Run-Time Reconfiguration of Multiprocessors Based on Compile-Time Analysis

Madhura Purnaprajna, Mario Porrmann, Ulrich Rueckert
Systems and Circuit Technology, Heinz Nixdorf Institute, Germany
{madhurap,mario,rueckert}@hni.upb.de
and
Michael Hussmann, Michael Thies, Uwe Kastens
Institute of Computer Science, University of Paderborn, Germany
{michaelh,mthies,uwe}@upb.de

In multiprocessors, performance improvement is typically achieved by exploring parallelism with fixed granularities, such as instruction-level, task-level, or data-level parallelism. We introduce a new reconfiguration mechanism that facilitates variations in these granularities in order to optimize resource utilization in addition to performance improvements. Our reconfigurable multiprocessor QuadroCore combines the advantages of reconfigurability and parallel processing. In this paper, a unified hardware-software approach for the design of our QuadroCore is presented. This design-flow is enabled via compiler-driven reconfiguration, which matches application-specific characteristics to a fixed set of architectural variations. A special reconfiguration mechanism has been developed that alters the architecture within a single clock cycle.

The QuadroCore has been implemented on Xilinx XC2V6000 for functional validation and on UMC's 90nm standard cell technology for performance estimation. A diverse set of applications have been mapped onto the reconfigurable multiprocessor to meet orthogonal performance characteristics in terms of time and power. Speedup measurements show a 2-11 times performance increase in comparison to a single processor. Additionally, the reconfiguration scheme has been applied to save power in data-parallel applications. Gate-level simulations have been performed to measure the power-performance trade-offs for two computationally complex applications. The power reports confirm that introducing this scheme of reconfiguration results in power savings in the range of 15-24%.

Categories and Subject Descriptors: [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*C.1.2*

General Terms: Reconfigurable Multiprocessors, Compilation for Multiprocessors

1. INTRODUCTION

Partitioning large applications onto multiple processing elements typically introduces performance speedups. However, the granularity of application partitioning is entirely application dependent. For instance, an application with a fine-grained parallelism allows operating instructions in parallel. Superscalar and VLIW processors are classical examples of architectures for instruction-level parallelism. At a coarse-grained level, larger structures of a program like loops, functions, threads, or tasks are executed in parallel. Vector and SIMD processors explore data-level parallelism (DLP). In addition, partitioning a given application also involves an overhead in terms of inter-processor communication and synchronization. Hence, the influence of the granularity of partitioning on the performance of the system identifies the parallelization paradigm to be adapted. Conventionally, executing applications on multiprocessors is restricted to a single granularity, which is predetermined by the programming style and by the inherent parallelism in the underlying hardware. Task-level or data-level parallelism is usually exploited manually by partitioning an application by hand or with special programming models. Automatic compilation is often limited to addressing parallelism at the instruction or loop-level. Here, we explore the advantage of enhancing a multiprocessor with reconfigurability that enables variations in terms of the degrees of parallelism, frequency of synchronization, and amount of communication. This gives rise to a single, unified architecture that offers task, data, and instruction-level parallelism.

In our approach, called CoBRA[1], shown in Figure 1, compile-time analysis determines the schedule for reconfiguration during run-time. Reconfiguration alters the architecture of the QuadroCore within a fixed set of operating modes, called *reconfigurable architectural variants* during run-time. A prominent example is to reconfigure between parallelization paradigms like MIMD and SIMD. Given a program that exhibits both regular and non-regular structures, the compiler determines the best execution mode by analyzing the parallelism during compilation. This has a number of benefits like improved performance, efficient resource utilization, reduced code-size, and power consumption. Further, the usage of a manageable set of variants leads to an enormous reduction in the design space, compared to fine-grained reconfigurable architectures. The compiler then addresses this finite design space efficiently by using well-known program analysis techniques [Muchnik 1997]. Primarily, our compiler enables transparent usage of the reconfiguration variants without any manual effort in order to reduce time-to-market. In the approach presented in this paper, reconfiguration is performed with a very low overhead during run-time by switching fixed, coarse-grained components like instruction decoders, ALUs and register banks. The preliminary concepts of the compiler-driven reconfiguration in QuadroCore have been presented in [Hussmann et al. 2007]. This approach is in contrast to research using fine-grained reconfigurable architectures, where reconfiguration typically incurs a significant overhead [Compton and Hauck 2002]. Reconfigurability in the QuadroCore is introduced within the existing architecture, such that the processor's base instruction set remains unaltered. QuadroCore is also different from processors such as Stretch [Gonzalez 2006], where additional instructions are mapped onto reconfigurable logic (called instruction set extension fabric) to enable performance acceleration. In Stretch, the added instructions are application-specific, whereas in QuadroCore the added instructions ensure co-operative processing between the processors and are independent of the application mapped.

The organisation of this paper is as follows: Section 2 motivates opportunities to reconfigure our multiprocessor architecture and compares it with existing approaches. Also, in this section the reasoning for switching between each of the reconfigurable modes is presented. The structure and concepts of both the hardware architecture and compiler backend are outlined in Section 3. Section 4 analyses scalability and provides and experimental setup for accelerated performance and functional validation of the QuadroCore architecture. Application mapping and

[1]**Co**mpiler-**D**riven **D**ynamic **R**econfiguration of **A**rchitectural Variants (merge two *D*s to a *B*)

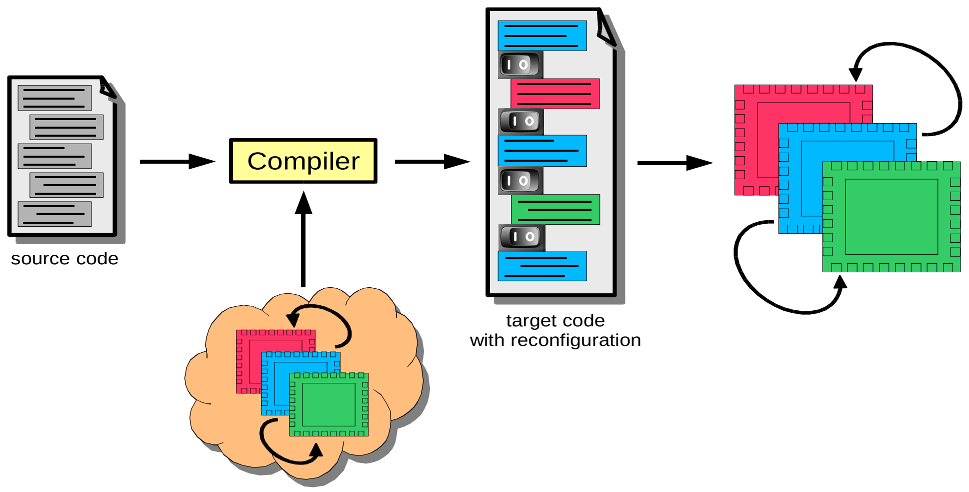

Fig. 1. Philosophy of Compiler-driven Reconfiguration of Architectural Variants

evaluations for instruction-level parallelism and application-specific power savings via reconfiguration are detailed in this section. Finally, conclusions and future work are discussed in Section 5.

2. RECONFIGURABILITY IN THE QUADROCORE ARCHITECTURE

Figure 2 shows a high-level representation of our QuadroCore architecture. The multiprocessor architecture comprises four RISC-based 32-bit embedded processors, called N-Core [Gruenewald et al. 2004]. The N-Core processors have a three-stage pipeline and a fixed 16-bit opcode length. Each processor has a 16×32 local register file and operates independently with individual program memories. Most arithmetic and logical operations provide a single-cycle execution time and load/store instructions have a three-cycle execution time. Sharing of data among the processors in the cluster is enabled via an external memory, accessible over a shared bus, in our case a Wishbone bus [Silicore 2002]. Access to this external memory is managed via a round-robin arbitration mechanism. This base architecture represents a typical MIMD mode of operation.

