Energy efficiency vs. programmability trade-off: architectures and design principles

J.P Robelly, H. Seidel, K.C Chen, G. Fettweis
Dresden Silicon GmbH.
Helmholtzstrasse 18, 01069 Dresden, Germany
robelly@dresdensilicon.com

Abstract

Performance achievements on programmable architectures due to process technology are reaching their limits, since designs are becoming wire- and power-limited rather than device limited. Likewise, traditional exploitation of instruction level parallelism saturates as the conventional approach for designing wider issue machines leads to very expensive interconnections, big instruction memory footprint and high register file pressure. New architectural concepts targeted to the application domain of media processing are needed in order to push current state-of-the-art limitations. To this end, we regard media applications as a collection of tasks which consume and produce chunks of data. The exploitation of task level parallelism as well as more traditional forms of parallelism is a key issue for achieving the required amount of MOPS/Watt and MOPS/mm^2 for media applications. Tasks comprise data transfers and number crunching algorithm kernels, which are very computing-intensive yet highly predictable. Moreover, most of the data manipulated by a task is of a local nature. Granularity and characteristics of these tasks will lead us in this paper to draw conclusions about memory hierarchy, task scheduling strategies and efficient low-overhead programmable architectures for highly predictable kernel computations.

1. Introduction

Advancements in VLSI technology together with massive exploitation of parallelism at its different levels: instruction, data, pipelining and task level, have enabled programmable architectures to deliver the required processing power at reasonable levels of power consumption for the implementation of media processing, a field which has been traditionally dominated by more ASIC-like designs. The irruption of programmable architectures to a domain dominated by hardwired solutions has created a variety of new business opportunities. The pillars of these new opportunities reside on the advantages of programmability, which broadly speaking can be summarized as: Non-permanent customization and application development after fabrication, economies of scale (amortizing large, fixed design costs) and time to market (evolving requirements and standards).

Yet media applications remain an application domain that challenges a software based implementation from the power consumption and processing power perspective. Let us for instance consider the implementation of video codecs. It is reported that the new compression standard H.264 outperforms the achievable compression factor of the MPEG-2 standard by a factor of two at the same level of quality [6]. This improvement on the compression factor is achieved at expenses of the standard complexity. H.264 uses quite sophisticated compression tools like intra prediction, quartel-pel tree-structured motion estimation/compensation, and adaptive deblocking filtering. These new compression tools accounts for an increment from two to three times for the decoder and from four to five times for the encoder complexity with respect to MPEG-2 [7]. Benchmarking a C model of a H.264 baseline decoder shows that a Pentium processor running at a clock frequency of 2.5 Ghz. is required for real-time decoding of a typical D1, 25 fps, 4:2:0, 10 Mbits compressed video stream. In applications where power consumption is an issue the increment on processing power via the mere increment of the processor clock frequency is not an acceptable approach. This example stresses out the necessity for domain-specific programmable architectural concepts that massively exploit parallelism and enable scaling of the available processing power via number of gates rather than clock frequency.

For the past two decades processors have doubled in performance every 18 to 24 months without substantial changes of the underlying instruction set architecture (ISA) [5]. The driving forces for this amazing growth have been the improvements on process technology and instruc-

3-9810801-0-6/DATE06 © 2006 EDAA

tion level parallelism (ILP). However, performance achievements due to progress on process technology are reaching their limits, since chip wiring is the main limiting factor in deep-submicrometer CMOS technology. Traditional exploitation of ILP has also reached a limiting factor. Conventional methods for designing wider issue machines leads to very expensive interconnections, big instruction memory footprint and high register file pressure. It is clear that for programmable architectures to deliver the required amount of MOPS/Watt and MOPS/mm^2 new architectural concepts that take into account the characteristics of the application domain have to be developed for overcoming current state-of-the-art limitations.

