A cycle-approximate, mixed-ISA simulator for the KAHRISMA architecture

Timo Stripf, Ralf Koenig, Juergen Becker
Karlsruhe Institute of Technology, Karlsruhe, Germany
{stripf, ralf.koenig, becker}@kit.edu

***Abstract*—Processor architectures that are capable to reconfigure their instruction set and instruction format dynamically at run time offer a new flexibility exploiting instruction level parallelism vs. thread level parallelism. Based on the characteristics of an application or thread the instruction set architecture (ISA) can be adapted to increase performance or reduce resource/power consumption. To benefit from this run-time flexibility automatic selection of an appropriate ISA for each function of a given application is envisioned. This demands a cycle-accurate simulator that is capable of measuring the performance characteristics of an ISA dependent on the target application. However, simulation speed of a cycle-accurate simulator of our reconfigurable VLIW-like processor instances featuring dynamic operation execution would become relatively slow due to the superscalar-like microarchitecture. Within this paper we address this problem by presenting our cycle-approximate simulator approach containing a heuristic dynamic operation execution and memory model that provides a good trade-off between performance and accuracy. Additionally, the simulator features measurement of instruction level parallelism (ILP) that could be theoretically exploited by VLIW processor instances running on our architecture. The theoretical ILP could be used as an indicator for the ISA selection process without the need to simulate any combination of the different ISAs and applications.**

I. Introduction

Within the KAHRISMA project [1] we research a novel multi-grained hypermorph reconfigurable processor architecture. Other reconfigurable processor architectures typically focus on dynamic extension of their instruction set (the set of instructions the processor is capable to execute) by fine-grained reconfigurable logic while relying on a RISC instruction encoding [2]. In contrast to that, KAHRISMA additionally features dynamical reconfiguration of the instruction format (switch between RISC and n-issue VLIW) to execute a configurable number of statically-scheduled instructions in parallel. Thereby, multiple processor instances executing different instruction formats may co-exists in parallel. Each instruction format requires thereby a different amount of resources and also provides different peak performance characteristics. This new degree of freedom leads to the decision problem which application should run on which instruction format to efficiently utilize the available reconfigurable hardware fabrics.

One major problem of a multiplicity of reconfigurable architectures is their programmability. That often arises from the methodology to first design the system from a hardware centric view without addressing the programmability of the developed architecture. In contrast, in our research the programmability plays an important role. It is one of our major design goals to offer a well-programmable reconfigurable architecture by the widely-used, high-level, general-purpose programming language C/C++ while improving application execution and architecture efficiency. To achieve this, a compiler-based software toolchain is designed and implemented in parallel to hardware architecture development [3]. The requirements of the software framework are respected in architectural design decisions from an early stage on to maintain programmability of the hardware architecture. The software toolchain targets mixed-ISA processors and is therefore based on an *Architecture Description Language* (ADL).

978-3-9810801-8-6/DATE12/©2012 EDAA

In contrast to a processor architecture with a fixed *Instruction Set Architecture* (ISA), the flexibility – introduced by reconfigurability of the instruction set and instruction format – raises the problem of selecting an appropriate ISA e.g. on function granularity of a given application while taking reconfiguration overhead, resource consumption, energy consumption, and performance into account. For function-based ISA selection we require detailed information about the performance characteristics of the application running on each ISA. Thereby, the achievable quality of a solution is directly related to the accuracy of performance values. Therefore, we require a cycle-accurate simulator in order to measure the performance of the ISAs for a given application. However, it is very time consuming to simulate our processor pipeline cycle-accurate, since our microarchitecture utilizes the *Dynamic Operation Execution* (DOE) [4] model and thus has a relatively complex pipeline (compared to RISC or VLIW processors).

In this paper we present our cycle-approximate simulator containing a simplified dynamic operation execution model that provides a good trade-off between performance and accuracy. Additionally, the simulator features measurement of *Instruction Level Parallelism* (ILP) that could be theoretically exploited by VLIW processor instances on our architecture. The theoretical ILP could be used as an indicator for the ISA selection process without the need to simulate any combination of the different ISAs and applications. The rest of this paper is organized as follows: Section II gives an overview of state-of-the-art simulator techniques. Section III describes the simulator-relevant aspects of the KAHRISMA microarchitecture in detail. A general overview of our software framework is given in Section IV. Section V presents the components of our cycle-approximate simulator while in Section VI our cycle-approximate algorithms are explained. In Section VII our results are presented. Finally, Section VIII concludes this paper.