Variations in synchronization, method of communication, and type of parallelism within the architecture are introduced via the added reconfigurable operating modes. A mode switch is achieved via reconfiguration, in between executing different applications or even within stages of the same application. During the execution of a single large application, this reconfiguration scheme offers a very low overhead, since the time to switch between modes is optimized to a single clock cycle. This scheme is advantageous, since the best suited mode is chosen for every application or within an application, with reconfiguration time making a very minimal impact on the total execution time (detailed in Section 4). The individual processors allow run-time modifications to the architecture and support self-reconfiguration, as determined during compilation. An explicit requirement for a reconfiguration controller, which is typical in FPGA-based designs, is eliminated

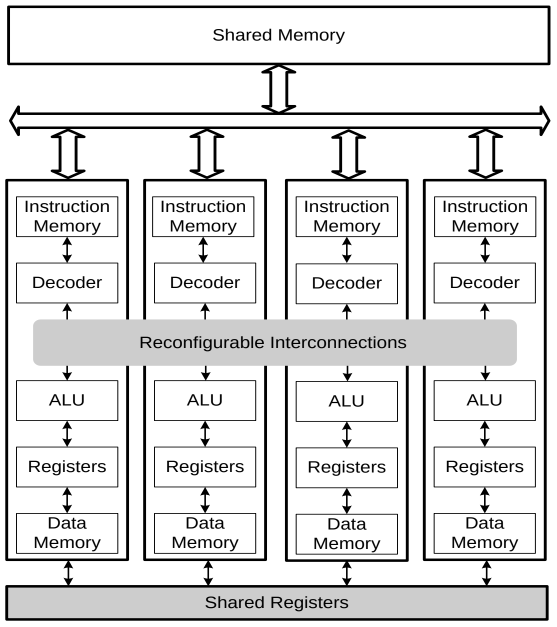

Fig. 2. Overview of QuadroCore Architecture

in this methodology, since the configuration information is included as additional instructions, as illustrated below. Additionally, the need for separate configuration memory space is avoided, since the configuration data is embedded into the instruction stream. Another architecture that configures to varying granularities and parallelism as per work loads, called TRIPS, is presented in [Sankaralingam et al. 2006]. Unlike TRIPS, the QuadroCore is based on introducing reconfigurability to our existing embedded processors (called N-core) without altering the base RISC instruction set. Our focus is to introduce methods of reconfiguration that can be extended to most multiprocessor architectures with minimal design changes. The adaptability based on the type of parallelism, viz., thread and instruction-level parallelism is in the same lines as Voltron [Zhong et al. 2007]. However, in QuadroCore the performance analysis using the standard cell implementation addresses orthogonal application-specific objectives such as time and power.

Reconfiguration Mechanism Figure 3 illustrates the principle of the reconfiguration mechanism in QuadroCore. The layer of multiplexers between the decode and execute stages of the processors allow reconfiguring the control path, when encountered with the reconfiguration instruction. These additional multiplexers and the associated control logic influence the timing characteristics and the chip area. The figure represents a high-level functional diagram of the QuadroCore control-path. Additionally, instruction set extensions to enhance collective branching and sharing branch conditions were added to ease co-operative processing. Overall, the goal was to minimize the impact of these architectural enhancements on the total area and clock frequency. The figure also illustrates that the instruction pipeline remains unaltered and reconfiguration in this context is enabled via a special instruction that configures the stage of multiplexers. Once configured, the processors operate using the base instructions as previously, but with an altered control path.

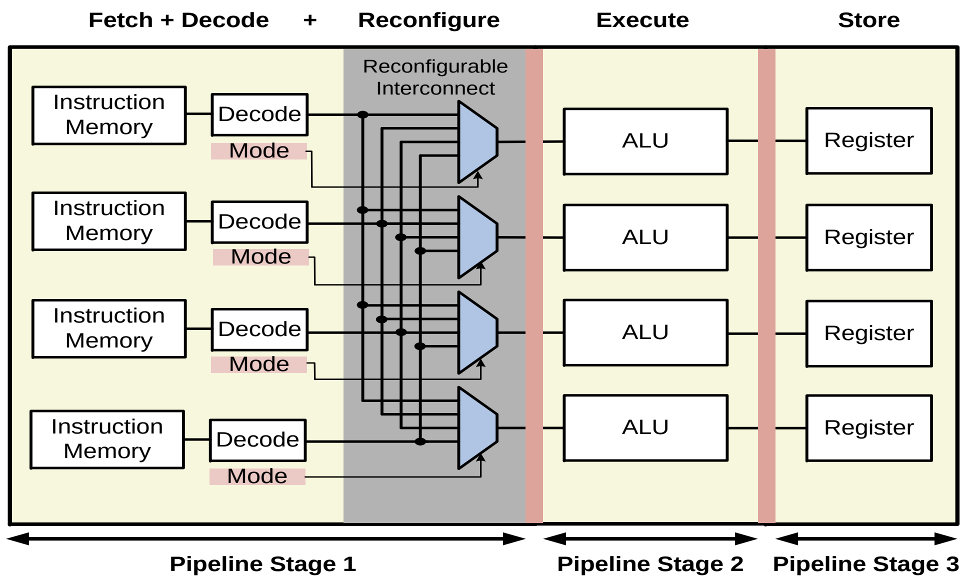

Fig. 3. Principle of Control Flow Reconfiguration in QuadroCore

Reconfiguration Instruction Format In FPGA-based designs, a configuration file includes both control and data information required to configure the individual configurable logic blocks, memories and to define their interconnections. In contrast, the reconfiguration in QuadroCore is triggered by a single instruction, where the choice of the mode is decoded from the instruction. The format is as shown in Figure 4, with SIMD mode as an example.

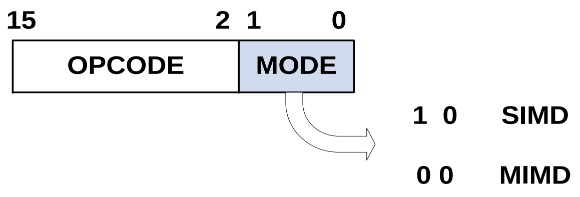

Fig. 4. Instruction Format for Reconfiguration

This reconfiguration instruction introduces variations in the architecture. The choice of the correct architectural variant is made using program analysis techniques by the compiler. The variations in application characteristics and the matching operating mode are listed in Table I. These alterations are introduced by executing the special reconfiguration instruction between application boundaries or within an application with diverse characteristics in terms of parallelism, amount of communication, and synchronisation.

Table I. Reconfigurable Architectural Variants

Operating Mode	**Application Characteristics**
Asynchronous MIMD	Coarse grained, task-level parallelism
Synchronous MIMD	Fine grained, instruction-level parallelism
SIMD	Data-level parallelism
Fast Memory Access	Large data exchange, data-level parallelism
Comm. via Shared Reg File	Few, frequent register exchange, fine-grained

The following subsections describe the reconfiguration modes to enables alterations within methods of synchronization, communication, and the reconfiguration between SIMD and MIMD.

2.1 Synchronization

Synchronization mechanisms employed in multiprocessors can be categorized as synchronous (lock-step) or completely asynchronous operations. Often, the synchronization overhead is a constraint for the granularity of parallelism explored. Fine-grained parallelism necessitates frequent synchronization on account of dependencies between instructions. In the presence of coarse-grained parallelism, processors can operate more independently and synchronize only when necessary. Hence in QuadroCore, synchronization is chosen as a run-time option that allows switching between the asynchronous and synchronous mode of operation with an overhead of a single clock-cycle. In the asynchronous mode of operation, barrier instructions allow synchronization between independently operating instruction streams. In the synchronous mode, the instruction streams operate in lock-step, synchronous at every instruction. The presence of two synchronization modes enables exploring the advantages of both lock-step architectures and asynchronous operation interchangeably. Hence, a variable granularity can be achieved depending on the mode of operation, varying between a fine-grained architecture exploring instruction-level parallelism and a coarse-grained architecture with task-level parallelism. The synchronous mode is selected in case of many inter-processor dependencies. Otherwise, infrequent synchronization using barriers is chosen.

In comparison to our approach, synchronization is always achieved using barriers in [Dietz et al. 1989]. Hence, an application with fine-grained parallelism incurs a large synchronization overhead. Similarly, a recent commercial multiprocessor architecture called AMBRIC [Halfhill 2006] consists of RISC processors (called *'brics'*) arranged in a cluster, where each individual core runs at its own clock speed. Processors are synchronized only when data needs to be exchanged. In both these cases [Dietz et al. 1989; Halfhill 2006], the architecture is more suited for applications with coarse-grained parallelism due to the overhead of explicit synchronization for each data transfer.

2.2 Communication

Inter-processor data dependencies demand a mechanism for exchanging register values. This can be implemented using message-passing between distributed memories or by using a single shared-memory. Message passing is more suited for large amounts of data exchange but infrequent communication. To enable fast and frequent exchange of register values between the processors in QuadroCore, a shared register file was included to ease communication. Exchange of infrequent, but large amounts data can be enabled via the external shared memory.

Considering related architectures, AMBRIC [Halfhill 2006] offers a point-to-point communication via channels, which automatically manage the synchronization between the processors. However, data transfer is limited to data-exchange between neighbouring processors. In [Gupta 1990], Gupta presents the integration of shared register channels into a RISC based multiprocessor, which provides a broadcast communication. Hence, communication and synchronization are combined in a single method. On the other hand, it is restricted to an asynchronous execution, because communication with channels always implies explicit synchronization. Silicon Hive's processor [Mei et al. 2005] consists of multiple cells with distributed register files. The interconnection network allows data transfer between the functional units and register files, always operating synchronously. In our architecture, a processor can communicate with a subset of the processors using a broadcast mechanism. Furthermore, two or more processors can be synchronized explicitly if necessary, or operate in lock-step, in case of frequent data exchange.

