<http://www.diva-portal.org>

Preprint

This is the submitted version of a paper presented at *Embedded Multicore/Manycore SoCs (MCSoc), 2014 IEEE 8th International Symposium on*.

Citation for the original published paper:

Podobas, A. (2014)

Accelerating Parallel Computations with OpenMP-Driven System-on-Chip Generation for FPGAs.

In: *Embedded Multicore/Manycore SoCs (MCSoc), 2014 IEEE 8th International Symposium on* (pp. 149-156). Aizu-Wakamatsu: IEEE conference proceedings

<http://dx.doi.org/10.1109/MCSoc.2014.30>

N.B. When citing this work, cite the original published paper.

Permanent link to this version:

<http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-161262>

Accelerating Parallel Computations with OpenMP-driven System-on-Chip Generation for FPGAs

Artur Podobas

*School of Information and Communication Technology
Royal Institute of Technology, KTH
Stockholm, Sweden
Email: podobas@kth.se*

Abstract—The task-based programming paradigm offers a portable way of writing parallel applications. However, it requires tedious tuning of the application for performance. We present a novel design flow where programmers can use application knowledge to easily generate a System-on-Chip (SoC) specialized in executing the application. Our design flow uses a compiler that automatically generates task-specific cores and packs them into a custom SoC. A SoC-specific runtime systems schedules tasks on cores to accelerate application execution. The generated SoC shows up to 6000 times performance improvement in comparison to the Altera NiosII/s processor and up to 7 times compared to an AMD Opteron 6172 core. Our design flow helps programmers generate high-performance systems without requiring tuning and prior hardware design knowledge.

Keywords—OpenMP FPGA, HLS, Hardware Generation, Parallel, Task-based, Software-Hardware

I. INTRODUCTION

Processors with multiple cores (multicores) form the bulk of today's general purpose market. Multicores overcome the thermal- and power-constraints encountered by their single counterpart by working in parallel. Today we also see pattern-specific accelerators. Accelerators such as General Purpose Graphics Processing Units (GPGPUs) improve performance by focusing on a specific programming paradigm— for GPUs this is the Single-Program Multiple Data (SPMD) model. Because GPUs are limited to the SPMD programming paradigm they can specialize in performing these patterns very well; there is no need to have a cache, memory protection, programmable interrupts, and silicon is spent on replicating many small-cores that work in parallel. *Parallelism* and *specialization* is the focus of the present paper. We use Field Programmable Gate Arrays (FPGAs) to specialize the hardware. A FPGA contains a sea of unconnected logic; a hardware designer can use Hardware-Description Languages (HDL) to describe digital logic, program a FPGA with it and simulate the hardware behavior.

Hardware designed manually by humans for FPGAs can achieve higher performance *and* power benefits compared to both general purpose processor and GPUs [1], [2], [3].

However, hardware-designers tend to be more scarce than software-designers. A hardware vendor designs a new processor and the software industry adapts their software to the new architecture. The opposite could be more productive—the application should drive the generation of hardware. Software, in particular parallel software have very firm properties: 1) the resources or instructions that a compiled software is using and 2) the number of cores (parallelism) the software can efficiently use.

We developed a High-Level Synthesis (HLS) design-flow called fpBLYSK that combines parallel computing, compiler and hardware design, which allows software developers to exploit parallelism and hardware acceleration with minimal effort. Our design-flow is based on the composable and modular *task-based* programming paradigm, where the application is decomposed into units called tasks. A task can execute concurrently with other tasks and is not bound to a specific core. We target the OpenMP programming model but the design-flow fits any task-based model. A software developer writes a program using OpenMP task directives and our compiler transforms the tasks down to hardware with only the resources they require. The programmer can also convey profiling information to our tool about task-graph properties such as the *span* [4], allowing our tool to design a system with as many *task-specific* cores as the application can utilize.

Our contributions are the following:

- A novel design-flow where we show how OpenMP can be used to drive system generation and what benefits it brings
- A High-level Synthesis Tool capable of generating hardware cores based on tasks as well as transform the OpenMP application to utilize them
- An evaluation in terms of execution performance and scalability that shows why task-based parallelism is more suited for reconfigurable rather than general purpose hardware

The rest of the paper is structured in the following way. Section II positions the present paper against previous

studies. Section III presents our novel design-flow including implementation details and examples. Section IV discloses how we evaluated our approach and section V shows and discusses the experimental results. Finally conclusions are presented in section VI.