II. Overview of State-Of-The-Art

In general, there exist two models for processor architecture simulation: *Cycle-Accurate Simulator* (CAS) and *Instruction Set Simulator* (ISS). The ISS simulates only the behavior of an ISA while the CAS provides a cycle-accurate simulation of a microarchitecture realizing an ISA. CASs are slower than ISSs since they must additionally ensure that all operations are executed at the correct time while taking branch prediction, caches misses, fetches, pipeline stalls, and many other microarchitectural aspects into account. They are often used to model and benchmark new microprocessors without actually building a physical chip. Thereby, the SystemC [5] language is widely used for structural modeling of the microarchitecture [6].

The majority of functional ISSs rely on the following techniques: (1) interpretation [7], (2) static compilation [8], [9], and (3) dynamic compilation [10]. Interpretation fetches, decodes, and executes instructions within the simulation loop. Execution is often realized by a large switch statement or indirect function calls.

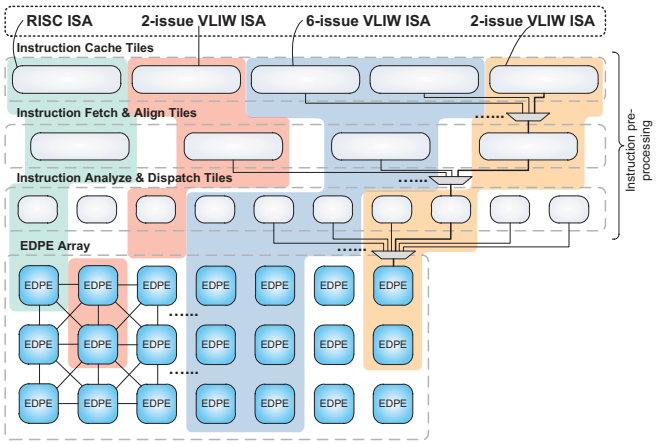

Fig. 1. Realization of Different ISA Configurations for the KAHRISMA Architecture

Decode and execute are the two major performance bottlenecks within interpretation-based simulators. The decoding costs can be significantly reduced by using instruction caching techniques [11]. In contrast, static and dynamic compilation translates the instructions of the simulated architecture into instructions of the host architecture. Static compilation performs the translation offline before simulation while dynamic compilation performs it during simulation. Static and dynamic compilation simulate on basic block level and thus offers the highest simulation speed but provide less information. In [10] dynamic compilation techniques exploiting just in-time compilation facilities of the LLVM compiler framework are presented. The authors show that cycle-accurate simulation of scalar RISC pipelines is possible with up to 800 MIPS. However, the model is not applicable to any superscalar-like processor pipeline. Furthermore, the processor cache is not modeled in detail. So the cycle-accuracy of the simulator is limited to processors without caches and constant memory access time using a simple RISC pipeline.

Besides CAS and ISS, there exist cycle-approximate simulators that try to estimate the cycle counts of the target architecture in order to provide a good trade-off between performance and accuracy. In [12] a prior training and regression based performance prediction technique based on high-level counters (e.g. total number of different types of instructions and cache reads and misses) is introduced. It is shown that the cycle counts of an ARM v5 processor could be estimated with an average error of 5.8% with a simulation speed of an ISS and an extra training phase. However, the training stage can take several hours and is thus impractical since the training stage would have to be repeated for every configuration of our architecture.

III. The KAHRISMA Processor Architecture

In the following the flexibility of the KAHRISMA architecture by reconfiguration of the ISA is presented. One major feature of the KAHRISMA architecture is the execution of multiple threads with different ISAs. In Figure 1 a global overview of the architecture is shown. On the bottom it consists of an array of *EDPEs* (Encapsulated Datapath Elements). An EDPE performs all required operations and consists of a local register file, *Arithmetic Logic Unit* (ALU), and a Synchronization Unit to synchronize neighboring EDPEs. On top of the figure the instruction preprocessing tiles are shown. They consist of three groups of separated instruction caches, fetch & align tiles, and instruction analyze & dispatch tiles.

The distributed tiles represent pipeline stages of the architecture. They are flexibly combined by reconfiguring the tiles and the communication network in between. In this way, the processor can instantiate different hardware threads as shown in Figure 1. The first thread executes a RISC ISA with a minimum of required resources. The third hardware thread executes a 6-issue VLIW ISA and can execute 6 operations in parallel by the six assigned EDPEs.

