Czech Technical University in Prague
Faculty of Information Technology
Department of Digital Design

Programmable and Customizable Hardware Accelerators for Self-adaptive Virtual Processors in FPGA

by

Jaroslav Sýkora

Doctoral Degree Study Program: Informatics

Dissertation thesis statement for obtaining
the academic title of “Doctor” abbreviated to “Ph.D.”

Prague, December 2013

The dissertation thesis was written during part-time doctoral study at the Department of Digital Design, Faculty of Information Technology of the Czech Technical University in Prague and full-time employment at the Department of Signal Processing of the Institute of Information Theory and Automation of the ASCR in Prague, Czech Republic.

Ph.D. Candidate: Ing. Jaroslav Sýkora
Department of Digital Design
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9, 160 00 Prague, Czech Republic
sykorj11@fit.cvut.cz

Supervisor: Ing. Martin Daněk, Ph.D.
Signal Processing, ÚTIA AV ČR, v.v.i.;
Computer Systems Group, TU Darmstadt
xdanek@email.cz

Reviewers: _____

_____

_____

The dissertation thesis statement was distributed on .....

The defence of the dissertation thesis will be held before the Committee for the presentation and defence of the dissertation thesis in the doctoral degree study program Informatics
on ..... at ..... in the meeting room No. ..... .

In accordance with Para. 9 of Art. 35 of the Study and Examination Code for Students of the CTU, those who are interested may look into the dissertation thesis and make notes, copies, or duplicates from it at their own expense. A copy of the dissertation thesis is available in the Science and Research Office of the Faculty of Information Technology, room No. 308.

Electronic version of the thesis is available at: `http://www.jsykora.info/thesis`

.....

Chairman of the Committee for the presentation and defence of the dissertation thesis
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9, 160 00 Prague, Czech Republic

Contents

1 Introduction and Motivation

1.1 The Need for Data-Driven Scheduling

A schedule of operations in a machine may be *driven* by the flow of *instructions* or the flow of *data*. Traditional von Neumann computer architectures are driven by instructions because operations are started in response to the arrival of explicit instructions. In *data-driven* (or *dataflow*) architectures, on the other hand, operations are started in response to the availability of input data.

Data-driven scheduling is needed when execution latencies of operations are *unpredictable*. In computers the usual example is accessing DRAM chips. In DRAMs the practical unpredictability of read/write access latencies stems from periodical refreshing of memory capacitors that blocks the memory for a moment, and from the state-full memory interface (access to a closed row takes longer than to an open one, reversing access direction takes time, etc.).

On the other hand, if all latencies are statically known (*predictable*) in advance during compile time (or system construction) then the compiler could statically reorder instructions and construct a perfect schedule. This holds for arbitrary long latencies as long as they can be determined in advance. Small perturbations in execution latencies might be tolerated using statistical approaches such as processor caches. In that case instruction-driven machines are typically more efficient because of their simplicity.

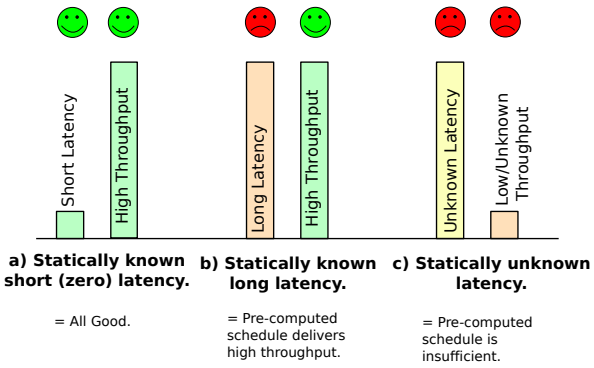

Figure 1: Statically predictable vs. unpredictable latencies.

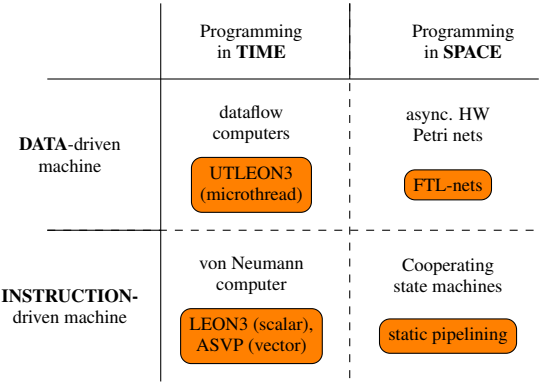

Figure 2: Instruction- and data-driven machines programmed in time and space.

1.2 The Need for Convergence of Procedural and Structural Programming

The classification of computing machines as *instruction-driven* or *data-driven* is one possible point of view. An orthogonal classification is whether the system is programmed *procedurally in time* or *structurally in space*.

Figure 2 shows a matrix that puts the instruction- and data-driven machines in relation with programming in time and space. Classical von Neumann computers are instruction-driven and programmed in time. Dataflow computers with stored program are data-driven and also programmed in time. Single-core out-of-order execution processors are hybrids programmed in time; the processor ISA (and the pipeline front-end) are based on the instruction-driven paradigm, the pipeline back-end and the execution core is data-driven. Many-core in-order execution processors are instruction-driven, with the fine-grain programming in individual cores done in time and the coarse-grain application mapping done in space. Reconfigurable hardware is programmed in space. The application designer can choose if the computation is instruction-driven or data-driven by coding the appropriate hardware structures from the primitives provided in the architecture.

In the past when frequency scaling in new silicon generations was still a norm the processor hardware was built to maximise sequential performance. Single-core superscalar processors were programmed *procedurally in time*. When frequency scaling had ended and many-core scaling had begun, the old sequential software programming models were no longer viable for new systems with more than a couple of processor cores. New programming models had to be invented to efficiently program many-core processor hardware with complex structure. One such new model is the

Self-adaptive Virtual Processor (SVP, [8, 9, 10, 12]). Besides the efficient utilization of many-cores the SVP model must also overcome unpredictable latencies in off-chip memories (DRAMs) and in on-chip shared interconnect. Hence, the SVP model is *data-driven* (dynamically scheduled) and it contains some aspects of the *structural programming in space* (e.g. delegation of threads to places).

Previous work in SVP envisioned large-scale homogeneous many-core chips because it assumed that low clock-frequency silicon is easily scalable in space [11]. However, the contemporary and future power constraints will favour *heterogeneous* many-cores in which parts of the chip could be powered off if the given special function is not used at the moment [2]. The thermal design power of a chip could be so low that not all cores may be powered up simultaneously.

Specialized hardware can deliver orders of magnitude higher performance. Computing in reconfigurable arrays allows to tap the performance of specialized solutions without the cost of manufacturing a single-purpose silicon hardware. A typical approach is to combine a general-purpose many-core processor cluster with special-purpose cores to accelerate selected key applications. In this work the use of (embedded) reconfigurable arrays in place of (some of) the special-purpose cores is postulated. The resulting system is programmed procedurally in individual general-purpose cores, and structurally among the general-purpose cores and in the reconfigurable arrays. The system is data-driven to overcome unpredictable latencies in off-chip memories, on-chip shared interconnect, and in interconnect in the reconfigurable arrays.

2 Problem Statement and Solutions

The coverage of the thesis is depicted in Figure 3 as a 'map' that puts thesis objectives and contributions in relations. In the centre there are six main topics that determine the research field: *heterogeneity, programming in space, multithreading, customized hardware, reconfigurable arrays, and the ease of programming*. This field is studied here through the optics of **two computing architectures**:

A1. Statically scheduled instruction-driven architecture for custom computing on FPGA, called ASVP and described in [1].

A2. Dynamically scheduled data-driven architecture for general-purpose computing using microthreading, described in [8, 12] and implemented in [X.12].

