e., 12% of its overall FUs. Although more than 64 FUs are used by the configuration with 32 rows in every benchmark, a maximum array with 64 FUs is reasonable. This is because the presented values show the maximum utilization by the benchmarks; the average utilization is much lower.

The number of allocated memory access units is shown in Figure 12. In the four and eight row configurations, every benchmark requires all available memory access units. At the 16 row version, *crc* uses only 14 and *jpeg* as well as *rijndael* and *dijkstra* all 16 memory access units. The other benchmarks allocate 15 memory access units. A saturation of memory access units is reached at the 32 row version of GAP: The maximum number of memory access units (31) is allocated by *jpeg* while *stringsearch* requires only 23.

These measurements indicate that a high number of memory access units seems to be required to get a good performance result. But, the presented values are only the maximum number of allocated memory access units i.e., they do not figure out if this particular configuration is relevant for the overall performance.

To clarify this question, the maximum number of rows that are used to accelerate loops is presented in Figure 13. Because

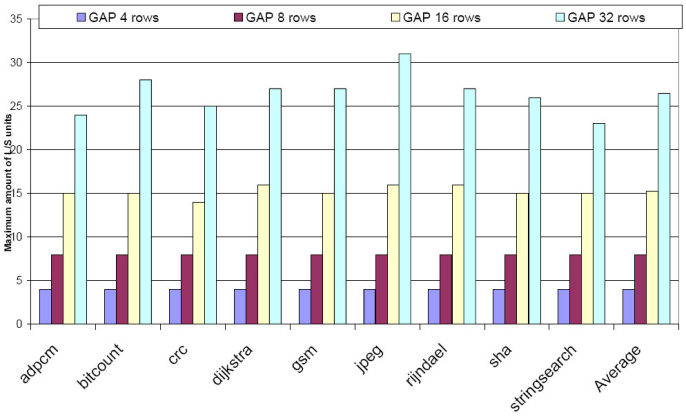

Fig. 12. Maximum number of used load/store units per configuration with branch prediction.

the GAP architecture supports only one memory access per row, the number of memory access units required for loop acceleration cannot be higher than the number of used rows. Hence, with only 17 memory access units most benchmarks would perform well if 32 rows are available. With less rows only 3, 6, resp. 12 units are required on average.

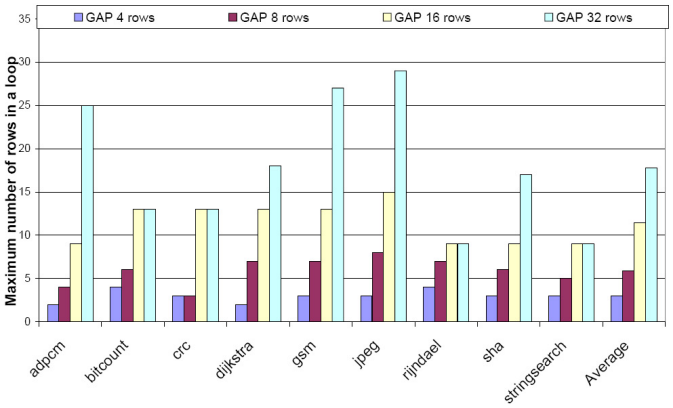

Fig. 13. Maximum number of used rows at loop acceleration with branch prediction.

Another area consuming resource is the multiply/divide unit. Although each row of the array contains a multiply/divide unit, the evaluations show a clear result: only one multiply/divide unit is required for all benchmarks and all configurations. Hence, a single multiply/divide unit, which can be accessed by all rows would be sufficient.

Besides the number of rows the number of columns also dictates the hardware effort. To find out a minimum array width we measured the number of used columns per configuration i.e., the number of columns with at least a single issued instruction. The histogram in Figure 14 shows how much configurations could be satisfied by a certain number of columns. Therefore we conservatively assumed a maximum array length of 32 rows i.e., using a shorter array would also lead to smaller configurations. As a result we can notice that only 8 columns are required to satisfy 95% of all configurations emerging in

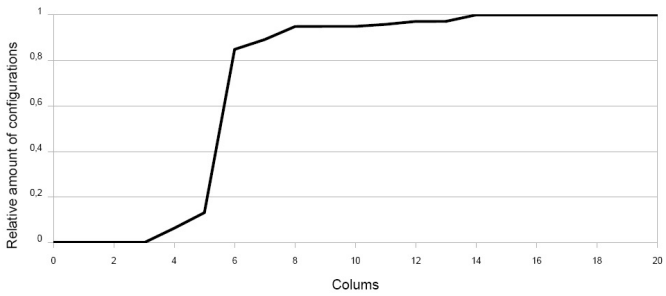

Fig. 14. Ratio of the satisfied configurations depending on the number of columns.

all applied benchmarks. With 14 columns, 99% and with 20 columns all of the configurations can be mapped completely into the array.

Besides the lower number of FUs also the input multiplexers of the FUs shrink if the array width is decreased. Choosing an array width where not all configurations can be issued completely does not mean that some applications cannot be executed at all. Rather these configurations require an additional array flush resulting in an additional configuration phase.

VI. Conclusion and Future Work

We presented the new GAP architecture comprising a processor back-end similar to a two-dimensional reconfigurable array. In contrast to well-known reconfigurable architectures the GAP is able to execute a conventional instruction stream generated by a standard compiler. The comparison to an out-of-order superscalar processor simulator shows a maximum performance factor of 2.56 when using the GAP with branch prediction and 4.65 when using it with additional predicated execution.

