Exploiting Parallelism on GPUs and FPGAs with OmpSs

Jaume Bosch
Barcelona Supercomputing Center
Barcelona, Spain
jaume.bosch@bsc.es

Antonio Filgueras
Barcelona Supercomputing Center
Barcelona, Spain
antonio.filgueras@bsc.es

Miquel Vidal
Barcelona Supercomputing Center
Barcelona, Spain
miquel.vidal@bsc.es

Daniel Jimenez-Gonzalez
Universitat Politecnica de Catalunya
Barcelona, Spain
djimenez@ac.upc.edu

Carlos Alvarez
Universitat Politecnica de Catalunya
Barcelona, Spain
calvarez@ac.upc.edu

Xavier Martorell
Universitat Politecnica de Catalunya
Barcelona, Spain
xavim@ac.upc.edu

ABSTRACT

This paper presents the OmpSs approach to deal with heterogeneous programming on GPU and FPGA accelerators. The OmpSs programming model is based on the Mercurium compiler and the Nanos++ runtime. Applications are annotated with compiler directives specifying task-based parallelism. The Mercurium compiler transforms the code to exploit the parallelism in the SMP host cores, and also to spawn work on CUDA/OpenCL devices, and FPGA accelerators. For the CUDA/OpenCL devices, the programmer needs only to insert the annotations and provide the kernel function to be compiled by the native CUDA/OpenCL compiler. In the case of the FPGAs, OmpSs uses the High-Level Synthesis tools from FPGA vendors to generate the IP configurations for the FPGA. In this paper we present the performance obtained on the matrix multiply benchmark in the Xilinx Zynq Ultrascale+, as a result of using OmpSs on this benchmark.

KEYWORDS

OmpSs Programming Model, Parallelism, GPU, FPGA

ACM Reference Format:
Jaume Bosch, Antonio Filgueras, Miquel Vidal, Daniel Jimenez-Gonzalez, Carlos Alvarez, and Xavier Martorell. 2020. Exploiting Parallelism on GPUs and FPGAs with OmpSs. In *Proceedings of* . ACM, New York, NY, USA, 6 pages. https://doi.org/

1 INTRODUCTION

Current trends in computer architecture go in the direction of providing heterogeneous execution environments. Heterogeneity comes in many different flavors. In plain shared memory platforms, big/LITTLE cores provide some kind of heterogeneity, where the runtime scheduling policy should deal with the different performance of the cores.

More complex environments incorporate accelerators. Tipically, those provide specialized hardware to better execute specific algorithms. Probably the most common are GPGPUs (General-Purpose Graphics-Processing Units). Along with them, two main approaches for programming (among others) have been developed. OpenCL is the low-level standard specified by the Khronos group, supported by most of the GPGPUs currently available. CUDA is the approach pushed by NVIDIA on their GPGPUs.

Unfortunately, both models incorporate many details from the architecture of the accelerators onto the programming model. For example, given that the GPGPUs have their own memory address space, the models provide the means to implement the data transfers between where the data is (usually the main host memory), to and from where the data needs to be (for computation onto the GPGPU hardware). The additional memory space is needed in order to achieve performance when running code on the accelerators, and data movements are a must for them to work. Although GPGPUs can be implemented in a discrete way, using a PCIe bus connection, or integrated on the same die as the host cores, their behaviour is pretty similar, and both CUDA and OpenCL support such approaches.

More recently, FPGA (Field-Programmable Gate Array) devices have started reaching the same level of architectural support. FPGA devices are programmed by means of bitstreams, usually generated by vendor-propietary tools, following an specification provided in the VHDL or Verilog hardware description languages.

Discrete FPGA devices can be programmed over PCIe, providing additional acceleration functions to the main host cores. Usually, the FPGA device incorporates the implementation of the bus protocol as part of its programming. The programmer needs to be aware of it, as he/she should incorporate it in the bitstream generation process.

In the case of the FPGAs, there is an additional characteristic to be taken into account. Vendor compilation tools to generate the place and route to configure the FPGA usually take from minutes to hours. This causes that the porting of new code onto these platforms is usually a slower process than with GPGPUs.

Vendors also provide FPGAs integrated with a few cores, that can be used as the host cores. In this case, the FPGA shares the physical memory with the cores, as it was the case in the GPGPUs.