The **main objectives** of the thesis are:

O1. To design and develop a new implementation of the architecture **A1**, to characterise its hardware and software performance. Based on the characterization

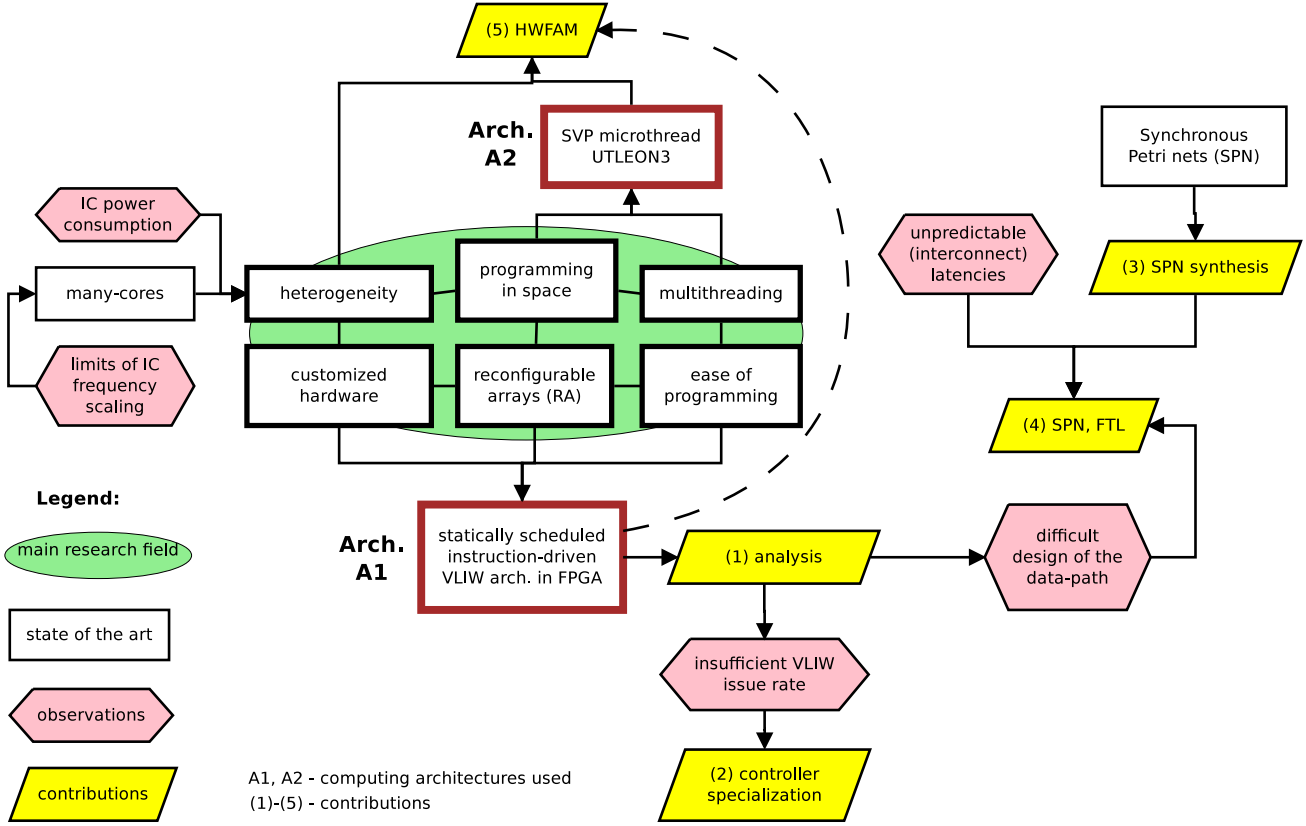

Figure 3: Map of the thesis coverage.

new features in the architecture **A1** are proposed, implemented and evaluated that improve the execution efficiency.

O2. To propose and develop a method for dataflow-oriented specification of problem-dependent (specialized) compute pipelines. Using a newly implemented software tool the method is tested by generating specialized compute units that can be used (not only) in the architecture **A1**.

O3. To propose and develop a method for integrating simple statically-scheduled architectures such as **A1** into the advanced multi-core data-driven processor architecture **A2**.

O4. To compare the performance of the architectures **A1** (ASVP), **A2** (microthreading) with the method developed in **O2** and with usual industry-standard approaches (manually coded HDL, scalar processor).

2.1 Objective O1: Evaluation and Improvements in ASVP

2.1.1 The Architecture A1: ASVP

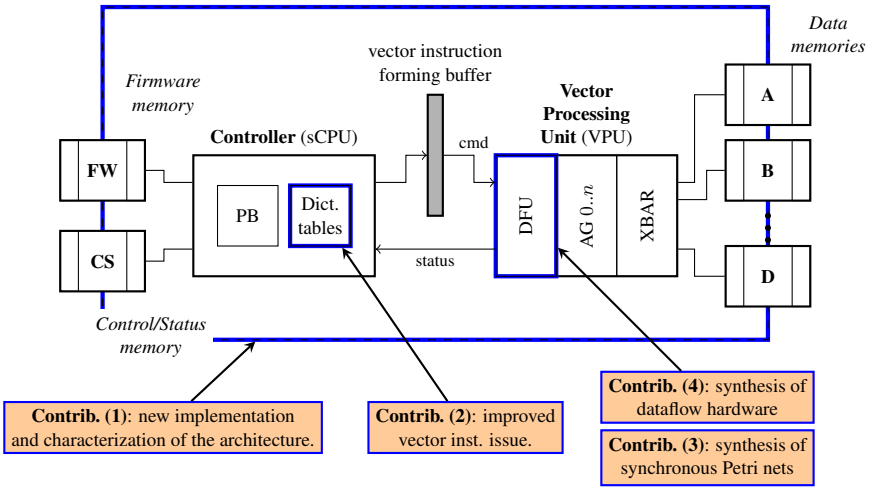

Figure 4: Internal organization of the ASVP core. *Components:* sCPU = 8-bit Xilinx PicoBlaze (PB) controller; DFU = Domain Function Unit; AG = Address Generators; XBAR = Crossbar.

The conceptual block diagram of the architecture **A1**: the *Application-Specific Vector Processor* core (ASVP) is shown in Figure 4. The ASVP is an accelerator

for computing tasks in the *Digital Signal Processing* (DSP) domain. Typical tasks in the DSP domain are data-parallel and operating on infinite input streams. Relevant parts of input data are internally stored in local data memories (A, B, ... in the picture), implemented using BlockRAMs on FPGA. Chunks of data are processed in the *Vector Processing Unit* (VPU), intermediate results are stored back in the local memories. The VPU is composed of a crossbar (XBAR), address generators (AG) and the *Domain Function Unit* (DFU). The DFU implements a set of basic vector operations, such as vector addition, multiplication, dot product etc. The set of operations is meant to be *customizable* for a given application.

The DFU and AG modules are controlled by *vector instructions* created and issued from an embedded microcontroller (sCPU). The v-instructions are created in a *forming buffer* at the interface between the sCPU and VPU using standard I/O instructions of the microcontroller. Once a v-instruction is issued to the VPU the forming buffer is free to receive the next vector instruction, while the issued one is still running. In the optimal case the preparation of a v-instruction and the execution of another one in VPU are overlapped.

2.1.2 Execution Efficiency of the ASVP

In the thesis the performance of the ASVP platform is characterised. The hardware is implemented in several FPGA generations and the resource usage and cycle time data are presented. Practical real-use performance is evaluated by the following three application kernels (firmwares): *matrix multiplication*, *Mandelbrot fractal set* computation, and *image segmentation* (foreground/background detection in video).

The authors who defined the ASVP platform in [1] expected that the *v*-instruction issue and execution will be overlapped, hence the compute hardware (VPU) will be kept busy all the time. The evaluation presented in the thesis shows this assumption to be incorrect. For instance, in a configuration in Spartan 6 FPGA the *image segmentation* kernel utilizes the compute hardware (VPU) only 48% of the total running time.

The problem is caused by a relatively low performance of the embedded microcontroller (sCPU) used in ASVP. Simulation results indicate that doubling the performance of the microcontroller would help, but the VPU would still be idle significant amounts of time. The solution proposed and implemented in the thesis is based on an observation that some particular fragments of v-instruction words are often repeated (reused) over the kernel execution time.

As shown in Figure 5, each v-instruction word is composed of the op-code ‘oper’ and of several fields that configure architectural address generators (dst, src1-3, idx). Typically the AG configuration data is wide, up to 46 bits as shown in the figure. In practical algorithms the AG config. data is often reused over several v-instructions, although in different positions within the v-instructions. Hence it is advantageous to

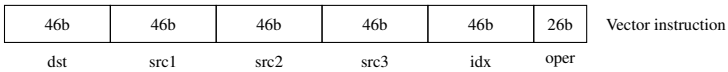

Figure 5: Structure of a vector instruction (*v-instruction*) in an ASVP with $5\times$AG.

