Heterogeneous Multicore Computing: Challenges And Opportunities

Experiences From The hArtes Project

Koen Bertels, Vlad-Mihai Sima and Georgi Kuzmanov
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
http://ce.et.tudelft.nl

Abstract—This paper discusses the different challenges that were encountered during the hArtes project and how those challenges were met. A key objective of hArtes was to find the best mapping of an application on a particular heterogeneous hardware platform. The mapping process involves determining what parts of the application should be executed by what hardware component. When viewed in isolation, kernel based acceleration can produce significant speedups. However, when mapping the entire application, this potential never seems to live up to its full potential, due to other concerns than mere Amdahl's law. Many problems have to do with communication bottlenecks. As hArtes always used sequential C-code as the starting point, finding enough parallelism in those applications was also one of the other limitations limiting overall performance improvement. Nevertheless, the hArtes project successfully addressed many of these problems resulting in a toolchain that assists the developer in mapping code on heterogeneous multicore computing platforms. The speedups obtained, measured for the whole applications, are between 1.94 and 31, compared to pure software execution.

Keywords: heterogeneous, multicore, reconfigurable hardware, toolchain, embedded

1. Introduction

The hArtes project addresses the development of embedded systems. It investigates hardware/software integration and its main objective was to develop an integrated toolchain that provides (semi-)automatic support for the entire HW/SW co-design process. The applications used as input were written in various high-level algorithm descriptions, and, using the toolchain, a semi automatic "best fit" mapping to the platform was generated. The toolchain was intended to provide a fast development trajectory from application coding to the design of a reconfigurable embedded computing system. Having a (semi-)automatic process meant that a constant idea was to allow developers experiment with different solutions.

How hArtes addresses the problems can be summarized as follows:

- Improved mapping and allocation algorithms were developed to be able to deal with non-uniform memory accesses and specific communications bottlenecks.
- OpenMP compliance allows to express and exploit parallelism while respecting the sequential consistency paradigm.
- Specific code optimizations such as loop unrolling or load balancing were developed to provide improved performance.

The flow is described, and the information flow is detailed. The hardware platform is presented next, by describing the high-level organization of the platform and the components used. The applications, used to validate the whole approach, are presented in Section 4.

The remainder of this paper is organized as follows. Section 2 presents the Molen programming paradigm and the Molen abstraction layer, which represents the foundations on which the hArtes toolchains are build. Section 3 presents all the tools involved in the hArtes tool-chain. Section 4 presents the application analysis.

In Section 5 a detailed description of the problems is presented. In the same section, we describe, for each problem, the solutions that were applied in the context of the project. We summarize the conclusions in Section 6.

2. Molen Programming Paradigm

The hArtes design approach assumes the Molen architecture and programming paradigm. The Molen programming paradigm targets machines that adhere to the Molen machine organization [11]. The Molen machine organization is based on the processor-coprocessor model and allows the processor to control the execution of the coprocessor by a set of fixed primitives. By using these primitives, any number of operations can be implemented as accelerated components. Started as an extension for reconfigurable architectures, this machine organization can be used for any heterogeneous architecture like the hArtes hardware platform. For each targeted architecture, different mechanism can be used to implement the Molen primitives; nonetheless, this will not reduce the generality of the approach. This is achieved by developing and implementing a hardware abstraction layer.
The Molen programming paradigm is based on the sequential consistency model that allows multiple processing elements to act as coprocessors to a General Purpose Processor (GPP) [12]. The paradigm defines five programming primitives that have to be implemented by a platform, together with their semantics. The five primitives are SET, EXECUTE, MOVET, MOVEF and BREAK.

The SET Primitive
The SET primitive function is to start the configuration of the processing element with the operation that has to be executed. This will represent different actions for different processing elements. For a Field Programmable Gate Array (FPGA), it invokes the partial reconfiguration. For a Digital Signal Processor (DSP) processor, it represents the loading of an executable file into memory. This operation can take a different amount of time depending of the characteristics of the processing elements and the specific operation. Having a separate primitive to manage the configuration allows the compiler to perform scheduling of reconfigurations or loading. An appropriate scheduling would be able to hide the configuration time, by making the configuration in parallel with other useful computations.