2.3 SIMD / MIMD

SIMD execution is well suited for regular program structures with data-level parallelism, which can be found in scientific or multimedia computations. Programs with non-regular structures can be executed in a MIMD manner to exploit the inherent instruction or task-level parallelism. In the SIMD mode introduced in QuadroCore, a single instruction stream is fetched and decoded by the first processor, but executed by all processors with data residing in their respective register banks and local memories. The absence of instruction fetch and decode operations by the participating processor results in reduced power consumption. Switching between MIMD and SIMD execution becomes useful if programs executed on the multiprocessor contain both regular and non-regular structures. Instead of selecting one execution mode statically, the compiler can identify the parts of a program suited for MIMD or SIMD execution and switch between the modes. The CHARISMA[2] module of our CoBRA compiler applies well-known scheduling and vectorization techniques [Kennedy and Allen 2002] at first and finally selects the best combination of modes. The selection heuristic can be based on parameters like execution time, code size, or estimated energy consumption.

In the same context, the authors in [Barretta et al. 2002] present a multi-clustered VLIW architecture, which can be switched between ILP (Instruction-level parallelism) and SIMD modes. The associated compiler is expected to identify pieces of code that can be executed in SIMD mode by determining accesses to disjoint

[2]**C**ompiler **H**andles **A**rchitectural **R**econfiguration **I**ntegrating **S**IMD **M**IMD **A**utomatically

memory blocks (provided in the source code or computed automatically). The CoBRA compiler uses the Superword Level Parallelism (SLP) approach [Larsen and Amarasinghe 2000], which targets sequential code in basic blocks instead of performing complex transformations on loop nests. In contrast to classical vectorization techniques, SLP can also be exploited when vector parallelism is scarce or loop transformations cannot be applied. The authors have shown that focusing on SLP leads to simple and robust compiler implementations while still achieving a good performance. Vector parallelism can be transformed to SLP by loop unrolling. The CoBRA compiler unrolls loops depending on the number of targeted processors.

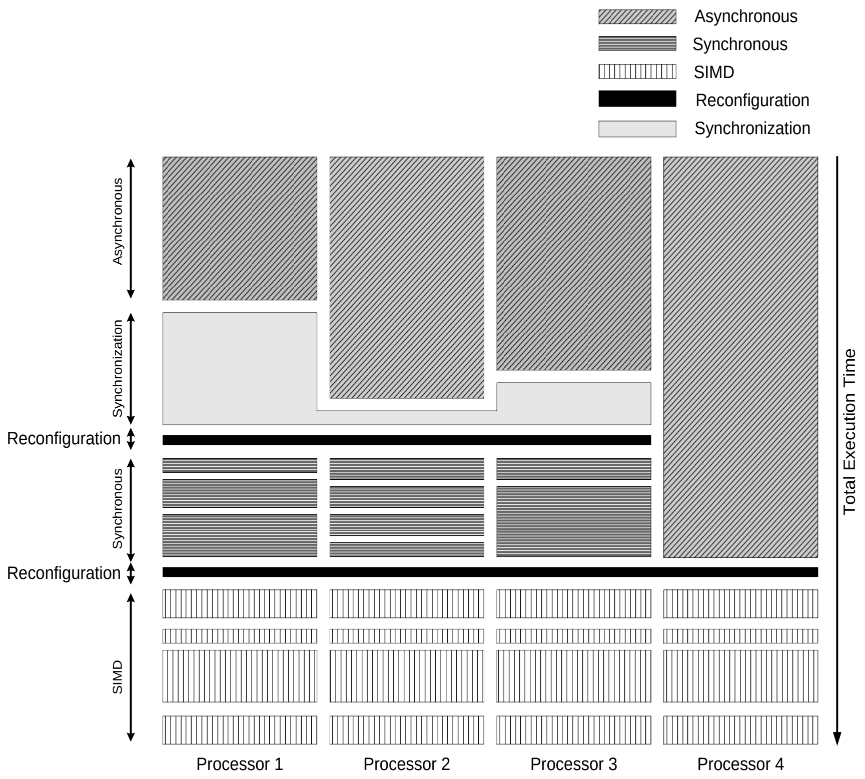

Fig. 5. Illustration of Reconfigurable Operating Modes in QuadroCore

To summarize, Figure 5 illustrates an example of mode changes during execution on four processors achieved via instruction streams in the QuadroCore architecture. Initially, all the processors operate in the default asynchronous mode. Due to application demands, processors 1-3 are required to be switched to the synchronous mode, which is done by synchronisation using barriers, followed by reconfiguration to switch to synchronous mode. During this time, processor 4 continues to operate in the default asynchronous mode. Next, all the four processors are reconfigured to operated in the SIMD mode. In all these cases, it has to be noted that the instruction stream itself includes the configuration information to allow switching between the fixed set of reconfigurable modes. The mode changes are determined during compile-time and the architectural changes (reconfiguration) are inferred during run-time.

3. CONCEPTS OF HARDWARE AND COMPILER

Figure 2 shows the QuadroCore architecture, where the extensions to the original base architecture are highlighted in grey. Additionally, introducing the reconfigurable modes marginally alters the control path of the processor; viz. address generation and the forwarding of control signal after the instruction fetch and decode stages. Each of the processors is composed of coarse-grained building blocks such as decoder, 16×32 registers file, 32-bit ALU and 32K local instruction and data memory. The memory hierarchy provides a single cycle access to the local register file, a two clock cycle access to the shared register file, a three clock cycle access to local memory, and requires a minimum of six cycle overhead to the shared external memory. Hence, a single external memory access requires six clock cycles and extends up to a worst case value of fifteen clock cycles, when all the processors make a simultaneous access, on account of the arbitration mechanism.

The instruction set architecture of the N-Core processor provides about 11% free opcode space to allow architectural enhancements. This free opcode space has been utilized to add instruction set extensions that permit run-time modifications to the architecture and support for co-operative operation of multiple instances of the same processor. These instructions include operations such as sharing of branch condition, collective branching, and reconfiguration to enable switching between the presented operating modes. The hardware modifications on account of introduced instruction set extensions have been optimized to minimize the effect of altering the processor's critical path. With area as a trade-off for the processor synthesis, the architecture is preserved such that the maximum operating frequency is only marginally affected (detailed analysis refer Section 4.2).

In order to alter the intra-cluster communication between processors, a layer of interconnect was introduced between decode and execute stages of all the processors in the cluster. The interconnect network allows alterations to the control flow via instruction streams. The layer of interconnect is steered by a special reconfiguration instruction, which defines the operating mode of the processor. This instruction controls the operation of the reconfigurable interconnects. As determined during compilation, the reconfigurable interconnect is configured as defined by the reconfiguration instruction in a single clock cycle. This additional reconfigurable interconnect marginally influences the timing characteristics of the cluster, but since it is an independent operation, it does not interfere with the existing instruction set architecture.

3.1 Structure of the Compiler Backend

Figure 6 illustrates the structure of the compiler backend, which has been derived from an existing backend for superscalar processors [Hussmann et al. 2005]. In contrast to the original backend, it features three additional phases (highlighted in grey), that are explained here. In order to support retargetability the QuadroCore was described by an abstract processor model, which was specified using our high-level processor specification language UPSLA[3] [Bonorden et al. 2003]. The UPSLA compiler is used to generate machine-specific parts of the compiler backend as well as the cycle-accurate software simulator.