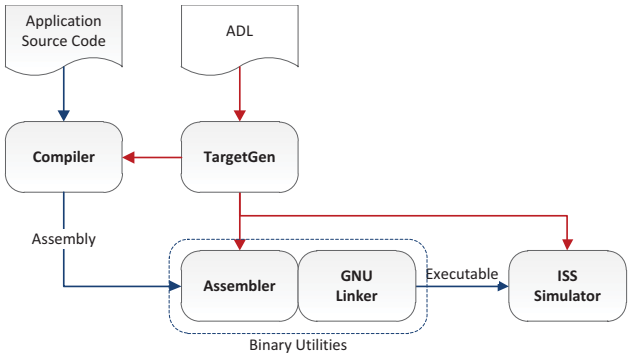

Fig. 2. Overview of the ADL-based software framework

During runtime the processor can dynamically instantiate new hardware threads as long as the required resources are available. It is also possible to change the ISA of one hardware thread during execution. That offers the possibility to adapt the resource consumption of one hardware thread to the individual requirements of the executed thread or application.

The KAHRISMA architecture uses the *Dynamic Operation Execution* (DOE) [4] model, i.e. all operations of one instruction need not be issued at the same time. Instead, the execution of one operation can start immediately if the data dependencies are fulfilled and all required resources are available. All operations of one slot are issued sequentially to the functional units. The individual slots can drift among each other so that the operations between slots (all within one instruction) can be executed out of order. While the ISA is comparable to VLIW processors, the microarchitecture is related to superscalar architectures with dynamic scheduling but without a dispatcher. A dispatcher is not necessary since a slot within the RSIW-instruction already specifies where the operation is executed.

IV. ADL-based Software Framework

Figure 2 gives an overview of the ADL-based software framework. The software framework uses an ADL description and the application's C/C++ source code as input. The ADL is processed by our *TargetGen* utility which generates source code fragments of the compiler, assembler, and simulator in order to retarget the compiler framework to any architecture described within the ADL. The ADL description contains specifications of all possible processor configurations and their ISAs in parallel [3].

Our retargetable compiler translates C/C++ source code into target-dependent assembly files. It is based on the LLVM compiler infrastructure. It allows developing mixed-ISA applications by three features. (1) It is capable to switch ISA during code generation process to target any ISA specified within the ADL description. (2) It outputs a special assembly pseudo directive to notice the assembler about the used ISA. (3) It prefixes the function name symbols by the target ISA identifier to enable mixed-ISA applications compilation containing multiple implementations of the same function, each using a different ISA.

The binary utilities – consisting of an assembler and linker – translate the assembly files into an executable file for the reconfigurable architecture. In a first step, the files are assembled into object files. Afterwards all object files are linked together into the application binary. Both, the object files and application binary, are stored in standard *Executable and Linkable Format* (ELF) format [13]. The assembler supports mixed-ISA assembly files. During assembling the ISA can be switched using a special assembly pseudo directive.

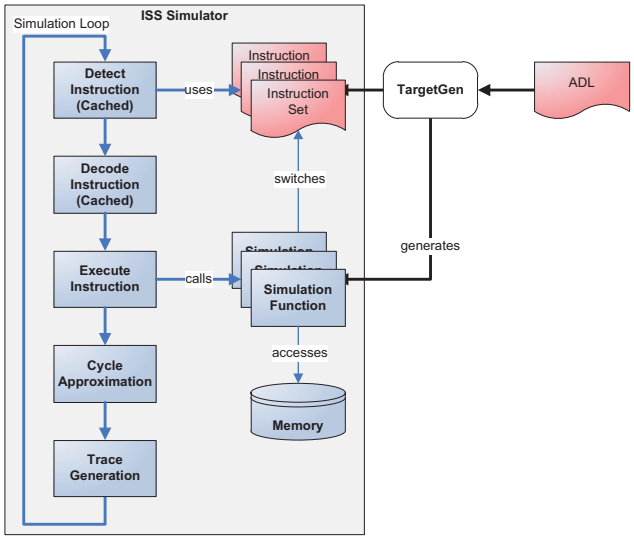

Fig. 3. Cycle-Approximate Simulator Design

The *Instruction Set Simulator* (ISS) is a cycle-approximate simulator that emulates all ISAs as specified within the ADL. As input data it uses the application's executable file generated by the binary utilities. The components of the simulator are described in detail in Section V. The ISS pursues several goals: (1) It is used to validate the compiler and binary utilities in combination with the simulated application. Only if the compiler, assembler, linker, and simulation are working correctly for a given (correct) application the simulator is able to finalize application execution and provide valid results. (2) The ISS gives cycle-approximate performance results in combination with dynamic program analysis, e.g. profiling. This is in our case especially important for the selection of appropriate ISAs for an application on function granularity. (3) The ISS is capable of trace file generation. The trace file contains the exact behavior of the processor for each cycle during simulation of an application. It is used to validate different implementations of the ISA, e.g. our *Register Transfer Level* (RTL) hardware implementation. The trace file can also serve as stimuli values for simulations of partial implementations of the ISA and is therefore very useful for early evaluation of hardware components. (4) It is used for error detections within applications. During compiler development it frequently happens that malicious code is generated. In that case the simulator provides mapping of instruction addresses to assembly and source code lines, an instruction pointer history, and trace file information.