With the advent of the accelerators, the runtime execution environments provided on top of the operating system started to leak. There was the need to provide the programmer with new API interfaces that were not part of the common system libraries (C, PThreads), and in some cases, the user is granted

.

direct access to the hardware resources of the accelerator. For example, the user can directly allocate memory on the accelerator (global memory), or on the registers of the running threads (shared memory).

In our work, we try to make the programmers life easy, by providing higher-level abstractions that could help him/her to generate the proper high performance code on them. For exemple:

- Providing SIMD code generation based on directives, instead of having to use the low-level SIMD intrinsics.
- Making the memory allocation and data copies automatic, based on directives.
- Generating the code to run on the accelerator automatically, provided the C/C++ implementation, if C/C++ tools to generate CPU or FPGA code are available.
- Allowing the use of parallelism based on tasking (instead of kernel invocations).
- Providing support for data dependent tasks, and implementing the execution based on such data dependences.

This makes the programming environment to (hopefully) completelly hide the target architectures, providing a clean, high-level, abstract interface to the programmers, and incorporating all the intelligence on management and scheduling onto the runtime system.

2 PROGRAMMING WITH OMPSS

The OmpSs [2, 11] programming model allows to express parallelism that will be executed in the available resources among the host SMP cores, or integrated/discrete GPUs and/or FPGAs. OmpSs is based on task parallelism, and very similar to OpenMP tasking. It is being used as a forerunner prototyping environment for future OpenMP features. On GPUs, both CUDA and OpenCL kernels are supported. For FPGAs, OmpSs uses the vendor IP generation tools (Xilinx Vivado and Vivado HLS [5, 9], or Altera Quartus [1]), to generate the hardware configuration from high-level code. OmpSs can also leverage existing IP cores, provided they adhere to the same interface with our software platform.

2.1 The OmpSs compilation environments

The compilation environment supporting OmpSs@CUDA is presented in Figure 1. In this environment, our Mercurium compiler transforms calls annotated with *target task* directives onto a call to the Nanos++ runtime system to spawn a task for the GPU CUDA helper threads. The runtime system executes a stub function generated by the compiler to invoke the CUDA kernel. An example of code annotation is shown in the listing of Figure 2.

In a similar way, compilation of code with OpenCL is shown in Figure 3. One difference between the OpenCL environment and the CUDA one is that the CUDA kernels are compiled during the compilation phase. Instead, when using OpenCL, the kernel code is compiled at runtime, when the OpenCL kernels are invoked. Compilation is done using the OpenCL compiler infrastructure available from the OpenCL runtime library. Nevertheless, OmpSs also works on OpenCL

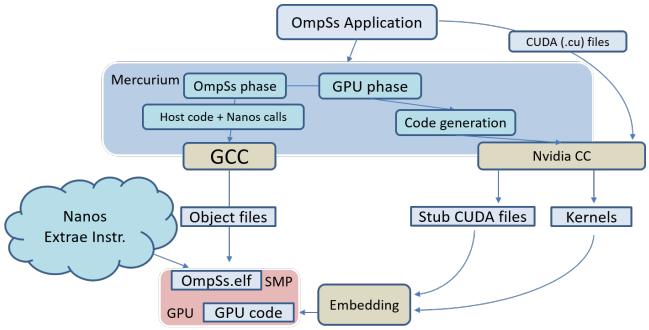

Figure 1: OmpSs compilation env. with CUDA support

```
#pragma omp target device(cuda) copy_deps ndrange( 1,n,128 )
#pragma omp task in([n]x) inout([n]y)
__global__ void saxpy(int n, float a,
                      float* x, float* y);

int main(int argc, char *argv[])
{
  ...
  // OmpSs task
  saxpy(N, a, x, y);

  ...
  #pragma omp taskwait
}
```

