Compiler-Driven Reconfiguration of Multiprocessors

Michael Hußmann
Institute of Computer Science
University of Paderborn
Paderborn, Germany
michaelh@upb.de

Michael Thies
Institute of Computer Science
University of Paderborn
Paderborn, Germany
mthies@upb.de

Uwe Kastens
Institute of Computer Science
University of Paderborn
Paderborn, Germany
uwe@upb.de

Madhura Purnaprajna
Heinz Nixdorf Institute
University of Paderborn
Paderborn, Germany
madhurap@hni.upb.de

Mario Porrmann
Heinz Nixdorf Institute
University of Paderborn
Paderborn, Germany
mario@hni.upb.de

Ulrich Rückert
Heinz Nixdorf Institute
University of Paderborn
Paderborn, Germany
rueckert@hni.upb.de

ABSTRACT

Multiprocessors enable parallel execution of a single large application to achieve a performance improvement. An application is split at instruction, data or task level (based on the granularity), such that the overhead of partitioning is minimal. Parallelization for multiprocessors is mostly restricted to a fixed granularity. Reconfiguration enables architectural variations to allow multiple granularities of operation within a multiprocessor. This adaptability optimizes resource utilization over a fixed organization.

Here, a unified hardware-software approach to design a reconfigurable multiprocessor system called QuadroCore is presented. In our holistic methodology, compiler-driven reconfiguration selects from a fixed set of modes. Each mode relies on matching program analysis to exploit the architecture efficiently. For instance, a multiprocessor may adapt to different parallelization paradigms. The compiler can determine the best execution mode for each piece of code by analyzing the parallelism in a program. A fast, single-cycle, run-time reconfiguration between these predetermined modes is enabled by executing special instructions which switch coarse-grained components like instruction decoders, ALUs and register banks. Performance is evaluated in terms of execution cycles and achieved clock frequency. First results indicate suitability especially in audio and video processing applications.

1. INTRODUCTION

Multiprocessor architectures allow executing single applications in parallel to improve the performance. 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 an application like loops, functions, threads, or tasks may be executed in parallel. Additionally, vector and SIMD processors explore data level parallelism. Partitioning a given problem also has the overhead of communication and synchronization. Hence, the level of granularity for partitioning without performance decrease identifies the most suited type of parallelism.

Conventionally, executing applications on multiprocessors is restricted to a single granularity predetermined by the programming style. Task level or data level parallelism is usually exploited manually by partitioning an application by hand or using special programming models. Automatic compilation often addresses parallelism at the instruction or loop level. Here, we present the advantages of augmenting a multiprocessor with reconfigurability to include varying degrees of synchronization and communication. This enables a single, unified architecture offering task, data, and instruction level parallelism.

In our approach (called CoBRA[1]) the compiler alters the architecture of the QuadroCore by choosing between a fixed set of operating modes. These modes (called *reconfigurable architectural variants*) can be enabled via reconfiguration at run-time. Applications may benefit from switching between these variants. A prominent example is to reconfigure between different parallelization paradigms like MIMD or SIMD. Given a program that exhibits both regular and non-regular structures, the compiler can determine the best execution mode by analyzing the parallelism. This has a number of benefits like improved performance, efficient resource utilization, reduced code size and power consumption.

The usage of a manageable set of variants leads to an enormous reduction in the design space, compared to fine grained reconfiguration. The compiler then addresses this finite design space efficiently using well-known program analysis techniques [1]. Furthermore, reconfiguration can be performed with a very low effort at run-time by switching fixed, coarse-grained components like instruction decoders, ALUs and register banks. This is in contrast to classical research towards reconfigurable architectures [2], where processors are typically coupled with reconfigurable logic to speed-up parts of a given application.

The rest of this paper is structured as follows: Section 2 motivates opportunities to reconfigure our multiprocessor architecture and compares it with existing approaches. Furthermore, we present the decision making for switching between the modes. The structure and concepts of both hardware and compiler are outlined in Section 3. The experimental evaluation and its results are discussed in Section 4. Section 5 concludes the paper and discusses future work.

