Run-time Reconfigurable Cluster of Processors

Madhura Purnaprajna and Mario Porrmann
Systems and Circuit Technology
Heinz Nixdorf Institute, University of Paderborn, Germany
{madhurap, porrmann}@hni.upb.de

***Abstract*—High performance requirements often necessitate redesigns for every new application, resulting in long time-to-market. Every architectural change involves costs in terms of hardware design, verification and fabrication. As an alternative, architectural flexibility provides easy adaptability to different application domains in order to avoid the high cost of redesigns. Hence, a method of reusing the basic building blocks within processors to enable co-operative multiprocessing is proposed. Run-time reconfiguration is used as a method for application-specific customisation. Here, a method of application description in conjunction with a flexible multiprocessor template is proposed. Finally, the costs and benefits of this approach are analysed for a computationally intensive algorithm in terms of execution time and power consumption. The impact of variations in application-specific characteristics on the proposed architecture, are also analysed.**

I. Introduction to template based designs

Most processors have evolved using existing legacy architectures in order to provide backward compatibility. Enhancements like additional functional units, customized hardware accelerators, register files, and instruction set extensions provide application-specific accelerations. These enhancements require both hardware redesigns and software regeneration. To avoid significant redesigns, configurable processors such as from ARC [1] and Tensilica [2] use template based designs, where configurability at design time provides application-specific customisation. The same philosophy is seen in some network processors (e.g., BCM1250 from Broadcom), where multiple legacy cores are packed together to enable parallel packet processing. Similarly, in embedded processing Silicon Hive's processors [3] use multiple programmable cores to add instruction slots in an ultra-long instruction word to support instruction-level parallelism. Also, the pipeline processor from Rapport called Kilocore [4] uses a collection of processing elements in a single pipeline for enhanced parallel processing. Using the same analogy, PicoArray provides a flexible two dimensional grid of processors using a 3-way VLIW processor, with four such processors in a cluster. These regular processing fabrics provide coarse-grained architectural customisations in comparison to fine-grained FPGA fabrics. Table I summarizes the above mentioned architectures in terms of the range of configurability using existing processors, as a method of application-specific adaptability.

Off-the-shelf products such as FPGAs have been used to circumvent the time-to-market delay in designing customized processors. FPGAs are cost-effective solutions for low-volume production. Additionally, reconfigurability allows time multiplexing designs for area savings, and eases in-field upgrades encountered during the application's lifetime. However, fine-grained reconfigurable architectures have the disadvantage of a significant reconfiguration overhead, in terms of area, reconfiguration time, and energy consumption. Hence, these architectures are often used as co-processors along with other legacy processors to accelerate compute-intensive parts of the application. Run-time reconfiguration of FPGAs is intensively discussed in research. But the absence of automatic design-flows currently limits its applicability to real-world problems. Similarly, multiprocessor architectures are advantageous on account of their on-chip parallelism, although usually suited for a fixed granularity. For example, parallel processor arrays such as Ambric [5], Tilera [6] and multi-core network processors are well suited for data parallel applications. In contrast, a coarse grained architecture called TRIPS allows introducing application-specific characteristics, such as parallelism within the architecture as described in [7]. The TRIPS architecture is composed of large coarse-grained components, which are partitioned as processor and memory subsystems. Point-to-point communication channels allow exposure to software for optimization. Overall, customisations have been introduced during design time for application-specific accelerations. This typically involves adding additional hardware or introducing major architectural restructuring. Another dynamically reconfigurable processors called, FlexCore [8], has an interconnect, which allows the datapath to be reconfigured. Further, design acceleration is achieved by inserting customized hardware within the same compilation framework. However, this framework requires redefining the instruction format and necessitates the use of a long instruction format. This in turn requires a three-fold higher bandwidth to ensure adequate datapath utilization.

Unlike the above-mentioned methods, we propose a reconfigurable multiprocessor called QuadroCore, where reconfigurability is introduced when using legacy processors. Adding limited reconfigurable logic in terms of reconfigurable interconnects allows using the same instruction set architecture and ensures binary compatibility. Further, application-specific customisation is achieved via co-operative processing within the multiprocessor. Our objective is to introduce a scheme of reconfiguration that can be extended to most multiprocessors, independent of the architecture of the processor cores that are used.