II. RELATED WORK

Choi et al. [5], [6] studied conversion of software OpenMP-threads to custom accelerators, where each accelerator has direct access to main memory. Compilation from C to HDL is provided. Aside from targeting a different paradigm (task-based), our work increases determinism since cores work in isolation. Our compiler also supports setting different constraints on the generated code (increasing design-space exploration). From a compilation perspective, our work is closely related and inspired by Leow et al.'s OpenMP compiler [7]– we use similar technique of mapping the C code to a state-machine.

OmpSs [8] is a parallel framework that supports FPGAs by offloading tasks similar to our approach, targeting both High-Performance Computers [9] and SoC's [10]. Unlike their work, which use third-party tools for the HDL generation and focuses more on the software extensions required, our approach promotes tight integration between parallel software and hardware layers for system generation.

The CerberO [11] architecture automatically generates a SoC for OpenMP applications, but is limited to soft-cores (MicroBlaze) rather than creating customized accelerators as we do. The SPMD paradigm is a popular target for FPGA acceleration, particularly for branch-less kernels. FCUDA by Papakonstantinou et al. [12] is one such framework that synthesized FPGA hardware based on nVidia CUDA kernels using third-party tools. Altera also provides a tool for generating synthesizable hardware cores executing OpenCL kernels.

Instead of creating isolated accelerators, computations can be speeded up by putting the accelerators inside the processor. The DURASE [13] system provides a methodology for finding application patterns that are converted to hardware accelerating modules incorporated into the NiosII processor, increasing the performance in the application's critical paths. NAPA C [14] follows a similar approach where RISC processor cores are extended with specialized hardware capable of accelerating parts of the application using custom logic. NAPA C uses similar scheduling mechanisms as we do when generating the hardware from the intermediate code. Commercial alternatives include both Altera's NiosII C2H and Xilinx's Vivado that are HLS tools used to accelerate subsets of the C language.

III. OPENMP-DRIVEN SYSTEM-ON-CHIP GENERATION

A. Idea

Our design-flow is divided into three phases as shown in Fig. 1. The first step (Fig. 1:a) is to program the

Figure 1. Overview of the proposed design-flow. Source-code written using OpenMP-tasks (a) is optionally profiled to determine available parallelism (b). Our compiler compiles the code (c) and emits a specialized SoC for the given application (d) and source-code that uses the generated SoC (e)

application using OpenMP-tasks and tune it. Tuning could involve exposing desirable amount of parallelism by changing the cutoff [15]. The application is then optionally analyzed (Fig. 1:b) by an OpenMP profiler [16]. A profiler can analyze the *span*, which is the property governing the maximum number of processing cores the application could scale to. Our compiler is then invoked (Fig. 1:c) and builds a System-on-Chip (SoC) based around the NiosII processor and an application that performs the OpenMP calculation on the generated SoC (Fig. 1:d and e).

The proposed approach allows several advantages over parallelism-oblivious approaches. Primarily, the programmer receives task-specific accelerators that contain only the resources required to perform a task's computation. The SoC itself is specialized to the needs of the application, and will contain no-more task-specific cores than the application can utilize, thus maximizing performance and reducing power consumption. Our compiler performs three steps: generate individual tasks written in C to hardware accelerators, building a SoC containing the task-specific cores and to transform the otherwise architecture-independent OpenMP code to exploit the generated SoC.

B. Task-specific core generation

We created a compiler for C89 and the OpenMP compiler directives. The compiler is hand-written and uses a Recursive Descend (RD) parser that transforms the code into an Abstract Syntax Tree (AST). Once the AST has gone through type-checking and semantic analysis, we transform the AST into a Three-Address-Code (TAC) based intermediate format. After optimizations and dependency analysis is complete, the hardware generation begins. The user or a profiler has several options governing the hardware that is generated. The following options are supported:

- Allowed number of resources of each type (**-resource= R**). The compiler may only generate hardware that has R similar resources. For example, if $R = 2$ then the hardware will have no-more than two adders even if instruction-level parallelism allows for more. This is a core regulating option.
- Instruction Re-Ordering (**-reorder**) allows instructions to be re-ordered assuming that there is available resources and no dependency between the instructions. This is a core regulating option.
- Pipelining of instructions (**-pipeline= P**). Pipelining allows P independent instructions to execute on a given resource simultaneously. If there are more than P instructions already executing in the resource, either a new resource will be allocated or the instruction will be scheduled later. This is a core regulating option.
- Result-forwarding (**-enable-forwarding**) allows a result to be immediately forwarded in the same cycle it was computed rather than storing it in a register before usage. This is a core regulating option.
- Number of cores to generate in the system (**-numcores= C**). This is a system regulating option.