The great advantage of the GAP architecture is the improvement of sequential program execution that cannot be provided by modern multithreaded and multicore architectures. Besides the performance, we also evaluated different sizes of the FU array and its utilization to find an optimal array size. The measurements show that the number of allocated FUs is much less than the total amount of FUs (about 6-12%).

In future, we will further improve the performance by implementing multiple configuration layers, which speed up frequently used configurations and allow to decrease the maximum number of rows and columns. Multiple configuration layers would support especially function calls and configurations that do not fit into a single configuration layer. Hence, the layers will at least compensate the drawback of a smaller array. Moreover, a possible misprediction penalty can be eliminated completely by having the target instruction stream already configured within another layer.

Further optimizations concern the horizontal connection network. Because the current implementation allows to transport data from any FU of one row to any FU of the next row, complex multiplexers are required at each data input of the FUs. To reach simpler horizontal connections, we will reduce

the number of horizontal busses. This reduction will restrict the number of output values that can be transported to the inputs of the next row's FUs. Hence, the configuration unit must be modified to take care of that circumstance. If more data must be transferred to succeeding FUs than busses are available, the concerned operations must be placed in the following row.

REFERENCES

[1] F. Bouwens, M. Berekovic, A. Kanstein, and G. Gaydadjiev. Architectural exploration of the adres coarse-grained reconfigurable array. In *ARC*, pages 1–13, 2007.

[2] D. Burger and T. Austin. The simplescalar tool set, version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3):13.

[3] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. H. Burrill, R. G. McDonald, and W. Yode. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, 2004.

[4] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles, and K. Flautner. An architecture framework for transparent instruction set customization in embedded processors. In *Proc. of the International Symposium on Computer Architecture*, pages 272–283, 2005.

[5] N. Clark, A. Hormati, and S. Mahlke. VEAL: Virtualized execution accelerator for loops. In *Proc. of the International Symposium on Computer Architecture*, pages 389–400, 2008.

[6] K. Compton and S. Hauck. Reconfigurable Computing: A Survey of Systems and Software. *ACM Computing Surveys*, 34(2):171–210, June 2000.

[7] S. Cotofana and S. Vassiliadis. On the design complexity of the issue logic of superscalar machines. *EUROMICRO Conference*, 1:10277, 1998.

[8] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and T. Brown. Mibench: A free, commercially representative embedded benchmark suite. *4th IEEE International Workshop on Workload Characteristics*, pages 3–14, December 2001.

[9] S. Hauck, T. Fry, M. Hosler, and J. Kao. The Chimaera Reconfigurable Functional Unit. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 1997)*, pages 87–96, 1997.

[10] J. Hauser and J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 1997)*, pages 12–, 1997.

[11] M.-H. Lee, H. Singh, G. Lu, N. Bagherzadeh, F. J. Kurdahi, E. M. Filho, and V. C. Alves. Design and implementation of the morphosys reconfigurable computingprocessor. *Journal of VLSI Signal Processing Systems*, 24(2–3), March 2000.

[12] R. Lysecky, G. Stitt, and F. Vahid. Warp processors. *ACM Trans. Des. Autom. Electron. Syst.*, 11(3):659–681, 2006.

[13] PACT XPP Technologies, July 2006. *http://www.pactxpp.com/main/download/XPP-III_overview_WP.pdf*.

[14] R. Razdan and M. D. Smith. A High-Performance Microarchitecture with Hardware-Programmable Functional Units. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 172–80, 1994.

[15] R. Santos, R. Azevedo, and G. Araujo. 2d-vliw: An architecture based on the geometry of computation. In *ASAP '06: Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors*, pages 87–94, Washington, DC, USA, 2006. IEEE Computer Society.

[16] J. Sicolo. A multiported nonblocking cache for a superscalar uniprocessor. Master's thesis, Department of Computer Science, University of Illinois, Urbana IL, 1990.

[17] F. Sun, S. Ravi, and N. K. Jha. A scalable application-specific processor synthesis methodology. In *Proceedings of the International Conference on Computer-Aided Design*, pages 9–13, 2003.

[18] M. B. Taylor, J. S. Kim, J. E. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmann, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. P. Amarasinghe, and A. Agarwal. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.

[19] M. B. Taylor, W. Lee, J. E. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. S. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. P. Amarasinghe, and A. Agarwal. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and streams. In *ISCA*, pages 2–13. IEEE Computer Society, 2004.

[20] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. pages 13–21, 1995.

[21] S. Uhrig. Processor with internal grid of execution units. Patent pending, WO 2007/143972 A3.

[22] S. Uhrig, S. Maier, G. Kuzmanov, and T. Ungerer. Coupling of a reconfigurable architecture and a multithreaded processor core with integrated real-time scheduling. In *13th Reconfigurable Architectures Workshop (RAW 2006), Rhodos, Greece*, Apr. 2006.

[23] S. Uhrig, B. Shehan, R. Jahr, and T. Ungerer. A two-dimensional superscalar processor architecture. In *The First International Conference on Future Computational Technologies and Applications, FUTURE COMPUTING 2009, Athens, Greece*, November 2009.

[24] S. Vassiliadis, S. Wong, and S. D. Cotofana. The molen pmu-coded processor. In *In in 11th International Conference on Field-Programmable Logic and Applications (FPL), Springer-Verlag Lecture Notes in Computer Science (LNCS) Vol. 2147*, pages 275–285, August 2001.

[25] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte. The molen polymorphic processor. *IEEE Transactions on Computers*, 53(11):1363–1375, 2004.

[26] F.-J. Veredas, M. Scheppler, W. Moffat, and B. Mei. Custom implementation of the coarse-grained reconfigurable ADRES architecture for multimedia purposes. *International Conference on Field Programmable Logic and Applications*, pages 106–111, 2005.