The MOVET, MOVEF Primitives
The role of the MOVET and MOVEF primitives’ is to send (MOVET) and receive (MOVEF) parameters to/from the processing element.

The EXECUTE Primitive
From the programmer’s perspective, the EXECUTE primitive will start the execution of a Custom Computing Unit (CCU). This operation can be done asynchronously. The semantic is that the operation will start on a processing element, while the execution continues on the GPP. It is the responsibility of the programmer to check for the status of the execution, using the BREAK primitive.

The BREAK Primitive
The role of the BREAK primitive is to synchronize the execution of the GPP with the execution of the FPGA. This primitive semantic is that the GPP will be stalled until the execution of the FPGA finishes.

In Section 5 we will discuss how these primitives were modified and extended in the context of the hArtes project.

Source Annotations
Molen programming paradigm can be used with any compilation flow that can partition an application. The partition is done between GPP and the other processing elements. The C language was chosen as it is one of the most used languages in the embedded system world. The structure of the C language was analyzed, and it was decided that functions offer the necessary abstraction level to model computations that run on processing elements. To specify which functions will be mapped to other processing elements, the standard language extension mechanism of C was used, namely, pragmas. Two types of pragmas were introduced as it can be seen in Figure 1:

- On a function declaration. The semantic is that all calls to that function will be translated to Molen primitives. The Molen backend compiler will provide one implementation for the corresponding processing element.
- On a function call. The semantic is that the call will be replaced by the corresponding Molen primitives. The Molen backend compiler will provide an implementation for the corresponding processing element. This allows more optimization opportunities, as different implementations can be generated for that function, for example, taking into account constant function parameters.

The information in the pragma includes:
- the name of the processing element to which the computation will be mapped.
- the implementation identifier. This makes a connection between implementations and the token (function declaration or function call) to which the pragma applies.

3. hArtes Toolchain And Hardware Platform
The hArtes toolchain main objective is to support the entire process of mapping an existing application onto a specific reconfigurable heterogeneous system. The application has to be described at a high algorithmic level or in a high level programming language. For the algorithm description, the developer has the choice of using GUI based tools or
specific signal processing oriented languages as long as the final output is ANSI C.

The toolchain is composed of several toolboxes, which provide different functionalities and together realize the main objective. A schematic representation of the toolchain is depicted in Figure 2.

The high level design alternatives (a graphic entry, a signal processing or computation oriented language, or just general purpose C language) are offered by the hArtes AET (Algorithm Exploration and Translation) toolbox. For the graphical part, NU-Tech was adopted as Graphical Algorithm Exploration (GAE) solution and Scilab as a computation-oriented language. NU-Tech [7] is a platform that supports the development of algorithms for real-time scenarios, emphasizing strict timing control. Scilab is a free and open source software for numerical computations, similar to Matlab [5].

The output from the Algorithm Exploration Toolbox will be processed by hArmonic which will partition the application and decide one one specific mapping. At each step, the application developer has full control over the process.

The tools available in the hArtes Framework are performing the following tasks:

- Automated partitioning of the high-level algorithm descriptions. Using a set of predetermined design criteria and information about available resources, the high-level algorithms are divided into tasks.
- Transformation of the high-level algorithm tasks. This includes, for example, transformation to make tasks compatible with specific processing elements, like the FPGA.
- Design space exploration. Exploring the potential mapping possibilities between the tasks available and the processing elements of the reconfigurable heterogeneous systems. This is done by using estimated or measured costs for each task.
- Code manipulations. This includes, for example, scheduling of lengthy operations like the reconfiguration.
- Mapping and generation of the code for the targeted GPPs and DSPs.

The final step invokes the backend tools that will produce the different branches for the different hardware components. Those tools are:

- VHSIC Hardware Description Danguage (VHDL) code generation, for the target FPGA.
- Synthesis of the VHDL code obtained, using vendor-specific tools.
- Compilation of C code for the GPP processors present in the system.
- Integration of all binary code generated and running on the platform.