1) *Internal Hardware Representation*: We have four basic primitives to internally (before emitting HDL-code) describe the hardware. Every connection point in our generated hardware will have a *node*. A node can be a driver, a sink or both. Connections between nodes are done using *wires*. Wires connects a node that drives the wire to one or several sink nodes. A wire can also be of *bus*-type and be driven by several nodes. Multiplexers form the basic primitive for flow-control in our system. A multiplexer can have several inputs of different widths, and one output. Multiplexers can extend the input width to the output width— this is important when a constant/variable is represented as a 4-bit value but requires sign-extension to 32-bit. The third primitive is a register of variable bit-length. Registers are used to track variables and compiler inserted temporary variables. The fourth and last primitive is an generic unit that has a variable number of ports to represent functions (Adders, Dividers, etc.).

The reason we internally construct our system with this low-level of detail is for validation- we can check for errors such as multiple nodes driving a wire during a certain phase in our hardware. Also, if given the logic for the functional units, we could generate SPICE models for simulation, but this is outside the scope of the present paper.

2) *Example Hardware Generation*: Consider the following, entry point C code:

```
1 void task_ex ( void ) {
2   int a,b,c,d;
3   a = a + 5;
4   b = 13 * a;
5   c = 12 * a;
6   d = b + c; }
```

The above C-code will be transformed to the following TAC code by our compiler:

```
1 <kernel_start>
2 a.int.signed = a.int.signed.32 + 5.int.signed.4
3 b.int.signed = 13.int.unsigned.32 * a.int.signed.
4 c.int.signed = 12.int.unsigned.5 * a.int.signed.32
5 d.int.signed = b.int.signed.32 + c.int.signed.32
6 <kernel_end>
```

Each operand in the TAC-code is in the format: `<name>.<type>.<signed/unsigned>.<width>`. Unlike a traditional software compiler, which targets a predefined set of registers (and constant register-size), we can generate registers of variable width and extend them when needed in other parts of the logic, which is the reason for the `width` field. For example, the constant 5 can be represented with 4-bits (3 bit + sign-bit). Variables can also be reduced in size using value-ranged optimizations. The width of a variable impacts the resources it consumes on the FPGA. The operations `kernel_start` and `kernel_end` are automatically inserted by the compiler and will generate the logic that governs how the task is started and when it ends.

To illustrate the impact of different options, assume that the example TAC-code is compiled with three different option-setups: **noopt** with all optimizations disabled, **pipe-online** with pipelining, instruction re-ordering and limited resources (`-pipeline=inf,-resource=1,-reorder`) and **super-scalar** which allows result forwarding and endless amount of resources but no pipelining (`-pipeline=1,-resource=inf,-enable-forwarding,-reorder`).

The TAC-code is synthesized into a Finite-State-Machine (FSM) with the control signals corresponding for each TAC-Operation (similar in approach to [7]):

Operation #1 ($a=a+1$): uses an *integer-adder* to perform the addition between the variable a and a constant 5. It has no prior dependencies (Fig. 2:a) and will be scheduled at the state immediately after the kernel has started execution. This is the first operation and a new integer-adder is allocated. The compiler connects the register holding the variable a to one of the adder's inputs, and creates a constant for the value 5 ("0101"), which is connected to the second input of the adder. The state will be given control-signals that control the multiplexers of the adder's input. The integer-adder in the current example finishes within the same clock-cycle, and the value is captured in the same state.