TABLE I
CONFIGURABILITY AND RECONFIGURABILITY IN EXISTING ARCHITECTURES

Core	PE	Inter PE Communication	Application Domain	Customization
ARC/Xtensa	RISC	Queue/FIFO based	Embedded processing	Register files, Hardware accelerators
SiliconHive	VLIW	Programmable Interconnect	Image, Video, Signal processing	Functional units, Register files, Instruction slots
Rapport Kilocore	8bit CPU	Queue and Register file interconnects	Packet processing	Pipelines
PicoArray	3-way, VLIW	Bus and Switch Box	Software Defined Radio	Inter-processor communication
Ambric	32-bit RISC	Queue based	Video, Image processing	Reconfigurable Interconnects, Scalability
Tilera	RISC	Nearest neighbour	Data-parallel	Inter-PE communication
FPGA	LUTs	Switchboxes, Routes	Compute Intensive	LUTs, Routing

II. PROPOSED RECONFIGURABLE MULTIPROCESSOR TEMPLATE

A fixed architecture often renders resources as unused or inefficiently used depending on the application that is being executed. This influences resource efficiency, power consumption and the overall system performance. In order to achieve a balance between application domain customisation while reusing the base architecture of a multiprocessor, a generalized template is proposed. The template in Figure 1 shows four loosely coupled processors in a cluster. Reconfigurable connectivity between the existing resources is added as an architectural feature. This feature extends the usability of the multiprocessor architecture to applications with varying resource requirements and degrees of parallelism. Hence, the proposed reconfiguration capabilities, when added to the underlying hardware, retain the original structure of the processor and its instruction set. Reconfigurable interconnects are introduced, which allow interchangeable resource connectivity. The building blocks within a processor are treated as distributed resources, accessible by all (or a subset of) processors. This modular and reconfigurable architecture allows easy reusability and scalability to suit application requirements. This arrangement allows cooperative resource sharing within the multiprocessor hierarchy.

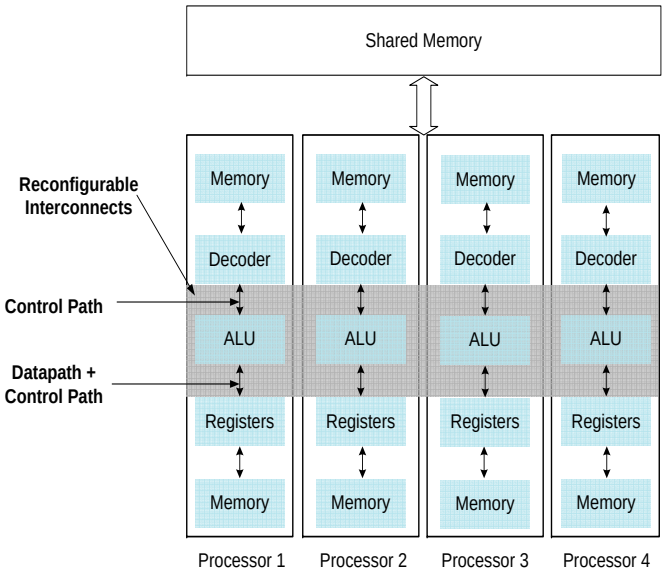

Fig. 1. Reconfigurable Multiprocessor Template

A. Instruction to Control Reconfiguration

The decision of altering the existing structure is driven by the instruction executed. Hence, the choice of resources and their variations are determined by the compiler and requested during run-time. A quick, single cycle run-time reconfiguration ensures low overhead in terms of time required to reconfigure this resource connectivity. A special reconfiguration instruction executed during run-time connects the existing resources. Thus, depending upon the resources demanded by the application these instructions are executed at boundaries between regions where a change in resource requirement is observed during program analysis. This reconfiguration instruction acts as the configuration information to determine the functionality of the reconfigurable interconnects between the intermediate stages of the instruction pipeline.

The interconnect introduced between the decode & execute stages, and execute & register read/write stages is composed of multiplexers that allow run-time selection between the multiple control paths (and/or data paths) among the four processors.

B. Reconfiguration Design Space

The reconfigurable architectural capabilities introduced in the QuadroCore cluster are listed below. The corresponding figures depict the logical representation of the modifications. Further, the method of invoking these variations via high-level programming is shown with sample code.