Taking as a starting point the task oriented nature of media applications, we explore in this paper different concepts and design principles that enable programmable architectures to cope with the required complexity at reasonable levels of power consumption. The remainder of this paper is as follows. In section 2 we explain the concept of task and the advantages of expressing media applications in terms of a task oriented computational model. In section 3 we present a programmable architectural concept that matches the characteristics of the task computational model of section 2. The advantage of considering media applications from the task perspective is that this enables an strict separation of data transfers from kernel computations. In section 4 we present a DSP concept based on an architectural template, which matches the high-predictable, high-data locality nature of number crunching kernel computations. Finally, in section 5 we present our conclusions.

2. Media Applications: A Task Oriented Perspective

Control flow and tasks are fundamental elements of every media application. Upon this perspective a media application can be regarded as a state machine where the control flow determines which tasks are to be executed according to either the input data stream or to data generated by previous tasks. Thus, a task consumes and produce chunks of data whereas control flow defines the tasks to be executed. A task comprises data transfers and computational kernels. Moreover, each task has its own local and independent memory space which is used to store temporary values and values involved in data transfers. In this context data transfers implies a memory hierarchy, which is intended to read and write the chunks of data that are necessary for the execution of a task. Moreover, read and write data transfers take place between the independent local memory of each task and an external memory space to the task. In a general case, transfer of data can be applied to data arrays of an arbitrary dimension and with an arbitrary access scheme. It is important to point out that even though our approach has some commonalities with previous works as the ones presented in [4],[9] it has an essential difference, namely our starting point is not a data flow graph.

```
if(x==0 & y==1)
    ta_begin();
    ta_g2dfetch(stmp,x,4,y,9,width);
    ta_launch(QB01);
    ta_g2dput(dest,4,4,width);
    ta_end();
```

Figure 1. Example code showing control code, data transfers and the definition of a task called QB01

The description of media applications in terms of tasks has a lot of advantages. On the one hand, it makes compulsory the differentiation within the application between control flow, data transfers and kernel executions. As we will discuss later in this paper this differentiation is of paramount importance for an efficient implementation of media applications into programmable architectures, since each of these parts: control flow, data transfers and kernel executions, have their own characteristics that determine the efficient underlying hardware architecture to be used.

On the other hand, a task oriented description of a media application exposes concurrency at the task level and data locality. The level of concurrency at the task level is limited by the data dependencies between tasks defined by the control flow. The description of media applications by means of tasks makes compulsory to differentiate between data which is produced and consumed within a task from data that is transferred from the external memory into the memory space of each task.

2.1. Task Oriented Execution Environment

In order to allow for programming an executable version of a media application expressed in terms of tasks, we have developed a C++ framework. The purpose of this C++ framework is not only to enable the implementation of an executable version of the media application, but it is also intended to be used at later stages of the design flow in order to automatically generate code for the multicore platform introduced in section 3.

An example of the coding style for programming with this C++ framework is shown in figure 1. In this example we can observe the three main ingredients of an application, namely control flow, data transfers and kernel computation. In the example we consider the definition of a task for computing a quartel-pel motion compensation taken from the implementation of a H.264 decoder. The C++ framework

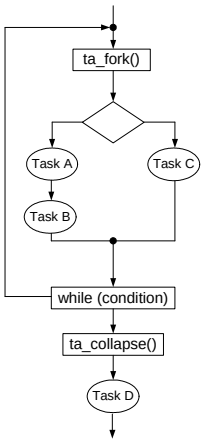

Figure 2. Example of the use of the Fork and Collapse Directives for Exposing Task Level Parallelism