Operation #2 ($b=13*a$): has a dependency (Fig. 2:a) on operation #1. All three versions will schedule operation #2 in the state following the state the first operation completes. The compiler will allocate an *integer-multiplier*, create the constant 13 and connects both source operands (13 and b) to the input multiplexers of the multiplier. Multiplexer control signals are added to the scheduled state. The integer-multiplier in the current example has a five cycle latency, and the value is captured five states after the state the instruction

Figure 2. (a) The TAC-operation dependency graph used when generating the hardware. (b) The hardware generated for the **noopt** and **pipeline** versions

was scheduled on.

Operation #3 ($c = 12 * b$): has a dependency (Fig. 2:a) on operation #1. The **noopt** version will schedule it after operation #2 has completed while the **pipeline** version will re-use the existing integer-multiplier and pipeline the operation. The **superscalar** version will on the other hand allocate one more integer-multiplier and schedule the instruction at the same state as operation #2, saving one clock-cycle compared to the **pipeline** version.

Operation #4 ($d = b + c$): has a dependency (Fig. 2:a) on both operation #1 and #2 and will schedule the instruction when variable b and c has been computed. The previously created integer-adder unit will be re-used, since no other operation is using it. The **superscalar** version will save one clock-cycle by immediately forwarding the results of b and c when they have been computed rather than waiting for the variables to be saved.

The final hardware created for the **noopt** and **pipeline** versions is shown in Fig. 2:b, where we also see the *core_ready* and *core_start* signals interfacing Altera's Avalon bus interface. These signals are used by the *kernel_start* and *kernel_end* to start and finish execution. The *core_ready* signal is asserted during the *kernel_start* state. Only when the *core_start* signal is asserted will the task-specific core start execution.

Each core has a *control-register* which is mapped at address zero in task-specific core's memory map. Starting a core is done by writing to the control-register and reading the control-register returns the status of the core (busy or ready).

The complete instruction schedules of the FSMs generated by the different versions can be seen in Table I. **noopt** scheduled all instructions back-to-back with no instruction-level parallelism, yielding an execution time of 14 clock cycles. **pipeline** took advantage of the parallelism of instruction

#2 and #3 and pipelined those to the one multiplier-unit existing, reducing the overall execution time by six cycles compared to **noopt**. **superscalar** manages to reduce the execution time by another two cycles by allocating another integer-multiplier (rather than pipelining it) and forwarding the computed values for variable b and c in the same cycle they were calculated.

Table I
FINITE-STATE-MACHINE SCHEDULE GENERATED WITH THE DIFFERENT
OPTIONS: **NOOPT**, **PIPEONLINE** AND **SUPERSCALAR**. TOTAL
EXECUTION TIME: **NOOPT**: 14 CYCLES, **PIPEONLINE**: 8 CYCLES,
SUPERSCALAR: 6 CYCLES

State	noopt	pipeline	superscalar
0	(kernel_start)	(kernel_start)	(kernel_start)
1	$a = a + 5$	$a = a + 5$	$a = a + 5$
2	$b = 13 * a$	$b = 13 * a$	$b = 13 * a$. $c = 12 * a$
3		$c = 12 * a$	
4			
5			
6			
7			$d = b + c$
8	$c = 12 * a$	$d = b + c$	(kernel_end)
9		(kernel_end)	
10			
11			
12			
13	$d = b + c$		
14	(kernel_end)		
15			

3) *Task Argument Passing*: Should the task contain any parameters, a block RAM is added to the task-specific core. The size of the block RAM is the size required to hold all the parameters. Consider the following function definition:

```
1 void task_ex ( int num , float result[128] )
2 { }
```

The above function definition would generate a block RAM with a size of 516 bytes (128+1 words). The variable *num* would be mapped to the first word in the block RAM and *result* would be mapped to the remaining 128 words. The task-specific core has uncontended access to the local argument block RAM during execution.

C. System Generation

Our compiler generates an Quartys System (QSYS) file specifying the number of task-specific cores in the system and connects them to a shared Avalon bus. Each core is assigned a unique memory-mapped address range, which is also saved for use in generating the OpenMP transformed application and run-time. We also instantiate a NiosII/s processor, on-chip RAM for holding data and instructions. An interval timer and JTAG interface is also generated to benchmark our solution.