1) SYNC / ASYNC: Depending on the amount and frequency of inter-processor data exchange, the processors in the cluster can operate synchronously at instruction level or asynchronously. The cluster can be adapted according to the application characteristics during run-time, since both fine-grained synchronization scheme (for instruction-level parallelism) and a coarse-grained independent operation (for task-level parallelism) are supported. The run-time change in synchronization is achieved by introducing a synchronization instruction between parts of the application where a change in application characteristics is noticed.
By default, the cluster operates asynchronously and exchange of data is made possible via explicit barrier synchronization, which has the following construct, where *mask* represents the processors (all or a subset) that need to be explicitly synchronized each time.

```
barrier(mask);
```

The execution time for this instruction is one clock cycle, which is dependent on the arrival time of all the processors executing this instructions.
The following instruction switches processors listed by *mask* to a synchronous mode of operation, where *mask* represents the processors (all or a subset) that operate synchronously.

```
synchronize(mask);
```

In this mode, the execution times for all the instructions need to be fixed during compilation.

2) SIMD / MIMD: The choice of data-parallel or task-parallel behaviour steers architectural characteristics. MIMD mode allows asynchronous operations on independent data and instruction streams. SIMD mode coordinates all the four data-paths with a single instruction stream, thus saving energy via reduced memory interactions. Figure 2 shows the operation for Processor1 and Processor2 in SIMD mode, where the control path between the decode and execute stages are reconfigured.

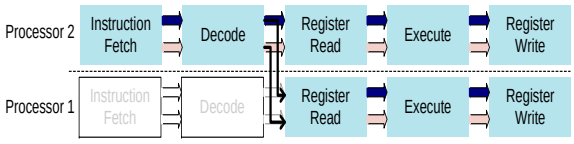

Fig. 2. Single Instruction Stream, Multiple Data Stream

The MIMD mode is the default mode of operation and a SIMD mode of operation can be via a reconfiguration instruction. Following is a case of loops, as shown below for a *for* loop.

```
for-simd(i=0; i<4; i++)
{
operations in SIMD mode
}
```

Where, *for-simd* results in instruction fetch and decode managed only by one of the processors. Thus making a corresponding reduction in code-size, instruction-fetch & decode. A fast memory access mechanism circumvents the communication delay in accessing the shared memory for all the participating processors by simultaneously fetching multiple memory locations followed by internal re-distribution.

3) Register Sharing: Register sharing among processors is possible on account of the reconfigurable interconnect introduced between the ALUs and the register files. For applications with high register pressure, registers from the neighbouring processors are used. Figure 3 shows the reconfigured register write (control and datapath), which helps to avoid the register read operation for the next operation and minimizes the data-exchange overhead between the two processors.

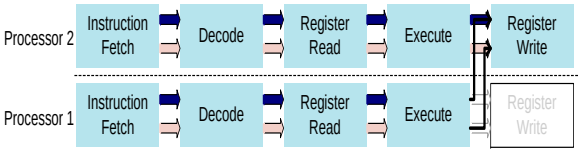

Fig. 3. Sharing Registers Contents

In the following code, the *share* construct moves the result stored in the local register file of Processor2 (*X[2]*) to the local register file of Processor1 by reconfiguring the register write stage of Processor2 to be directed to the local register file of Processor1. This mode saves data transfer time (via memories), hence resulting in reduced number of instructions and execution time.

```
share(X[2],2);
```

4) Word-length Configurability: For a given architecture, the effective resource utilization depends on the mapping of the application. Thus, a variation in the valid word-length as defined in the application directly influences resource utilization and power dissipation. Here, a variable word-length property of the processor allows using only the required word length as defined by the application. Also, multiple 32-bit ALUs from the neighbouring processors are merged to expand the word-length of a processor. Figure 4 shows the representation, where two 32-bit data paths are merged to form a single 64-bit data path, also avoiding redundant instruction fetch and decode operations.