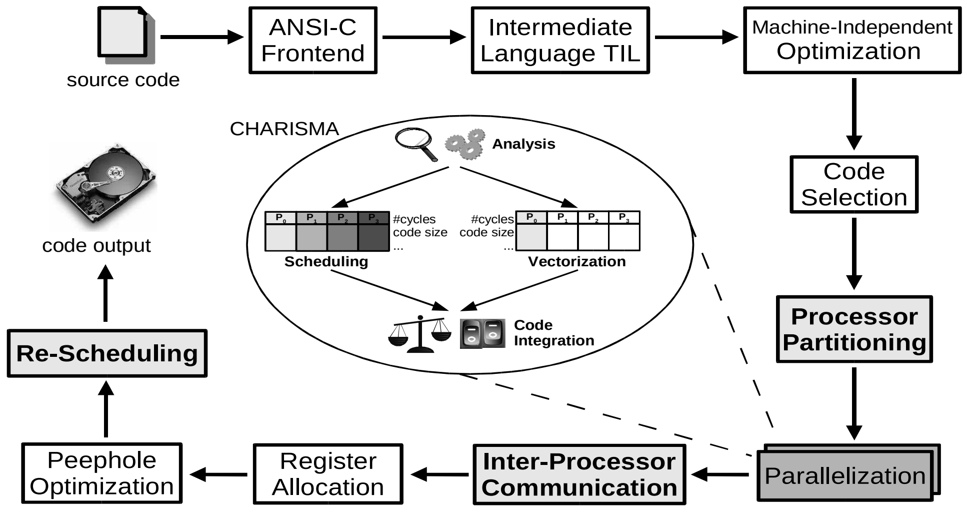

Fig. 6. Backend Structure: Compiler Driven Dynamic Reconfiguration of Architectural Variants

Processor partitioning decomposes into partitioning of data objects and allocation of functional units, which is needed as input for the parallelization phase. The data partitioning neglects global data, which is stored in the external memory and therefore can be accessed by all processors. Instructions accessing local structures are assigned to the processor whose stack contains the data. Concretely, data partitioning is based on an affinity graph whose nodes correspond to the variables of a function. The affinities between variables are modelled as edge weights and indicate communication costs incurred if such variables are stored on different processors. The size of variables can be represented by node weights in order to balance the register and memory requirements. The resulting graph is partitioned using common graph partitioning techniques. Our current prototype uses the graph partitioning tool set METIS [Karypis and Kumar 1998], which aims at achieving approximately equally sized partitions, for load balancing between processors. Functional units are allocated using the BUG algorithm by Ellis [Ellis 1986]. We envision a holistic processor partitioning method based on affinity graphs, which considers both data objects and instructions.

The *parallelization phase* is implemented as a separate component called CHARISMA, whose basic idea has been presented in Section 2.3. An important challenge affects the granularity of the code integration: The referred scheduling and vectorization techniques operate on quite different contexts like basic blocks, loops, or traces. In order to simplify the first prototype implementation, we decided to perform fine-grained parallelization on basic block level. Otherwise, a schedule for

[3]**U**nified **P**rocessor **S**pecification **La**nguage

a loop might correspond to multiple schedules for the basic blocks of the loop, for instance. Furthermore, additional glue code is needed for software-pipelined loops to integrate them into a machine. In the future, we will also handle other techniques operating on loop level or optimizing traces. Currently, we use list scheduling for the scheduling part of the parallelization phase and vectorization is based on an adapted version of SLP [Larsen and Amarasinghe 2000] (see Section 2.3).

Immediately after scheduling, the *remote data dependencies* between different processors are determined. This information is used for the placement of *communication code* to exchange register values, which is described in Section 3.3.2. After register allocation and *peephole optimization*, *re-scheduling* is performed to produce a more compact schedule. Instead of just applying local optimizations to the existing schedule, the Data Dependence Graph (DDG) is re-constructed and scheduled again. This phase is also capable of inserting barrier instructions within a basic block on-the-fly. The compiler backend relies on a coherent model of the targeted processor. Commonly used methods are implemented in modules to enable code sharing, which offers an efficient validation of the compiler as well as adaption to similar architectures. Further details can be found in [Hussmann 2008].

The following sections present concepts for both – the QuadroCore architecture and the associated compiler design.

3.2 Realization of Synchronization

Synchronization between a certain set of processors P is realized by executing a special `barrier` instruction on each processor in P. As soon as all processors in P have executed their barrier instruction, they can continue execution. If only a proper subset $P' \subset P$ has reached a barrier, the processors in P' must wait for the remaining processors. In order to synchronize disjoint sets of processors at the same time independently, a `barrier` instruction has an immediate field which represents the set P as a bitmask, called the *barrier mask*. This mask is matched with the corresponding set of processors.

3.2.1 *Hardware Support for Synchronization.* In the asynchronous mode of operation, barrier instructions synchronize between independently operating instruction streams. Since the task of barrier placement is optimized during compilation, the hardware architecture has to ensure a very low cost instruction execution time without affecting the system's operating frequency. In our architecture, this synchronization is achieved in a single clock cycle, where each processor accesses a barrier status register asynchronously. Depending on when each of the processor encounters a barrier instruction, the barrier status register is set accordingly. When the required subset of barriers has been reached, the register is reset and the status is provided simultaneously to all the processors. Hence, constant polling of an external memory address, employed in classical synchronization methods is avoided. The single cycle restriction introduces a minimal variation in the system's operating frequency, discussed later in Section 4. This method of synchronization is faster than techniques implemented via software barriers in recent implementations such as in [Ito et al. 2008].

In the synchronous mode, the instruction streams operate in lock-step, synchronous fashion at every instruction. This ensures a predictable behaviour to allow the compiler to schedule the instruction to explore the maximum degree of instruction-level parallelism. The instructions are restricted to fixed cycles per instruction, explicitly fixing the execution time for all instructions. For example, instructions with data-dependent execution lengths, such as early exits in multiplications, are disabled. Although the execution time of each instruction is forced to a worst-case value, there is no additional delay involved in synchronizing between instruction streams. Here, the maximum operating frequency of the system remains unaltered. The choice of the variant is made in the re-scheduling phase during compilation.

3.2.2 *Placement of Barriers.* The re-scheduling phase of the compiler backend (see Section 3.1) can place barrier instructions to synchronize processors explicitly wherever necessary. However, it inserts local barriers within basic blocks and global barriers at function calls. Global barriers beyond basic blocks are added by a heuristic that avoids unintended overwriting of communication registers (in the shared register file) and takes global memory dependences into account. Rescheduling is based on reconstruction of the DDG followed by a list scheduling of each basic block. The selection heuristic was extended as illustrated in Figure 7: When list scheduling selects a node u from the ready list, all successors v which are executed on a processor other than u, will be marked with the barrier mask $\{u, v\}$. In the example, the two right-most cases of use (grey and dark-grey) are marked when the definition is extracted from the list.

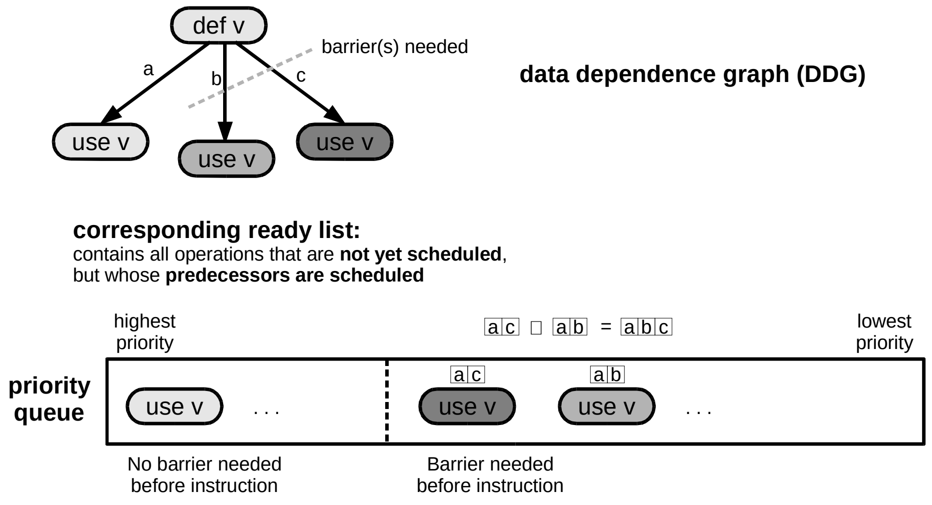
def v
barrier(s) needed
a
b
c
use v
use v
use v
data dependence graph (DDG)
corresponding ready list:
contains all operations that are **not yet scheduled**,
but whose **predecessors are scheduled**
highest priority
a c □ a b = a b c
lowest priority
priority queue
use v
...
a c
use v
a b
use v
...
No barrier needed before instruction
Barrier needed before instruction

Fig. 7. Barrier Insertion by List Scheduling

Each time an instruction with a marker is selected, a barrier will be inserted before this instruction. Such a barrier synchronizes the processors denoted by the markers of the instructions in the ready list. Thus, the combination of the barrier masks of all marked instructions is reduced to a single barrier, synchronizing all relevant processors. Then, the markers are removed. Consequently, a barrier between two dependent instructions $u \rightarrow v$ is always inserted *after* placing u and *before* placing v. In order to minimize the number of inserted barriers, marked instructions are selected with a lower priority. Concretely, the existence of a marker is used as a primary criterion, while the original criterion becomes the secondary criterion. Hence, synchronization instructions are inserted as late as possible in order to support coalescing of multiple barriers into fewer barriers.

