Automated Generation of Reconfigurable Systems-on-Chip by Interactive Code Transformations for High-Level Synthesis

Silvano Brugnoni[1], Thomas Corbat[1], Peter Sommerlad[1], Toni Suter[1],
Jens Korinth[2], David de la Chevallerie[2], Andreas Koch[2]

[1]: IFS Institute for Software, FHO HSR Hochschule für Technik Rapperswil, Switzerland
[2]: Embedded Systems and Applications Group (ESA), TU Darmstadt, Germany

Abstract

Despite the advances in high-level hardware synthesis (HLS), the programming style required by the design tools for generating efficient hardware implementations still differs significantly from that used in conventional software development. To ease the development of high-quality hardware accelerators using HLS, we propose the use of automated interactive source code transformations. Guided by the user, the transformations help to avoid much of the tedious and potentially error-prone manual code re-development process. The automation also takes aspects of the system-on-chip architecture into account, e.g., addressing data-movement and the creation of heterogeneous pools of processing elements, which can then be accessed in a multi-threaded manner from software. We demonstrate the technique targeting both reconfigurable systems-on-chip, as well as PCIe Gen3-attached compute platforms.

Despite the advances in high-level hardware synthesis (HLS), the programming style required by the design tools for generating efficient hardware implementations still differs significantly from that used in conventional software development. To ease the development of high-quality hardware accelerators using HLS, we propose the use of automated interactive source code transformations. Guided by the user, the transformations help to avoid much of the tedious and potentially error-prone manual code re-development process. The automation also takes aspects of the system-on-chip architecture into account, e.g., addressing data-movement and the creation of heterogeneous pools of processing elements, which can then be accessed in a multi-threaded manner from software. We demonstrate the technique targeting both reconfigurable systems-on-chip, as well as PCIe Gen3-attached compute platforms.

1 Introduction

The difficulties of increasing the performance of individual processors, combined with the scalability problems when attempting to employ many parallel processors to cooperatively execute an application, have led hardware architects to consider more *heterogeneous* architectures [1]. The scalability problems of homogeneous parallel processing can be alleviated here by being able to choose different accelerators that are especially suitable for specific parts of the application, e.g., dataflow, vector arithmetic, bit-manipulation. The potential gains in performance (and often also energy efficiency) are offset, however, by an increasing complexity of programming such systems. Different families of accelerators usually follow different programming paradigms, often requiring multiple programming languages. This is especially true for reconfigurable computing using FPGAs: While they offer the greatest flexibility for matching application needs, their programming often requires expert knowledge in topics such as computer architecture, digital logic design, and the use of special EDA tools; all fields generally unfamiliar to application software developers. To lower this barrier, *high-level synthesis (HLS)* tools [2] attempt to translate more abstract descriptions, usually conventional software programming languages such as C or C++, into hardware descriptions that can be mapped to FPGAs or ASICs. Despite these advances, HLS tools are not yet completely successful in relieving a software developer of the intricacies of hardware, as the required source code often has to obey numerous restrictions (e.g., with regard to pointer use), or has to be written in a stylized fashion to be compiled into efficient hardware [3]. Also, many HLS flows just create individual IP cores, and do not address the complex task of actually integrating these into a complete *systems-on-chip (SoC)*. Thus, even using HLS has so far required significant engineering effort to exploit the potential of reconfigurable computing. In this work, we present tool support to reduce this effort by two approaches: First, automated source code transformation techniques encapsulate expertise of transformations beneficial for HLS, and make them accessible to non-expert developers (Section 4). Second, these code

transformations are aware of the system-level aspects, and transform the code not just for HLS, but toward a portable tool flow for the automatic assembly of entire heterogeneous parallel SoC architectures, including data movement, operating system integration, and user-level APIs (Section 5).

2 Related Work

The compilation from high-level programming languages to hardware implementations has been the subject of intense study in both the academic as well as the commercial domains. Some recent academic tools are LegUp [4], Nymble [5], and PandA [6], while examples for commercial efforts include Catapult C [7], PICO [8], and Vivado HLS [9]. All of these tools attempt to translate from a subset of C/C++ to hardware blocks for implementation on ASICs and/or FPGAs. However, the compilers require additional tool support to actually integrate the generated blocks as usable accelerators into complete system-on-chip architectures.

Recent examples of such tools are CMOST [10], ReconOS [11], LEAP OS [12], MARC II [13], and the commercial tool SDSoC [14]. Both ReconOS and SDSoC exclusively target Xilinx Zynq-series FPGAs and provide no support for PCIe devices. LEAP OS supports several PCIe Gen2 devices, but neither PCIe Gen3 nor any current Xilinx FPGAs (i.e., 7-series). MARC II unifies accelerator memory interfaces and low-level control/status registers, but does not offer any higher-level abstractions. The approach most closely related to ours is the OpenCL-based CMOST, which also does not support PCIe Gen3 or Zynq devices. Furthermore, we use the ThreadPoolComposer [15] toolchain to evaluate the accelerators, which does not depend on availability of an OpenCL compiler for the target platform.

Winterstein *et al.* have published a number of tools [16] [17] that provide code transformations for enabling HLS of unsupported C language features, such as dynamic heap-allocated data structures. However, these tools do not provide extraction and generation of entire kernels. The transformation tools described in our paper have been designed to follow the usability guidelines established in [18] and [19].

3 Selected Requirements on C Code for HLS

Many current HLS tools raise the level of abstraction in the description of the behavior of complex hardware modules (compared to classical HDLs such as Verilog, VHDL) by providing support for C/C++ syntax. In theory, this allows any programs written using these languages to be synthesized for FPGA. However, several implicit and explicit assumptions about the system infrastructure and execution environment simply do not hold true in this context. Because of these principal differences, some language constructs exhibit different semantic behavior, and some others are not supported at all. In order to be amenable to high-level synthesis, program code must adhere to significant restrictions (cf. [3]): It must neither use any system calls, nor dynamic memory (`malloc`/`new`) and the source code must be available in its entirety, which excludes libraries in binary or pre-compiled form. Furthermore, all language constructs must be of fixed size and unambiguous implementation; this excludes, e.g., variable sized arrays and classes with virtual methods, as well as most implementations of common data structures (e.g., in the standard template library STL). Beyond that, several restrictions apply to the use of pointer types.