Figure 2: Sample OmpSs@CUDA kernel invocation

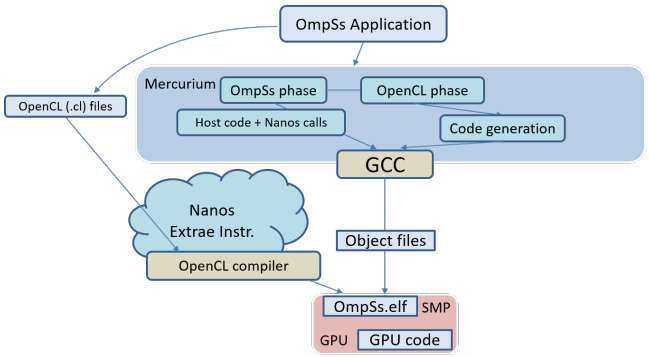

Figure 3: OmpSs compilation env. with OpenCL support

for FPGA devices. In this case, the kernels are compiled with the vendor tools (Xilinx SDAccel, Intel-Altera Quartus), and the binary code for FPGA configuration is available previously to the execution.

We have upgraded the OmpSs infrastructure to incorporate the support for Xilinx FPGAs using the Vivado HLS tool. Figure 5 shows the toolchain flow. The OmpSs application is split in two parts according to the OmpSs directives (see Figure 6). All functions annotated with the *target device(fpga)* directive are defined as tasks that will be transfered to the Vivado HLS

```
#pragma omp target device(opencl) copy_deps ndrange( 1,n,128 )
#pragma omp task in([n]x) inout([n]y)
__kernel void saxpy(int n, float a,
                    __global float* x, __global float* y);

int main(int argc, char *argv[])
{
  ...
  // OmpSs task
  saxpy(N, a, x, y);

  ...
  #pragma omp taskwait
}
```

Figure 4: Sample OmpSs@OpenCL kernel invocation

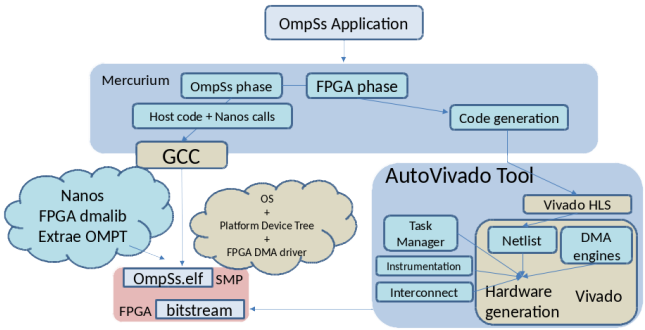

Figure 5: OmpSs compilation env. with FPGA support

tool for compilation to IP cores. Additionally, the Mercurium compiler generates the stub function used to invoke the IP cores from our Nanos++ runtime system, adapting the parameter passing. Vivado HLS transforms the stub functions and the FPGA-annotated functions onto VHDL, and the Vivado tool generates the IP cores. In the code example, the *onto* clause indicates to the compiler that the identification number of this IP core is zero (0). This information is used to generate a configuration file our of the compilation of the application, with the list of accelerators available in the FPGA. Additionally, the *num_instances* clause is used to express how many instances of the given IP core the programmer decides to generate to potentially run simultaneously this type of tasks. If the programmer uses a number larger than one, the runtime system will now through the configuration file that it will have as many instances available as the number indicated.

There is a further step to encapsulate the IP cores onto the bitstream that will be used for configuring the target FPGA. This step is left to the Petalinux tool from Xilinx, currently targeting the Zynq 7000 and Zynq Ultrascale+ chips.

Tasks can be annotated with the *implements(funcname)* clause, indicating that such task is a different version of the same algorithm that *funcname* implements. This allows the runtime system to select the *best* version to run at any given point in time. This is done by applying a scheduling policy that takes these alternative implementations into account.

```
#pragma omp target device(fpga) copy_deps \
            onto(0) num_instances(1)
#pragma omp task in( vec_a[0:CONST_BS-1], \
                    vec_b[0:CONST_BS-1]) \
                 out(vec_c[0:CONST_BS-1])
void vector_mult(int* vec_a, int* vec_b, int* vec_c)
{
  int i;
#pragma HLS ARRAY_PARTITION variable=vec_a complete
#pragma HLS ARRAY_PARTITION variable=vec_b complete
#pragma HLS ARRAY_PARTITION variable=vec_c complete

  for (i=0; i < CONST_BS; i++) {
#pragma HLS PIPELINE II=1
    vec_c[i] = vec_a[i] * vec_b[i];
  }
}
```