The hArtes Hardware Platform (hHP) was designed in such a way that it represents a reference target for the hArtes toolchain, while also providing computing resources for several high-performance applications in the audio domain. We present, briefly, the main characteristics of this platform.

Following the Molen Architectural template, the hHP consists of a GPP, several co-processing units. As many of the hArtes applications were audio oriented, special care was given to audio input and outputs from the platform.

The basic independent block of the system includes an ARM processor, a DSP processor (ATMEL Magic) and an application-specific reconfigurable block (Xilinx Virtex4 FPGA). This hardware block is called in hArtes terminology a Basic Configurable Element (BCE). In case these resources are insufficient for an application, an extension mechanism was designed, which would allow multiple BCEs to be connected.

For the actual hardware implementation, two BCEs were put on the hArtes board. The general organization of the hHP is shown in Figure 3. The current board contain two BCEs each, but multiple BCE could be chained if needed. The BCE are independent of one another, and can run any selected thread or application mapped by the hArtes toolchain.

For massive data streaming, dedicated hardware is configured on the board, represented by the "Audio I/O subsystem" block in Figure 4. Eight input and eight output Alesis Digital Audio Tape (ADAT) Lightpipe interfaces are available on the board. The ADAT Lightpipe is a standard for transfer of digital audio that uses fiber optic cables and has Toslink connectors at either end.
4.1 Profiling

For this application, optimizations are written directly in assembly for various architectures. As we will see, this affects also the application structure and, because of that, an automated toolchain has to do more work to uncover the real structure.

We chose to run the encoder with the following default parameters: no B frames, subpixel motion estimation and partition decision quality of 5 on a scale from 1 (fast) to 7 (best), one reference frame, integer pixel motion estimation method hex. The profile information might differ when using other parameters. This analysis represents the basis for optimizing the application.

In order to profile any application and obtain useful information for the hArtes toolchain at least one thing is needed, namely, transforming macro definitions to function calls. As this is a very specific optimization, we performed it inlining optimization active will skew the result. Specifically for the x264 application another source transformation was needed, namely, transforming macro definitions to function calls. As this is a very specific optimization, we performed

4.1.1 Profiling

In this section we describe the process for mapping the hArtes applications to the available resources. The hArtes project developed the following strategy on which the hArtes toolchain is based:

- Identify from the profile information, hot-spots, which take a significant portion of the application execution time.
- Select a function or only a fraction of a code, for mapping to another processing element.
- Profile the function on the processing element.

The function profiling on a different processing element is affected by the way in which memory is allocated. If the data does not reside in the processing’s element local memory, then a transfer will be needed between main memory and the local memory. This transfer can represent a significant overhead that might negate the speedup obtained.

We focus in this paper on FPGA based mapping, but the framework also supports function mapping on the DSP.

In this section we describe the process for mapping the real world application provided by the individual partners of the project.

To give an idea of the complexity of the applications, the number of files and total lines of code are given in Table 1.

<table>
<thead>
<tr>
<th>Application</th>
<th>Number of files (headers)</th>
<th>Total number of lines (in headers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>x264</td>
<td>4565(38167)</td>
<td>38167(4565)</td>
</tr>
<tr>
<td>wfs</td>
<td>53(76)</td>
<td>76(18)</td>
</tr>
<tr>
<td>incar</td>
<td>16(2860)</td>
<td>2860(510)</td>
</tr>
<tr>
<td>tcf</td>
<td>16(30)</td>
<td>30(16)</td>
</tr>
</tbody>
</table>

4. Video applications - H.264 codec

H.264 is the standard for video compression and decompression, developed jointly by ITU-T and ISO/IEC (with the name MPEG-4 AVC) [6]. To use this standard, the industrial partners in the project decided to use the free software library implementation, namely x264 [13].

4.1 Video applications - H.264 codec

To give an idea of the complexity of the applications, the number of files and total lines of code are given in Table 1.