D. OpenMP transformation

The final phase is to take the OpenMP compiler pragmas and source-to-source transform them to suit the NiosII and

the hardware we generated. We created a run-time system compatible with the OpenMP task-directives that can handle parallelism on the generated SoC. Consider the following task-definition and invocation (with syntax borrowed from [9]):

```

1 #pragma omp task device(FPGA)
2 void matmul ( float A[64], float B[64],
3             float C[64])
4 { <matrix multiplication code> }
5 ...
6 int main (...) {
7     #pragma omp task
8     matmul(inA, inB, inC);
9 }

```

The compiler knows that `matmul` is an accelerator and the argument-structure of that task. The compiler also has prior knowledge about the memory-mapped addresses where each task-specific core is mapped. The following code demonstrates how to start and immediately synchronize with a task:

```

1 /* #pragma omp task
2 / matmul(inA, inB, inC); */
3 unsigned int *sched_core;
4 while ( ( sched_core=get_free_core() )!=NULL)
5     synchronize();
6 Data_Transfer ( inA , sched_core[1+0],64);
7 Data_transfer ( inB , sched_core[1+64],64);
8 sched_core[0] = 1;
9 while (! sched_core[0]);
10 Data_transfer ( sched_core[1+128], inC , 64);

```

The shown code finds a core that is not currently occupied by sweeping through all the task-specific cores' control-registers. When a free core is found, we copy the arrays `inA` and `inB` into the core's local-memory at the same address increased by one (the control-register is mapped to adress zero). We start the core by asserting the control-register, and wait until the computation is complete. When the computation is finished we transfer the result array to `inC` and the task is done.

In our run-time system we do not immediately synchronize with a task. Instead, the task is executed and placed in a running queue. If all task-specific cores are occupied, then the task is placed in a ready-queue awaiting execution until a task-specific core becomes available. The code is:

```

1 /* #pragma omp task
2 matmul(inA, inB, inC); */
3 { struct tsk_info *task = alloc_task ();
4   task->A = inA; task->A_size = 64;
5   task->B = inB; task->B_size = 64;
6   task->C = inC; task->C_size = 64;
7   enqueue_and_try_launch (task); }
8
9 void enqueue_and_try_launch ( tsk_info *task) {
10  unsigned int *sched_core = get_free_core();
11  if (sched_core == NULL) enqueue_ready_task(task);
12  else
13  { Data_transfer ( task->A , sched_core[1+0],64);
14    Data_transfer ( task->B , sched_core[1+64],64);
15    task->Core = sched_core; sched_core[0] = 1;
16    enqueue_running_task ( task); } }

```

E. Restrictions

Our proposed methodology contains one intentional restriction: a task-specific core may not reference data using pointers. This forces the run-time system to explicitly copy all used data to the task-specific core before executing, and the task execution time becomes deterministic for a particular input set. Coupled with a quality-of-service capable interconnect [17], our design-flow can support soft or firm real-time guarantees. Another limitation is that function recursion is not allowed and task-specific cores cannot expose more parallelism (create new tasks). Distrubted memory also simplifies debugging as there is no corruption of the host processor's memory. Several recent programming models have promoted distributed memory computing in the task-based paradigm, such as OpenMP4.0 and OmpSs [8] and XKaapi [18]– our generated SoC fits well to take advantage of the previous research concerning scheduling of tasks to reduce data-transfers.

IV. METHODOLOGY

A. Experimental Platform

We used the Altera DE5 development board for all our experiments. The DE5 board contains a Stratix V (dev: 5SGXEA7N2F45C2) FPGA device. Synthesis was done using Altera's Quartus tool. We used a Nios II/s processor with 2kB of instruction cache, a 5-stage pipeline design, static branch prediction and hardware multiply/divide instructions. The clock-frequency used for evaluation was 50 MHz for all cases involving the FPGA board, but we have also successfully constrained and synthesized all our task-specific cores at 200 MHz. Performance was compared against a single-core NiosII/s processor and, to put the performance in perspective, against an AMD Opteron 6172 processor core running at 2.6 GHz.

Our HLS compiler used components with characteristics shown in Table II from Altera's MegaCore library. We used the shortest available pipeline for the components because they occupy the least amount of space on the device.

Table II
COMPONENT CHARACTERISTICS USED BY OUR TOOL.