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

2. RECONFIGURABLE MODES IN QUADROCORE ARCHITECTURE

Figure 1 shows, a high-level representation of our QuadroCore. The multiprocessor architecture comprises four RISC-based 32-bit processors, called N-Core [3]. The N-Core processors have a three-stage pipeline structure and a fixed 16-bit opcode length. Each processor has its own register file and local memory and operates independently in the cluster. Most arithmetical and logical operations provide a single-cycle execution time and load/store instructions have a two-cycle execution time. Sharing of data among the processors in the cluster is provided via an external memory, accessible over a shared wishbone bus. 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 parallelism, synchronization and method of communication can be achieved by adding reconfigurable operating modes to the architecture. These modes can be introduced via reconfiguration between execution of different applications or even within stages in 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 these modes is optimized to a single clock cycle. Hence, the advantage of using the most suitable mode is achieved without making a significant impact on the execution time and the maximum operating frequency. The individual processors allow run-time modifications to the architecture and support self-reconfiguration, as defined by the compiler. Hence, an explicit requirement for a reconfiguration controller, which is typical in FPGA-based designs, is entirely avoided in this methodology. Also, the need for separate configuration memory space is avoided, since the configuration data is embedded well within the instruction stream.

The following subsections describe the reconfigurable modes with methods of synchronization, communication as well as the reconfiguration between SIMD and MIMD.

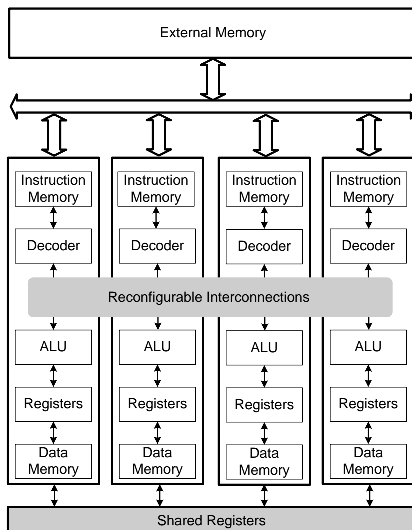

Figure 1: QuadroCore Architecture

2.1 Synchronization

Synchronization mechanisms employed in multiprocessors differ between synchronous (lock-step) and completely asynchronous operation. Often the synchronization overhead is a constraint for the level of parallelism explored. Fine grained parallelism necessitates frequent synchronization on account of dependencies between processors. In the presence of coarse grained parallelism processors could operate more independently and only synchronize when necessary. Hence, synchronization is chosen as a reconfigurable option which 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.

In comparison to our approach, the synchronization in [4] is always achieved using barriers. But an application with fine grained parallelism would imply a large overhead for synchronization. Also, a recent commercial multiprocessor architecture called AMBRIC [5] consists of RISC processors (called 'brics') arranged in a cluster, where each individual core runs at its own clock speed. If data needs to be exchanged, the processors are synchronized. In both these cases [4, 5], the architecture is more suited for applications with coarse grained parallelism due to the overhead of explicit synchronization for each data transfer. The synchronous mode of operation in our architecture allows operating in a lock-step fashion. This enables exploring the advantages of both lock-step architectures and asynchronous operation interchangeably. Hence, a variable granularity could be achieved depending on the mode of operation, varying between a fine grained architecture exploring instruction level parallelism to 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.

2.2 Communication

Inter-processor data dependencies demand a mechanism for exchanging register values. The same could be achieved using message passing between distributed memory or a single shared memory. Message-oriented architectures have no common address space, but the data is distributed to the different processors. Communication is realized by messages which copy data objects between the local memories. Message passing is more suited for large amount of data exchange but infrequent communication. As the processors of our architecture demand fast and frequent exchange of register values, we decided to use a shared memory for communication. Such a shared memory can either be based on external memory of the multiprocessor or a dedicated shared register file. Since access to external memory incurs a large overhead, a shared register file was introduced, which can be simultaneously accessed by all the processors efficiently. Although the access to this shared register file can be enabled via reconfiguration of the data communication mode, it was set as the default mode of operation on account of the performance improvement observed.