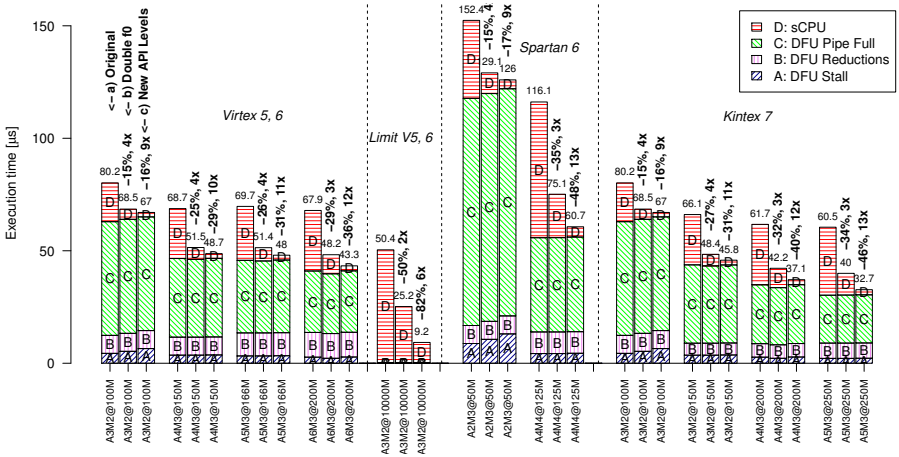

Figure 6: Execution times of the *image segmentation* kernel (lower is better). Horizontal axis: hardware implementation configurations. Compared approaches: (a) original; (b) $2\times$ faster sCPU (alternative); (c) using the dictionary tables (proposed).

store these *fragments* in a separate dictionary table that serves as a software-controlled cache. When a fragment is needed in an v-instruction a single instruction of the sCPU controller is able to pull it from the dictionary into the preparatory buffer.

Experiments show that the controller overhead, i.e. the time *not* overlapped by a useful computation in VPU, is reduced typically $9\times$ (max. $13\times$). The absolute execution time is reduced by 10% (min 6%, max 15%, speed-up $1.1\times$) for the Mandelbrot application, and 30% (min 16%, max 48%, speed-up $1.3\times$) for the ImgSeg application (Figure 6).

The resource cost of the additional hardware is around 100 slices in Virtex 6 and Spartan 6, and 55 slices in Kintex 7. This is around 7% and 3.3%, respectively, of the total area of the core in FPGA. The new hardware is speed-up/area efficient.

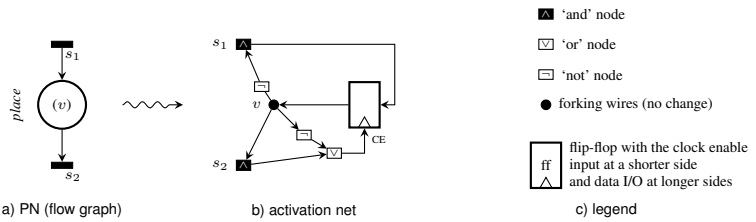

Figure 7: A single SPN *place* and the associated Activation Net (AN) fragment.

2.2 Objective O2: Dataflow-Oriented Specification of Specialized Compute Units

The *Domain Function Unit* (DFU) in ASVP implements a customizable set of vector operations. The previous work [1] did not specify any structured means of how the DFU could be built. Author's experience showed that the ad-hoc manual approach is error-prone and cumbersome. Hence the motivation for a dataflow-oriented structured specification method of custom compute pipelines (not only) for the DFU.

The solution is based on an innovative method of hardware synthesis from *Synchronous Petri nets* (SPN). The first sub-problem is an efficient synthesis of SPN to hardware; the method can be used for controller synthesis. The second sub-problem is an extension of SPN to enable dataflow processing that is not throttled by unavailability of bubbles in the pipeline.

2.2.1 Sub-problem: Controller Synthesis Using Synchronous Petri Nets

Compared to the usual well-known Petri nets, *Synchronous Petri nets* (SPN) have a notion of *global clock* in the environment (SynPN in [6], SIP-net in [3]). All firing events are synchronised to the global clock ticks. For an external observer firing events that occur during the same clock-tick are truly simultaneous.

Activation net (AN) is a new tool I have devised to enable the translation of the SPN specification into synchronous hardware. The AN computes which *transitions* in the corresponding SPN fire in a clock-tick. The net is a finite static graph of logic operators (and: $\wedge$ or: $\vee$, not: $\neg$) in the standard Boolean set $\mathbb{B} = \{0, 1\}$. There is also a special *flip-flop* (ff) operator that represents a clock-synchronous register.

Activation nets are constructed by expansion of SPN components (i.e., places, transitions) using pre-defined AN fragments (patterns). Figure 7.a shows a *place* in an SPN net, with one input (s_1) and one output transition (s_2). The symbol '(v)', $v \in \mathbb{B}$, inside the *place* symbolizes dynamic marking of the place. When $v = 0$ the place is empty, when $v = 1$ the place holds a token. *Transitions (s_1, s_2) are mapped to* wired-and *nodes which are depicted as '* ∧ *'.*

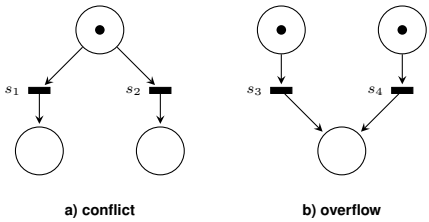

Figure 8: Two undesirable situations in synchronous Petri nets [3].

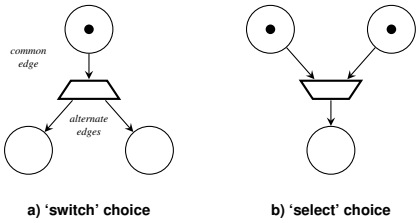

Figure 9: The 'switch' and 'select' choice transitions deterministically resolve conflicts.

Figure 8 shows two examples of Petri nets where transitions marked s_1 and s_2 cannot fire simultaneously. In picture (a) there is an alternate choice between s_1 and s_2 firing, in (b) there is similar choice between s_3 and s_4. In the standard, *asynchronous* Petri nets transition firings are individual events, each separate from others. When several transitions happen to be enabled at the same time they will *not* fire simultaneously. Rather, the firings are non-deterministically serialised, perhaps based on probabilities. In Figure 8 one of the transitions s_1 or s_2 (or s_3/s_4) would fire earlier than the other, thus resolving the indeterminism.

In *synchronous* Petri nets the situation is more severe. All transitions that are enabled always fire at the next clock tick; if the transition should not fire it shall not be enabled in the first place! The two situations in Figure 8 called 'conflict' and 'overflow' should be resolved by deterministically disabling one of the transitions, perhaps via a priority scheme. In previous work [3] the authors dismiss networks that may evolve conflicts or overflows as incorrectly formed.

My solution is to define a completely new kind of a transition node called a '*choice*'. There are two types of the *choice* transitions, as depicted in Figure 9:

1. the '**switch**' choice has a single common input arc and multiple alternate outputs (think: *switch to mult. places*), and

2. the '**select**' choice has a single common output arc and multiple alternate inputs (think: *select from mult. places*).

The function of *choice* transitions is conceptually carried out in null time (it is a combinatorial function in hardware). The *switch* (*select*) choice transition guarantees that *at most one* output (input) alternate arc is enabled at any time, iff the single common input (output) arc is enabled as well. This is in contrast to the normal SPN transition node that guarantees *all* its arcs being disabled or enabled simultaneously.

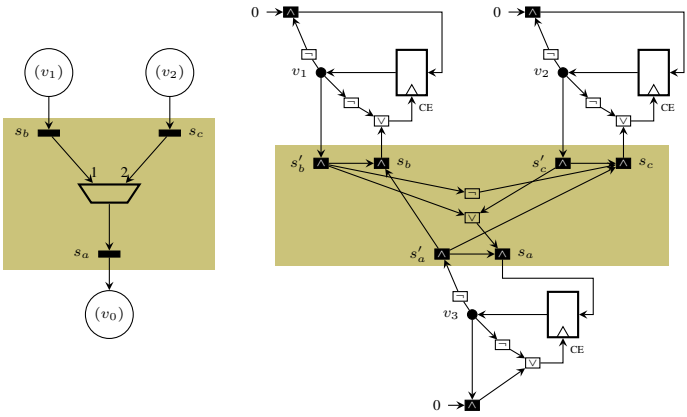

Figure 10: A simple SPN with the *select* choice, and its AN.