Software code that actually adheres to all of these restrictions is hard to find and it often requires significant effort to implement suitable workarounds in existing code. Each transformation presented in the following addresses one of the code restrictions and automates a workaround that yields more efficiently synthesizable C/C++ code: Non-array pointers that are not written to can be replaced by scalar values of the pointed-to type by the **Pointer Elimination** transformation; this yields a more efficient hardware interface, since no bus master interface is required for pass-by-value. The **Pointer to Array** transformation can convert array pointers in the function signature to corresponding fixed-size arrays; this enables Vivado HLS to correctly infer the address bit-width and synthesize bus master interface(s) for memory access. Also, Vivado HLS can synthesize efficient bus master interfaces that perform *burst transfers*, but only for calls to `memcpy`; all other accesses are implemented as individual transfers with the same bit-width as the array element type. Furthermore, Vivado HLS considers pointers in the function signature as *off-chip memories*, which limits parallel accesses. To remedy these problems, the **Memory Localization** transformation generates a *local buffer consisting of on-chip memories (BRAM, LUTs)* that is transferred via `memcpy` at the beginning and end of the kernel execution, which provides highly parallel access to arrays.

Vivado HLS requires *access to all code and data structures which are relevant to the kernel*, which can be difficult to achieve; unfortunately, Vivado HLS suffers from severe stability and performance issues when dealing with huge code bases. To alleviate this problem, the **Kernel Extraction** transformation isolates a user-selected portion of the code, determines all its dependencies, and moves the code and its dependencies into a separate self-contained translation unit, which can be used by Vivado HLS.

In the following section, we will illustrate these transformations in more detail, using a Sobel filter (cf. Listing 1) as a running example, which is inspired by Canis' discussion in [20].

```
struct image { uint8_t data[WIDTH * HEIGHT]; };
bool check_bounds(int32_t x, int32_t y) {...}
uint8_t bound(int32_t in_dir) {...}
int8_t const stencil_x[3][3] = ...;
int8_t const stencil_y[3][3] = ...;

void sobel(uint8_t const *in_image, uint8_t *out_image) {
 for (size_t y = 0; y < HEIGHT; y++) {
  for (size_t x = 0; x < WIDTH; x++) {
   if (check_bounds(x, y)) {
    int32_t x_dir = 0, y_dir = 0;
    for (int8_t x_os = -1; x_os <= 1; x_os++) {
     for (int8_t y_os = -1; y_os <= 1; y_os++) {
      int32_t img_i = (y + y_os) * WIDTH + x + x_os;
      uint8_t pixel = *(in_image + img_i);
      x_dir += pixel * stencil_x[1 + x_os][1 + y_os];
      y_dir += pixel * stencil_y[1 + x_os][1 + y_os];
     }
    }
    uint8_t edge_weight = bound(x_dir) + bound(y_dir);
    int32_t out_index = y * WIDTH + x;
    *(out_image + out_index) = 255 - edge_weight;
   }
  }
 }
}

int main() {
  image * images = ..., results = ...;
  for (size_t img_i = 0; img_i < NOF_IMAGES; img_i++) {
    sobel(images[img_i].data, results[img_i].data);
  }
}
```

Listing 1 Sobel filter example

4 C/C++ Code Transformations for HLS

In this section, we present the code transformations for HLS. For each transformation, we give a short statement of its intent, followed by an overview of the transformation actions, and conclude by illustrating the application of each transformation on the Sobel filter example. The transformations are implemented as plug-ins for the Eclipse-based Cevelop IDE [21], which provides a sophisticated analysis and transformation infrastructure for C++. They can be easily triggered by selecting the target source code element in the code editor and invoking the corresponding transformation command in the Cevelop graphical user interface. The Cevelop infrastructure, combined with the analyses that were specifically developed for the presented transformations, provides a platform that can be used to implement additional transformations for HLS.

4.1 Pointer Parameter Elimination

Most HLS tools are capable of dealing with a limited set of pointer and array types; e.g., by convention, Vivado HLS implements stack-allocated data in on-chip memory, whereas pointers indicate off-chip memory [3]. Access to the latter is usually implemented using a bus master interface, i.e., via address and data channels, or streaming. This approach is suitable for large amounts of data, but can incur prohibitive overhead for small, isolated pieces of data and limits parallelism. Unfortunately, even fundamental types are often passed via pointers in code intended to run on CPUs, since the dereferencing-overhead is negligible (for modern compilers). To remedy the adverse effect of such code on the resulting hardware circuits, we provide two transformations to remove pointer parameters:

Pointer Elimination replaces a pointer parameter by a corresponding value parameter.

1. Replace the pointer parameter declaration with a value parameter declaration:

   `void f(int *param)` ⇒ `void f(int param)`

2. Remove all pointer dereference expressions in the function body:

   `x = *param` ⇒ `x = param`

Note that side-effects on the data may be lost at the call site, since they are now local to the copied data in the kernel.

Pointer to Array Conversion replaces a pointer parameter by an array parameter. It is composed of three transformation actions:

1. Replace the pointer parameter declaration with an array parameter declaration:

   `void f(int *param)` ⇒ `void f(int param[SIZE])`

2. Replace all pointer dereference expressions with corresponding array access expressions:

   `y = *(param + x)` ⇒ `y = param[x]`

3. Introduce an offset variable for tracking modifications of the parameter, if necessary:

   ```
   x = *param++;   ⇒   int offset = 0;
                      x = param[offset++];
   ```

The results of this transformation when applied to the Sobel example are shown in Listing 2: The type of the parameters `input_image` and `output_image` is replaced in the function declarator. All array memory accesses are performed using the index operator. Introduction of an `offset` variable is not required because the pointers are not modified.

```
void sobel(uint8_t const in_image[WIDTH * HEIGHT],
           uint8_t out_image[WIDTH * HEIGHT]) {
      ...
      uint8_t pixel = in_image[img_i];
      ...
      out_image[out_index] = 255 - edge_weight;
}
```