4.1.1 Profiling

For this application, optimizations are written directly in assembly for various architectures. As we will see, this affects also the application structure and, because of that, an automated toolchain has to do more work to uncover the real structure.

We chose to run the encoder with the following default parameters: no B frames, subpixel motion estimation and partition decision quality of 5 on a scale from 1 (fast) to 7 (best), one reference frame, integer pixel motion estimation method hex. The profile information might differ when using other parameters. This analysis represents the basis for optimizing the application.

In order to profile any application and obtain useful information for the hArtes toolchain at least one thing is needed, namely, transforming macro definitions to function calls. As this is a very specific optimization, we performed
it manually and didn’t integrate it in the toolchain. The final profiling information obtained after these modifications is show in Figure 5. We used input data videos from a repository for freely-redistributable test sequences [14].

4.1.2 Mapping

Using the profiling information, it was decided to map the two most CPU intensive functions to the FPGA. The first step was to evaluate the speedup obtained for each kernel. Not all kernels give a speedup when moved to another processing elements. So, even if, they are high in the profile list, that does not mean they should be mapped to the FPGA.

The steps performed to obtain the speedup obtained by mapping a kernel on a computation element, are:

- Create, using the processing compiler, in our case Delft Workbench Automated Reconfigurable VHDL Generator (DW ARV), an FPGA implementation for both pixel_sad_wxh (referred from now, simply as SAD) and pixel_satd_wxh (referred from now, simply as SATD).
- Instrument the application to obtain all the possible data sets used by the kernels.
- Run the kernels using the data collected, using either local or shared memory.

The last step is necessary in order to determine the communication overhead.

After we performed the instrumentation, we observed that the kernel was called with some parameter combinations more often. This affected the execution time and the size of the data inputs. As an example, we show the results obtained for SATD kernel in Table 2. For example the case s1=16, s2=16 lx=16, ly=16 is show in Figure 5. We used input data videos from a repository for freely-redistributable test sequences [14].

### Table 2: Number of calls for each combination of parameters, and percentages from total number of invocations for SATD when running on more videos

<table>
<thead>
<tr>
<th>Video</th>
<th>s1=16, s2=8 lx=4, ly=4</th>
<th>s1=16, s2=16 lx=8, ly=8</th>
<th>s1=32, s2=16 lx=4, ly=4</th>
<th>s1=16, s2=8 lx=8, ly=8</th>
<th>s1=16, s2=16 lx=16, ly=16</th>
<th>s1=32, s2=16 lx=8, ly=8</th>
<th>s1=16, s2=8 lx=8, ly=4</th>
<th>s1=16, s2=240 lx=8, ly=8</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>1.30</td>
<td>1.64</td>
<td>1.23</td>
<td>2.28</td>
<td>2.78</td>
<td>3.72</td>
<td>0.72</td>
<td>0.92</td>
</tr>
<tr>
<td>seconds</td>
<td>309.32</td>
<td>304.85</td>
<td>301.07</td>
<td>304.13</td>
<td>304.85</td>
<td>304.13</td>
<td>304.13</td>
<td>304.13</td>
</tr>
<tr>
<td>calls</td>
<td>4.47</td>
<td>4.64</td>
<td>5.08</td>
<td>5.04</td>
<td>5.04</td>
<td>5.04</td>
<td>4.78</td>
<td>4.78</td>
</tr>
<tr>
<td>ref</td>
<td>121344040</td>
<td>123456040</td>
<td>123456040</td>
<td>123456040</td>
<td>123456040</td>
<td>123456040</td>
<td>123456040</td>
<td>123456040</td>
</tr>
<tr>
<td>ms/call</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>ms/call</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