Figure 10 shows a simple SPN with 3 *places* and the *select* choice node. It explicitly shows the transitions s_a, s_b and s_c for clarity. The AN on the right is constructed as follows: First, the AN fragments for the three *places* are instantiated. Second, the relevant boundary *wands* of the fragments are split. Third, the AN fragment for the *choice* transition is instantiated, using the newly split *wands*.

Synthesis of Synchronous Petri nets into hardware (**Contrib. (3)**) is demonstrated on an example of a transputer serial-link controller. This example was selected because it is relatively simple, yet non-trivial. The same example was used in [3]: it allows me to contrast my approach to the previous work. The key improvement of the new approach are the *alternate choice* nodes that allow convenient specification of priorities among multiple alternate arcs. The use of *switch* and *select choices* instead of unstructured *ad-hoc* boolean guards on transition nodes improves readability of the graphical representation because the priorities among potentially competing arcs are explicitly declared.

2.2.2 Sub-problem: Dataflow Hardware Synthesis from Petri Nets

The fundamental difference between Petri nets (with capacity=1 *places*) and the standard clock-synchronous RTL design as used in hardware is the problem called a '*bubble wall*' (Figure 11). In both cases the token moves are synchronized to a common global clock. In RTL pipeline all data tokens in registers are moved forward simultaneously on each clock tick. In Petri net a data token moves forward only when the next place is empty. The consequence is a lower throughput of Petri nets. Throughput is an average number of tokens transported over a link (wire, edge) per

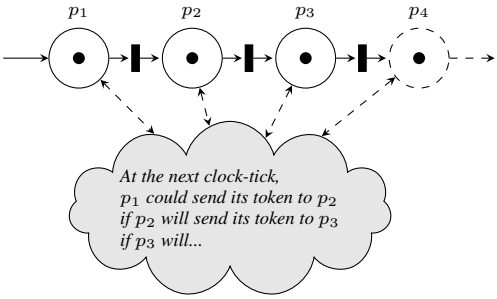

Figure 11: Synchronous flow of tokens requires global information about the state of Petri net.

clock-tick. Using RTL registers the throughput is determined only by the amount of tokens (valid data values) in pipeline. Higher number of tokens means higher throughput. In Petri net tokens can jump to the next place only when the place is unoccupied; hence, empty places (bubbles) are equally important for throughput as tokens.

The 'bubble wall' is caused by the *principle of locality* of Petri nets. The locality principle says that the behaviour of transitions depends *exclusively* on all input and output places and on the transition itself. Each transition 'sees' the state only of the places it directly connects to. If capacity=1 places are used a transition fires if the source place has a token and the destination place has a bubble. This limits the firing rate of each transition to 0.5 firings per clock-tick in synchronous SPN.

To overcome the 'bubble wall' Petri nets must be taught to 'think globally'. The critical situation just happens when two places connected by a transition both hold a token. The token in the place p_1 in Figure 11 could be transported to the place p_2 at the next clock-tick *if and only if* the token in p_2 is simultaneously transported to p_3 which in turn requires that its token is moved to p_4, and so on.

Processing of global information to enable synchronous token flows can be realized in activation nets. Figure 12 shows the conceptual program: I started with the existing *synchronous Petri nets* (SPN) semantics. Based on that the *activation nets* (AN) were defined, a graph that computes when transitions fire and tokens move in SPN. It turns out now that a small modification in the AN fragment of the *place* node allows to overcome the bubble wall in SPN. As this is a radical departure from Petri nets towards the RTL hardware the new approach is called the *Flow Transfer Level* (FTL).

The new component in FTL that replaces a *place* is called *cell*. Figure 13 shows the AN fragments of both components. Compared to the standard *place* component the *cell* allows for an additional transition state called *flow-through*. In the flow-

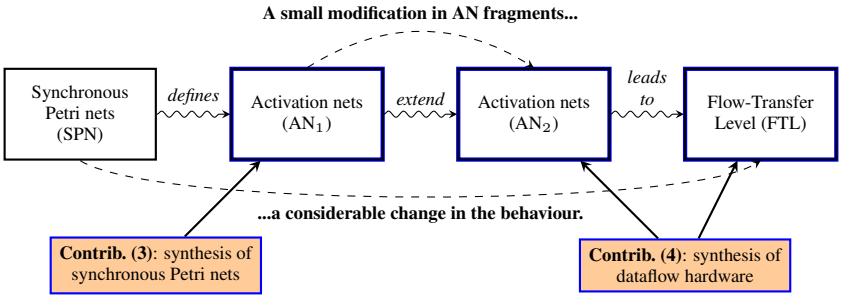

Figure 12: The thought process starts at SPN.

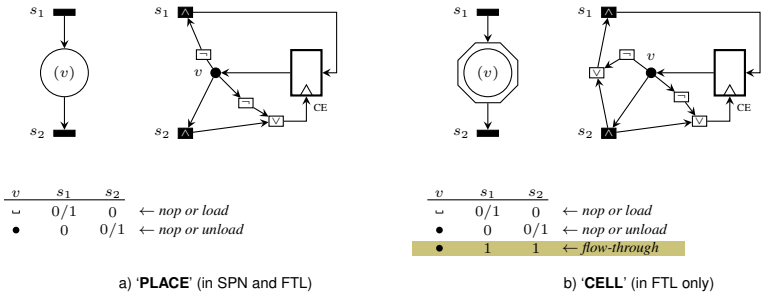

Figure 13: Definitions of the standard *place* and the new *cell* nodes in SPN and FTL.

through state the *cell* holds a token and both the input and output transitions are enabled; at the next clock-tick the existing token will be unloaded and a new token loaded simultaneously. In Petri net establishing a flow-through state in a node requires global information about the markings of many *places* and it depends on the topology of the net. In activation nets the flow-through state is computed locally by the AN fragment.

Dataflow synthesis using the FTL-net method (**Contrib. (4)**) is demonstrated on a design of a single-purpose domain function unit (DFU) for the ASVP core. The new DFU implements only a single operation: a pipelined computation of pixels in the Mandelbrot fractal set (MF-set). The solution is contrasted with an implementation in ASVP that uses general-purpose DFU. Synthesised hardware of both solutions occupies practically the same area in FPGA, but the FTL-net version is 22× faster. The price paid is the deep specialization of the DFU which precludes reconfiguration via firmware upload in ASVP.

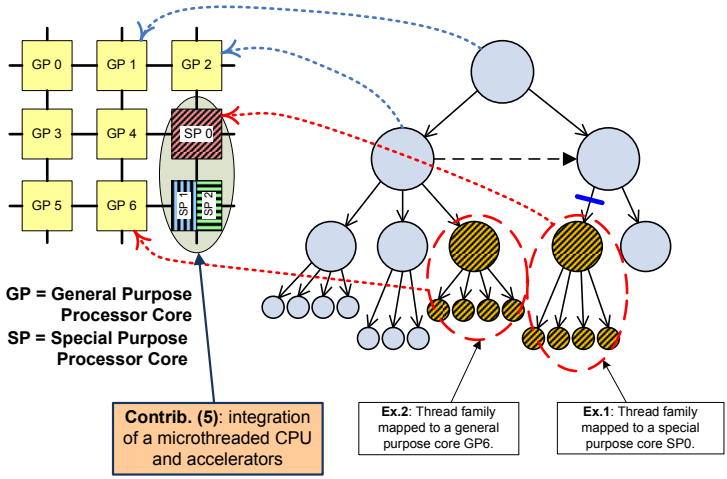

Figure 14: Thread mapping in a heterogeneous microgrid.

2.3 Objective O3: Integration of Custom Accelerators in Data-Driven Multicore Processor Systems

In the objective **O1** dataflow scheduling is practically limited to ordering complete kernels executed in the ASVP architecture. The objective **O2** dealt with specifications of dataflow computation at the level of individual circuits and gates. Finally, in objective **O3** dataflow scheduling in a processor is discussed and a method is proposed for connecting architectures scheduled at the task level (such as in **A1**) to advanced multi-core data-driven processor architectures represented by **A2**.

2.3.1 Self-adaptive Virtual Processor

Dynamic scheduling at the fine-grained instruction level in von Neumann computers is not scalable as it is limited by the complexity of reorder buffers. Attempts are made to perform scheduling at coarse granularities but without sacrificing performance. In *microthreading*, discussed in the thesis, the unpredictable latencies also happen at load/store instructions but the scheduling is performed at the microthread level. Individual microthreads are not correlated and the programming model–the *Self-adaptive Virtual Processor* (SVP, [10, 8, 9])–encourages creating many of them.