Listing 2 Replacement for pointer parameter

4.2 Memory Localization

As mentioned earlier, the differentiation between on-chip and off-chip memories has significant impact on the performance of HLS hardware and a suitable trade-off is imperative. When dealing with large amounts of data, a common hardware design technique is to repeatedly fetch a small part of the data into a local buffer (using burst transfers, if available) and then operate locally on the buffer. In HLS,

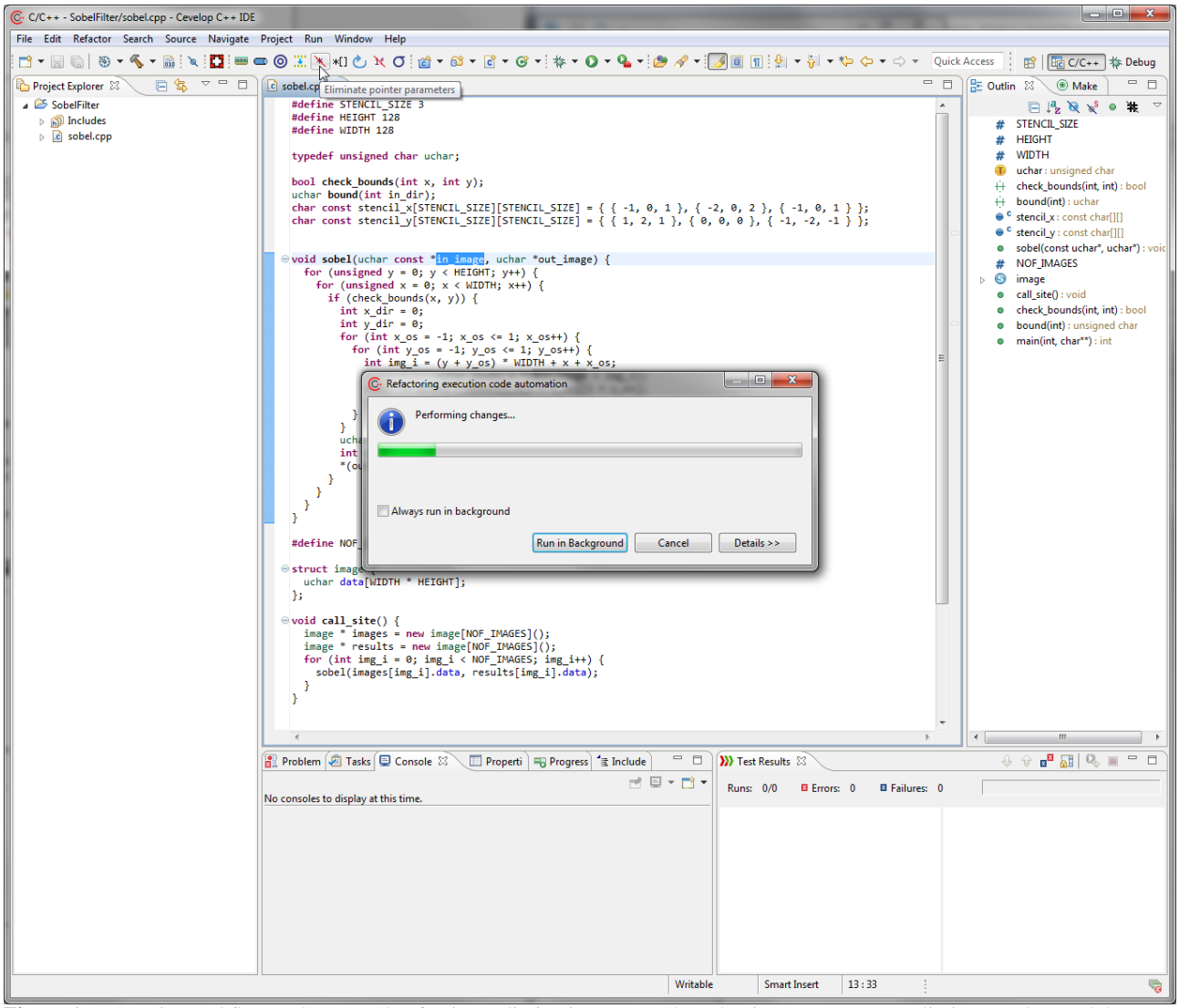

Figure 1 Interactive tool flow at the example of pointer elimination; User selected pointer parameter to eliminate and started the transformation with a single click.

this can be expressed by copying a pointer parameter to a stack allocated buffer variable; if `memcpy` is used to copy the data, Vivado HLS will also generate burst transfers [3]. The *Memory Localization* transformation can apply this technique automatically to pointer structures on the function interface. It is composed of three transformation actions:

1. Rename the parameter:

   ```
   void f(int x[5])    ⇒ void f(int param_x[5])
   ```

2. Add a local array with same name and size as the parameter prior to 1) to avoid further renaming.

3. Insert memory copy operations where necessary: For input parameters, copy its data to the newly introduced local array; for output parameters, copy the data back from the local array. Only parameters serving as input and output require both copy operations.

Example for actions 2 and 3:

```
void f(int param_x[5]) {
  int x[5];
  memcpy(x, param_x, sizeof(x)); // function start
  ...
  memcpy(param_x, x, sizeof(x)); // function end
}
```

The results of this transformation when applied to the Sobel example are shown in Listing 3. The data of `in_image` is copied into the local scope. Conversely, the data of `out_image` is copied back to the parameter, in order to retain the side-effect on the parameter.

```
void
sobel(uint8_t const param_in_image[WIDTH * HEIGHT],
      uint8_t param_out_image[WIDTH * HEIGHT]) {
 uint8_t out_image[WIDTH * HEIGHT];
 uint8_t in_image[WIDTH * HEIGHT];
 memcpy(in_image, param_in_image, sizeof(in_image));
 ...
 memcpy(param_out_image, out_image, sizeof(out_image));
}
```