Figure 6: Vector multiply function targeting the FPGA

```
#pragma omp target device(smp) copy_deps \
            implements(vector_mult)
#pragma omp task in( vec_a[0:CONST_BS-1], \
                    vec_b[0:CONST_BS-1]) \
                 out(vec_c[0:CONST_BS-1])
void vector_mult_smp(int* vec_a, int* vec_b, int* vec_c)
{
  int i;

  for (i=0; i < CONST_BS; i++) {
    vec_c[i] = vec_a[i] * vec_b[i];
  }
}
```

Figure 7: Vector multiply function targeting the SMP cores

The *vector_mult* example can be completed with the SMP version of the task, to be executed on the SMP cores, as represented in Figure 7. Observe as, in the case of the FPGA devices, the SMP code can be the same as the *target device(fpga)* code, as the FPGA vendor tools understand the same type of code. In this case, the difference is that in the FPGA code we had the HLS directives passed to Vivado HLS for IP core generation, while in the SMP version there is no need of such directives. Potentially, the SMP version can be annotated with additional OmpSs directives, for example to distribute the iterations of the loop among the SMP cores, with an inner level of tasking.

In the case of targeting CUDA or OpenCL GPU devices, the kernel code to be executed on them has to be expressed in CUDA or OpenCL, as we do not have an OmpSs translator from C/C++ onto CUDA/OpenCL code.

Finally, Figure 8 shows how a normal procedure call from the program towards the *vect_mult* function gets invoked as a new task. This task is presented to the OmpSs runtime as being able to execute in both the SMP cores and the FPGA device. Then, the runtime system may apply a specific scheduling policy to decide which of the environments is more suitable to execute each particular instance of the task.

```
int vect_a[N];
int vect_b[N];
int vect_c[N];

int main(int argc, char * argv[ ])
{
  int i;
  ...
  // vector initialization
  ...

  for (i=0; i < N; i += BS) {
    vect_mult (&vect_a[i],
               &vect_b[i],
               &vect_c[i]);
  }
#pragma omp taskwait
}
```

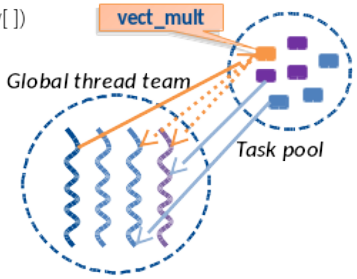

Figure 8: Invoking heterogeneous vector multiply task

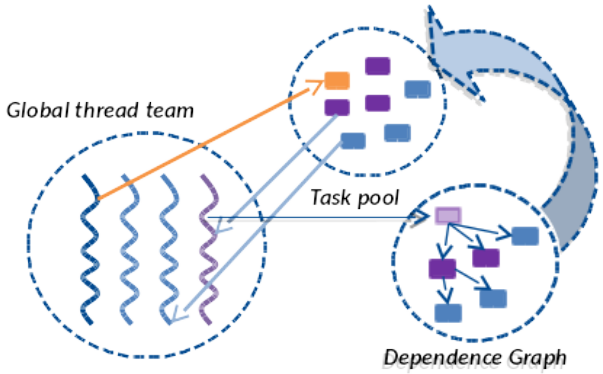

Figure 9: High-level representation of the Nanos++ execution environment

2.2 The OmpSs runtime system

Nanos++ is the OmpSs runtime system. It takes care of executing the tasks annotated by the programmer in the available resources. The high-level view of the execution environment is presented in Figure 9.

The Nanos++ environment has a *thread team* created by default, the *dependence graph* used to organize the tasks that still have pending data dependences to be resolved, and the *task pool* representing the task ready queue. Executing threads create tasks and insert them into the dependence graph. When data dependences have been fulfilled, the thread detecting this situation moves the tasks now free of dependences to the task pool. When a thread finishes the execution of a task, it becomes idle, and it searches for work in the task pool.

On the heterogeneous environments described in subsection 2.1, Nanos++ has a specific subset of threads that represent each of the heterogenous devices. We call these threads *helper threads*. The purple thread (thread number 4, on the right-hand side of the Global thread team) in the figure is one of those helper threads. In this particular example, it may represent one of a GPU using CUDA or OpenCL code, or an IP core in an FPGA.