Considering related architectures, AMBRIC [5] offers a point to point communication via channels, which manage synchronization between the processors automatically.

However, data transfer is limited to neighbouring processors. In [6], Gupta presents the integration of shared register channels into a RISC based multiprocessor, which also 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 using channels always implies explicit synchronization. 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. Silicon Hive's processor [7] consists of multiple cells with distributed register files. The interconnection network allows data transfer between the functional units and register files. Reconfigurability in our context implies altering the control flow and synchronization.

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 level parallelism. In the SIMD mode, there is only one instruction stream decoded by the first processor, but executed on all processors with different data in their respective register banks. 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 the CoBRA compiler applies well-known scheduling and vectorization techniques [8] at first and finally select the best combination of modes. The selection heuristic can be based on parameters like execution time, code size, or estimated energy consumption. Currently, it aims at a fast execution time by minimizing the effort in reconfiguration.

In the same context, Barretta et al. [9] proposed a multi-clustered VLIW architecture, which can be switched between so-called ILP and SIMD modes. Currently, there only exists a simulator, but neither a proper hardware implementation nor a corresponding compiler. The proposed compiler is expected to identify pieces of code which can be executed in SIMD mode by determining accesses to disjoint memory blocks (provided in the source code or computed automatically). The CoBRA compiler uses the Superword Level Parallelism (SLP) approach [10], 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 by the number of targeted processors.

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

3. CONCEPTS OF HARDWARE AND COMPILER

Figure 1 shows the QuadroCore architecture, where the additional resources are highlighted in grey. This multiprocessor organization itself is scalable and reusable. However, here we confine the details of the architecture to the co-operative operation of four processors in a cluster. A single processor is composed of coarse grained building blocks such as decoders, registers files, and local instruction and data memory. The memory hierarchy allows 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 a six cycle overhead to the shared external memory. Hence, a single external memory access requires six clock cycles and extends upto fifteen clock cycles when all the processors make a simultaneous access, on account of the arbitration mechanism.

The instruction set architecture provides about 11% free opcode space to allow architectural enhancements. This free opcode space has been utilized to add instruction set extensions to allow runtime 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 reconfigurable operating modes. The hardware modifications on account of introduced instruction set extensions have been optimized to avoid altering the maximum operating frequency of the processor cluster.

In order to reconfigure the interconnections between the building blocks, a layer of interconnect was introduced between decode and execute stages of all the processors in a cluster. The interconnect network allows alterations to the control flow via instruction streams. The control flow between processors in the cluster 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 directed by the compiler, changes in terms of modifications to the inputs and outputs of the interconnect layer are introduced in a single clock cycle. The addition of this instruction and the associated hardware does not introduce a change in the timing characteristics of the entire cluster, since it is an independent operation and does not interfere with the existing instruction set architecture.

3.1 Structure of the Compiler Backend

Figure 2 illustrates the structure of the compiler backend, which has been derived from an existing backend for VLIW machines [11]. In contrast to the original backend, it features three additional phases (highlighted in grey) which are explained here.

Processor partitioning decomposes into partitioning of data objects and allocation of functional units, which is needed as input for the parallelization. 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 express communication costs that will occur, if such variables are stored on differ-

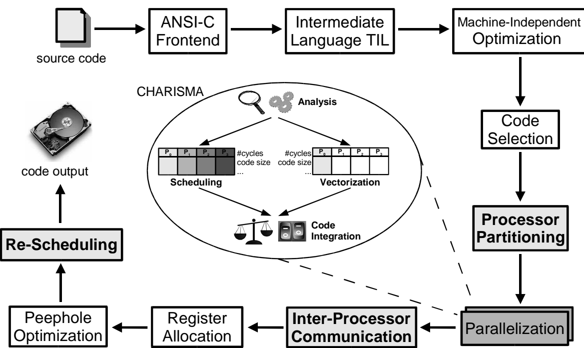

Figure 2: Structure of CoBRA backend

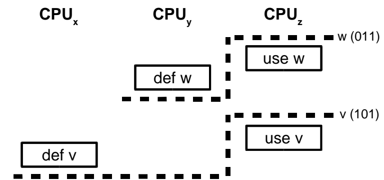

Figure 3: Example of barrier synchronization for the QuadroCore

ent 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 [12], which aims at achieving approximately equally sized partitions. Functional units are allocated using the BUG algorithm by Ellis [13]. We envision a holistic processor partitioning method based on affinity graphs which considers both data objects and instructions.