Listing 3 Local copy of `in_image` and `out_image`

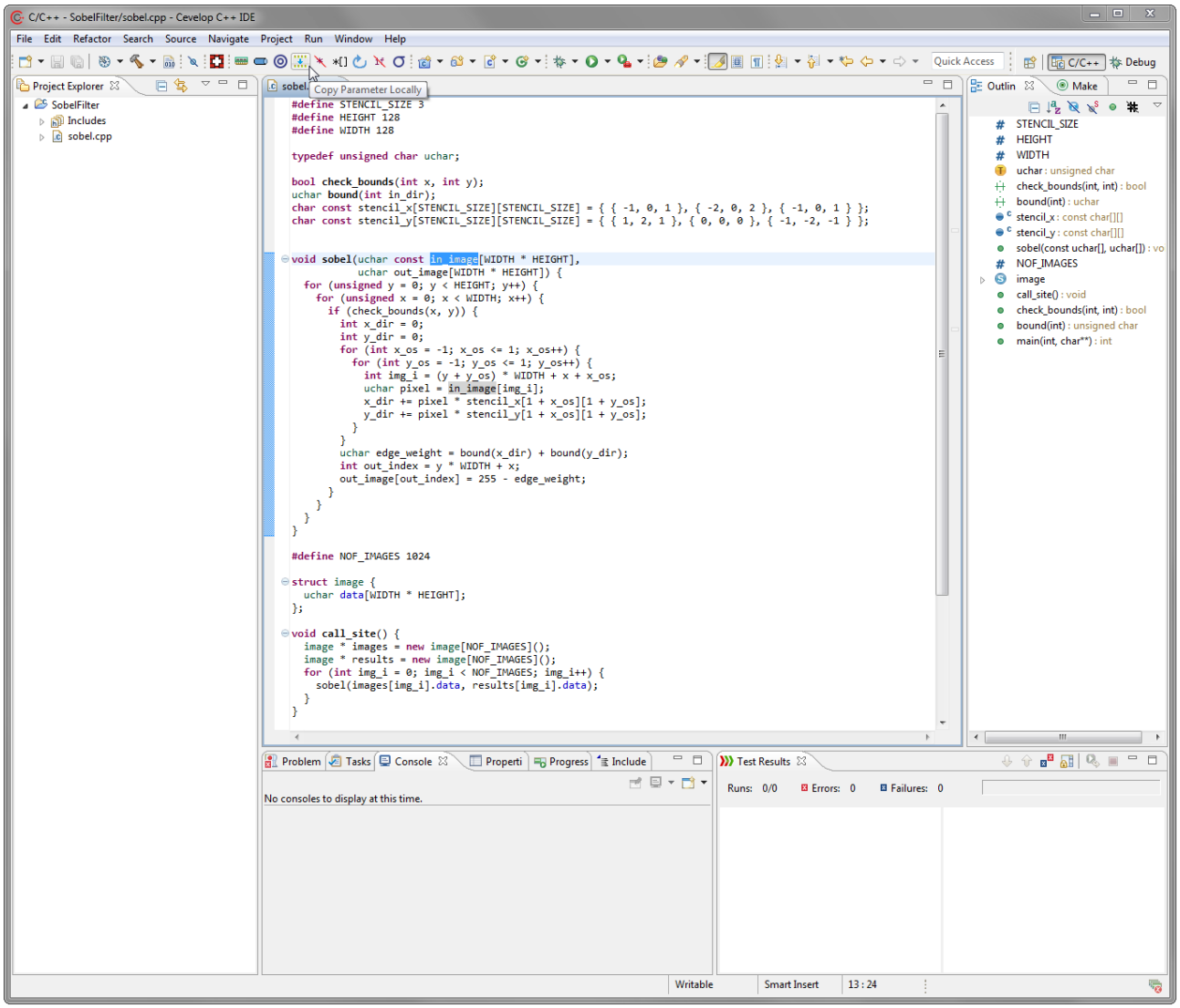

Figure 2 Interactive tool flow for memory localization; User selected a function parameter and can generate the local copy with a single click.

4.3 Kernel Extraction

Most traditional C/C++ code is not as neatly structured into cohesive kernel functions as in the Sobel example. Individual pieces of a particular computation are often scattered across several translation units, data structures, or classes. We provide a *Kernel Extraction* transformation, which automatically prepares specific program sections for HLS. When applied to a kernel function call, this transformation extracts the function body and all of its code dependencies, as well as any data pertaining to this particular function call, into a separate independent translation unit that is ready for synthesis. The extraction process encompasses the following steps:

1. Identify all data and code dependencies of the kernel function and copy them into a separate translation unit.
2. Generate the kernel interface and put it into the same translation unit as the data dependencies.
3. Generate the kernel computation logic, based on the target function, and put it into the translation unit.
4. Modify the function call site to use the newly generated kernel instead of the target function.

The interface of the extracted kernel consists of a data container and three functions:

- `kernel_data`: This struct serves as a container for all data that must be copied to and from the FPGA, to ensure that all data transfers happen as a single DMA operation.
- `prepare()`: This function stores the kernel arguments in the data container prior to the kernel execution.
- `retrieve()`: This function fetches the computation results from the data container after the kernel execution has finished.