The SVP expresses fine-grained concurrency by composition of *microthreads* (Figure 14, right). A microthread[1] comprises only a few processor instructions which

[1]The terms 'thread' and 'microthread' are used interchangeably throughout the text.

typically implement a body of a loop and share only a small portion of the processor (ISA) register file. A *family of threads* is an ordered set of threads, all created by one processor instruction, called `create`. The ordering is defined by a sequence of integer index values specified during *create event* as a {start, step, limit} triple. A family of threads is akin to a parallel *for*-loop, but the threads can execute in parallel by default.

The SVP model is oblivious to the number of processing cores in the system. A family of threads can be dynamically scheduled entirely on a single processor or distributed across multiple cores as shown in Figure 14. In the latter case each thread executes in one given core, but different threads can be scheduled in different cores. Threads of the family are scheduled concurrently within their local cluster. Previous work on SVP assumed that cores within clusters are homogeneous (identical) in their ISA to enable the transparent distribution of threads among cores. In my work this view is extended to *heterogeneous* clusters composed of both the *general-purpose* and *specialized processors* (= accelerators), as illustrated in Figure 14 on the left.

The use of accelerators in multi-threading or multi-core environments is complicated by the need for locking and synchronization between otherwise unrelated threads as they compete for hardware resources. In [7] an all-software framework called FUSE for the management of hardware accelerators in traditional multi-threaded (single-core) environment was proposed. However, the FUSE scheme is not suitable in situations when both smaller and larger datasets are processed by the same thread functions.

The coupling of multithreaded execution in processors with a concurrent execution in custom and reconfigurable hardware introduces the problem of sharing the hardware accelerators between multiple threads [16]. This requires synchronization between potentially unrelated threads, and/or a context-switch mechanism in the accelerator. In RCC [13] (the multithreaded extension of the Molen [14] protocol), only one thread at a time is allowed to execute in the reconfigurable unit. In Chimaera [4] the reconfigurable functional unit (RFU) is always stateless (purely combinatorial) so there is not any context to be saved on a thread switch. The Garp [5] RFU architecture provides special instructions for context save and restore on a thread switch, to be used by the operating system.

2.3.2 Proposed Solution

The interfacing of a microthreaded processor and specialized accelerator cores is handled within a dedicated subsystem called *Hardware Families of Threads* (HWFAM). The HWFAM allows for a complete *transparency* of the execution in accelerators relative to the general-purpose microthreaded processors. The activation of accelerators is based on SVP's *creation events* which normally commence execution of new families of threads. To recognize which family of threads should be executed

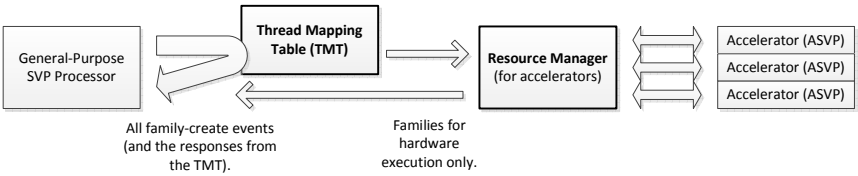

Figure 15: Flowchart of the create events in the HWFAM subsystem.

in an accelerator the HWFAM uses the *thread function address*, set-up for each family by the `setthread` instruction.

The circulation of create events in a simple system consisting of one general-purpose processor and an accelerator is shown in Figure 15. The key component of the proposed HWFAM interface is the *Thread Mapping Table* (TMT), located in the middle of the picture. The TMT unit decides which create events, originating in the microthreaded processor on the left, should be passed on to accelerators. According to the decision, create events are either handed over to the *resource manager* to orchestrate the accelerated execution, or they are sent back to the GP processor to resume regular thread initialization. The resource manager autonomously translates SVP create events in command sequences for accelerators. Once the execution in accelerators has finished the resource manager sends a *completion message* to the GP processor to release the family ID in the thread scheduler and to signal family's parent thread.

2.3.3 Results

The placement of thread families into a processor or accelerator is decided at run-time when the family is created. The decision in TMT considers the current resource occupancy–the availability of accelerators. The mapping of threads to processor or accelerators is controlled by the blocksize parameter, a standard SVP mechanism for performance throttling. The experiments have shown that the values 'blocksize = numberOfAccelerators' or 'blocksize = numberOfAccelerators+1' are usually good choices.

If the work-load size is correlated with the number of threads the HWFAM scheme can steer creation events between accelerators and processors at run-time. Small-sized thread families will not be transferred to an accelerator even if one is available because the fixed-costs associated with the move would outweigh benefits.

Thread-family creation events are routed through the TMT in HWFAM. The microthreaded processor well tolerates new latencies that arise in TMT. The load on TMT processing can be lowered by marking a subset of families as *virtual*; the processor then sends only these 'virtual' families into TMT. The marking is indicated

	Sched?	Hw?	Benchmarks *FIR*	*DCT*	*MF-set*
GP-ASVP: vector processing	SS	PI	✓		✓
Manual HDL: static pipelining	SS	PD	✓	✓	✓
FTL: pipelining	DS	PD	✓	✓	✓
LEON3: scalar processor	SS	PI	✓	✓	✓
UTLEON3: microthreading processor	DS	PI	✓	✓	✓

Table 1: The benchmarks and platforms used for the comparison. **SS**=Static Schedule. **DS**=Dynamic Schedule. **PI**=Problem-Independent hardware (= predefined, static). **PD**=Problem-Dependent hardware (= specialized).

		Benchmarks *FIR*	*DCT*	*MF-set*	**Techno.**
GP-ASVP	Area:	1781 slices			Virtex 6
	Frequency:	200MHz			(xc6vlx240t)
Manual HDL	Area:	(various)	231 sl.	1878 sl.	Virtex 6
	Frequency:	150MHz	150MHz	200MHz	(xc6vlx240t)
FTL	Area:	(various)	516 sl.	2186 sl.	Virtex 6
	Frequency:	150MHz	150MHz	200MHz	(xc6vlx240t)
LEON3	Area:	1927 slices			Virtex 5
	Frequency:	80MHz			(xc5vlx110t-1)
UTLEON3	Area:	7988 slices			Virtex 5
	Frequency:	40MHz			(xc5vlx110t-1)

Table 2: Platform characterization data used for the evaluation.

by setting the LSB of thread function addresses, hence it does not require new ISA instructions or registers.

2.4 Objective O4: Comparison of the Dataflow Approaches

The approaches discussed in the thesis are compared using three micro-benchmarks: *Finite Impulse Response* filter (FIR), *Discrete Cosine Transform* (DCT), and the *Mandelbrot Fractal* set (MF-set), see Table 1.

Hardware platform characterization data used for the evaluation are reported in Table 2. In the GP-ASVP, LEON3 and UTLEON3 platforms the hardware is problem-independent. In the Manual HDL and FTL approaches the hardware is problem-dependent and implements one of the benchmarks. Based on the platform data the absolute execution times in seconds of the individual benchmarks were obtained.

The standard metrics used for the comparison are the execution time speed-up S

	MaxIter Pixels: 100%				
	time [ms]	IPC	ta	S	F
GP-ASVP	0.5493		978	1×	1×
Manual HDL (SysGen)	0.0325		61	16.9×	16.0×
FTL (MF-ASVP)	0.0334		71	16.5×	13.8×
LEON3	8.9	0.61	17064	0.062×	0.057×
UTLEON3	12.2	0.90	97204	0.045×	0.010×

Table 3: MF-set: All platforms, finishing times. Computed over a block of 100 pixels with the ratio of the in-set pixels 100%. Resource utilization in GP-ASVP and FTL as for Virtex 6 at 200MHz.

(all cases) and the Instructions Per Cycle (IPC, for processors). Since the hardware platforms require different amounts of resources in an FPGA the speed-up S is not fair. The performance metric that relates latency and resource usage is the product '$time \times area$', or ta in short. The $time$ is the task execution latency in seconds; the $area$ is the occupied area measured in FPGA slices. The concept was defined in [15] as the *functional density*: $D = \frac{1}{t \cdot a}$. For example, an implementation that occupies $2\times$ more slices and is $2\times$ faster has the same $time \times area$ performance as the original one. The ta values presented below are in $milliseconds \times Slices$.