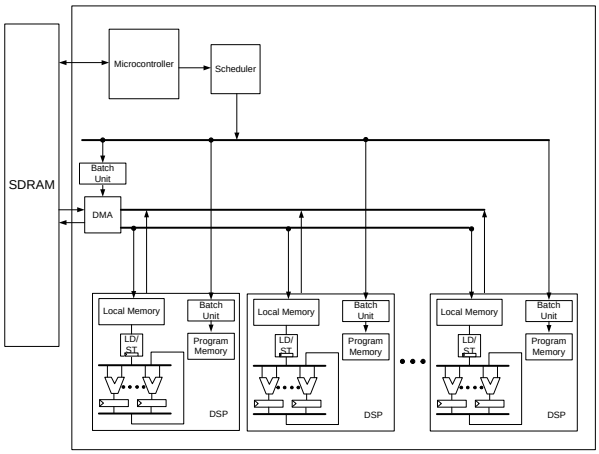

Figure 3. Block Diagram of the multicore platform

offers many directives, which allows for writing and reading an arbitrary number of parameters to and from the task. In the example, the launched task is called QB01 and it reads a 4x9 block of data. This block of data is read from the main memory at the position determined by the pointer `stmp`. The task actuates upon this block of pixels and produces via interpolation a 4x4 block of data. The code that implements the functionality of this task is implemented elsewhere and it is not shown in this example. After the task is launched the resulting block of data is written back to the main memory location defined by the pointer `dest`. More complex accesses schemes can be implemented on the data transfers. For example, in order to exploit data level parallelism within a task it is very useful for many applications to read and write a block of data with a stride or transposed access scheme.

The C++ framework also offers directives for exposing parallelism at the task level. This is illustrated in figure 2. In this figure we observe a flow diagram which main part illustrates the body of a while loop. Within the body loop we can find some control flow that determines the tasks to be executed in each loop iteration. Let us assume that in each iteration the data transfers required by the tasks to be executed do not present direct data dependencies. Taking as an example video decoding, we can imagine that in each loop iteration a new macroblock of data pixels is processed by the tasks as they are defined by the control flow. The `ta_fork()` directive allows for defining tasks which can be executed in parallel. Let us also assume that the task D of figure 2 consumes data which was previously generated within the loop body. In order to define this situation we can use the directive `ta_collapse()`. This directive means that previous tasks have to be completed in order to launch the new one. There are many types of fork and collapse directives. In order to be able to express the parallelism at different levels of granularity we are working with a hierarchical fork and collapse concept. In the next section we present a programmable multicore platform that matches this simple computational model.

3. A Programmable Multicore Platform for Media Applications

In order to map media applications described with the simple task-oriented programming model we introduced above, we propose a programmable multicore platform as the one illustrated in figure 3. We envision the programmable platform having at lest two levels (for most applications three levels) of memory hierarchy. The external memory enables mass storage and it offers a rather low communication bandwidth. Local memory offers a very high communication bandwidth and it is used for computing local data produced and consumed within a task. An additional memory with a medium bandwidth can be devised for storing data that is consumed by many tasks. In order to cope with the control intensive nature of media applications we aim at implementing an out of order execution scheme of the differen tasks. To this end, we are executing the control code in a RISC-like micro-controller. This RISC-like processor also fills execution queues consisting of tasks. In turn, this execution queues are used by an out of order execution unit that schedules data transfers and kernel computations. In this context tasks can be regarded as an instruction stream of a higher granularity, which is processed by the scheduler in order to fire data transfers and kernel computations. In order to facilitate the scheduling the in-

formation regarding the execution time of kernel computations as well as data transfers have to be known in advance. We are also using a global clock reference for the whole multicore platform so that scheduling can be done using this common time basis. In our concept, data transfers are carried out by a dedicated DMA channel. Thus execution queues contain data transfers and computational kernels, which are scheduled according to a simple *First Come First Served* approach. Computational Kernels are executed in programmable accelerators based on our novel microarchitectural template for DSPs called STA (Synchronous Transfer Architecture), which are specially designed to exploit data, instruction level parallelism and the predictable nature of number crunching algorithms. This concept is explained in section 4 in more detail. Note that in our concept every DSP core is able to execute any tasks so that the scheduler can choose any core for executing kernels computation according to DSP load. The scheduling unit of the platform annotates into the batch control units a time stamp when data transfers for the DMA unit and kernel computations for DSP units are to be executed.

It is important to point out that it is not our ambition to create a new parallel computation model for media applications as in [9]. These works aim at a programming model that is general enough for allowing code synthesis to a variety of parallel architectures. In our approach, we start from the architectural concept presented in this section in mind and then we developed a simple computational model that matches this platform.