Unit	Latency	Unit	Latency
SP Add/Sub	7	SP Multiply	4
SP Divide	5	INT Add/Sub	1
SP->INT conv.	5	INT->SP conv.	5
INT Multiply	5	INT Divide	6
INT Comparator	0	SP Comparator	1

B. Compilation

For all benchmarks we allowed the compiler to pipeline any operation as long as a stall did not occur. If the compiler detected a stall then it would instead allocate a new resource (**-pipeline=7**, **-reorder**, **-enable-forwarding**, **-resource=inf**). The number of cores generated varied for the

benchmarks, ranging from 120 cores for the PrimeNumber-kernel down-to 24 cores for the BLAS-3 kernel.

C. Benchmarks

Four benchmarks were used for evaluation purposes, all shown in Table III. Mandelbrot generates a 320x320 pixel fractal where we expose each scan-line in the image as a task. PI-kernel estimates the value of PI using 2 millions iterative steps. PrimeNumber-kernel calculates all prime-numbers between 2-20000. Our BLAS-3 algorithm is a *blocked* Matrix-Multiplication of two 256x256 matrix arrays where each block is 8x8 elements large. PrimeNumber-kernel is the only benchmark that does not use single-precision floating point operations.

Table III
BENCHMARK DESCRIPTION AND INPUT SETS USED FOR THE EVALUATION.

Benchmark	Input set	Behavior
Mandelbrot-kernel	320x320 pixels	kernel
PI-kernel	2 million steps	kernel
PrimeNumber-kernel	primes 1-20000	kernel
BLAS level-3 (MatMul)	256x256 (8x8 BlockSize)	scientific

D. Performance Measurements

Altera's Interval Timer was used to measure execution time for all experiments involving the FPGA device. For the AMD Opteron timing measurements we used the hardware time-stamp counter. The speed-up performance of our systems was calculated using:

$$Speedup = \frac{t_{ser}}{t_{par_n}}$$

where t_{par_n} is the execution time for our parallel solution for a particular number of task-specific cores n , and t_{ser} is the serial time for either the NiosII/s processor or the AMD Opteron 6172 processor core.

V. RESULTS

A. Single-core performance

The performance for the non-parallel version of the benchmarks is shown in Table IV for all three systems. The AMD Opteron 6172 is by far the fastest, mainly because it runs at a clock frequency that is 52 times faster than our board's frequency (50 MHz). In addition, the AMD Opteron core uses vector instructions that improves performance. Our task-specific core is only 5.6x slower on the PI-kernel and 22x slower on the BLAS-3 core execution. By unrolling loops in the the PI- and BLAS-3 kernel, our compiler manages to reorder and utilize instruction-level parallelism, which is why these two kernels perform well. One can argue that since the AMD Opteron 6172's frequency is 52 times higher, our task-specific cores are more performant. We have also successfully constrained and generated designs

for clock-frequency of 200MHz, which would put the AMD Opteron 6172 core and the task-specific core at a similar performance level. The BLAS-3 performance that includes transfers (BLAS L3 256x256) is 64x slower than the AMD Opteron, due to the slow data transfer rate in our system. The performance of NiosII/s is poor, mainly because the lack of hardware floating-point units¹. Still, even on an integer application such as the PrimeNumber-kernel our task-specific core is 50% faster than the NiosII/s. To conclude, using our methodology one can expect to get between 50% and 1750% performance speed-up without using parallelism compared to NiosII/s, and be as little as 5x slower than a state of the art ASIC processor without using OpenMP or having prior knowledge of hardware design.

Table IV
PERFORMANCE OF A SINGLE TASK-SPECIFIC CORE COMPARED TO THE NIOSII/S AND AMD OPTERON 6172 CORE

Benchmark	Task-Specific	NiosII/s	AMD Opteron
Mandelbrot	171.71 ms	14238 ms	4.69 ms
PI	417.81 ms	73386 ms	74.08 ms
PrimeNum.	6385 ms	9801 ms	406.53 ms
BLAS L3 (8x8, excl. transfer)	94.76 us	8056 us	4.28 us
BLAS L3 (256x256)	4819 ms	227488 ms	75.96 ms

B. Generated Systems

For the parallel performance evaluation we generated the systems shown in Table V. The largest system in terms of ALM cells is the PrimeNumber-kernel which also has the most cores. Some of the systems utilize all the hard DSP blocks for floating-point operations– this is not a hard limit since these functions can be implemented using logic. The amount of block-memory bits used by the BLAS level-3 kernel is larger relative to the others– this is because we need to generate a large on-chip RAM module to hold the three 256x256 matrices. All the generated hardware has been generated to satisfy the parallelism of the application but there is room for more task-specific cores should the application allow it and run-time system handle it.