### Table 3: Processing times and speedups in various scenarios for SATD unrolled

<table>
<thead>
<tr>
<th>Case</th>
<th>Times µs</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA</td>
<td>FPGA with transfer</td>
<td>ARM vs FPGA</td>
</tr>
<tr>
<td>s1=16, s2=8 lx=4, ly=4</td>
<td>9</td>
<td>114</td>
</tr>
<tr>
<td>s1=16, s2=16 lx=8, ly=8</td>
<td>9</td>
<td>141</td>
</tr>
<tr>
<td>s1=32, s2=16 lx=4, ly=4</td>
<td>8</td>
<td>137</td>
</tr>
<tr>
<td>s1=16, s2=8 lx=8, ly=8</td>
<td>11</td>
<td>140</td>
</tr>
<tr>
<td>s1=16, s2=16 lx=16, ly=16</td>
<td>19</td>
<td>156</td>
</tr>
<tr>
<td>s1=32, s2=16 lx=8, ly=8</td>
<td>9</td>
<td>144</td>
</tr>
<tr>
<td>s1=16, s2=8 lx=8, ly=4</td>
<td>8</td>
<td>135</td>
</tr>
<tr>
<td>s1=16, s2=240 lx=8, ly=8</td>
<td>8</td>
<td>139</td>
</tr>
<tr>
<td>s1=16, s2=240 lx=8, ly=8</td>
<td>8</td>
<td>196</td>
</tr>
<tr>
<td>s1=16, s2=16 lx=16, ly=16</td>
<td>17</td>
<td>165</td>
</tr>
<tr>
<td>s1=16, s2=16 lx=8, ly=16</td>
<td>14</td>
<td>152</td>
</tr>
<tr>
<td>s1=16, s2=16 lx=16, ly=8</td>
<td>12</td>
<td>143</td>
</tr>
<tr>
<td>s1=16, s2=240 lx=16, ly=8</td>
<td>12</td>
<td>197</td>
</tr>
<tr>
<td>s1=16, s2=240 lx=16, ly=16</td>
<td>17</td>
<td>272</td>
</tr>
<tr>
<td>s1=16, s2=240 lx=8, ly=16</td>
<td>12</td>
<td>270</td>
</tr>
</tbody>
</table>

iterations count) while the best is obtained for a large number of iterations (when lx and ly are 16).

When taking into account the transfer of memory performed between the ARM and the FPGA, the overall execution time is greater on the FPGA as can be seen from column 3. This is due to a extremely inefficient transfer speed between the main memory and the local FPGA memory.

4.1.3 Conclusion

From analyzing this application we can see that just taking the profiling information is not sufficient as code transformation might give a skewed view over which functions take most of the application execution time. More than that, bulk profiling information is not sufficient in all the cases. Rather, a way of detecting the correlation between parameters should be available in an instrumentation tool to help the designer
making the best decision when optimizing. We refer such a method in Section 5.

For this application, because of the memory transfer overhead, the FPGA is slower than the GPP. Optimizations that reduce this overhead are needed in order to be able to use the FPGA for a faster application execution. We present one such optimization in 5.1

4.2 Immersive audio - Beamforming and Wave-field Synthesis

In this section, we will discuss the implementation of a multi-beam, broadband beamforming and wave-field synthesis (WFS) [4]. These audio algorithms can be used in audio-visual transmission scenario like a telepresence application. A camera will tracks the recording directions (for example, by tracking human faces in the recorded scene) and using the beamformer algorithm the sound waves will be filtered based on its direction and sent through a transmission medium to the rendering side. There, the spatio-temporal properties of the recording space will be reproduced using a wave-field synthesis algorithm. Compared to other stereophonic approaches, the properties that are reproduced are not limited to a sweet spot, but to a much wider area, depending on the number of speakers array.

4.2.1 Profiling

This application is simpler than the x264 application and does not contain any assembly optimization. Most of the computations performed are floating point, which is a significant drawback for the simple ARM processor embedded in the Artes platform, which does not have a FPU unit.

The application will be executed with the default parameters: the sampling frequency (48 khz), the number of sources (2), the room size (15m). By profiling the application, we observe that 80% of the time is spent in the main function, _fFD_RealFIR_Pair_fpga_ (referred from now simply as FFD).

4.2.2 Mapping

After using DWARV to generate the VHDL, synthesizing and implementing, we see that the area occupied by FFD kernel is 25% of the available area on the FPGA. This is due partly to the extensive use of floating point units (8 units, including addition, multiplication and division).