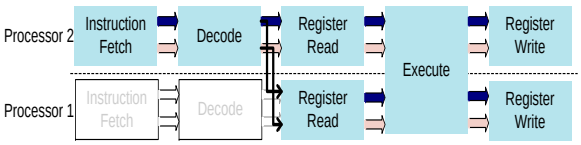

Fig. 4. Varying the ALU Word-length

```
tX = add64(X[0], X[1]);
```

The merging of the two ALUs is enabled by the instruction *add64*, which ensures that the addition operations include the carry over of the lower order bytes to the higher order ALU in the neighbouring processor. Reconfiguring the adders permits carry-over logic, and saves instruction fetches for word-width operations provided all the participating ALUs execute the same instruction. Similarly, the same is applicable to the other arithmetic units and ALU operations, viz. subtraction, division, multiplication, etc.

5) Asymmetric Clustering: In this mode of operation, parts

of the non-functional parts of the processors are replaced by the corresponding building blocks borrowed from the nearest neighbour, by actively reconfiguring the interconnections. This is feasible since all the processors are identical and the resources of the neighbouring processors are easily accessible. Figure 5 shows the reconstructed control and datapath using the fault-free resources of two adjacent processors.

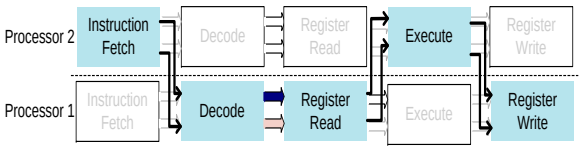

Fig. 5. Reusing Resource to Restore Normal Operation

In the following construct, the *test()* operation gathers the health of the individual processing elements (represented by *mask*) and chooses the fault-free resources to restore normal processor operation.

```
test(mask);
```

The identical resources and loose coupling between the processor building blocks enable this mode of operation.

C. Scalable Reconfigurable Multiprocessor

The processor's constituent components, such as ALUs, register files, memories and communication interfaces, constitute a library of reusable building blocks. This library is used to build processors with varying number and combination of components. Figure 6 shows multiple QuadroCore processors, interconnected via switch boxes. All the above discussed reconfiguration schemes exist in all the processors and can be used interchangeably, as demanded by the application. An automated compiler-driven design flow for the QuadroCore processor is presented in [9]. The switch-boxes use packet-based inter-cluster communication. A detailed description of the proposed mechanism is given in [10]. Scalability, easy adaptability, and extensibility are the main advantages of this architecture.

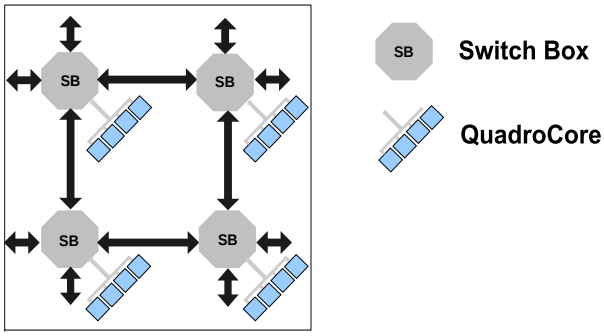

Fig. 6. Scalable Reconfigurable Multiprocessors

D. Costs and Benefits

The layer of interconnects are controlled via instruction set extensions to alter the control and dataflow between the decode, execute, and register access stages. These enhancements ensure that the base instruction set architecture is reused and reconfiguration is managed at high-level of abstraction, as suggested using the above-mentioned programming model. All the proposed configurations and the programming constructs result in a reduction in the number of instructions, hence reduced memory transactions, hence lower power. The benefits in terms of clock cycles are at a cost of additional multiplexers and additional routing interconnects between the processing stages, which result in reduced overall frequency. Therefore, the proposed methodology is a trade-off between achieved clock frequency and the cycles required for application execution.

III. QuadroCore Performance

QuadroCore is our reconfigurable multiprocessor architecture composed of four 32-bit RISC-based processors called N-Core (described in [11]), targeted for network processing applications. A combination of four processors forms a cluster called QuadroCore. This hierarchy of processors is extended to multiple clusters interconnected by a network on chip, with a packet-based inter-cluster communication as proposed in [12]. The architecture was implemented on UMC's 90nm standard cell technology. On adding reconfigurable modes, a change of 10% in area and operating frequency of 5% was observed as compared to the original architecture [13], as show in Table II. The reduction in dynamic power observed is on account of the reduced operating frequency.