V. Cycle-Approximate Simulator

The simulator is an essential part of the complete framework. It is an mixed-ISA ISS that emulates the behavior of multiple ISAs and optionally approximates the cycle count of the KAHRISMA microarchitecture. We used the interpretation technique to realize our simulator since the cycle-approximation calculation is performed for each simulated instruction. Static or dynamic compilation would not make sense in that case since their performance gain results from operating on basic block level. Figure 3 shows the design of our simulator. Based on the ADL description, the TargetGen utility generates fragments of the simulator's source code, especially a register table, multiple operation tables, and simulation functions. To support multiple ISAs in parallel, each supported ISA has its own operation table and only the active operation table is used during instruction detection. The operation table contains a list of all operations of an ISA. Each operation within an operation table contains its name, size, fields, implicit registers, and pointer to the simulation function. The fields represent the organization of the operation's instruction word, e.g. the encoding and location of the opcode or destination/source registers. The implicit registers are registers that are read or written by the operation without explicit encoding with a field of the instruction word, e.g. each jump operation implicitly writes the *Instruction Pointer* (IP). For each operation a simulation function exists that is called to execute the operation. To automatically generate the simulation function, the ADL description contains for each operation a simulation source code fragment in C++.

The simulator gets as input the compiled executable in ELF format. The ELF file is loaded into the simulated memory of the processor. The start address is extracted and used to initialize the IP. Within the simulation loop, first the instruction addressed by the IP is detected by checking the constant fields for each operation of the current active ISA (e.g. opcode field). The detected operation is decoded by extracting all fields of the operation. These are stored into a *decode structure* to provide fast access to the information during execution. The decoded instruction is executed by calling the simulation function generated by TargetGen. Thereby, parts of decode structure are directly passed to the simulation function.

After an instruction is executed optional tasks are performed. These optional tasks include the cycle approximation (see section VI) and trace file generation. A trace file tracks the behavior of the simulated processor. For each executed operation the cycle number, opcode, input/output register numbers and values, and immediate values are appended to the trace file. The trace file is used to validate our hardware implementation.

A. Decode Cache

The operation detection and decoding is a major performance bottleneck within interpretation-based simulators. To compensate that, all detected and decoded instructions are stored in a cache tagged by the instruction address. Thereby, each executed instruction is only detected and decoded once. Due to the principle of program locality the number of decoded instructions can be dramatically reduced compared to the number of executed instructions. In that way the processing time spent for detection and decoding becomes insignificant. To realize our *decode cache* we use the unordered_map container from the boost library [14]. It implements a hash map and has an average time complexity close to O(1) to lookup an entry.

Further, we speed up the cache entry lookup by using instruction prediction. The idea is that for non-branch instructions the following instruction is always identical. Also for branch instructions the following instruction is often the same. Therefore, we store within each decode structure the IP and decode structure pointer of the following instruction. To get the decode structure of the current instruction we first compare the current IP to the predicted IP of the previous instruction. If both match, we will use the predicted decode structure pointer. Otherwise, we perform a cache lookup and update the prediction IP and decode structure of the previous instruction. This mechanism is comparable to a 1 bit branch predictor in hardware and avoids unnecessary expensive cache lookups within our simulation loop.

B. Simulation of Parallel Operations

To execute RISC instructions consisting of one operation we can call the execution function provided by TargetGen. The execution of parallel operation within VLIW instructions is more complex. For each operation one simulation function exists. It is important that the registers of all parallel operations are loaded before any operation writes back its results. So it is not possible to simple call the simulation functions sequentially. One

solution is to generate for each possible combination of operations another simulation function. However, in that case the number of simulation functions would grow exponentially with the number of parallel operations within one VLIW instruction. Instead, in our solution only one simulation function per operation exists and it is called recursively. After the operation is performed and before the registers are written, the simulation function calls the simulation function of the next parallel operation. In that case, first all operations are calculated and their results are stored within local variable on the stack. Afterwards the results are written into the register file.