3.3 Realization of Communication

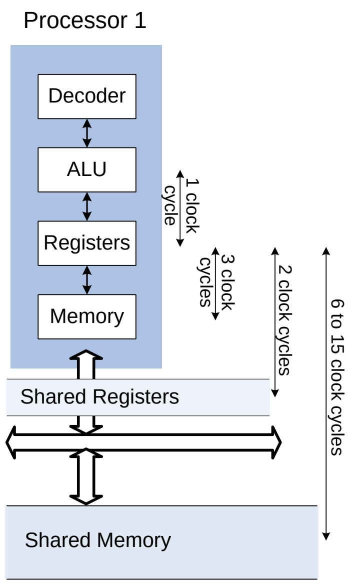

Fig. 8. Register and Memory Access Times in QuadroCore

3.3.1 *Shared Register File for Data Communication.* A shared register file has been introduced to ease the data exchange mechanism between the processors, as shown in Figure 2. This shared register file consists of 32 registers, accessible by all the processors via dedicated ports at all times. This set of registers is in addition to the 16-entry local register file that exists for each processor. Since there are independent read and write ports for each processor, no hardware arbitration mechanism is required for registers access, since the valid read-write sequences are scheduled during compilation. This ensures a two clock cycle access time for read or writes operations, enabled via special load and store instructions. As access to the external memory for data exchange takes 6-15 clock cycles, it is not used for communication. Hence, the round-trip time (write and read) is 4 clock-cycles for the shared register file in comparison to 16 to 30 clock cycles using the shared memory. Further, the compiler manages data dependency and read-write sequencing. A similar mechanism is added via instruction set extensions to allow sharing (or broadcasting) the condition flag of one of the processors for collective branch operations.

Figure 8 shows the access times within the memory hierarchy of QuadroCore for the current FPGA and standard cell implementations. The shared register file is only used for inter-processor communication, because its access time is longer than accessing the local registers of a processor. In order to utilize the shared registers for all instructions the encoding of register operands would have to be extended to store the additional register numbers. This would result in larger instructions, and hence an increase in code size.

3.3.2 *Placement of Communication Code.* Figure 9 illustrates the basic principle of integrating copy instructions into the schedule in terms of the shared register file. In the upper left corner of the picture, an excerpt of a DDG with three nodes is shown. Obviously, the *use* node depends on the two *def* nodes. The right hand data dependence is called a *local* dependence, because the participating nodes are scheduled on the same processor. The left hand data dependence is denoted a *remote* dependence, because the nodes are executed by different processors. Consequently, communication code is needed to transport the value v defined by `def v` from processor x to y where it is used by `use v, w`.

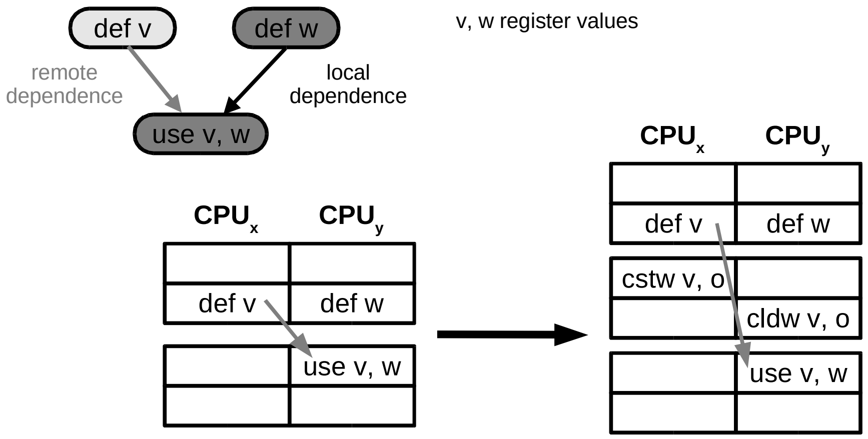

Fig. 9. Integration of Communication code using Instruction Set Extensions

In order to reduce the communication effort, the CoBRA compiler aims at handling as many remote dependencies as possible with one copy operation. For instance, a broadcast communication is chosen automatically, if a value is used by several processors. Furthermore, our placement strategy moves communication code out of loops by determining the most suitable position in terms of execution time. Concretely, we first identify all basic blocks which are located on *all* paths from a definition to its uses and then select the basic block with lowest execution frequency.

Our compiler uses the nesting depth of blocks as a static estimate. If there exists multiple definitions for a register value, the placement strategy selects a basic block, which is reached by as many definitions as possible. A performance comparison of the communication using the shared register file and the external memory can be found in Section 4.

3.4 Realization of SIMD

Typical SIMD architectures, like the well-known vector machines or multimedia extensions to general-purpose microprocessors [Hennessy and Patterson 2006], have special vector registers. In the QuadroCore, each processor has a separate register bank to store scalar values. Clearly, this architecture is only useful for the default MIMD mode. In order to minimize the alterations of the existing architecture to accommodate SIMD operations, the vector registers only exist conceptually. Let C be the number of processors and R be the number of registers per processor. Then, the j-th entry of the vector register r_i is mapped to the register $r_{i,j}$ of processor j, for $i \in \{0, \ldots, R-1\}$ and $j \in \{0, \ldots, C-1\}$, as illustrated in Figure 10. These registers $r_{i,j}$ for a certain i and all j are denoted as *homonymous* registers.

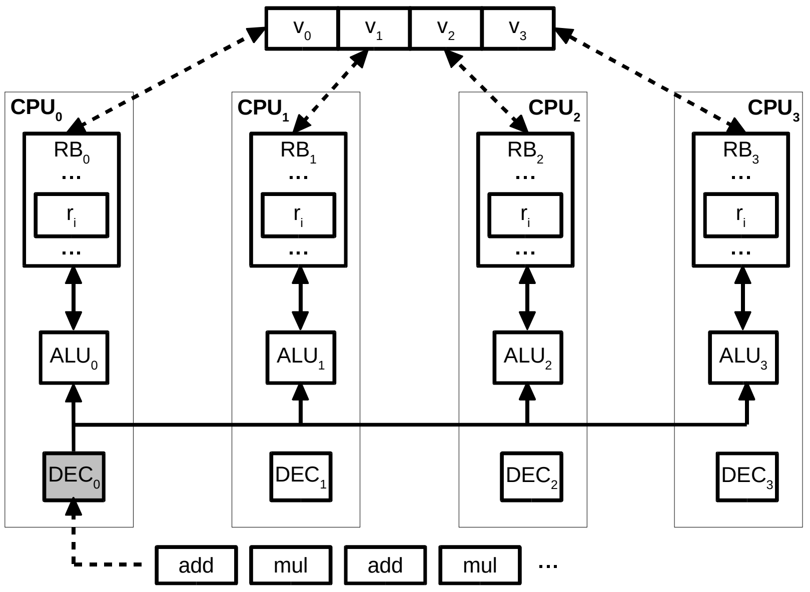

Fig. 10. Functionality of Vector Registers in SIMD mode

Consequently, a single instruction with encoded register operand r_i is executed by all processors j with different values stored in their registers $r_{i,j}$, respectively. In the SIMD mode, a processor c accesses memory data of word size w using $c * w$ as an offset to a base address.

3.4.1 *Vectorization and Register Allocation.* CHARISMA (see Section 3.1) utilizes the SLP approach [Larsen and Amarasinghe 2000] for vectorization (also characterized briefly in Section 2.3). The fundamental idea of the SLP approach is to identify adjacent memory accesses as an initial set of SIMD instructions. Further vectorizable statements can be found by traversing the def-use/use-def chains of the operands. According to [Larsen and Amarasinghe 2000], adjacency can be determined using both alignment information [Larsen et al. 2000] and array analysis. Our adjacency module is based on an extension of Common Subexpression

Elimination (CSE), that computes all expressions which only differ in constants of address computations. Such constants are annotated at the intermediate nodes in order to determine the adjacency afterwards. If a basic block contains both regular and irregular structures, the SLP algorithm produces code consisting of both SIMD and MIMD instructions. Finally, in order to reduce the overhead in reconfiguration, the scheduler aims at maximizing contiguous sections executed in either SIMD or MIMD mode.

As vector registers are mapped to multiple homonymous scalar registers in our architecture, the register allocation has to arrange the register values accordingly: At first, the virtual scalar registers used in SIMD mode are replaced by virtual vector registers, which actually represents a certain combination of those scalar registers. Then, transport instructions are inserted at the boundaries between SIMD and MIMD code to arrange register values properly. Finally, the registers are allocated using conventional techniques known from literature [Muchnik 1997]. We have developed a heuristic to place transport instructions efficiently, which takes def-use/use-def chains into consideration.

3.4.2 *Hardware Augmentation for SIMD Mode.* When multiple processors execute the same set of instructions for different data streams, the instruction fetch and instruction decode stage of the processors are redundant for all the participating processors. The task of instruction fetch and decode can be administered by a single processor. Hence, to support a single instruction stream to be executed on all the four processors, the decoder allows forwarding of its control and data signals from its instruction memory to all (or a subset of) the processors via the reconfigurable interconnect. The participating processors execute the same instruction as long as they operate in this mode.