TABLE II
Standard Cell Synthesis Reports - Typical Operating Conditions

Architecture	Clock Period	Area	Dyn. Power
Original multiprocessor	4.74 ns	0.77 sq mm	42 mW
Reconfigurable QuadroCore	5.00 ns	0.85 sq mm	40 mW

A. Application Mapping

Partitioning a compute intensive application onto multiple processors incurs the overhead of communication (T_{comm}) and synchronization (T_{sync}) for data exchange between processors, apart from the computation time ($T_{compute}$) on each of the processors. Each of the above mentioned modes have an influence on the time, power, and performance of the application. In general, the total execution time of an application with N instructions may be expressed as:

$$T_{total} = \sum_{i=1}^{i=N} (T_{compute_i} + T_{comm_i} + T_{sync_i}) \qquad (1)$$

Application partitioning can be achieved via divide and conquer algorithms, for task partitioning or data partitioning. The objective of task partitioning is to distribute the compute intensive operations onto the available processing elements.

TABLE III
QUADROCORE RECONFIGURATION DESIGN SPACE: COSTS AND BENEFITS

Steps	Operating Mode	Cycles	Benefits	Construct	Comments
Step 1	MIMD	$\frac{15*N}{4}$	None	Default	Instruction and Data in local memory
	SIMD	$\frac{7*N}{4}$	T_{total}, P_{im}	$copy-simd(x[i])$; $copy-simd(a[i])$;	Distribute a(k) and x(n-k) from shared memory
	Register Sharing	$\frac{2*N}{4}$	T_{reg}, T_{comm}	$share(x[0],x[1],x[2],x[3])$;	Share variables residing in local memory of first processor
Step 2	MIMD	$N*T_a$	None	$for(i=0;i<4;i++)$ $y(i)=a(i)*x(i)$	All variables reside in local memory of one processor
	SIMD	$N*(T_a$)+ T_{recon}	T_{total}, P_{im}	$for\text{-}simd(i=0;i<4;i++)$ $y=a*x$;	One processor performs the instruction fetch & decode on behalf of the participating processors
Step 3	SYNC	0	None	$synchronize(mask)$	Synchronize processors defined by *mask*. Wait until instructions on all processor are executed
	ASYNC	N (min.)	P_{im}	$barrier(mask)$	Explicit barriers only when data exchange is initiated
Step 4	MIMD	$\frac{15*N}{4}$	None	default	Wait till data is copied to/from shared memory
	Register Sharing	$\frac{7*N}{4}$	T_{reg}, T_{comm}	$share(y1,y2,y3,y4)$	Wait till data is copied to/from register files
Step 5	MIMD	$N*T_a$	None	default	Single processor computes the sum

Data partitioning distributes data to the multiple processing elements. For an application mapped to QuadroCore, the total execution time ($T_{QuadroCore}$) is expressed as follows:

$$T_{QuadroCore} = \frac{T_{compute}}{P} + T_{comm} + T_{sync}$$

Where P is the number of processors being used for the particular application. T_{sync}, T_{comm} correspond to the inter-processor synchronization and communication times. The time required to execute a given application can be expressed as a product of the total number of instructions (N_{total}), and the operating frequency or the clock period (T_{period}) :

$$\begin{aligned} T_{total} &= N_{total} * T_{period} \\ N_{total} &= (N_{dm} + N_a + N_{reg}) + N_{recon} \end{aligned}$$

Where N_{dm} is the number of instructions for data memory transactions, N_a corresponds to ALU operations, N_{reg} to the number of register operations, and N_{recon} corresponds to the number of reconfiguration instructions. Further, depending on the execution time for each of the instructions, the corresponding time may be expressed as T_a, T_{reg}, and T_{recon} respectively. Similarly, power can be expressed as a sum of the individual power components – the instruction memory (P_{im}), data memory (P_{dm}), ALU (P_a), and register files (P_{reg}). Thus, the power consumed by the QuadroCore can be modelled given below:

$$P_{QuadroCore} = 4 * (P_{im} + P_{dm} + P_a + P_{reg})$$

Where, 4 represents the four identical QuadroCore processors.

It has to be noted that the model for time and power are independent of the target implementation and thus can be extended to other instruction set architectures.