C. Debugging

For debugging purpose and statistics generation the simulator can map an instruction address to the corresponding assembler file line number, C/C++ source file line number, or function name. Therefore, the assembler stores the assembler file mapping into a custom data section within the ELF file. Additionally, the compiler can generate debugging information into the assembly file. The debug information is in standard DWARF format and a DWARF reader within the simulator extracts the source line numbers. Within the ELF file the start address and end address of each function is stored.

D. Mixed-ISA Support

To address the reconfigurability of our processor architecture, the simulator can switch the ISA during runtime. We extended the state of the processor (that contains the register file and memory) to also include the currently active ISA. Each ISA is identified by a unique number that is provided by the ADL. On startup the initial ISA must match the ISA of the entry code of the executable. Therefore, the initial ISA can optionally be specified per command line parameter. Otherwise, the default ISA – as specified in the ADL – is used. We added a new "SWITCHTARGET" instruction within the ADL. This instruction accepts one immediate as operand and changes the ISA to the given ISA identification number. The simulation code – as specified within the ADL – calls a simulator specific function that updates the active ISA state of the simulator. The next instruction is then detected and decoded using the new ISA. Therefore, for each ISA specified within the ADL a separate opcode table and simulation functions is provided. During instruction detection only the opcode table of the currently active ISA is used.

E. C Standard Library Emulation

Input/output functions are one critical point when developing mixed-ISA applications and testing the overall framework. In general, they are provided by the C library implementation. In contrast, we directly provide the functionality of required C standard library functions within our simulator. Therefore, we embedded a special simulation operation into the ADL and the simulator. The library function is encoded as immediate within the operation. Each library function is made visible to the linker by providing an automatically generated assembly file containing a small function body for each library function that only executes the simulation operation and returns afterwards. Within the simulator an emulated library function has direct access to the register file and memory. It reads the input parameters from the registers and stack according to the calling convention, executes the corresponding C library function natively, and writes the result back to the registers.

The native execution of C library functions allows the fast retargetability of our framework since we must not recompile a complete C library in order to support a new ISA or to reflect any changes to a given ISA. Instead, we can execute the simulation operation from any ISA that is currently executed within the simulation. However, the native execution has the disadvantage that the cycles required for these functions are not counted. Therefore, we support to replace any native C library function with real implementations on the simulated ISA.

VI. Cycle-Approximation Models

Besides functional application execution, the simulator supports several cycle models to approximate the application execution time on the microarchitecture. In contrast to a cycle-accurate simulator, we do not model the exact KAHRISMA microarchitecture within our simulator. Instead, we approximate the cycles based on a heuristic model in order to provide a trade-off between accuracy and simulation speed. At the moment we provide three cycle models that are explained in the following sections: *Instruction-Level Parallelism* (ILP), *Atomic Instruction Execution* (AIE), and *Dynamic Operation Execution* (DOE).

A. Instruction-Level Parallelism

The ILP cycle model performs a fast theoretical ILP measurement that calculates the theoretical upper limit for operations per cycle that could be achieved by our architecture with unlimited resources. It predicts the performance of a KAHRISMA VLIW processor instance with unlimited number of operations in parallel, unlimited number of renaming registers, and an ideal memory architecture with three cycles delay (the delay of our L1 cache) and unlimited number of parallel memory accesses. In such a theoretical architecture the parallelism is limited by true data dependencies. As input we simulate a RISC ISA. The RISC instructions are executed by the simulator in the sequence given by the compiler. However, such a theoretical architecture would allow to execute all parallelizable operations with an instructions as early as possible. To mimic that we calculate an individual start and completion cycle for each RISC instruction.

For each register we store the cycle of its last write access that is given by the completion cycle of the last instruction that wrote the register. The instruction's start cycle is dependent on its source registers. The start cycle becomes the maximum write cycle of all source registers. The instruction's completion cycle is then given by its start cycle plus its delay. On VLIW processors only the operations until the next branch instruction can be scheduled in parallel. So the start cycle must be additionally higher than the completion cycle of the last branch instruction. For memory instructions we use a pessimistic model that assumes memory operations are dependent on each other. A load/store instruction is always dependent on the last store instruction and can therefore be executed earliest on the start cycle of store instruction. This reflects the optimization potential of our compiler since we do not have an alias analysis and use at the moment the same pessimistic model for scheduling.

B. Atomic Instruction Execution

Within the atomic instruction execution model we assume that all operations of an instruction are issued in the same clock cycle(s). The following instruction can only be issued if all operations of the previous instruction finished execution. Within our simulator we calculate the delay of one instruction from the maximum delay of its operations.