Tasks annotated with the *implements* clause can be executed on an SMP core or on one ore more devices. This means that when the runtime system finds one of these tasks in the ready queue, it can be grabbed by a regular worker thread, that will execute the SMP version of the task in an SMP core. Or the task can be grabbed by one of the *helper threads*, and then the device version of that task will be executed in the device represented by the thread.

Characteristic	ZC706	AXIOM Board
Architecture	32-bit	64-bit
SMP Cores	2x ARM Cortex A9	4x ARM Cortex A53
SMP Freq.	666 MHz	1.5 GHz
Total memory	1 GByte	4 GBytes
FPGA	XC7Z045	ZU9EG

Table 1: Characteristics of the execution environments

3 EVALUATION

This section presents the evaluation of the OmpSs execution environment on Xilinx Zynq devices, with the Matrix Multiply benchmark.

3.1 Execution environments

The OmpSs execution environment for Xilinx Zynq FPGAs has been evaluated on two different Zynq platforms. The environment was first developed for the 32-bit Zedboard [3], and the Xilinx ZC702 [6] and ZC706 [7] evaluation kits. In this paper, we present results of Matrix Multiply on the ZC706 board.

We also ported OmpSs to the Zynq Ultrascale+ chip [8], and specifically to the AXIOM board [4]. Table 1 shows the main characteristics and differences of the two execution environments.

The matrix multiply benchmark implements the common sgemm [10] algorithm, to multiply input matrices A and B, accumulating the results on the in/out matrix C. Matrix size is set to 1024x1024 single precision floating point values, and the block size in which it is split is 128x128 elements. The matrices are represented as hypermatrices of (8x8) pointers to blocks of (128x128) consecutive elements. Having the matrices represented in this way improves the performance of the memory transfers to and from the FPGA device, and also improves the locality on the execution on the SMP cores.

3.2 Results

Figure 10 shows the results obtained. It shows the number of GFlops achieved by matrix multiplication in the various environments. On the left side of the graph, we have the

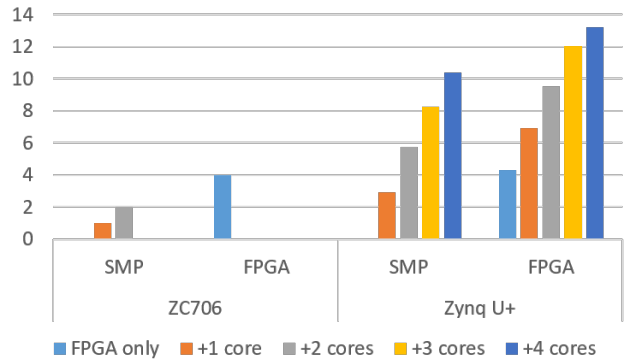

Figure 10: Evaluation of Matrix Multiply on the different OmpSs environments

results obtained in the ZC706 board. Overall, the performance obtained is 1 GFlop per ARM Cortex A9 core, and 4 GFlops on the FPGA. In this environment, the reduced number of cores does not allow to achieve additional advantage of using the *implements* feature to distribute the work among the FPGA and the cores. When doing so, the results are varying so much, and the performance is not better than that of the FPGA.

On the right-hand side, the results obtained in the AXIOM board (Zynq U+) are shown. In this case, we can observe that the plain use of SMP cores offers good scalability. The performance on a single core is already improving the performance of the SMP cores on the ZC706, reaching 2.9 GFlops. The SMP environment scales up to 10.3 GFlops obtained in 4 cores. The FPGA fabric behaves similarly to that of the ZC706, reaching 4.2 GFlops.

In the AXIOM board, we can appreciate the benefit of using the *implements* approach, in which each core added to the execution of matrix multiply adds up to 2.5 Gflops. The step from the execution on 3 cores to the execution on 4 cores is a little less (1.2 Gflops), because in this case the OmpSs environment is running 5 threads on 4 cores: there is the additional *helper thread* driving the FPGA, which causes a light system oversubscription, reducing the performance that the additional core is able to contribute into the results. Nevertheless, we think that the fact that we can exploit the heterogeneous resources of the Zynq Ultrascale+ to execute parts of the same application, achieving this scalability, is a very good result obtained from the OmpSs environment.