Table IV lists the proposed reconfiguration capabilities and their corresponding influence on the above mentioned time and power parameters. For instance, the choice of the type of synchronization is a trade-off between the need for inter-processor synchronization which influences T_{sync}, N_{recon}, and P_{im}. Similarly, in comparison to the MIMD mode, the SIMD mode has instruction fetches only for the 'master' processor, which results in savings in terms of P_{im} at the cost of addition time (T_{reconf}). In case of register-sharing the additional power required is only for the register file borrowed from the neighbouring processor and the power required for reconfiguring the register inputs. The configurable ALUs permit computing multiple identical operations simultaneously, affecting $T_{compute}$, T_{recon}, and P_{im}. Thus, the choice of operating mode based on application-specific characteristics, which can be altered during application description.

TABLE IV
PERFORMANCE IMPACT BASED RECONFIGURABLE MODES

Architecture	Area	Time	Power
ASYNC/SYNC	+5%	T_{sync}, N_{recon}	P_{im}
SIMD/MIMD	+13%	N_{recon}	P_{im}
Register Sharing	+5%	N_{recon}, N_{dm}	P_{dm}, P_{im}
Configurable ALUs	+2%	$T_{compute}, N_{recon}$	P_{im}

B. Application: FIR filter

As an example, consider partitioning an FIR filter onto the 4-processor QuadroCore using the proposed reconfiguration

design space. The equation for realizing an FIR filter is given as follows by Equation 2.

$$y(n) = \sum_{k=1}^{N} a(k) * x(n-k) \quad (2)$$

Where *a(k)* is the coefficient of the FIR filter at tap *k*, *x(n)* is the input, *y(n)* is the output at time *n* and N is the length. The computation can be divided into the following steps:

1) Distribute *a(k)* and *x(n-k)* among the four processors (corresponds to T_{comm})
2) *y(k) = a(k) * x(n-k)* (corresponds to $T_{compute}$)
3) Wait for end of computation, for all k elements (corresponds to T_{sync})
4) Collect *y(n)* from all the other three processors (corresponds to T_{comm})
5) $\sum_{k=1}^{N} y(k)$ (corresponds to $T_{compute}$)

Table III summarizes the costs and benefits for each of the reconfigurable modes for these steps. It has to be noted, that the choice of the mode is a compromise between the number of cycles required for execution (optimizing for time) and the corresponding impact on power (optimizing for power). The clock cycles depict the variation in execution time for each chosen mode, and the benefits reflect the corresponding savings in terms of clock cycles or power that is achieved in each of the modes. Since the choice of the modes are entirely application dependent, accordingly are the costs and benefits evaluated. E.g., in Step 2, the MIMD mode of operation is beneficial in terms of time, but results in higher power consumption in comparison to SIMD . Similarly, in Step 3, the SYNC mode results in lower synchronization time, but ASYNC is beneficial in terms of lower power on account of fewer instruction memory accesses.

C. Comparison to Amdahl's Law

Figure 7 shows the overlay of the classical Amdahl's law with increasing number of processors and the performance impact of mapping an application (FIR Filter) on QuadroCore. The speedup achieved using Amdahl's law applied to one, two, three and four processors are represented by $S1$, $S2$, $S3$ and $S4$ respectively. The speedup achieved for a 10-tap FIR filter is represented by S_{fir10} and for a 40-tap filter by S_{fir40}. The intersection of the plot S_{fir10} (and S_{fir40}) with the graphs for Amdahl's law coincide with the parallelisable fraction for one, two, three and four processors respectively.

As can be seen, with increase in the number of processors, the parallelisable fraction decrease because of the additional communication and synchronization overhead.

IV. Conclusions

A method of customizing a cluster of processors to adapt to application-specific requirements is proposed. The method reuses an existing processor architecture and permits replicating processor. These processors are further interconnected via a network-on-chip. Programming suggestions to allow application-specific partitioning with time and power as the performance trade-offs are described. Design space exploration is made possible via a proposed set of reconfigurable schemes. The possibility of introducing the proposed modes during application description, which results in a diverse power-performance trade-off, is shown. Although the addition of flexibility in the architecture results in a reduction in the maximum operating frequency, and increase area, it is compensated by the reduction in the number of clock cycles required to perform the same task. An example of an N-tap FIR filter is analysed, and is indicative of the possible use of the proposed modes. Further, comparison to Amdahl's law shows the impact of the partitioning overhead, which is dependent on application-specific characteristics.