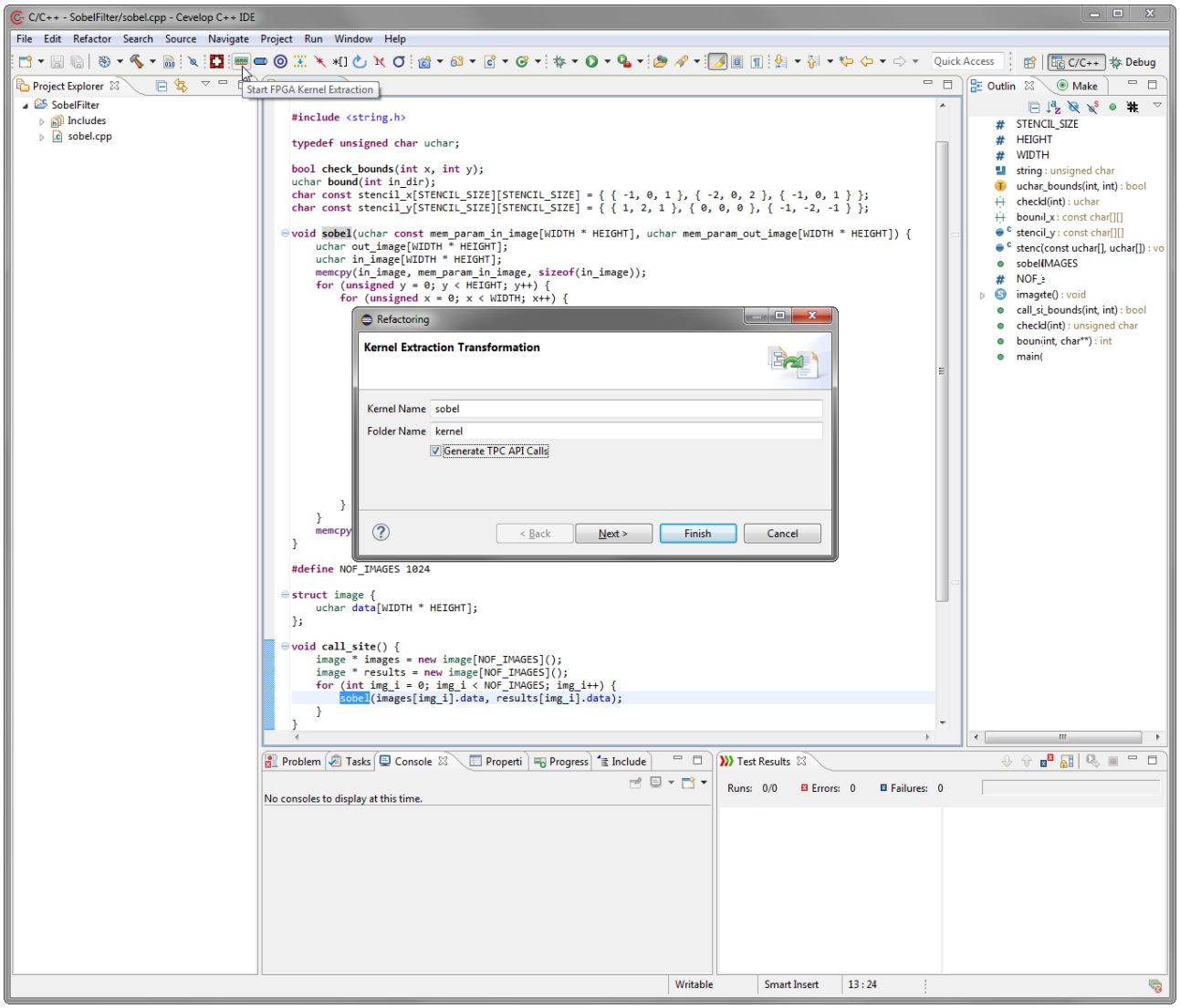

Figure 3 Interactive tool flow for kernel extraction; User selected a function call site (blue selection below) and with a single click the user can start the extraction of both code and data structures into separate compilation units, as well as generate the TPC API code that replaces the original call site code and executes on the FPGA instead.

- `apply()`: This function contains the computation logic that will be synthesized and executed on the FPGA.

The translation unit generated during the extraction consists of two files: a header containing the declarations of the kernel interface as well as the declarations of the kernel dependencies, and a source file containing the corresponding definitions.

In the following, we illustrate the results of the kernel extraction when applied to the Sobel filter example, that has been preprocessed using the previously describe transformations. Listing 4 shows the header of the extracted `sobel` kernel: The data container, `kernel_data_sobel`, contains the kernel input and output parameters, `in_image` and `out_image`. The parameters are copied to and fetched from the data container using `prepare` and `retrieve`, respectively. In addition, the header contains the declarations of any types, macros, typedefs, and constants, which the kernel depends on.

```
extern const int8_t stencil_x[3][3];
extern const int8_t stencil_y[3][3];
struct kernel_data_sobel
{
  uint8_t in_image[WIDTH * HEIGHT];
  uint8_t out_image[WIDTH * HEIGHT];
};

inline void
prepare(kernel_data_sobel &data,
        const uint8_t in_image[WIDTH * HEIGHT]) {
  memcpy(data.in_image, in_image,
         sizeof(data.in_image));
}

void apply(kernel_data_sobel &data);

inline void
retrieve(kernel_data_sobel &data,
         uint8_t out_image[WIDTH * HEIGHT]) {
  memcpy(out_image, data.out_image,
         sizeof(data.out_image));
}
```

Listing 4 Extracted kernel header - `kernel_sobel.h`

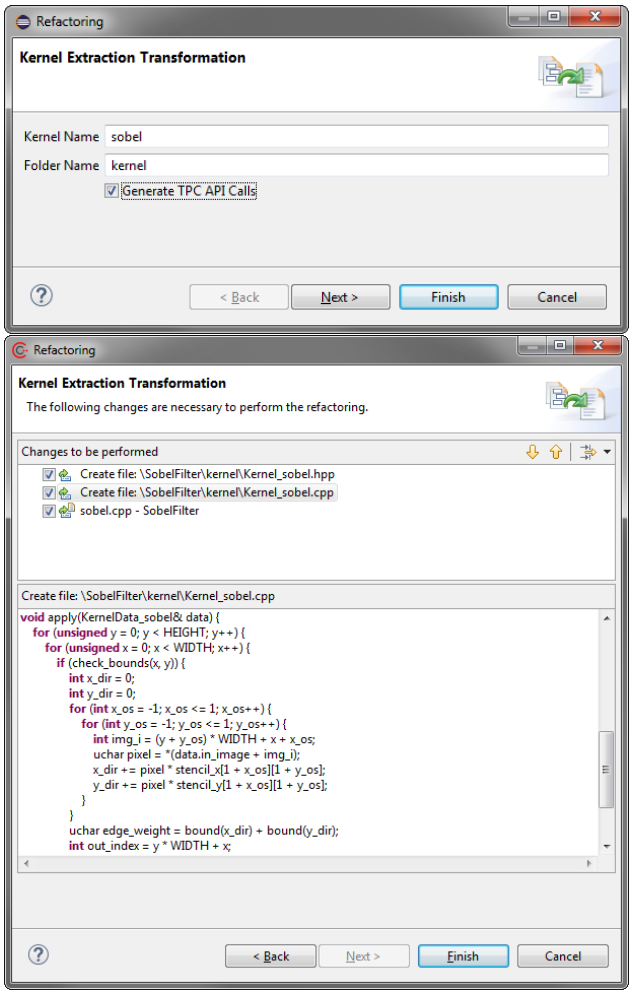

Figure 4 Kernel extraction separates the extracted code from the original code base; the user can select both name of the kernel, as well as the location of the new source files containing code and data definitions.