Table V
RESOURCE UTILIZATION OF OUR GENERATED SoCs FOR THE DIFFERENT BENCHMARKS IN THE EVALUATION

Benchmark	Cores	Logic(ALMs)	BRAM(KBits)	DSP Block
Mandelbrot	48	77,364(33%)	2,636(5%)	256(100%)
PI	64	180,356(77%)	2,453(5%)	256(100%)
PrimeNum.	120	116,261(50%)	2,155(4%)	1(<1%)
BLAS L3	24	42,919(18%)	8,582(16%)	73(29%)

¹ Instead, the hardware designer is supposed to add custom instructions and configure GCC to use them

C. Parallel performance

The speedup (performance increase) for each benchmark when compared to our baseline processors is seen in Fig. 3, where the horizontal axis indicates the amount of task-specific cores used and the vertical axis indicates the performance increase we received when using our design flow. Comparing with the NiosII/s processor, our design flow can reach orders of magnitude of performance improvements. This superlinearity is attributed to the high performance differences between a NiosII/s processor core and our task-specific core, where the task-specific core is much faster. For the PI-kernel we reach up to 5130x (Fig. 3:b) faster than a single NiosII/s processor. One can argue that better performance can be obtained by using several NiosII/s processors [19], but each NiosII/s core consumes more space than a single task-specific core and is also slower. Fig. 3:c also shows the impact of the application-dependent properties; we see two performance curves labeled *fine-grained* and *coarse-grained*. These correspond to two different executions of the same benchmark with different amount of parallelism exposed. We found that sometimes, even if the amount of parallelism is high, the performance does not scale to the theoretical limit. For the fine-grained case, we expose parallelism that should theoretically scale to 2500 processors, and yet we only manage scaling up to 32 task-specific cores. However, reducing the amount of parallelism in the coarse-grained case, we manage to scale to 52 cores. The NiosII/s processor is simply too slow to deal out the work to the different task-specific cores. The task-specific cores are so-fast that when the NiosII/s processor wants to send a task to core 53, core 0 will already have finished execution and thus parallelism is lost. This scenario can also be seen with BLAS-3 Matrix multiplication in Fig. 3:e, but due to a different reason. Since the time taken to transfer the data (the three matrices) consumes more time compared to the computation on the task-specific core it is simply not possible to use all 24 cores. We also evaluated a locality-aware scheduler that keeps track of where data is located, prioritizing cores with the least amount of transfers (similar to HEFT [20]), and even forcing unused cores to work in order to reduce future transfers. The locality-aware scheduler performed even worse than transferring all data, attributed to the slowness of the NiosII/s. A way to overcome the data-transfers' cost is to integrate several Direct Memory Access (DMA) controllers into the system or change the memory hierarchy. This will be addressed in future work. We reach at the very minimum equivalent performance to one AMD Opteron 6172 core. Exceptions are the Prime-Number kernel (Fig.3:a) that reaches 4.6x performance and the PI-kernel (Fig. 3:d) that peaks at 7x faster. Note that we did not use the full FPGA resources, and we could have fitted many more task-specific cores on the chip if we would have altered the benchmarks to expose more parallelism.

Figure 3. Performance results for the generated systems compared (normalized) against one NiosII/s core and one AMD Opteron 6172 core in terms of execution time speedup. The horizontal axis has the number of task-specific cores that the application used while the vertical axis has the speed-up benefit (how much faster our system).

VI. CONCLUSION

We have demonstrated a novel design-flow that uses OpenMP to drive system generation using our HLS tool fpBLYSK. Any software programmer adept in parallel programming can use our methodology to receive a hardware platform that fits the target application's parallelism well without any prior knowledge of hardware design. We have experimentally shown that even at a low clock frequency of 50MHz, our system can be up-to seven times faster than a state of the art AMD Opteron 6172 core, and is several orders of magnitude faster than a single NiosII/s processor. The present study shows that any computer system should, in addition to GPUs, be equipped with a FPGA to fully take advantage of parallelism inside applications. A limitation found was to use the use of soft-cores for managing and exposing parallelism on-to the task-specific cores. Future work will investigate the possibility to generate hardware that exposes, schedules and synchronizes the parallelism of the application. Such an investigation could increase the performance of fine-grained parallelism.