The speed-up ratio F compares the $time \times area$ products of two implementations 1, 2:

$$F_{2/1} = \frac{ta_1}{ta_2} = \frac{D_2}{D_1} \tag{1}$$

When $F_{2/1} = 1$ the performance scaling of the two implementations is identical. For example, if the second implementation is $3\times$ faster it is also $3\times$ larger in area. When $F_{2/1} > 1$ an FPGA slice in the implementation 2 is 'better used' (more productive) than in the implementation 1.

Table 3 reports typical results, measured for the *MF-set* benchmark. The GP-ASVP (the architecture **A1**) is the baseline, hence S_{GPASVP}=F_{GPASVP}= $1\times$. The S-speed-up (execution time only) is the highest for the Manual HDL approach ($16.9\times$), closely followed by the FTL ($16.5\times$) (objective **O2**). The F-speed-up (execution time *and* resource usage) is the highest for the Manual HDL ($16.0\times$), followed by the FTL ($13.8\times$). Higher resource utilization in FTL is caused by issues in the way legacy pipelined FP cores are embedded and in proper sizing of FIFO queues in cyclic (closed) pipelines.

The processors are always much slower than the other approaches; it is the cost of the ease of programming and generality. Typically the microthreaded UTLEON3 processor has better (higher) IPC than the LEON3. However, the current implementation of UTLEON3 in FPGA is $4.2\times$ larger in slices and it has $2.0\times$ slower clock

frequency, hence the absolute performance is always worse than in LEON3.

Implementations in GP-ASVP are typically an order of magnitude slower than in the fully specialized hardware (Manual HDL or FTL). The GP-ASVP hardware is specialized for an application *domain* but it can be programmed for different tasks using firmware. Depending on the particular use-case the lower performance may or may not be acceptable.

3 Thesis Contributions

There are five main **contributions**, labelled (1)–(5) in Figure 3 and listed here:

(1) **Analysis of a statically scheduled instruction-driven vector processing architecture for customized computing realized in a reconfigurable array.**
The particular architecture **A1** was disclosed in previous work [1]. The architecture is characterised here separately in hardware implementation and in application mapping. It is shown here that (contrary to previous design expectations) a high-frequency issue of vector instructions is needed in spite of using batched processing in the data path.

(2) **A method for achieving high-frequency instruction issue using controller specialization in an architecture with wide horizontal instructions generated on the fly.**
A quantitative analysis of the structure of vector instructions and of the issue frequencies are performed. A technique is proposed that reduces overheads in the instruction issue by storing frequently used combinations in a dictionary table.

(3) **Structured and extensible approach for synthesis of hardware controllers from synchronous Petri nets.**
The concurrency in hardware controllers is limited by a decomposition step that partitions the functional specification into cooperating sequential automatons. Controller specification using Petri nets has the advantage that decomposition for hardware synthesis is not needed.

(4) **A new technique for dataflow hardware synthesis from Petri nets.**
The idea is that traditional HDLs can describe logic structure (gates) of hardware but neglect the importance of interconnect delays. The new technique based on SPN synthesis (3) gets around the problem by decoupling functionality from timing, hence hardware specifications are ‘elastic’ and adaptable to interconnect latencies in a changing implementation technology.

(5) **Bridging the gap between the data-driven microthreaded procedural computation (arch. A2) with the special-purpose data-driven hardware in re-**

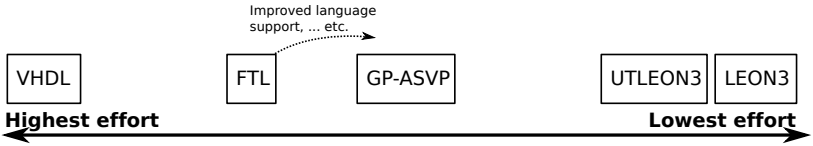

Figure 16: Author's subjective estimation of the effort needed for using the platforms.

configurable arrays (arch. A1).
A transparent software/hardware interface that couples a general-purpose microthreaded model and custom accelerators is demonstrated. Concurrency issues are handled in hardware using the SVP model.

4 Conclusions

The author's subjective estimation of the effort required for using the approaches discussed in the thesis is shown in Figure 16. The horizontal axis goes from the highest effort needed at the left to the lowest effort at the right. The LEON3 scalar processor is put at the far right as the easiest to use, while a manual VHDL approach is at the far left as the most challenging. Microthreading (UTLEON3) is not difficult to understand conceptually, though there are peculiar details that must be considered at the assembly level. The GP-ASVP approach was put in the middle because detailed hardware/software interactions must be observed by the programmer. At the moment the FTL approach is more difficult to use than the GP-ASVP, hence it was put to the left of it. Future improvements at the language and feature level might put it at least on-par with ASVP.

The statically scheduled hardware architecture **A1** (ASVP) is suitable for data-parallel applications only. The memory model requires that applications must be partitioned into tasks and all the I/O data for each task must fit in local memory banks. Dataflow scheduling is limited to the task level. The disadvantage of the architecture is the requirement to decompose algorithms over several layers: (1) operations constructed in hardware; (2) tasks executed in firmware; and (3) the management software in a host processor. The decomposition must respect that: (1) hardware operations implemented in the DFU should be simple and reusable; and (2) firmware code and the task I/O data must fit in local memory banks. Furthermore, vector-oriented processing requires high regularity of data parallelism down to the level of elementary operations. The advantage of the architecture is the software-oriented flow once the primitive hardware operations are defined and implemented in DFU.

The FTL (objective **O2**) increases the specification level of hardware modules. Dataflow scheduling is performed for individual hardware operations. The hard-

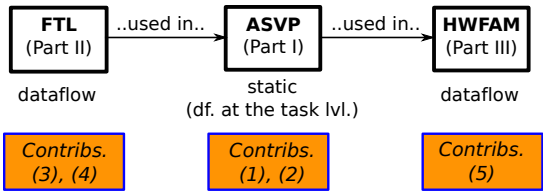

Figure 17: Practical composition of the three methods (ASVP, FTL, HWFAM) as discussed in the thesis.

ware that performs the scheduling is custom-generated for a given pipeline, hence the overheads are small. Parallel processing is implemented by pipelining. Individual high-level operations executed by algorithms are mapped to distinct pipeline stages. Irregular codes, such as conditional blocks, insert the alternate-choice nodes and split the pipelines. Pipelining is more flexible than the vector processing.

Microthreading implements dataflow scheduling in a general-purpose computing platform. Compared to previous dataflow machines and processors the dynamic scheduling of threads is more coarse grained than scheduling of instructions. This has the advantage of lower hardware resource requirements. The HWFAM interface (objective **O3**) views families of threads as posted tasks of work. It takes advantage of the memory model which guarantees visibility of memory updates only after threads have been explicitly synchronized. A family of threads is a black-box that can be executed anywhere.

Experiments show that while microthreading is able to improve pipeline efficiency (IPC), the current implementation in UTLEON3 is not competitive with a classical scalar in-order processor such as the LEON3. Future implementation of microthreading must therefore achieve on-par clock-cycle delay, reasonable resource requirements (say at most $2\times$ than a classical core) and–most importantly–it must support the cooperative multi-core operation. This in turn requires high-performance pipelined memory buses and cache coherency.

Parallel execution in microthreading is based on efficient utilization of many independent but similar (interchangeable) processing units with fully programmable interconnect, realized partly in hardware and partly in shared memory. Pipelining in FTL on the other hand specializes each pipeline stage to a single program step, including the connections to the previous and next processing stages. The decision to use the first (microthreading) or the second (pipelining) approach is guided by an observation if there are ‘more instructions or more data in the algorithm’. Unfortunately, there is hardly any universal measure that could quantify the ratio of instructions vs. data.

Future Work

The greatest potential for future work, both theoretical and practical, is in the *Flow Transfer Level*–the dataflow processing using Petri nets, presented in Part II. At the current theoretical level the synthesis is limited to FTL nets that decompose into acyclic activation nets. Cyclic activation nets would be transformed into cyclic combinatorial circuits in hardware. Attempts to overcome this limitations have already been made by the author, but the solutions obtained so far are unsatisfactory: while some practical experiments were successful (synthesis of testing FTL nets), the method itself could not be proved to be correct for all cases.

A practical limitation of FTL is the absence of a proper design input language. The CAD tool presented in the thesis is being developed in the reverse direction: from the VHDL code generator back to the middle end. An input FTL net is given as a short program in Python that instantiates appropriate objects and constructs the graph. This strategy proved successful to test quickly the core transformation and synthesis algorithms.