The concept of out of order execution is a well established technique within the processor architecture community. We are applying the same ideas. However, it is our intention to apply these ideas at the task level rather than at the instruction level. It is our conviction that exploiting dynamic scheduling at higher levels of granularity is a more powerful approach. Above all if we consider that due to the predictable nature of computational kernels, these do not take advantage of dynamic scheduling. Computational kernels call for a static compilation technique for exploiting instruction, data and pipeline parallelism. Since this can ba calculated by a compiler in compilation time, we are able to build low overhead DSP cores with a lot of ALUs for computation and very low control overhead. This leads to efficient low-power low-overhead high-performance programmable architectures for kernel computation as it is explained in the next section.

4. Efficient Low-Overhead Programmable Architectures

Signal processing applications are characterized by number crunching algorithm kernels at which an application conventionally spends most of its time. Although these kernels are computing intensive, they are highly predictable, and often they do not take advantage of sophisticated features like branch prediction or dynamic scheduling mechanisms. Such kernels are amenable to static scheduling and software pipelining by means of compiler backend techniques, thus shifting the complexity from the hardware to the compiler. A key issue in DSP design for the coming years will be the application of more sophisticated compiler approaches in order to minimize the hardware overhead of the architecture. The underlying objective is to utilize the available die size for efficiently placing very fast data paths.

Another paramount characteristic of kernel computations is the data locality these algorithms exhibit. Corporaal [3] concluded after analyzing the benchmark Stanford that 40 % of read accesses to values are performed in the same cycle as these values are defined. Furthermore, he points out that 45% of all defined values are dead after being used at lifetime zero. These observations make clear that there is potential for a considerable reduction of the register file traffic by means of bypassing values directly between functional units. A fundamental conclusion is that high performance DSP architectures have to exhibit producer-consumer locality that can be exploited to reduce data transfer traffic. Moreover, it is very well understood in the processor design community that it is data transfers and not performing arithmetic which governs power consumption. Thus, a key question for DSP designers is how to create an architecture that exposes and exploits data locality, thereby avoiding power dissipation that would otherwise solely result from moving operands.

One of the driving forces for the increase of processor performance has been ILP. However, the way how ILP has been conventionally implemented is becoming very expensive and does not scale for high concurrency machines. Efficient exploitation of ILP requires large fully interconnected register files. Moreover, in order to overcome data hazards, bypass networks have been added to these architectures, which together with hardware support enable the utilization of data forwarding techniques. All these trends in processor architecture have led to a situation where the amount of available concurrency is limited by the associated hardware cost. From this discussion it seems quite clear that in order to push ILP to even higher levels of concurrency a new architectural concept needs to be envisioned, which scales in a better way with the degree of ILP.

Single instruction multiple data (SIMD) vector signal processors offer the potential to increase the data transfer rates between memory and computational resources, since data vectors residing on memory are accessed and processed in a parallel fashion. This enables the achievement of speed-up gains in the implementation of DSP algorithms into these processor architectures. While SIMD has its origins in supercomputers, the advancements of VLSI tech-