The *parallelization phase* is implemented in a separate component called CHARISMA, whose basic idea has already 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 prototypical implementation, we decided to perform fine-grained parallelization on basic block level. If not, a schedule for a loop might correspond to multiple schedules for the basic blocks of the loop, for instance. Furthermore, additional glue code would be needed for software-pipelined loops in order 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 [10] (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*, a *rescheduling* is performed to produce a more compact schedule. Instead of just applying local optimizations to the existing schedule, the Data Dependence Graph (DDG) is reconstructed and scheduled again. This phase is also capable of inserting barrier instructions within a basic block on-the-fly.

In the following sections the proposed concepts are presented both in terms of the compiler and hardware architecture.

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. Figure 3 illustrates an example.

3.2.1 Hardware Support for Synchronization

In the asynchronous mode of operation, barrier instructions allow synchronization between independently operating instruction streams. Since the task of barrier placement is optimized by the compiler, the role of hardware architecture is to provide a very low overhead instruction execution (in terms of the clock cycles) without affecting the system's operating frequency. Hence, this synchronization was achieved in a single clock cycle, where each processor accesses a synchronization status register asynchronously. Constant polling of an external memory address is avoided, since the status of the barrier is always available to all the processors simultaneously. The single cycle restriction introduces a minimal variation in the system's operating frequency, discussed later in Section 4.

In the synchronous mode, the instruction streams operate in lock-step, synchronous at every instruction. This provides a predictable behavior to allow the compiler to schedule the instruction so as to explore maximum degree of instruction level parallelism. The instructions are restricted to fixed cycles per instruction, explicitly retaining the execution time for all instructions. Although the execution time of each instruction is forced to a worst case value, there is no overhead involved in synchronizing between instruction streams. Here, the maximum operating frequency of the system remains unaltered.

3.2.2 Placement of Barriers

Rescheduling is based on a reconstruction of the DDG followed by a list scheduling of each basic block. However, it inserts only local barriers within basic blocks as well as global barriers at function calls. Global barriers beyond basic blocks are added by a heuristic which avoids unintended overwriting of communication registers and takes global memory dependences into account.

The selection heuristic was extended as follows: 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 $\{u, v\}$.

Each time an instruction with a marker is selected, a barrier will be inserted before this instruction. Such barrier synchronizes the processors denoted by the markers of the instructions in the ready list. Obviously, the combination of the barrier masks of all marked instructions reduced to a sin-

gle 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

3.3.1 Shared Register File for Data Communication

In Figure 1, a shared register file has been introduced to the ease the data exchange mechanism between the processors. The register bank consists of 32 registers, accessible to all the processors via dedicated ports, at all times. Since there are independent read and write ports for each processor, no arbitration mechanism is required for registers access. This ensures a 2 clock cycle access time for read and write operations, enabled via special instructions `cstw` and `cldw`. As the external memory for data exchange takes 6 to 15 clock cycles, it is not used for communication. Further, data dependency and read-write sequencing is managed by the compiler. A similar mechanism is implemented to allow sharing (or broadcasting) the condition flag of one of the processors for co-operative branch operations.

The shared register file is only used for communication, because its access time is longer than accessing the registers of a processor. Furthermore, the encoding of register operands must be extended to store the additional register numbers in order to utilize the shared registers for all instructions. This implies larger instructions, and hence an increase in code size.

3.3.2 Placement of Communication Code

Figure 4 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 definition 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`.

In order to reduce the communication effort, the CoBRA compiler aims at handling as many remote dependences 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. Such information may result from a profiling. 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 have special vector registers. In the QuadroCore, each processor has a separate register bank to store scalar values. Clearly, such design is useful for the default MIMD mode.