Résumé (English)

The roots of all evil are the latencies that are statically unpredictable. Dynamic schedule of operations, constructed on-the-fly in data-driven machines, is needed to overcome them. Microthreading is a unified data-driven and dynamically scheduled model for efficient programming of many-core general-purpose processors. It overcomes unpredictable latencies in off-chip memories (DRAMs) and in on-chip shared interconnect. As silicon chips became power-limited, causing the shift from frequency scaling to many-core scaling, the previous work envisioned large-scale homogeneous many-core chips because it assumed that low-clock frequency silicon is easily scalable in space. However, the contemporary and future power constraints will favour heterogeneous (specialized) rather than homogeneous (general-purpose) many-cores because the thermal design power of a chip could be so low that not all cores may be powered up simultaneously.

Besides the power issues the other negative side-effect of silicon scaling is an increase in latency of interconnect (metal wires) relative to that of gates: new designs are becoming limited by interconnect delays. As the interconnect delays depend on details of physical placement of modules in a chip or in a reconfigurable array they are difficult to predict accurately early on in the design process. Consequently, future hardware will be special-purpose and customized due to the power issues, and it will be data-driven to overcome on-chip interconnect latencies.

This dissertation explores dataflow latency-tolerant techniques with a focus on customized hardware design using reconfigurable hardware arrays. Dataflow is stud-

ied at the gate and chip levels: gate-level dataflow overcomes on-chip interconnect delays, and chip-level dataflow allows for the composition of scalable heterogeneous many-cores.

The first contribution is an analysis of a contemporary statically scheduled instruction-driven architecture for customized computing realized in an FPGA. In contrast to the original design bases of the architecture it is shown here that high-frequency instruction issue is needed even in an architecture with batch (vector-based) data processing. The second contribution is a method to achieve the high-frequency instruction issue by using dictionary tables of instruction fragments.

Statically scheduled data-path used to be preferred because all latencies (including interconnect) were assumed to be fully known early in the design time. The third contribution is a new structured and extensible approach for synthesis of hardware controllers from synchronous Petri nets. The fourth contribution is a new technique for dataflow hardware synthesis from Petri nets. The technique is based on augmented synchronous Petri nets with optimal throughput.

The fifth contribution is a technique that combines the data-driven microthreaded procedural computation model with the special-purpose data-driven hardware in structurally programmed reconfigurable arrays. Adaptive transparent migration of microthreads between the general-purpose and special-purpose hardware is demonstrated.

Résumé (Czech)

U kořene mnoha problémů v *computer science* vězí nepredikovatelné latence. Pro jejich překonání je třeba operace plánovat dynamicky za běhu stroje. *Microthreading* je unifikovaný datově řízený model výpočtu s dynamickým plánováním instrukcí pro efektivní programování mnoho-jádrových procesorů. Model překonává latence při přístupu do hlavní paměti (DRAM) a při komunikaci mezi jádry na čipu. Dominantním omezujícím faktorem moderních výkonných čipů je elektrický příkon konvertovaný na teplo. Důsledkem je upuštění od škálování hodinové frekvence a nástup škálování v počtu jader. Předchozí výzkumy předpokládaly vznik velkých čipů s mnoha homogenními jádry, jelikož se předpokládalo, že design běžící na malých frekvencích je lehce škálovatelný co do počtu jader. Nysí se však ukazuje, že omezení příkonu spíše nahrává heterogenním (specializovaným) spíše než homogenním (obecným) řešením, protože maximální povolený tepelný výstup čipu může být tak nízký, že ne všechna jádra mohou být aktivní současně.

Dalším problémem škálování křemíku je nárůst latence v propojení modulů na čipu. Latence propojovacích vodičů jsou těžko odhadnutelné v počátcích návrhu, jelikož závisí na detailech jejich fyzického umístění. Z výše uvedeného plyne, že hardware bude spíše specializovaný než obecný kvůli příkonu, a bude datově řízený, aby překonával nepredikovatelné latence.

Tato dizertace se věnuje technikám *dataflow* a tolerování latencí se zaměřením na zákaznicky specializovaný hardware využívající rekonfigurovatelná pole. Dataflow je studováno na úrovni hradel a čipů: na úrovni hradel jde o překonávání latencí v propojení mezi moduly, na úrovni čipů jde o kompozici škálovatelných heterogenních mnoho-jádrových systémů.

Prvním přínosem práce je analýza soudobé staticky plánované architektury řízené instrukcemi a realizované na FPGA. V kontrastu k předchozím očekáváním moje analýza ukazuje, že vysoká četnost vydávání nových instrukcí je potřebná i v architektuře s dávkovým (vektorovým) zpracováním instrukcí. Druhým přínosem práce je právě metoda k dosažení rychlejšího vydávání instrukcí.

Datové cesty se statickým plánem provádění operací bývaly preferovány, protože všechny latence (včetně propojení) byly známy již v počátcích návrhu. Třetím přínosem práce je nová strukturovaná a rozšiřitelná metoda pro syntézu řadičů ze synchronních Petriho sítí (SPN). Čtvrtým přínosem práce je metoda pro dataflow syntézu z SPN.

Pátým přínosem práce je metoda kombinující datově řízený *microthreadový* výpočetní model se zákaznicky specializovaným a datově řízeným hardware implementovaným na programovatelných polích. Je demonstrována adaptivní transparentní migrace microthreadů mezi obecným a specializovaným hardware.

Bibliography

[1] M. Danek, J. Kadlec, R. Bartosinski, and L. Kohout. Increasing the level of abstraction in FPGA-based designs. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 5 –10, 2008.

[2] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[3] J. Fernandes, M. Adamski, and A. Proenca. Vhdl generation from hierarchical petri net specifications of parallel controllers. *Computers and Digital Techniques, IEE Proceedings* -, 144(2):127–137, 1997.

[4] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera Reconfigurable Functional Unit, 1997.

[5] J. Hauser and J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. In *FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, pages 12 –21, 1997.

[6] R. Hilal and P. Ladet. Synchronous petri nets: formalisation and interpretation. In *Systems, Man and Cybernetics, 1993. 'Systems Engineering in the Service of Humans', Conference Proceedings., International Conference on*, pages 246–251 vol.2, 1993.

[7] A. Ismail and L. Shannon. FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 170 –177, may 2011.

[8] C. Jesshope. Scalable instruction-level parallelism. In *Computer Systems: Architectures, Modeling, and Simulation*, pages 383–392. Springer Berlin / Heidelberg, 2004.

[9] C. Jesshope. μTC - an intermediate language for programming chip multiprocessors. In *Asia-Pacific Computer Systems Architecture Conference*, pages 147–160, 2006.

[10] C. Jesshope. A model for the design and programming of multi-cores. *Advances in Parallel Computing*, High Performance Computing and Grids in Action(16):37–55, 2008.

[11] C. Jesshope, M. Lankamp, and L. Zhang. The implementation of an SVP many-core processor and the evaluation of its memory architecture. *SIGARCH Comput. Archit. News*, 37:38–45, July 2009.

[12] R. 'kena' Poss. *On the realizability of hardware microthreading—Revisting the general-purpose processor interface: consequences and challenges*. PhD thesis, University of Amsterdam, 2012.

[13] S. Uhrig, S. Maier, G. Kuzmanov, and T. Ungerer. Coupling of a reconfigurable architecture and a multithreaded processor core with integrated real-time scheduling. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 4 pp., 2006.

[14] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. Panainte. The MOLEN Polymorphic Processor. *Computers, IEEE Transactions on*, 53(11):1363 – 1375, 2004.

[15] M. J. Wirthlin and B. L. Hutchings. Improving functional density through run-time constant propagation. In *In ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 86–92, 1997.

[16] P. G. Zaykov, G. K. Kuzmanov, and G. N. Gaydadjiev. Reconfigurable multithreading architectures: A survey. In *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS '09, pages 263–274, Berlin, Heidelberg, 2009. Springer-Verlag.

Publications of the Author

NOTE: The share of authorships of the publications listed below is given in percentages in superscripts (when the shares are not equal). The marking (S) means the author is a supervisor. All the publications are referred in the thesis.