C. Dynamic Operation Execution

The DOE cycle model approximates the performances of our KAHRISMA architecture (see Section III). In contrast to the AIE model, each operation of one instruction need not be issued at the same clock cycle. Instead, the slots of the VLIW instructions may drift among each other. An operation within a slot is issued if the previous operation within the same slot has been issued and the true data dependencies of the input registers are fulfilled.

Within our simulator we model the true data dependencies of DOE identically to the ILP model. For each register the write cycle is stored. For one operation the earliest start clock cycle can be calculated using the maximum of the write cycles of all source registers. Additionally, we store for each slot the start cycle of the last operation. Within one slot all operations must be issued in their order. Thus, the start cycle of one operation must be at least the start cycle of the last operation within the slot plus one. The one is added to ensure that only one operation is issued per slot and clock cycle.

With this simple model we can approximate the performance of our KAHRISMA architecture without simulating the pipeline of the microarchitecture in detail. The model is heuristic for three reasons: (1) The resource constraints are not considered, e.g. a multiplication may be shared between two slots within our architecture. (2) The drift between the issue slots is limited to a maximum value within our hardware to enable precise interrupts but it is not limited within the simulator. (3) The memory operations are executed within the simulator in the order given by the program and not in the order issued within the hardware.

D. Memory Approximation

Within our simulator we approximate the delay of each memory access. Thereby, the delay approximation is performed in order of the instruction stream executed by our behavioral model and not in the order as executed in hardware. To approximate the delay, we modeled a memory hierarchy consisting of three types of modules: caches, connection limits, and main memory. Each module has the same interface containing a function to calculate the completion cycle of a memory access. Within the cache and connection limit modules a pointer to submodule is stored that follows in the memory hierarchy. During delay calculation the function of the submodule is called, e.g. to pass the memory access to the next hierarchy in case of a memory miss. During DOE and AIE cycle mode calculation, the memory delay calculation starts by calling the function of the first module. The memory address, access type (read or write), slot, and start cycle are given as input parameters to the delay function. The function returns the completion cycle of the memory access.

The simplest module is the memory module whereas the memory access delay is configurable. It calculates the completion cycle by adding the fixed delay to the start cycle. The cache module emulates a n-way set associative cache with write-back write policy and least recently used replacement policy. The line size, associativity, cache size, and access delay are configurable. Within the delay function the current cycle is initialized by the start cycle plus the access delay. If the cache contains the memory address the function immediately returns. Otherwise, the access to the memory address is passed to the next submodule (e.g. a second cache). The current cycle is used as start cycle for the subaccess and the calculated completion cycle becomes the new current cycle. The same procedure is performed a second time if a write-back is required. After the subaccess the data must be stored inside the cache, so the cache delay is added again to the current cycle. At the end the current cycle is returned. Additionally, since the delay function can be called out of order, we store within each cache line the cycle the cache line was written. The completion cycle in case of a cache hit is the maximum of the current cycle and the write cycle of the cache line.

The cache module can calculate the delay of a cache access but the resource constraints of the cache are not modeled. A cache can perform only a limited number of accesses at the same time. This resource limit is modeled using the connection limit module. It can be configured by the maximum number of access ports and is typically placed before a cache or memory module. The connection limit module checks and stores for each start cycle if a port is available within the start cycle. Otherwise, the start cycle is increased until a free cycle has been found. Afterwards, the submodule is called using the new start cycle. The same mechanism is applied to the completion cycle that is returned from the submodule.

VII. Results

We used our framework to compile and simulate several applications using the cycle models introduced in Section VI. We will first give an overview of the simulator performance and provide afterwards results obtained from the simulation. We run the simulator on an Intel Xeon X5680 CPU at 3.33 GHz pinned to one core. We compiled the simulator with gcc and -O3 optimization.

For memory approximation we used the three modules introduced in Section VI-D to model a three layered memory hierarchy consisting of a L1 cache (2 KiB, 4-way, 3 cycles delay), L2 cache (256 KiB, 4-way, 6 cycles delay), and main memory (18 cycles delay). In front of the L1 cache we placed a limit connection module with one port to limit the L1 memory access to one access per cycle. For result generation we used a set of applications comprising the JPEG encoder/decoder (used from the MiBench), a fixed-point *Fast Fourier Transform* (FFT) implementation, a Quicksort sorting algorithm, a fully-unrolled *Advanced Encryption Standard* (AES) implementation, and a 4x4 integer *Discrete Cosine Transform* (DCT) approximation as used in H.264. All applications were compiled with maximum performance optimization.