Listing 5 shows the source file of the extracted Sobel kernel. It contains the definition of `apply()`, an adapted version of the `sobel` function, where the parameters are replaced with the corresponding fields of the data container. In addition, the source file contains the definitions of the functions `check_bounds()` and `bound()`, which the kernel depends on.

```
const int8_t stencil_x[3][3] = ...;
const int8_t stencil_y[3][3] = ...;
bool check_bounds(int32_t x, int32_t y) {...}
uint8_t bound(int32_t in) {...}

void apply(kernel_data_sobel &data) {
  uint8_t out_image[WIDTH * HEIGHT];
  uint8_t in_image[WIDTH * HEIGHT];
  memcpy(in_image, data.in_image,
         sizeof(in_image));
  ...
  memcpy(data.out_image, out_image,
         sizeof(out_image));
}
```

Listing 5 Extracted kernel source - `kernel_sobel.cpp`

The modified call site is shown in Listing 6. The `sobel` function invocation is replaced by calls to the kernel interface, which prepare the data, run the computation, and retrieve its results. Furthermore, the *Kernel Extraction* also generates calls into the TPC API (the software API of ThreadPoolComposer, see Sec. 5) to launch the kernel on an accelerator.

```
int main() {
  ...
  FOO foo;
  for (size_t img_i = 0;
       img_i < NOF_IMAGES;
       img_i++) {
   kernel_data_sobel data;
   prepare(data, images[img_i].data);
   foo.launch(0, &data);
   retrieve(data, results[img_i].data);
  }
}
```

Listing 6 Kernel extraction call site

5 System Environment for Accelerators

A major advantage of the HLS-based approach to hardware development is significantly increased productivity, and thus reduced time to the first working prototype. This time can be further reduced by using an automatic compilation flow that generates the system-on-chip architecture required to organise multiple instances of the hardware modules generated by HLS tools. In the following, we use the ThreadPoolComposer [15] compilation flow and system infrastructure to demonstrate this approach and evaluate the transformations presented in previous sections.

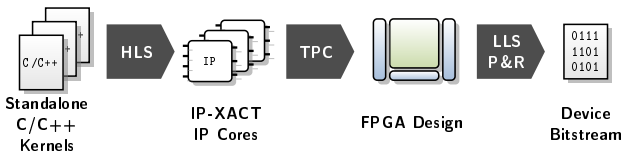

Figure 5 ThreadPoolComposer Compilation Flow

The overall flow is depicted in Fig. 5: Starting from a set of *standalone C/C++ kernels* which have been prepared for HLS using the software transformations described in the previous sections, ThreadPoolComposer performs HLS in batch mode to generate behaviorally equivalent IP Cores. In the next step, ThreadPoolComposer composes a complete design in two phases: First, the Architecture instantiates the HLS IP cores (each instance is called Processing Element (PE) in the following) and organizes them into a ThreadPool (cf. Fig. 6); the Architecture is the *device-independent part* of the design, which can be re-used with different Platforms. In the second phase, the chosen Platform instantiates *device-dependent infrastructure* and connects the ThreadPool with the host system; i.e., it provides the host with control access to the ThreadPool registers, the ThreadPool with access to device memory (if required), and a basic signaling interface from the ThreadPool to the host to notify of completed tasks. The resulting design is a complete system-on-chip, which is synthesized into a device

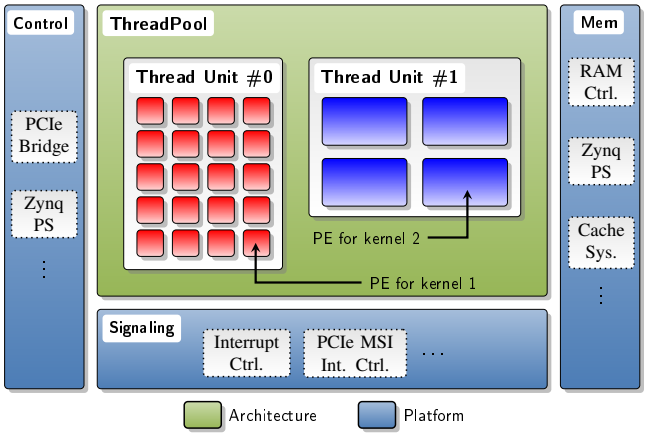

Figure 6 FPGA Design Organization

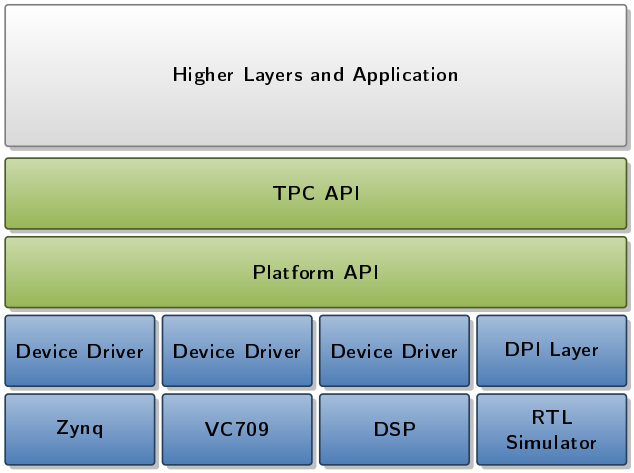

Figure 7 ThreadPoolComposer Stack

bitstream by the vendor's tool chain. This entire process is automatic; the user can provide a Composition, i.e., specify how many PEs of which kernels shall be instantiated.