VII. ACKNOWLEDGMENTS

Thanks goes to Altera for donating a Altera DE5 board to us. The author is a member of Scalable Computing Systems, SCALE. This work was funded by the Artemis PaPP Project nr. 295440.

REFERENCES

- [1] B. Van Essen, C. Macaraeg, M. Gokhale, and R. Prenger, "Accelerating a random forest classifier: Multi-core, GP-GPU, or FPGA?" in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE, 2012, pp. 232–239.
- [2] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. ACM, 2012, pp. 47–56.
- [3] R. Kalarot and J. Morris, "Comparison of fpga and gpu implementations of real-time stereo vision," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer Society Conference on*. IEEE, 2010, pp. 9–15.
- [4] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the non-determinism of reducers)," in *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*. ACM, 2010, pp. 303–314.
- [5] J. Choi, S. Brown, and J. Anderson, "From software threads to parallel hardware in high-level synthesis for FPGAs," in *Field-Programmable Technology (FPT), 2013 International Conference on*. IEEE, 2013, pp. 270–277.
- [6] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: high-level synthesis for FPGA-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2011, pp. 33–36.
- [7] Y. Leow, C. Ng, and W.-F. Wong, "Generating hardware from OpenMP programs," in *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*. IEEE, 2006, pp. 73–80.
- [8] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.
- [9] D. Cabrera, X. Martorell, G. Gaydadjiev, E. Ayguade, and D. Jiménez-González, "OpenMP extensions for FPGA Accelerators," in *Systems, Architectures, Modeling, and Simulation, 2009. SAMOS'09. International Symposium on*. IEEE, 2009, pp. 17–24.
- [10] A. Filgueras, E. Gil, D. Jimenez-Gonzalez, C. Alvarez, X. Martorell, J. Langer, J. Noguera, and K. Vissers, "OmpSs@Zynq All-programmable SoC Ecosystem," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA '14. ACM, 2014, pp. 137–146.
- [11] A. Tumeo, M. Monchiero, G. Palermo, F. Ferrandi, and D. Sciuto, "A design kit for a fully working shared memory multiprocessor on FPGA," in *Proceedings of the 17th ACM Great Lakes symposium on VLSI*. ACM, 2007, pp. 219–222.
- [12] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W.-M. Hwu, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *Application Specific Processors, 2009. SASP'09. IEEE 7th Symposium on*. IEEE, 2009, pp. 35–42.
- [13] K. Martin, C. Wolinski, K. Kuchcinski, A. Floch, and F. Charot, "Constraint-Driven Instructions Selection and Application Scheduling in the DURASE system," in *Proceedings of the 2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, ser. ASAP '09. IEEE Computer Society, 2009, pp. 145–152.
- [14] M. B. Gokhale and J. M. Stone, "NAPA C: Compiling for a hybrid RISC/FPGA architecture," in *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*. IEEE, 1998, pp. 126–135.
- [15] A. Duran, J. Corbalán, and E. Ayguadé, "An adaptive cut-off for task parallelism," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*. IEEE, 2008, pp. 1–11.
- [16] A. Muddukrishna, P. A. Jonsson, V. Vlassov, and M. Brorsson, "Characterizing task-based openmp programs," [urn:nbn:se:kth:diva-141201](http://nbn:se:kth:diva-141201), Tech. Rep., 2014.
- [17] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny, "QNoC: QoS architecture and design process for network on chip," *Journal of Systems Architecture*, vol. 50, no. 2, pp. 105–128, 2004.
- [18] T. Gautier, F. Lementec, V. Faucher, and B. Raffin, "X-Kaapi: a multi paradigm runtime for multicore architectures," in *Parallel Processing (ICPP), 2013 42nd International Conference on*. IEEE, 2013, pp. 728–735.
- [19] E. Fernandez-Alonso, D. Castells-Rufas, S. Risueño, J. Carrabina, and J. Joven, "A NoC-based multi-soft core with 16 cores," in *Electronics, Circuits, and Systems (ICECS), 2010 17th IEEE International Conference on*. IEEE, 2010, pp. 259–262.
- [20] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.