A. Simulator Performance

We used the cjpeg application compiled for a KAHRISMA RISC processor instance to measure the performance of our simulator. First of all, we analyzed the impact of the detection and decoding cache as well as the instruction prediction as described in Section V-A. Without cache we achieved a poor performance of 0.177 *Million Instructions per Second* (MIPS). By activating the cache we could avoid 99.991% of detected and decoded instructions and improve our performance to 16.7 MIPS. By activating the instruction prediction we could avoid 99.2% of the hash table lookups and speedup our simulation to 29.5 MIPS. If we ignore the overhead for the instruction prediction (which is one or two memory accesses and a comparison), we are able to calculate the execution time per instruction (by solving a system of linear equations) of the simulator components Execute, Cache Access, and Detect & Decode as seen in Table I.

Simulator Components	Average Execution Time per Instruction
Execute (1 Operation)	$33.2ns$
Cache Access	$26.0ns$
Detect & Decode	$5602.0ns$
ILP	$21.5ns$
AIP (including memory)	$19.7ns$
DOE (including memory)	$32.3ns$
Memory Model	$9.5ns$

TABLE I
Simulator Performance

If we activate our cycle approximation the performance of the simulator decreases. For ILP measurement we still reach 18.3 MIPS, for AIE 18.9 MIPS, and for DOE 15.3 MIPS. We calculated the execution time of the three cycle approximation models (see Table I). We further measured the impact of the memory model that is active during AIE and DOE approximation. Surprisingly, the memory model requires only $9.5ns$ and is thus comparable fast although 24.6% of all instructions access the memory.

B. Instruction-Level Parallelism

To evaluate the ILP cycle model described in Section VI-A, we compared the ILP to real results obtained from different VLIW processor instances with different issue widths: RISC (1-issue VLIW), 2, 4, 6, and 8-issue VLIW. The results of this comparison are shown in Figure 4. From our five applications the DCT and AES offer a high ILP while FFT, jpeg compression/decompression and Quicksort offer only a small ILP. For FFT the low ILP is remarkable since the Fourier transform is in general a very dataflow-dominant algorithm and therefore one would expect a higher ILP. However, the FFT implementation in our case uses a recursive approach which limits the parallelism by executing many functions consisting of small basic blocks. Figure VI-A shows that the ILP cycle model provides a good estimation of the available KAHRISMA-exploitable parallelism within an application. The AES benchmark is here an exception since the 8-issue VLIW processor instance can only utilize a fraction of the available ILP compared to DCT which only provides a little more ILP. This results from the larger working set of our AES algorithm that does not fit into the L1 cache and therefore causes 14% L1 cache misses. The L1 cache misses are not modeled within our ILP measurement.

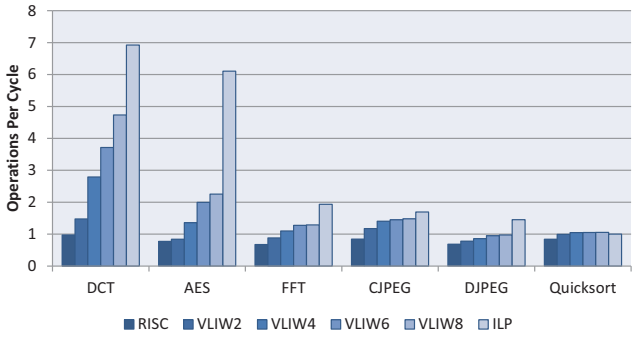

Fig. 4. ILP Measurement Compared to Several Configurations

C. Dynamic Operation Execution

The DOE cycle model approximates the cycle count required by the real hardware as accurate as possible. Therefore, we compared the results from the RTL hardware simulation against the results obtained from our cycle-approximate simulator. In order to filter errors resulting from branch misprediction, we rely on a perfect branch prediction for both simulators. We used the DCT application for comparison since it provides the highest parallelism. Table II shows the cycles required to execute the DCT compiled for different issue widths from the RTL simulation and the cycle-approximation simulation. The DOE model including the memory model is capable to approximate the required cycle with an error of up to 2.8%. The RTL simulator requires 8 ms per instruction, so our simulator is roughly $100,000$ times faster while providing nearly the same cycle counts.