Based on this Composition, ThreadPoolComposer can perform basic *design space exploration (DSE)*, where the design space is defined by *number of PEs* on the one hand, and *design operating frequency* on the other. During the HLS step the IP cores generated for each of the kernels are evaluated regarding their maximal frequency, as well as their area utilization. The initial composition (specified by the user) determines the overall distribution of kernel PEs, which is kept (approximately) constant. On the area axis, discrete steps can then be generated by enumerating all compositions with the same distribution of PEs ranging from at least one PE per kernel up to a platform-defined utilization limit (e.g., 70% Slice LUTs for PEs). The steps on the frequency axis are determined by the hardware, i.e., the oscillators and their multipliers on the device determine which frequencies can be generated. This design space is currently ordered by a very simplistic throughput metric: On each platform we measured the average time for writing two control registers and the average interrupt response time, i.e., the time between the hardware raising the interrupt and user space being notified. In sum, this yields the minimal *kernel execution overhead*. During the HLS step, kernel runtimes are evaluated either statically (if runtime is independent of the input) or dynamically (based on a user-specified example input). Putting it all together, we can now compute an estimate of the number of jobs per second given the design frequency and number of parallel PEs for each composition/frequency pair and use it as an idealized metric M to order the design space:

$$M = \sum_{i=1}^{K} \frac{N_i}{t_{min} + \frac{t_i}{F} + t_{int}}$$

where K is the number of different kernels, N_i is the number of PEs for kernel i, t_i is the runtime of kernel i in clock cycles, F is the design frequency, t_{min} is the start time (i.e., writing two control registers) and t_{int} is the interrupt response time. ThreadPoolComposer will iterate through the design space ordered by M and stop at the first composition/frequency pair that achieves timing closure. This simple DSE process has been used in the following evaluation.

ThreadPoolComposer provides a *two-level software stack* (see Figure 7) to close the remaining gap to the application: The Platform API library manages the Platform parts of the design; it provides abstractions for *device memory management*, *signaling* and *low-level access* to the hardware via the device driver. Both the device driver and the Platform API need only be implemented once per Platform. Above this layer resides the TPC API, which is the user-facing, high-level API the application uses to launch jobs on the accelerator (in style reminiscent of OpenCL, but much more lightweight). A code example can be found in [15].

ThreadPoolComposer currently supports *three different Platforms* for the following FPGA boards: The **zedboard** is Zynq-7000 series SoC board featuring a Xilinx XC7Z020(-1) FPGA and a dual core ARM Cortex A9, the **ZC706** is a larger version of the same system with a XC7Z045(-2) FPGA; finally, the **VC709** is a PCIe Gen3 device with a XC7VX690T(-2).

6 Evaluation

In this section, we evaluate the Memory Localization transformation (see Section 4.2) using the running example of the Sobel filter in three steps: First, we observe the results of our source code transformations on the high-level synthesis results. Second, we observe the resulting effect on the designs which can be built automatically by ThreadPoolComposer using its DSE mode in terms of number of PEs (∼ area) and design frequency. Finally, we evaluate the real-world performance of the designs by comparing the wall-clock runtimes of a multi-threaded benchmark program.

Note that Vivado HLS does not use *pipelining* by default.

Variant	F	Latency	$\propto$	BRAM
sobel	291	3256367	1.0×	0
sobel-ml	304	1176895	2.8×	16
sobel-p	288	291879	11.2×	0
sobel-p-ml	298	65831	49.5×	16

Table 1 High-Level Synthesis results for Sobel variants (ml = w/memory localization, p = pipelined) on ZC706: Latency in clock cycles, F = est. max. frequency in MHz.

As the Sobel algorithm is known to benefit significantly from pipelining, we also provide additional results with explicitly activated pipelining (indicated by the suffixes).

6.1 Environment

The *zedboard* is an embedded system that features a Xilinx Zynq-7000 SoC with a dual ARM Cortex A9 running at 666 MHz and a FPGA frequency of up to 100 Mhz. Xilinx' *ZC706* evaluation kit is a larger version of the same system, with the CPU running at 800 MHz and FPGA frequency of up to 250 Mhz. Finally, the *VC709* is a 8x PCIe Gen3 device that has been evaluated with an Intel Xeon E5 1620v2 host CPU @3.7 Ghz (3.9 GHz TurboBoost); maximum fabric frequency is also 250 MHz. All three evaluation Platforms are running Linux 3.19, the examples were compiled with `gcc 4.9.2`, both for the `x86_64` and the `armv71` targets, Vivado Design Suite 2015.2 was used for synthesis.

6.2 HLS Results

Table 1 shows the high-level synthesis results of the Memory Localization transformation in terms of latency for the ZC706 (the results for zedboard and VC709 are very similar and have been omitted for brevity). Compared to the original version without pipelining, the buffers require $< 2\%$ of the available BRAM18K resources; however, this yields an improvement by 2.8× in terms of latency. When applied to the pipelined version, the effect is more dramatic, yielding a 49.5× improvement compared to the original version, and 4.4× compared to the pipelined version without Memory Localization.

6.3 Full Design Synthesis Results

To quantify the effect of these changes on a complete design, we used ThreadPoolComposer [15] in DSE mode using the baseline Architecture, which uses only off-the-shelf Xilinx AXI infrastructure IP. The results are shown in Table 2, where the last column indicates the percentage of area used for the PEs only vs. overhead for communication and infrastructure. Zynq designs scale well to large numbers of PEs, all designs use $> 50\%$ of the total area for the PEs. On the VC709, multiple clock-domains between PCIe, DDR and user logic produce a significantly larger overhead. Furthermore, the total area usage is below optimal, which is due to the current ThreadPoolComposer limit of 64 PEs.

6.4 Performance Results

Finally, Table 3 shows the wall-clock runtimes of a multi-threaded benchmark program working on random data. Speedups range from 2.3× up to 377.6× (with PCIe-based VC709 benefiting most from the aggregated data transfers) compared to the IP core resulting from the original source code, which confirms that Memory Localization is a highly useful, portable optimization for C/C++ HLS code.