As directed by the reconfiguration instruction, one of the processors can switch to a master mode and allow forwarding of the decoded instructions. The instruction memory and the decoding units of all the other processors can be switched to an idle mode. Further, when used in conjunction with instructions that allow fast access to adjacent memory locations, the overhead involved in accessing data in external memory is nullified. Depending on the application, one (or more) processor(s) can operate in the 'master' mode. A subset of processors in a cluster can operate in SIMD or MIMD mode of operation or in a combination of the two, simultaneously. Although processors share the same instruction stream, the data stream remains independent. Figure 11 shows the operation in SIMD mode, where processor 1 is the master processor and the instruction fetch and decode stages of processors 2, 3 and 4 are switched off. Memories make a significant impact on the total power consumption, as also stated in [Lambrechts et al. 2005] and nearly 80% in QuadroCore. The reduced instruction memory accesses and the unused decode stages of the slave processor results in significant power savings.

Fast Access to Adjacent Memory Locations The external memory enables sharing of data streams, and is accessible by all the processors via an arbitration mechanism. When multiple processors access this external memory, a round robin arbitration mechanism provides access in a sequential order. This procedure adds a significant overhead to external memory accesses, which is essential in data-parallel applica-

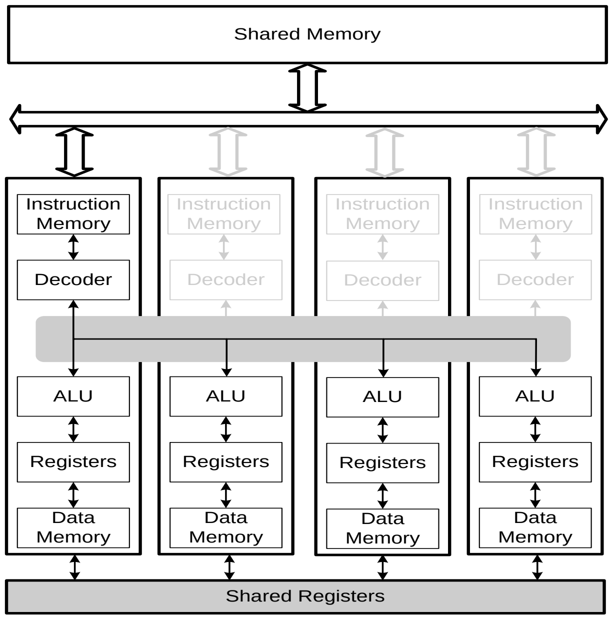

Fig. 11. QuadroCore in SIMD mode

tions. A significant bottleneck is introduced during simultaneous access to memory, which is inevitable in a multiprocessor organization, especially in the SIMD mode. To circumvent this bottleneck, fast access to adjacent memory locations is added via instruction set extensions. Using this memory access mechanism, a single transaction is sufficient to read (or write) multiple data-memory locations, which may represent consecutive locations of an array. This data read (or written) is then distributed (or collected) internally among the four processors. These special instructions avoid the delay involved during arbitration and reduce the total memory access time from a worst case of 15 clock cycles to exactly 7 clock cycles. Figure 8 shows the variations in the access time, based on the hierarchy of data access.

4. EXPERIMENTS AND PERFORMANCE ANALYSIS

In order to analyse the benefits of the presented architecture and design methodology, it is necessary to verify both the functional and performance characteristics of the architecture along with the compilation methodology. To validate the functionality of the QuadroCore architecture, the design was mapped on to our prototyping environment. For performance analysis, the design has been mapped onto UMC's 90nm standard cell technology. The compiler-driven reconfiguration has been validated for a set of networking applications, both in terms of mode changes and performance improvements. Additionally, the advantage of introducing this runtime reconfiguration as a mechanism for power savings is analyzed for two large data-parallel applications. The following sections present the set of experiments and performance analysis reports.

4.1 Prototype Implementation and Scalability

The choice of four processors in a cluster was a performance trade-off with respect to the memory access overhead in case of simultaneous access and to address the automatic extraction of parallelism. In [Fischer et al. 2003] it was demonstrated that irregular parallelism can be best exploited using a small number of processors (up to 4). However, the design description is parametric and can be scaled as per application requirements. To ensure functional validation, the QuadroCore was mapped onto a Xilinx Virtex2 XC2V6000 FPGA. The FPGA implementation has an operating frequency of 12.5 MHz and occupies 58% of the slices and 53% of the Block RAMs on a Virtex-II 6000. The purpose of mapping QuadroCore on the FPGA was to verify functional validation; hence, the design was optimized such that the entire QuadroCore and one switch-box could be mapped on to a single FPGA. For performance estimations in terms of timing, power and area, the architecture was implemented on UMC's 90nm standard cell technology.

As shown in Figure 12, the entire architecture can be replicated with intermediate stages of switch boxes [Niemann et al. 2007]. Scalability is achieved via hierarchical stages. In QuadroCore, sharing between processors is possible via a shared register file, which is limited to 4 processors in the cluster. Beyond this stage, scalability is achieved via introducing the network-on-chip inter-connectivity between the clusters. Similarly, access to memory is enabled via multiple stages. The final stage is the external memory coupled via high-speed-links to the individual switch boxes. This level of hierarchy ensures access to large amounts of data, as per application demands via high bandwidth interface to external memory. In contrast to the automatic parallelization of applications mapped to the QuadroCore architecture, manual partitioning is used across the NoC for applications exhibiting task-level parallelism.

Multiple QuadroCore processors are interconnected via a network-on-chip as discussed in [Niemann et al. 2006]. Depending on the application-specific computational demands, the resource requirement can be scaled accordingly. To enable accelerated prototyping, the entire architecture has been mapped to our scalable rapid prototyping system, RAPTOR2000 [Porrmann et al. 2009]. Using this environment, accelerated architectural prototyping for performance analysis of applications (described in C), is achieved. This experimental setup facilitates convenient cycle-accurate performance estimations for large benchmarks. The presence of this prototyping environment provides an accelerated mechanism to feedback application-level performance estimates to alter the architectural dimensions.

4.2 Core Area and Performance Estimation

The architecture was synthesized in UMC's 90 nm standard cell technology under typical operating conditions, using Design Compiler from Synopsys. Table II shows the variations in terms of maximum operating frequency (the clock period), area, total dynamic power, and mW/MHz, comparing the original multiprocessor core (both without memories) with the reconfigurable implementation. Since the memory architecture is unaltered, it has not been included in the synthesis reports. As seen from the results, the synthesized architecture shows an increase of about 14% in area. The area increase is mainly on account of the additional shared regis-

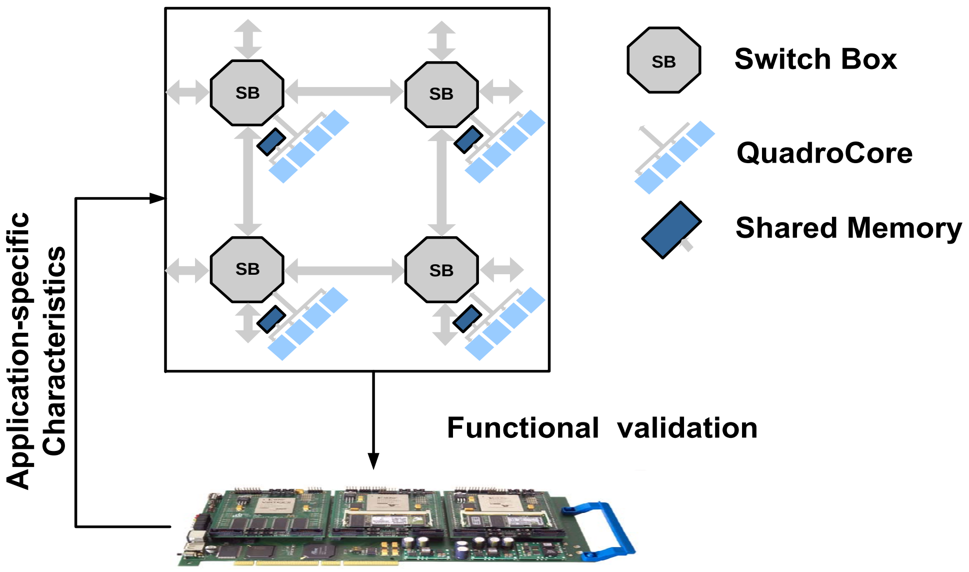

Fig. 12. Scalable Architecture and Prototyping Environment

ter file (about 6%) and the additional instruction set extensions (about 4%). The maximum operating frequency of the system is altered by about 3%, which is due of the additional layer of interconnects in the critical path. The reduction in the dynamic power (assuming 50% switching activity at the input) in comparison to the original architecture is attributed to the reduction in operating frequency of the reconfigurable multiprocessor. As observed in Table II, the mW/MHz ratio remains almost constant for both architectures, justifying the low overhead incurred on account of the reconfigurable capabilities. A more detailed power analysis is made in Section 4.4.