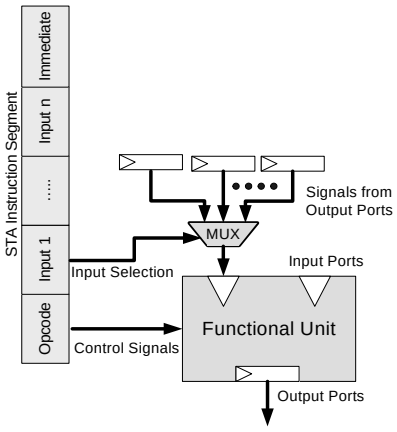

Figure 4. STA Architectural Template

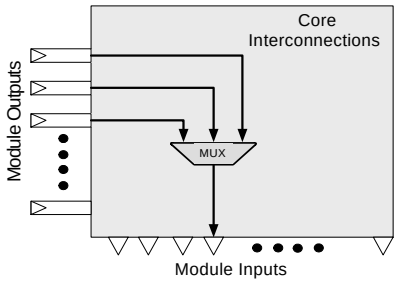

Figure 5. STA Bypass Network

nology have enabled SIMD to make its way to the world of embedded real-time signal processing. This renewed attention on SIMD processors has been driven by the demand for low-power and applications with ever increasing algorithm complexity, where a programmable device is favored over a fixed wired solution. Vector processing has been proven to be beneficial for media applications like video, image and audio processing, where algorithms are applied to chunks of data like subblocks, macroblocks, slices, and frames. Finally, it is worth pointing out that SIMD vector processing is consistent with the paradigm of low overhead architectures: a chip can be filled with arithmetical units and the overhead cost remains constant and is amortized over the different parallel units.

4.1 Putting it all Together

Our DSP architecture is based on basic modules of the form shown in figure 4. Such a basic module has an arbitrary number of input and output ports. In our concept, the output of each module is a register. Input and output ports can deal with a certain data type, e.g. boolean, 16-bit integer, 32-bit floating point, vectors of 16-bit integer, vectors of 32-bit floating point, etc. Basic modules implement some functionality and a DSP core is build up from a set of such basic modules. Ports of the same data type are connected with each other through a bypass network formed by multiplexers. This is sketched in figure 5. Both the functionality of basic modules and the multiplexers are explicitly controlled by processor instructions. At each cycle the instruction configures the bypass network and the functionality of the basic modules.

As a result, the whole system forms a synchronous network, which at each clock cycle consumes and produces some data. The produced data will in turn be consumed by other basic modules in the next cycles. This synchronous transfer of data has given the architecture its name: synchronous architecture (STA). Basic modules can be data paths or memory blocks. Memory blocks can be either register files or memories. There is a register file for each data type available in the processor. However, the register files have only a limited number of input and output ports. The STA architecture offers a high degree of data locality: Data that is produced in the current cycle can be directly routed to other processing units in the following cycles without going through the register file or memory. This not only speeds up computations, but also lowers power consumption and register file pressure. The STA architecture also supports data and instruction level parallelism. In fact, SIMD vector data parallelism is implemented by letting input ports, output ports and data paths deal with vector data types. At the same time, instruction level parallelism is supported since at each cycle a wide instruction controls each basic module as well as the multiplexing network. Two important issues regarding the architecture have to be taken into account. On the one hand, for large STA systems the bypass network becomes a critical part of the design. On the other hand, a wide instruction memory is needed. The complexity of the bypass network is alleviated by reducing the number of connections between ports. An obvious strategy for this is to determine those connections which allow for reusing values that present a high data locality. This is a viable approach, since applications are known at design time of the processor and thus, a customization of the bypass network can be carried out in order to meet die size and power consumption requirements. For those connections which are not frequently reused, a connection via the register file suffices. To control instruction memory footprint we are applying code compression techniques.

4.2 Automatic Integrated Design Flow

The simplicity and modularity of the STA concept enables the automatic generation of RTL and simulation models of processor cores from a machine description. Processor cores with different characteristics, e.g. size of register file, memory capacity, bypass network, functional units,

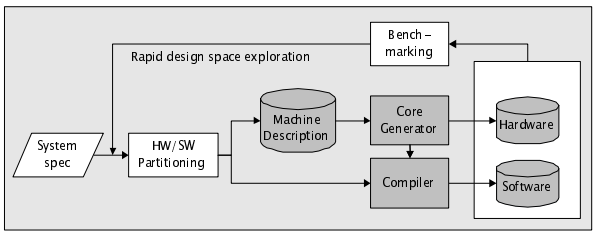

Figure 6. Automated Design Flow for DSP Cores

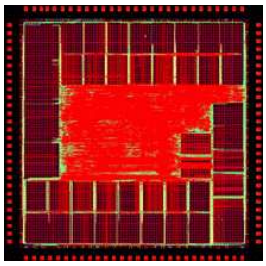

Figure 7. Die Photo of the SAMIRA DSP

data types and degree of SIMD-vector parallelism are automatically generated from a machine description file. This is illustrated in figure 6. In this figure we can observe that this design flow enables a design space exploration, where most of the design tasks are completely automated. We are also working on a retargetable compiler backend in order to allow for automatic code generation for these DSP cores.