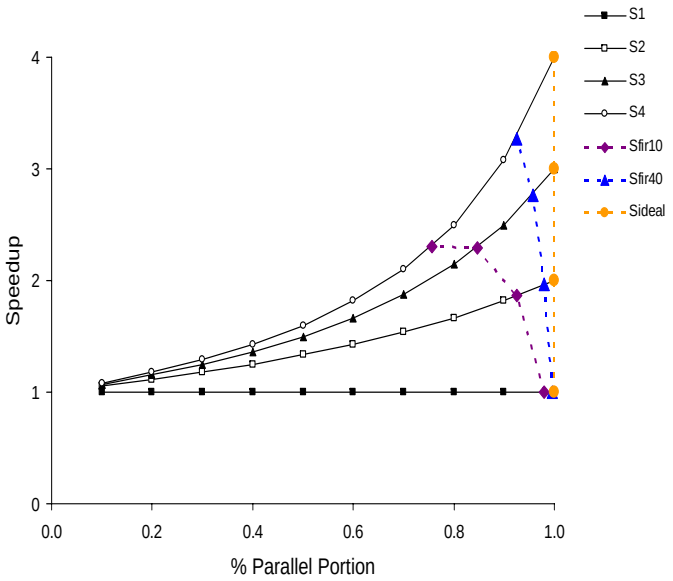

Fig. 7. Impact of Application Mapping on QuadroCore

V. Acknowledgements

This work was supported by 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 Infineon Technologies, Prof. Ramacher. The authors of this publication are fully responsible for its content.

References

[1] *ARC Configurable CPU/DSP Cores*, ARC Inc., Aug. 2008, available from http://www.arc.com.
[2] R. E. Gonzalez, "Xtensa: A configurable and extensible processor," *IEEE Micro*, vol. 20, no. 2, pp. 60–70, 2000.
[3] G. Burns, M. Jacobs, M. Lindwer, and B. Vandewiele, *Silicon Hive's Scalable and Modular Architecture Template for High-Performance Multi-Core Systems*, 2006.
[4] *Rapport' KC256, Technical Overview*, Rapport Inc., 2008, available from http://www.rapportincorporated.com.
[5] T. R. Halfhill, *Ambric's New Parallel Processor*, Oct. 2006, available from http://www.ambric.com.

[6] *Tilera64 Processor Family*, Tilera Corporation, Aug. 2007, available from http://www.tilera.com.

[7] K. Sankaralingam *et al.*, "Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture," *IEEE Micro*, vol. 23, no. 6, pp. 46–51, 2003.

[8] M. Thuresson, M. Sjalander, M. Bjork, L. Svensson, P. Larsson-Edefors, and P. Stenstrom, "Flexcore: Utilizing exposed datapath control for efficient computing," *Embedded Computer Systems: Architectures, Modeling and Simulation, 2007. IC-SAMOS 2007. International Conference on*, pp. 18–25, July 2007.

[9] M. Hussmann, M. Thies, U. Kastens, M. Purnaprajna, M. Porrmann, and U. Rueckert, "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)*, pp. 3–10, 2007.

[10] J.-C. Niemann, C. Puttmann, M. Porrmann, and U. Rueckert, "Resource efficiency of the giganetic chip multiprocessor architecture," *Journal of System Architecture*, vol. 53, no. 5-6, pp. 285–299, May 2007, special issue on Architectural premises for pervasive computing.

[11] M. Gruenewald, U. Kastens, D. K. Le, J.-C. Niemann, M. Porrmann, U. Rueckert, 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 Sep. 2004, pp. 209–214.

[12] C. Puttmann, J.-C. Niemann, M. Porrmann, and U. Ruckert, "GigaNoC - A Hierarchical Network-on-Chip for Scalable Chip-Multiprocessors," in *DSD '07: Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 495–502.

[13] J.-C. Niemann, C. Puttmann, M. Porrmann, and U. Rueckert, "GigaNetIC - a scalable embedded on-chip multiprocessor architecture for network applications," in *ARCS'06 Architecture of Computing Systems*, 13 - 16 Mar. 2006.