7 Conclusion

We have shown that automated transformations on the source code level are useful to accelerate the development of hardware kernels using HLS, and that HLS-specific optimizations on this level are portable and can significantly improve the overall performance. Implementing common hardware design techniques as repeatable, interactive transformations is a promising approach to tackle the severe difficulties HLS tools are facing and to reduce the gap between software code and hardware acceleration. In addition to the running example discussed above, the use of the automated user-guided source code-transformations in HLS-based design flows has already proven beneficial in trial use at the industrial partners of [22]. This also holds true for ThreadPoolComposer, which can not only be used to create and access FPGA-based accelerators in a portable manner, but also even more heterogeneous architectures encompassing many-core processors and DSPs.

Availability of Tools: ThreadPoolComposer has been released as open-source and is currently available in the *Downloads* section of `www.esa.cs.tu-darmstadt.de`. The automated source code transformations described are implemented as custom extensions to the Cevelop framework, available at [21].

Acknowledgment: This work is part of the Reengineering and Enabling Performance and poweR of Applications project (ICT-609666) [22], funded by the Seventh Framework Programme (FP7) of the European Union. The authors would also like to thank Xilinx, Inc. for supporting their work by hardware and software donations.

8 Literature

[1] R. Iyer and D. Tullsen, "Heterogeneous computing," *IEEE Micro*, no. 4, 2015.

[2] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, 2011.

zedboard	*N*	*F*	*U*	*L*	O (% Slice LUTs)
sobel	15	100	88.4%	52.8%	**52.8** **35.5**
sobel-ml	16	50	90.9%	53.4%	**53.4** **37.5**
sobel-p	13	50	79.0%	47.5%	**47.5** **31.5**
sobel-p-ml	15	50	88.5%	52.7%	**52.7** **35.8**
ZC706	*N*	*F*	*U*	*L*	**O (% Slice LUTs)**
sobel	64	167	86.7%	55.1%	**55.1** **31.6**
sobel-ml	64	167	86.9%	55.1%	**55.1** **31.8**
sobel-p	60	167	86.9%	57.3%	**57.3** **29.7**
sobel-p-ml	64	125	86.7%	54.5%	**54.5** **32.2**
VC709	*N*	*F*	*U*	*L*	**O (% Slice LUTs)**
sobel	63	100	69.2%	22.2%	**22.2** **47**
sobel-ml	64	100	70.9%	23.3%	**23.3** **47.6**
sobel-p	62	100	71.7%	25.3%	**25.3** **46.4**
sobel-p-ml	64	100	70.9%	23.3%	**23.3** **47.7**

Table 2 Synthesis results using automatic DSE mode (*N* = number of PEs, *F* = design frequency (MHz), *U* = Total Utilization (Slice LUTs), *L* = Utilization (Slice LUTs) PEs only, O = rel. platform + comm. overhead).

Kernel	zedboard	∝	ZC706	∝	VC709	∝
sobel	24421	1.0	15606	1.0	12839	1.0
sobel-ml	10815	2.3	575	27.1	192	66.9
sobel-p	6830	3.6	3920	4.0	11489	1.1
sobel-p-ml	721	33.9	190	82.1	34	377.6

Table 3 Wall-clock runtime of multi-threaded benchmark in *ms*.

[3] *Vivado Design Suite User Guide: High-Level Synthesis, Version 2015.2*, Xilinx Inc., 2015.

[4] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. S. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An Open Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems," *Embedded Computing Systems (TECS), ACM Transactions on*, 2013.

[5] J. Huthmann, B. Liebig, and A. Koch, "Hardware/software co-compilation with the Nymble system," *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 8th Int. Workshop on*, 2013.

[6] V. G. Castellana and F. Ferrandi, "An automated flow for the high level synthesis of coarse grained parallel applications," in *Field-Programmable Technology (FPT), 2013 Int. Conf. on*, 2013.

[7] T. Bollaert, "Catapult synthesis: a practical introduction to interactive c synthesis," in *High-Level Synthesis*, 2008.

[8] V. Kathail, S. Aditya, R. Schreiber, B. R. Rau, D. C. Cronquist, and M. Sivaraman, "Pico: automatically designing custom computers," *Computer*, vol. 35, no. 9, 2002.

[9] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, "Autopilot: A platform-based esl synthesis system," in *High-Level Synthesis*, 2008.

[10] P. Zhang, M. Huang, B. Xiao, H. Huang, and J. Cong, "Cmost: A system-level fpga compilation framework," in *Proceedings of the 52Nd Annual Design Automation Conference*, 2015.

[11] A. Agne, M. Happe, A. Keller, E. Lubbers, B. Plattner, M. Platzner, and C. Plessl, "ReconOS: An operating system approach for reconfigurable computing," *IEEE Micro*, 2014.

[12] K. Fleming, H.-J. Yang, M. Adler, and J. Emer, "The leap fpga operating system," in *Field Programmable Logic and Applications (FPL), 2014 24th Int. Conf. on*. IEEE, 2014.

[13] H. Lange, T. Wink, and A. Koch, "MARC II: A Parametrized Speculative Multi-Ported Memory Subsystem for Reconfigurable Computers," in *DATE'11*, 2011, pp. 1352–1357.

[14] *SDSoC User Guide: Platforms and Libraries*, Xilinx Inc., 2015.

[15] Jens Korinth and David de la Chevallerie and Andreas Koch, "ThreadPoolComposer - An Open-Source FPGA Toolchain for Software Developers," *CoRR*, vol. abs/1508.06821, 2015.

[16] F. Winterstein, S. Bayliss, and G. Constantinides, "Separation logic-assisted code transformations for efficient high-level synthesis," in

Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on, 2014.

[17] ——, "High-level synthesis of dynamic data structures: A case study using Vivado HLS," in *Field-Programmable Technology (FPT), 2013 International Conference on*, 2013.

[18] P. Sommerlad, G. Zgraggen, T. Corbat, and L. Felber, "Retaining comments when refactoring code," in *Companion to the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, 2008.

[19] E. Murphy-Hill, "Improving usability of refactoring tools," in *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, 2006.

[20] A. C. Canis, "LegUp: Open-Source High-Level Synthesis Research Framework," Ph.D. dissertation, University of Toronto, 2015.

[21] Institute for Software, "Cevelop IDE," 2014–2015. [Online]. Available: http://www.cevelop.com

[22] "REPARA - Reengineering and Enabling Performance and poweR of Applications," 2015. [Online]. Available: http://www.repara-project.eu