In order to minimize the changes of the existing architecture for the SIMD mode, 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 $i \in \{0, \dots, R-1\}$ and $j \in \{0, \dots, C-1\}$, as illustrated in Figure 5. In the following, such registers $r_{i,j}$ for a certain i and all j are denoted as *homonymous* registers.

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 SIMD mode, a processor c accesses memory data of word size w by using $c * w$ as an offset to a base address. The following paragraph describes the hardware support to access adjacent memory locations.

3.4.1 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 could be administered by a single processor.

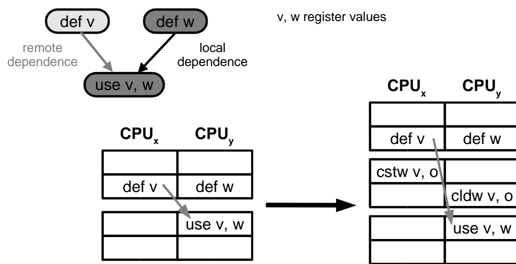

Figure 4: Communication of register values

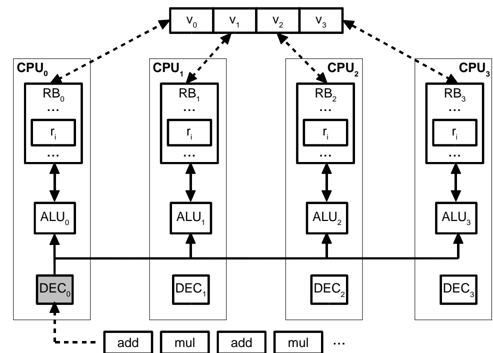

Figure 5: Functionality of SIMD mode

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. 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 could switch to a master mode and allow forwarding of the decoded instructions. The instruction memory and the decoding units of all the other processors could be switched to an idle mode, to allow power savings. Further, when used in conjunction with instructions which 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) could operate in the 'master' mode. A subset of processors in a cluster could 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.

Fast Access to Adjacent Memory Locations The external memory allows sharing of data streams, accessible by all the processors via an arbitration mechanism. When multiple processors access this external memory, the round robin arbitration mechanism provides access in a sequential order. This procedure adds a significant overhead to external memory access essential for sharing streams of data. 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. A single transaction allows accessing consecutive memory locations, which (could) represent consecutive locations of an array. Thus, the data accessed is distributed (or collected) internally among the four processors. A similar procedure is also applicable for storing data arriving from the four individual processors via a single write operation to external memory. These special instructions avoid the delay involved during arbitration and reduce the total access time from 15 clock cycles to 7 clock cycles. Figure 1 shows the variations in the access time, based on the hierarchy of data access.

3.4.2 Vectorization and Register Allocation

CHARISMA (see Section 3.1) utilizes the Superword Level Parallelism (SLP) approach [10] for vectorization, which has been 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 [10], adjacency can be determined using both alignment information [14] 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. In order to reduce the effort in reconfiguration, our 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 represent 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 [1]. We have developed a heuristic algorithm to place transport instructions efficiently, which considers def-use/use-def chains.

The current version of the compiler employs vectorization for generating MIMD code only in order to avoid additional transport instructions and SIMD registers. Hence, this so-called *pseudo* SIMD mode can be regarded as a good estimation of the results with the *real* SIMD mode.

4. EXPERIMENTS AND RESULTS

Experiments were performed using the parallelizing compiler on the hardware implementation of the proposed reconfigurable multiprocessor architecture. The same was simulated with a cycle accurate simulator. For exhaustive testing the model can be mapped to our FPGA based prototyping environment [15] for rapid evaluation of large benchmarks on hardware. The current prototype implementation of the CoBRA compiler performs a fine-grained parallelization on basic block level. The following sections present hardware estimates of area, power, clock frequency, evaluations of performance improvements using the shared register file, and finally performance comparison of the reconfigurable operating modes. For the initial evaluation we have selected small excerpts from practical audio and video applications. These computational blocks constitute typical transcoding algorithms for aggregation network access nodes. All benchmarks were evaluated in terms of cycles per instruction and the operating frequency.