4 CONCLUSIONS AND FUTURE WORK

In this paper we have shown the OmpSs approach to exploit task-based parallelism in SMP cores and accelerator devices. OmpSs has traditionally supported directive-based parallel programming with GPU accelerators. Kernel codes for the GPUs are compiled with the native CUDA or OpenCL compilers. With the advent of the FPGAs, our approach is to support general-purpose parallel programming by the same technique. Configuration bitstreams for the FPGA are generated using the FPGA vendor tools, after our Mercurium compiler outlines the annotated code. This process is now automatic, without programmer intervention. We have evaluated this approach on the Xilinx Zynq 7045, and the Xilinx Zynq Ultrascale+ FPGAs. We have used a single-precision matrix multiplication benchmark to show that it is possible to obtain performance out of these FPGAs, by leveraging the IP cores generated by the Xilinx Vivado HLS toolchain. And, in addition, we have shown that it is possible to have the SMP cores and the FPGA collaborating in the execution of matrix multiplication, each contributing to increase performance. This has been demonstrated in the Zynq Ultrascale+ platform, by obtaining 4.29 GFlops out of the FPGA, 2.9 GFlops out of a single ARM A-53 core, and a total of 13.2 GFlops out of the FPGA and the 4 ARM Cortex A53 cores.

Our current work goes in the direction of making the OmpSs toolchain for FPGAs more stable, and covering a wider spectrum of FPGA devices. Also, we are currently working on incorporating additional benchmarks and applications exploiting the use of FPGAs, including Cholesky, Kmeans, face detection and audio analysis.

ACKNOWLEDGMENTS

This work is partially supported by the European Union H2020 program through the AXIOM project (grant ICT-01-2014 GA 645496) and HiPEAC (GA 687698), by the Spanish Government through Programa Severo Ochoa (SEV-2011-0067), by the Spanish Ministerio de Economia y Competitividad under contract Computacion de Altas Prestaciones VII (TIN2015-65316-P), and the Departament d'Innovacio, Universitats i Empresa de la Generalitat de Catalunya, under project MPEXPAR: Models de Programacio i Entorns d'Execucio Paral·lels (2014-SGR-1051).

REFERENCES

[1] Intel Corp. 2017. Quartus Prime. (2017). https://www.altera.com/products/design-software/fpga-design/quartus-prime/what-s-new.html

[2] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters* 21, 2 (2011), 173–193. https://doi.org/10.1142/S0129626411000151

[3] Avnet Inc. 2017. Zedboard. (September 2017). http://zedboard.org/product/zedboard

[4] SECO Inc. 2017. The AXIOM Board. (2017). http://www.axiom-project.eu/2017/02/the-axiom-board-has-arrived/

[5] Xilinx Inc. 2017. Vivado High-Level Synthesis. (2017). http://www.xilinx.com/hls

[6] Xilinx Inc. 2017. Xilinx Zynq-7000 All Programmable SoC ZC702 Evaluation Kit. (September 2017). https://www.xilinx.com/products/boards-and-kits/ek-z7-zc702-g.html

[7] Xilinx Inc. 2017. Xilinx Zynq-7000 All Programmable SoC ZC706 Evaluation Kit. (September 2017). https://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.html

[8] Xilinx Inc. 2017. Zynq Ultrascale+ MPSoC. (2017). https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html

[9] Stephen Neuendorffer and Fernando Martinez-Vallina. 2013. Building Zynq® Accelerators with Vivado® High Level Synthesis. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '13)*. ACM, New York, NY, USA, 1–2. https://doi.org/10.1145/2435264.2435266

[10] University of Tennessee. 2017. BLAS - Basic Linear Algebra Subprograms. (2017). http://www.netlib.org/blas/

[11] Florentino Sainz, Sergi Mateo, Vicenç Beltran, José Luis Bosque, Xavier Martorell, and Eduard Ayguadé. 2014. Leveraging OmpSs to Exploit Hardware Accelerators. In *26th IEEE International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2014, Paris, France, October 22-24, 2014*. 112–119. https://doi.org/10.1109/SBAC-PAD.2014.26