Configuration	Hardware	Approximation	Error
RISC	21768	22062	1.4%
VLIW2	14111	13922	1.4%
VLIW4	9774	9878	1.1%
VLIW8	7774	7992	2.8%

TABLE II
SIMULATOR ACCURACY OF DYNAMIC OPERATION EXECUTION

VIII. CONCLUSION AND FUTURE WORK

Within this paper we presented a cycle-approximate, mixed-ISA, interpretation-based simulator for reconfigurable processor instances of our KAHRISMA architecture. The simulator estimates the cycle counts of various VLIW processor instances (each with a different ISA) utilizing the *Dynamic Operation Execution* (DOE) model. DOE is characterized by dynamic issuing of operations within VLIW-instructions once true data dependencies of the operation are fulfilled. This would require a complex pipeline design within a corresponding cycle-accurate simulator. Instead, we provided a heuristic, approximate cycle model for DOE in combination with a memory delay approximation. The memory model supports handling of out of order memory accesses while being called in order. We were able to show that the model provides a good trade-off between performance and accuracy by integrating it into our interpretation-based instruction set simulator. We achieved a maximum accuracy loss of 2.8% at a simulation speed of 15 MIPS.

Additionally, the simulator features measurement of upper bound of *Instruction-Level Parallelism* (ILP) exploitable by VLIW processor instances using infinite number of resources of our architecture. The theoretical ILP could be used as an indicator for the ISA selection process without the need to simulate any combination of the different ISAs and applications. Both cycle models, the DOE and ILP, could be used to select an appropriate ISA for each function of a given application. In future we will use the cycle-approximate simulator as basis to address the problem of selecting an appropriate ISA e.g. on function granularity of a given application while taking reconfiguration overhead, resource consumption, energy consumption, and performance into account. Therefore, we plan to integrate cycle-approximation models for branch misprediction into our simulator.

ACKNOWLEDGMENT

We thank the German Research Foundation (DFG) for funding this work within the *KAHRISMA* project.

REFERENCES

[1] R. Koenig, L. Bauer, T. Stripf, M. Shafique, W. Ahmed, J. Becker, and J. Henkel, "Kahrisma: A novel hypermorphic reconfigurable-instruction-set multi-grained-array architecture," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, 8-12 2010, pp. 819 –824.

[2] L. Bauer, M. Shafique, S. Kramer, and J. Henkel, "RISPP: Rotating Instruction Set Processing Platform," in *DAC'07: Proceedings of the 44th annual Conference on Design Automation*, 2007, pp. 791–796.

[3] T. Stripf, R. Koenig, and J. Becker, "A Novel ADL-based Compiler-Centric Software Framework for Reconfigurable Mixed-ISA Processors," in *Embedded Computer Systems (SAMOS), 2011 International Conference on*, july 2011, pp. 157 –164.

[4] R. Koenig, T. Stripf, J. Heisswolf, and J. Becker, "A scalable microarchitecture design that enables dynamic code execution for variable-issue clustered processors," in *Proc. of the 25th IEEE International Parallel & Distr. Proc. Symp., Workshops and Phd Forum*, 2011.

[5] "Open systemc initiative," http://www.systemc.org/, 2006.

[6] Yen-Ju Lu et al., "Microprocessor modeling and simulation with systemc," in *VLSI Design, Automation and Test, 2007. VLSI-DAT 2007. International Symposium on*, april 2007, pp. 1 –4.

[7] J. Lee, J. Kim, C. Jang, S. Kim, B. Egger, K. Kim, and S. Han, "Facsim: a fast and cycle-accurate architecture simulator for embedded systems," *SIGPLAN Not.*, vol. 43, pp. 89–100, June 2008.

[8] C. Mills et al., "Compiled instruction set simulation," *Software: Practice and Experience*, vol. 21, no. 8, pp. 877–889, 1991.

[9] V. Zivojnovic et al., "Supersim - a new technique for simulation of programmable dsp architectures," 1995.

[10] F. Brandner et al., "Fast and accurate simulation using the llvm compiler framework," in *1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO)*, Paphos, January 2009.

[11] M. Lv, Q. Deng, N. Guan, Y. Xie, and G. Yu, "Armiss: An instruction set simulator for the arm architecture," *Embedded Software and Systems, Second International Conference on*, vol. 0, pp. 548–555, 2008.

[12] B. Franke, "Fast cycle-approximate instruction set simulation," in *Proc. of the 11th intl. workshop on SW & compilers for embedded systems*, ser. SCOPES '08. New York, NY, USA: ACM, 2008, pp. 69–78.

[13] TIS Committee, "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2," http://refspecs.freestandards.org/elf/elf.pdf.

[14] "Boost c++ libraries," http://www.boost.org.