Table II. Standard Cell Synthesis Reports [4]

Architecture	Frequency	Area	Total Dynamic Power	mW / MHz
original	294 MHz	0.41 sq mm	6.45 mW	0.0219
reconfigurable	288 MHz	0.48 sq mm	6.26 mW	0.0217

4.3 Instruction-level Parallelism

Experiments were performed using the parallelizing compiler on the standard cell implementation of the QuadroCore reconfigurable multiprocessor architecture. The synthesized gate-level netlist was simulated for accurate power estimations. The current prototype implementation of the CoBRA compiler performs a fine-grained parallelization on basic block level. The following sections present evaluations of

[4]Power reports include only the Processor Core

performance comparison for the reconfigurable operating modes. For the initial evaluation, excerpts of integer benchmarks from practical audio and video applications were chosen. These computational blocks constitute typical transcoding algorithms for aggregation network access nodes.

convolution : Computes the discrete convolution of a 50 element array with a 16 element array.

fft : Represents the variable access pattern of a Fast Fourier Transformation with two arrays of 16 elements each.

poly: Evaluates a polynomial of degree 16 with variable coefficients.

sharpening : Computes the image sharpening algorithm for images with dimension of 10x10 pixels.

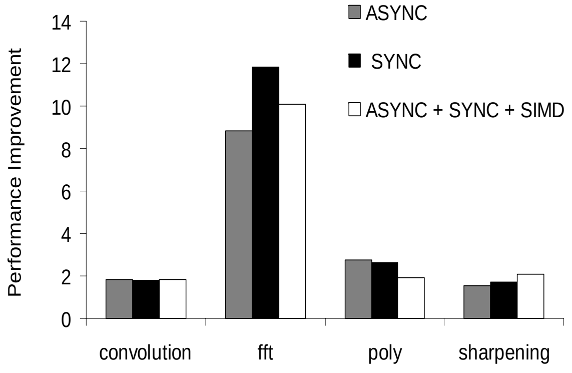

Fig. 13. Performance Improvement Compared to a Single Processor

In the current prototype, the compiler selects between asynchronous, synchronous or SIMD modes of operation for each basic block. The parallelization phase (see Section 2.3) produces both a vectorized (SIMD) and a scheduled (MIMD) version for the code of a basic block and selects the best result with respect to the estimated runtime. As a block may not be fully vectorizable, the SLP vectorizer may also yield some MIMD code and inserts reconfiguration instructions at the boundaries automatically. The decision of switching between synchronous and asynchronous mode is based on the inter-processor dependencies after parallelization (see Section 2.1). The barrier instructions are inserted in the re-scheduling phase (see Section 3.2.2), finally.

Run-time mode change is enabled via the presented single-cycle reconfiguration mechanism. In this paper, we focus our comparisons to a single processor and a cluster of four processors, nevertheless evaluations of the architecture with two, or three processors can also be performed. In Figure 13, the speedups obtained for the QuadroCore using the reconfigurable operating modes are shown, as compared to a single N-Core. The speedup is the ratio of the total execution time on QuadroCore versus the corresponding time on a single N-Core. The calculations for a single processor are solely for a single N-Core and do not include the overheads of interconnects etc. present in QuadroCore. In the plot, `ASYNC` corresponds to the asynchronous MIMD mode, `SYNC` corresponds to the synchronous MIMD mode, and `ASYNC+SYNC+SIMD` represents the combination of all the three modes, where decision of switching between the modes made by the compiler in order to optimize time. Additional timing analysis and execution time in each of the modes can be found in [Hussmann et al. 2007]. Some execution modes cannot be used exclusively for all parts of a given application. For instance, the SIMD mode is only applied for data-parallel code instead of redundant execution of single instructions on all processors when lacking of DLP. From the plot it can be seen that no single mode of operation is a true winner for all the applications. This further emphasizes that fixed hardware architecture with a single mode operation may not be best suited, even within the same application domain. The results suggests that the performance improvements depend on the type of the application and the corresponding mode of operation. For `convolution`, there is very little change in performance is noted between the various modes. However, for `fft` the CoBRA compiler achieves a significant improvement in performance by partitioning the algorithm onto four processors. Parallelizing `fft` yields a speed-up of 11, because the well-balanced register need avoids spill-code in contrast to a single processor. For `poly`, the asynchronous mode proves to be the most suited mode. In the case of `sharpening`, a performance improvement is observed when the `ASYNC+SYNC+SIMD` mode is chosen in. It must be noted, that the SIMD mode of operation also achieves power savings (see Section 4.4) and reduced code size (up to 22%). These results confirm the advantages of using the selected reconfigurable modes in our multiprocessor.

4.4 Power Savings via Reconfiguration in Data-parallel Applications

The MIMD mode of operation is the default mode, which is well suited for task-parallel applications. To analyse the benefits of the SIMD (data-parallel) mode of operation in terms for power, two computationally intensive applications that exhibit data-level parallelism were chosen. For stages of the application where the sequence of operations being executed on all the four processors are the same, a switch is made to the SIMD mode, to save power. Hence, in the SIMD mode of operation, a single instruction-fetch and decode results in four instructions being executed on the individual processors on different data.

4.4.1 *Multiplier in Elliptic Curve Cryptography.* A finite field multiplication in $GF(2^{233})$ used in Elliptic Curve Cryptography was chosen as a sample application to evaluate the performance on the QuadroCore reconfigurable multiprocessor architecture [Purnaprajna et al. 2008]. By applying the Karatsuba method iteratively, the multiplication of binary polynomials of degree 232 can be calculated with 27 finite field multiplications at word-level. The word-level multiplications are distributed among the four-processor QuadroCore architecture. Thus, four partial products can be computed in parallel to allow collective computation of the final results. Table III compares the variation in execution time for the application on a single processor, using the QuadroCore in MIMD mode, and using MIMD−SIMD reconfigurable mode in QuadroCore, for an operating frequency of 200 MHz. In order to achieve power savings QuadroCore is predominantly operated in the SIMD mode. As seen, a speedup of 3 has been observed in the MIMD mode of operation and a speedup of 2.88 in the MIMD−SIMD mode, in comparison to a single processor. Additionally, using the MIMD−SIMD results in power savings of 24% in comparison to the MIMD mode. The power savings are due to the reduced instruction fetches, and the subsequent reduction in instruction memory transactions. A small change in speedup is encountered on account of the additional clock cycles required to switch between the two modes, the resultant energy savings in the MIMD−SIMD mode of 16% confirm the advantage of reconfiguration.

Table III. ECC: Performance variations with Operating Mode [5]

Operating Mode	Execution Cycles	Speedup	Power(mW)	Energy (μJ)
Single Processor	9311	1	20.38	0.949
MIMD	3077	3.03	64.64	0.994
MIMD−SIMD	3237	2.88	49.51	0.801

4.4.2 *Self Organizing Maps.* As a second example, a neural network application - Kohonen's self-organizing map, which has been proven as a very effective tool in data analysis and exploration of high dimensional datasets [Kohonen 1989], was chosen. The algorithm involves executing the same set of operations on large amounts of parallel data. The common set of operations was mapped to all the four processors and data was distributed among all four processors. As the operations executed on all the processors are the same, MIMD−SIMD mode, switches are introduced wherever the control flow differs depending on the input data. Table IV shows the variations in the execution time, power, and energy for the operation in the single processor mode, the MIMD mode, and the MIMD−SIMD mode for a sample problem size. As compared to a single processor, a speedup of 3.45 was noted in the MIMD mode and of 3.41 in the MIMD−SIMD. The power savings in the MIMD−SIMD mode is about 15% in comparison to the MIMD mode. The resulting energy savings between the MIMD and MIMD−SIMD are 13%, which validates the advantage of reconfiguration.

Table IV. SOM: Performance variations with Operating Mode [6]

Operating Mode	Execution Cycles	Speedup	Power(mW)	Energy(μJ)
Single Processor	53,870	1	18.0	4.89
MIMD	15,608	3.45	52.0	4.05
MIMD−SIMD	15,790	3.41	44.0	3.50

[5]Power reports include both the Processor Core and Memory
[6]Power reports include both the Processor Core and Memory

5. CONCLUSION AND FUTURE WORK

We have presented a mechanism for introducing fast, run-time reconfiguration in our QuadroCore multiprocessor that marginally alters the architecture of the original legacy processors. A holistic evaluation of the reconfigurable QuadroCore multiprocessor is presented. The analyses show that, at the cost of about 14% increase in area and about 3% decrease in the maximum frequency of operation, a maximum performance increase of about 11 times in terms of cycles of operation has be achieved. The increase in area is mainly due to the shared register file, which contributes to nearly 6% of the total area. Hence, the area overhead on account of reconfiguration is small. Further, the instruction set architecture of the original legacy processor was preserved and extensions were included to allow co-operative multiprocessing.