Books (Non-Peer Reviewed)

[X.1] Martin Daněk, Leoš Kafka, Lukáš Kohout, Jaroslav Sýkora, Roman Bartosiński. *UTLEON3: Exploring Fine-Grain Multi-Threading in FPGAs*. Monography, 237 pages. ISBN 978-1461424093. Springer, (New York 2012) Circuits & Systems, [2012], `http://www.springer.com/engineering/circuits+&+systems/book/978-1-4614-2409-3`

[X.2] Torquati, M.; Bertels, K.; Karlsson, S.; Pacull, F. (Eds.) *Smart Multi-core Embedded Systems*. Chapters in a book. 175 pages, Springer, 2014. ISBN 978-1-4614-8800-2. http://www.springer.com/engineering/circuits+%26+systems/book/978-1-4614-8799-9

Journal Papers (Peer Reviewed)

[X.3] Martin Daněk, Leoš Kafka, Lukáš Kohout, Jaroslav Sýkora. *Hardware Support for Fine-Grain Multi-Threading in LEON3*. In Carpathian Journal of Electronic and Computer Engineering, Volume 4, Number 1 – 2011. ISSN 1844-9689.

Conference Papers (Peer Reviewed)

[X.4] Jaroslav Sýkora. *Composing Data-driven Circuits Using Handshake in the Clock-Synchronous Domain*. In: 2013 IEEE 16th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), CZ, IEEE CS, 2013, s. 211-214, ISBN 978-1-4673-6133-0, http://www.jsykora.info/p/ddecs2013-ftl.pdf

[X.5] Jaroslav Sýkora, Sven-Bodo Scholz. *Towards Self-Adaptive Concurrent Software Guided by On-line Performance Modelling*. In: 2nd HiPEAC Workshop on Feedback-Directed Compiler Optimization for Multicore Architectures (FD-COMA'13), Berlin, Germany, 2013. http://www.jsykora.info/p/fdcoma2013-snet-adapt.pdf

[X.6] Jaroslav Sýkora$^{(65\%)}$, Roman Bartosinski$^{(20\%)}$, Lukáš Kohout$^{(10\%)}$, Martin Daněk$^{(S)}$, Petr Honzík$^{(5\%)}$. *Reducing Instruction Issue Overheads in Application-Specific Vector Processors*. In Proceedings of the 15th Euromicro Conference on Digital System Design, DSD 2012, Eds: Niar Smail, 15th Euromicro Conference on Digital System Design, (Cesme, Izmir, TR, 5.9.2012-8.9.2012), ISBN-13: 978-0-7695-4798-5, http://www.jsykora.info/p/dsd2012-asvp.pdf

The paper has been cited in:

- Bařina, D., Zemčík, P.: *Wavelet Lifting on Application Specific Vector Processor*, In: GraphiCon'2013, Vladivostok, RU, GraphiCon, 2013, s. 83-86, ISBN 978-5-8044-1402-4

[X.7] Jaroslav Sýkora$^{(44\%)}$, Lukáš Kohout$^{(17\%)}$, Roman Bartosinski$^{(17\%)}$, Leoš Kafka$^{(17\%)}$, Martin Daněk$^{(S)}$, Petr Honzík$^{(5\%)}$. *The Architecture and the*

Technology Characterization of an FPGA-based Customizable Application-Specific Vector Processor. In Proceedings of the 2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), ISBN 978-1-4673-1185-4. 2012 IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits and Systems, (Tallinn, EE, 18.04.2012-20.04.2012) `http://www.jsykora.info/p/ddecs2012-asvp.pdf`

[X.8] Raphael Poss, Mike Lankamp, M. Irfan Uddin, Jaroslav Sýkora, Leoš Kafka. *Heterogeneous integration to simplify many-core architecture simulations*. In 4th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO'12), Paris, France, 23.1.2012.

[X.9] Roman Bartosinski, Martin Daněk, Jaroslav Sýkora, Lukáš Kohout, Petr Honzík. *Video surveillance application based on application specific vector processors*. In Proceedings of the 2012 Conference on Design & Architectures for Signal & Image Processing, Eds: Morawiec Adam, Hinderscheit Jinnie, Conference on Design & Architectures for Signal & Image Processing, (Karlsruhe, DE, 23.10.2012-25.10.2012), ISBN: 978-1-4673-2089-4, [2012].

[X.10] Jaroslav Sýkora$^{(50\%)}$, Leoš Kafka$^{(25\%)}$, Martin Daněk$^{(S)}$, Lukáš Kohout$^{(25\%)}$. *Microthreading as a Novel Method for Close Coupling of Custom Hardware Accelerators to SVP Processors*. In 2011 14th Euromicro Conference on Digital System Design Architectures, Methods and Tools DSD 2011. Oulu, Finland : IEEE Computer Society Conference Publishing Services, 2011. S. 525-532. ISBN 978-0-7695-4494-6. ISSN N. [14th Euromicro Conference on Digital System Design Architectures, Methods and Tools DSD 2011, Oulu, 31.8.2011-2.9.2011, FI]. `http://www.jsykora.info/p/dsd2011-hwfam.pdf`

[X.11] Jaroslav Sýkora$^{(50\%)}$, Leoš Kafka$^{(25\%)}$, Martin Daněk$^{(S)}$, Lukáš Kohout$^{(25\%)}$. *Analysis of Execution Efficiency in the Microthreaded Processor UTLEON3*. In Architecture of Computing Systems - ARCS 2011. Berlin : Springer-Verlag Berlin Heidelberg, 2011. S. 110-121. ISBN 978-3-642-19136-7. ISSN 0302-9743. [ARCS 2011. International Conference on Architecture of computing systems /24./, Como, 24.02.2011-25.02.2011, IT]. `http://www.jsykora.info/p/arcs2011-utleon3.pdf`

The paper has been cited in:

- Michiel W. van Tol, Chris R. Jesshope. *An Operating System Strategy for General-purpose Parallel Computing on Many-core Architectures*. In Advances in Parallel Computing, Volume 20: High Performance

Computing: From Grids and Clouds to Exascale. Pages 157-181. ISBN 978-1-60750-802-1, DOI 10.3233/978-1-60750-803-8-157.

- Roy Bakker and Michiel W. van Tol. *Experiences in porting the SVP concurrency model to the 48-core Intel SCC using dedicated copy cores*. 4th Many-core Applications Research Community Symposium (MARC'11), ISBN 9783869561691, Potsdam, Germany, [2012].

[X.12] Martin Daněk, Leoš Kafka, Lukáš Kohout, Jaroslav Sýkora. *Instruction Set Extensions for Multi-Threading in LEON3*. In Proceedings of the 13th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems. Los Alamitos : IEEE, 2010. S. 237-242. ISBN 978-1-4244-6610-8. [DDECS 2010 : 13th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems, Vienna, 14.04.2010-16.04.2010, AT]. `http://www.jsykora.info/p/ddecs2010-utleon.pdf`

The paper has been cited in:

- Salgado, F. and Garcia, P. and Gomes, T. and Cabral, J. and Monteiro, J. and Tavares, A. and Ekpanyapong, M. *Exploring metrics tradeoffs in a multithreading extensible processor*. In 2012 IEEE International Symposium on Industrial Electronics (ISIE). ISSN 2163-5137. [2012]
- Garcia, P.; Gomes, T.; Salgado, F.; Cabral, J.; Monteiro, J.; Tavares, A., *RAPTOR-Design: Refactorable Architecture Processor to Optimize Recurrent Design*, Computing System Engineering (SBESC), 2012 Brazilian Symposium on, ISBN 978-1-4673-5747-0, pp.188,191, 5-7 Nov. 2012.
- Olivier Serres, Abdullah Kayi, Ahmad Anbar, Tarek El-Ghazawi. *Hardware Support for Address Mapping in PGAS Languages; a UPC Case Study*. Submitted on 9 Sep 2013, available at `http://arxiv.org/abs/1309.2328`

[X.13] Jaroslav Sýkora. *A Method for Evaluating Latency Tolerance of Micro-Threaded Processor UTLEON3*. In Počítačové Architektury & Diagnostika PAD 2010. Češkovice, 13. 9. 2010 - 15. 9. 2010.

Other Non-Peer Reviewed Publications

[X.14] Jaroslav Sýkora. *LLVM-Based C Compiler for the PicoBlaze Processor*, 2012, Technical report. `http://www.jsykora.info/p/`

pblaze-llvm-cc.pdf; *Optimizing C Compiler and an ELF-Based Toolchain for the PicoBlaze Processor*, 2012, Software. http://sp.utia.cz/index.php?ids=pblaze-cc