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.

mm: Multiplies two 4x4 matrices.

sharpening: Sharpening algorithm for images with dimension of 10x10 pixels.

vectormuladd: Multiply-accumulate on vectors of 10 elements.

Area and Performance Estimation. Table 1 shows the variations in terms of maximum operating frequency (the clock period), area, total dynamic power, and mW/MHz, comparing the original multiprocessor architecture with the reconfigurable implementation. The architecture was synthesized in UMC 130nm standard cell technology using Synopsys' design compiler. As seen from the results, the synthesized architecture shows an increase of about 10% of area. The maximum operating frequency of the system is altered by about 5%. The reduction in the dynamic power calculations is attributed to the reduction in operating frequency of the reconfigurable multiprocessor, as can be seen by the fact that the mW/MHz ratio stays constant for both architectures.

Table 1: Standard Cell Synthesis Reports -Typical

Architecture	Clock Period (ns)	Area (sq mm)	Total Dynamic Power (mW)	mW / MHz
original multiprocessor	4.74	0.77	42	0.2
reconfigurable multiprocessor	5.00	0.85	40	0.2

Table 2: Register File: Performance Reports

Benchmark	With Shared Register File	With External Memory	Performance Improvement
convolution	***12428*** cycles	22328 cycles	1.79
sharpening	***35602*** cycles	49187 cycles	1.38

Advantages of Shared Register File. The shared register file is used for communication by default as a consistent performance improvement was observed, although a reconfiguration could enable access to external memory (see Section 3.3.1). For all the benchmarks mentioned above, a performance improvement was observed and Table 2 shows two sample test cases, with the execution time in terms of clock cycles.

Reconfigurable Modes. In the current prototype, the compiler selects between asynchronous, synchronous or SIMD modes for each basic block or just uses a single processor. Runtime mode change is enabled via a single cycle reconfiguration. We restrict our comparisons to a single processor and a cluster of four processors, although evaluations of the architecture with two, three of four processors may be performed.

In Table 3, *ASYNC* represents the asynchronous mode, *SYNC* is the synchronous mode and *SIMD* is the SIMD mode. A combination of one or more modes is achieved via reconfiguration, as suggested by the compiler. It has to be noted that no single mode of operation is a true winner for all the applications. This further emphasizes the point that a fixed hardware architecture may not be suitable for application, even within the same application domain.

The results imply that the performance improvements depend on the type of the application and the corresponding mode of operation. For `convolution`, `mm` and `fft` the CoBRA compiler achieves a significant improvement in performance by partitioning the algorithm into four processors. Parallelizing `fft` yields a speed-up of 10, because the well-balanced register need avoids much spill code in contrast to a single processor. Even when multiple processors access the external memory with a significant overhead, a performance increase can be observed compared to a single processor. However, in case of `vectormuladd` it may be seen that an increase in the number of processors does not have a positive effect on the performance (as in the case of synchronous and asynchronous mode). These results demonstrate that using the selected reconfigurable modes in our multiprocessor is beneficial, since the architecture allows switching between these modes and the compiler selects the optimal implementation for each piece of code.

5. CONCLUSION AND FUTURE WORK

The holistic evaluation of the reconfigurable multiprocessor shows that at the cost of about 10% increase in area and about 5% decrease in the maximum frequency of operation, a maximum performance increase of about 10 times in terms of cycles of operation can be achieved. The initial evaluations are complete both in terms of the reconfigurable hardware architecture and the associated retargetable compiler. According to the evaluations, a fixed multiprocessor organization may not be efficient for all applications. As observed from the examined subset of audio and video processing computational building blocks, alterations in terms of operating modes are required. Larger applications usually combine multiple blocks, where reconfiguring between modes is 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.

Beyond evaluating larger benchmarks, we aim at dynamically adapting the number of active processors in different sections of a program to optimize the energy consumption. Furthermore, we plan to evaluate a reconfiguration of the register banks at run-time, which has been proposed in [16]. In the QuadroCore, this technique is expected to further accelerate inter-processor communication. Processors with high register needs may borrow registers from a neighbouring processor by reconfiguring their respective input interconnects.