As observed from the subset of audio and video processing computational building blocks, alterations in terms of operating modes are required. Larger applications usually combine multiple such computational blocks, where reconfiguring between modes can be profitable. A typical example could be an aggregation network access node (like DSL Access Multiplexers) where multimedia data is transcoded to suit the customer's equipment. Further, in [Dreesen et al. 2007], a scheme for dynamic reconfiguration of the connections between processors and registers banks has been reported. Ongoing work intends to extend this scheme of register reconfiguration to the QuadroCore architecture. This would enable processors to borrow registers from their neighbours in order to avoid spilling, which is costly in terms of time and power.

Our scheme of reconfiguration has been used as a mechanism for power and energy savings. Analysis in terms of power and energy for data-parallel applications (Karatsuba's multiplication and Self Organizing Maps) show power savings in the range of 18-24% and energy savings of 16-26% via reconfiguration. These two examples being a proof of concept, demonstrate applicability to most data-parallel applications. As a next step, the impact of post-layout characteristics is presently being analyzed.

6. ACKNOWLEDGEMENTS

The authors would like thank the reviewers for their valuable comments and suggestions.

This work was supported by the International Graduate School of Dynamic Intelligent Systems, the German Research Foundation (DFG) under the Collaborative Research Center (SFB 614) - Self-Optimizing Concepts and Structures in Mechanical Engineering, the Federal Ministry of Education and Research (BMBF) under PT-KT-01BU0661-MxMobile, and by Prof. Ramacher from Infineon Technologies. The authors of this publication are fully responsible for its content.

REFERENCES

Barretta, D., Fornaciari, W., Sami, M., and Pau, D. 2002. SIMD Extension to VLIW Multicluster Processors for Embedded Applications. In *ICCD '02: Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02)*. IEEE Computer Society, Washington, DC, USA, 523.

Bonorden, O., Brüls, N., Le, D. K., Kastens, U., Meyer auf der Heide, F., Niemann, J.-C., Porrmann, M., Rueckert, U., Slowik, A., and Thies, M. 2003. A holistic methodology for network processor design. In *Proceedings of the Workshop on High-Speed Local Networks held in conjunction with the 28th Annual IEEE Conference on Local Computer Networks (LCN2003)*. 583–592.

Compton, K. and Hauck, S. 2002. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv. 34,* 2, 171–210.

Dietz, H., Schwederski, T., O'Keefe, M., and Zaafrani, A. 1989. Static synchronization beyond VLIW. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*. ACM Press, New York, NY, USA, 416–425.

Dreesen, R., Hussmann, M., Thies, M., and Kastens, U. 2007. Register Allocation for Processors with Dynamically Reconfigurable Register Banks. In *Proceedings of the 5rd Workshop on Optimizations for DSP and Embedded Systems (ODES) held in conjunction with the 5rd IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2007)*.

Ellis, J. R. 1986. *Bulldog: A Compiler for VLIW Architectures.* MIT Press.

Fischer, D., Teich, J., Weper, R., and Thies, M. 2003. BUILDABONG: A Framework for Architecture/Compiler Co-Exploration for ASIPs. *Journal of Circuits, Systems, and Computers 12,* 3 (Juni), 353–375.

Gonzalez, R. E. 2006. A software-configurable processor architecture. *IEEE Micro 26,* 5, 42–51.

Gruenewald, M., Kastens, U., Le, D. K., Niemann, J.-C., Porrmann, M., Rueckert, U., Thies, M., and Slowik, A. 2004. Network application driven instruction set extensions for embedded processing clusters. In *PARELEC 2004, International Conference on Parallel Computing in Electrical Engineering, Dresden, Germany*. 209–214.

Gupta, R. 1990. Employing register channels for the exploitation of instruction level parallelism. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*. ACM Press, New York, NY, USA, 118–127.

Halfhill, T. R. 2006. Ambric's new parallel processor. Tech. rep., Microprocessors Report. Oct. Available from http://www.ambric.com.

Hennessy, J. L. and Patterson, D. L. 2006. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, San Mateo, CA.

Hussmann, M. 2008. Compiler-Driven Dynamic Reconfiguration of Architectural Variants. Ph.D. thesis, University of Paderborn.

Hussmann, M., Thies, M., and Kastens, U. 2005. Parallelizing Compilation through Load-Time Scheduling for a Superscalar Processor Family. In *Proceedings of the 3rd Workshop on Optimizations for DSP and Embedded Systems (ODES) held in conjunction with the 3rd IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2005)*.

Hussmann, M., Thies, M., Kastens, U., Purnaprajna, M., Porrmann, M., and Rueckert, U. 2007. Compiler-driven reconfiguration of multiprocessors. *in Proceedings of the Workshop on Application Specific Processors (WASP) 2007 held in conjunction with the Embedded Systems Week, 2007 (CODES+ISSS, EMSOFT, and CASES)*, 3–10.

Ito, M., Hattori, T., Yoshida, Y., Hayase, K., Hayashi, T., Nishii, O., Yasu, Y., Hasegawa, A., Takada, M., Ito, M., Mizuno, H., Uchiyama, K., Odaka, T., Shirako, J., Mase, M., Kimura, K., and Kasahara, H. 2008. An 8640 mips soc with independent power-off control of 8 cpus and 8 rams by an automatic parallelizing compiler. *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, 90–598.

Karypis, G. and Kumar, V. 1998. Multilevel Algorithms for Multi-Constraint Graph Partitioning. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*. IEEE Computer Society, Washington, DC, USA, 1–13.

Kennedy, K. and Allen, J. R. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Kohonen, T. 1989. *Self-organization and associative memory.* Springer-Verlag New York, Inc. New York, NY, USA.

Lambrechts, A., Raghavan, P., Leroy, A., Talavera, G., Aa, T., Jayapala, M., Catthoor, F., Verkest, D., Deconinck, G., Corporaal, H., Robert, F., and Carrabina, J. 2005. Power breakdown analysis for a heterogeneous noc platform running a video application.

Application-Specific Systems, Architecture Processors, 2005. ASAP 2005. 16th IEEE International Conference on, 179–184.

Larsen, S. and Amarasinghe, S. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation.* ACM Press, New York, NY, USA, 145–156.

Larsen, S., Rugina, R., and Amarasinghe, S. 2000. Alignment Analysis. Tech. Rep. LCS-TM-605, Massachusetts Institute of Technology. June.

Mei, B., Lambrechts, A., Verkest, D., Mignolet, J.-Y., and Lauwereins, R. 2005. Architecture exploration for a reconfigurable architecture template. *IEEE Des. Test 22,* 2, 90–101.

Muchnik, S. S. 1997. *Advanced Compiler Design Implementation.* Morgan Kaufmann Publishers.

Niemann, J.-C., Puttmann, C., Porrmann, M., and Rueckert, U. 2006. Giganetic a scalable embedded on-chip multiprocessor architecture for network applications. In *ARCS'06 Architecture of Computing Systems.*

Niemann, J.-C., Puttmann, C., Porrmann, M., and Rueckert, U. 2007. Resource efficiency of the GigaNetIC chip multiprocessor architecture. *Journal of System Architecture 53,* 5-6 (May), 285–299. Special issue on Architectural premises for pervasive computing.

Porrmann, M., Hagemeyer, J., Romoth, J., and Strugholtz, M. 2009. Rapid prototyping of next-generation multiprocessor socs. *In Proceedings of Semiconductor Conference Dresden, SCD 2009, Dresden, Germany, April 29-30, 2009, invited paper*.

Purnaprajna, M., Puttmann, C., and Porrmann, M. 10-14 March 2008. Power Aware Reconfigurable Multiprocessor for Elliptic Curve Cryptography. *Design, Automation and Test in Europe, 2008. DATE '08*, 1462–1467.

Sankaralingam, K., Nagarajan, R., McDonald, R., Desikan, R., Drolia, S., Govindan, M. S., Gratz, P., Gulati, D., Hanson, H., Kim, C., Liu, H., Ranganathan, N., Sethumadhavan, S., Sharif, S., Shivakumar, P., Keckler, S. W., and Burger, D. 2006. Distributed microarchitectural protocols in the trips prototype processor. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE Computer Society, Washington, DC, USA, 480–491.

Silicore. September, 2002. Wishbone system-on-chip (SoC) Interconnection Architecture for Portable IP cores. Tech. rep. Available from http://www.opencores.org.

Zhong, H., Lieberman, S. A., and Mahlke, S. A. 2007. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture.* IEEE Computer Society, Washington, DC, USA, 25–36.