4.3 Samira: a First Silicon Prototype

In order to proof our DSP architectural concept we have taped out a generic DSP core with the following characteristics: 17 single-precision floating point units, 8x SIMD, 480k NAND gates, 2.4mm^2 logic area on UMC 0.13μm, 2.9 Mbit on-chip SRAM and clock frequency 212 MHz.

The die photo of the DSP core is presented in figure 7. The prototype DSP outperforms well-known competitors. A 256-point complex FFT consumes 1889 cycles, which corresponds to an execution time of 9.45 μs when clocked at 200 MHz. This nicely compares to Analog Devices' ADSP-21161N (26 μs at 100 MHz), Texas Instruments' TMS320C6713 (16 μs at 225 MHz), and Renesas' SH7750R (24 μs at 200 MHz). The true potential, however, stems from the processor's power efficiency. SAMIRA consumes 4.0 Watt-μs per 256-point complex FFT. By contrast, the ADSP-21161N consumes some 23 Watt-μs, and the TMS320C6713 calls for 12 Watt-μs. SAMIRA's closest rival among these processors is Renesas' SH7750R, which requires twice the energy per FFT (8 Watt-μs) as compared to SAMIRA. In fact, SAMIRA reaches the power efficiency of fixed point DSPs. For example, it outperforms Texas Instruments' TMS320VC5510, which exhibits a significantly higher execution time per FFT as well as a higher energy consumption per FFT than SAMIRA (4.3 vs. 4.0 Watt-μs)[1]. It is worth noting that SAMIRA reaches this performance without featuring any support for FFT specific instructions, since it was devised as a general purpose STA processor.

[1]Based on figures reported in [2][1][8]. Execution times and energy consumption for the ADSP-21161N, the TMS320C6713 and the Renesas SH7750R from [2]. Data for the TS203S compiled from clock cycles reported for FFTs in [1]. Data for TMS320VC5510 from [8].

5. Conclusions

Starting from a task oriented prospective of media applications we have presented in this paper a multicore DSP platform based on a simple task-based computational model. The most important characteristic of this computational model is the strict separation of control code, data transfers and kernel computations. Control code is executed in a microcontroller, data transfers are executed using DMA channels and kernel computations are implemented in low-overhead high-performance DSP cores. This led to the idea of an architectural template that exploits the characteristics of these kernel computations. We have shown that low power and low gate count can be achieved by means of this methodology.

References

[1] AnalogDevices. Writing efficient floating-point ffts for adsp-ts201 tigersharc processors. March 2004.

[2] BDTI. A bdti analysis of the texas instruments tms320c67x. *available from www.bdti.com/products/*, 2003.

[3] H. Corporaal. *Microprocessor Architectures: from VLIW to TTA*. John Wiley and Sons, 1997.

[4] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. *IEEE Computer*, pages 54–62, aug 2003.

[5] M. H. M. and W. Dally. How scaling will change processor architecture. *In Proc. of ISSCC 2004*.

[6] J. Ostermann, J. Bormans, P. List, D. Marpe, M. Narroschke, F. Pereira, T. Stockhammer, and T. Wedi. Video coding with h.264/avc: Tools, performance and complexity. *IEEE Circuits and Systems Magazine*, pages 7–27, January 2004.

[7] R. Schaefer, T. Wiegand, and H. Schwarz. The emerging h.264/avc standard. *EBU Technical Review*, (293), January 2005.

[8] TexasInstruments. C55x dsp benchmarks. *available from http://dspvillage.ti.com/*.

[9] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. *Proc. of the Int. Conf. on Compiler Construction 2002*, 2002.