6. ACKNOWLEDGEMENTS

This work was supported by the International Graduate School of Dynamic Intelligent Systems, the Federal Ministry of Education and Research(BMBF) under PT-KT-01BU0661-MxMobile, and Infineon Technologies, Prof. Ramacher. The authors of this publication are fully responsible for its content.

7. REFERENCES

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

[2] R. Hartenstein, "A Decade of Reconfigurable Computing: a Visionary Retrospective," in *Proceedings of the Conference on Design, Automation and Test in Europe.* IEEE Press, 2001, pp. 642–649.

[3] M. Grünewald, U. Kastens, D. K. Le, J.-C. Niemann, M. Porrmann, U. Rückert, M. Thies, and A. Slowik, "Network application driven instruction set extensions for embedded processing clusters," in *PARELEC 2004, International Conference on Parallel Computing in Electrical Engineering, Dresden, Germany,* 7 - 10 Sept. 2004, pp. 209–214.

[4] H. Dietz, T. Schwederski, M. O'Keefe, and A. Zaafrani, "Static synchronization beyond vliw," in *Supercomputing '89: Proceedings of the 1989*

Table 3: Reconfigurable Modes: Performance Reports

Benchmark	Modes	Clock Period	Execution Cycles	Performance Change
mm	Single Processor	4.74 ns	681	
	ASYNC	5.00 ns	***207***	3.28
	ASYNC + SYNC	5.00 ns	226	3.01
	ASYNC + SYNC + SIMD	5.00 ns	236	2.88
fft	Single Processor	4.74 ns	34431	
	ASYNC	5.00 ns	4105	8.38
	ASYNC + SYNC	5.00 ns	***3165***	10.87
	ASYNC + SYNC + SIMD	5.00 ns	3662	9.40
convolution	Single Processor	4.74 ns	16871	
	ASYNC	5.00 ns	***12330***	1.36
	ASYNC + SYNC	5.00 ns	12428	1.35
	ASYNC + SYNC + SIMD	5.00 ns	16231	1.03
sharpening	Single Processor	4.74 ns	40069	
	ASYNC	5.00 ns	38486	1.04
	ASYNC + SYNC	5.00 ns	35602	1.12
	ASYNC + SYNC + SIMD	5.00 ns	***29026***	1.38
vectormuladd	Single Processor	4.74 ns	883	
	ASYNC	5.00 ns	1482	***0.56***
	ASYNC + SYNC	5.00 ns	1438	***0.61***
	ASYNC + SYNC + SIMD	5.00 ns	***755***	1.17

ACM/IEEE conference on Supercomputing. New York, NY, USA: ACM Press, 1989, pp. 416–425.

[5] T. R. Halfhill, “Ambric's new parallel processor,” Microprocessors Report, Tech. Rep., Oct. 2006, available from http://www.ambric.com.

[6] R. Gupta, “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.* New York, NY, USA: ACM Press, 1990, pp. 118–127.

[7] B. Mei, A. Lambrechts, D. Verkest, J.-Y. Mignolet, and R. Lauwereins, “Architecture exploration for a reconfigurable architecture template,” *IEEE Des. Test*, vol. 22, no. 2, pp. 90–101, 2005.

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

[9] D. Barretta, W. Fornaciari, M. Sami, and D. Pau, “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).* Washington, DC, USA: IEEE Computer Society, 2002, p. 523.

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

[11] O. Bonorden, N. Brüls, D. K. Le, U. Kastens, F. Meyer auf der Heide, J.-C. Niemann, M. Porrmann, U. Rückert, A. Slowik, and M. Thies, “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)*, Oct. 2003, pp. 583–592.

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

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

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

[15] H. Kalte, M. Porrmann, and U. Rückert, “A prototyping platform for dynamically reconfigurable system on chip designs,” in *Proceedings of the IEEE Workshop Heterogeneous reconfigurable Systems on Chip (SoC)*, Hamburg, Germany, 2002.

[16] R. Dreesen, M. Hußmann, M. Thies, and U. Kastens, “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)*, Mar. 2007.