The FFD kernel will be called with a constant processing window size of 1024. We tested more window sizes. The results are presented in Table 4. We can see that when the data is located in the local FPGA memory the speedup is between 5.99 and 6.43 and, if a data transfer is needed, the speedup is between 3.38 and 4.27.

<table>
<thead>
<tr>
<th>Size</th>
<th>FPGA</th>
<th>FPGA with transfer</th>
<th>ARM</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>2047</td>
<td>3898</td>
<td>13169</td>
<td>6.43</td>
</tr>
<tr>
<td>512</td>
<td>4410</td>
<td>6965</td>
<td>26435</td>
<td>5.99</td>
</tr>
<tr>
<td>1024</td>
<td>9370</td>
<td>13671</td>
<td>58407</td>
<td>6.23</td>
</tr>
</tbody>
</table>

4.2.3 Conclusion

Floating point operations are one of the most computational intensive operations of an application. Still, the

4.3 In Car Audio - Enhanced Listening Experience

The focus of this application is enhancing the listening experience for travellers in a car. This is challenging due to the inherent dynamic nature of the environment with a lot of outside noise. The noises, spatial and spectral properties of the reproduced field, change the rendering of the sound from the loudspeakers. The system used as an example here is a real-time application with two objectives:

- to develop a complete set of audio algorithms for improving the audio quality, taking into account some features of the cabin.
- to have a modular system that can be adapted to other environments or that can use other algorithms

4.3.1 Profiling

Rather than profiling, the developer of the application suggested the kernel to be accelerated, namely FRACSHIFT.

4.3.2 Mapping

The kernel FRACSHIFT is represented by a delay line, followed by a floating point addition and accumulation.

As given in Table 5, we distinguish again between the speedups obtained with and without taking memory transfers into account. When viewed in isolation, the FPGA is up to 1.73 times faster then the ARM. This reduces to maximum 1.69 when including the memory transfers.

We also constructed an experiment to assess to what extent loop unroll and scalar replacement could provide additional speedups. To this purpose we manually implemented a design that fully unrolled two of the loops, partially unrolled the main computation loops and used a local array to store one of the parameters during the function execution. From
Table 5 we can see that the optimized version performs 4.25 times better than the unoptimized version, including the transfer time needed to transfer the data to and from the FPGA. The price for the gain in performance is an area increase of 7.3 times in slices.

When using the full capabilities of the platform (including the DSP processor and the two BCEs) the total speedup obtained is 31x compared to executing just on the GPPs of each BCE.

4.3.3 Conclusion

A similar conclusion as in the previous cases can be drawn. Taking the memory transfers into account limits the overall speedups. The use of specific optimizations such as loop unrolling can substantially improve the performance.

5. Identified Problems and Proposed Solutions

On the basis of the applications mapping experience we now discuss in a more generic way the different issues that were encountered and for which solutions had to be found. We distinguish below platform related issues, toolchain related issues and application related issue.

5.1 Platform related issues

Problem: The hardware platform developed for the hArtes project was designed for audio processing. The initial design decisions were taken considering that the FPGA will be used for the audio processing. For this reason, communications resources were allocated to the path between audio I/O and the FPGA, as the FPGA is the natural choice for processing a large number of audio streams. The connection between the FPGA and the ARM or the main memory was not considered to be important. The implication is that there is no Direct Memory Access (DMA) available when transferring between main memory and the FPGA local memory. Even if the absence of DMA is acceptable for reading or the writing of audio data, this limits the efficiency of the FPGA when dealing with generic applications or applications that have their input data stored somewhere else than in dedicated buffers.

This is not a core architecture problem, but rather a design decision, determined by the objective of the hardware board and its imagined uses. In order to evaluate the memory speed, we performed a series of experiments.

Since the DMA was not available to access the FPGA memory, all the transfers between the main memory and the FPGA local memory had to be performed by the GPP. Table 6 illustrates the results obtained when transferring a block of 32 kb between various memories. As we can see, the transfer between SDRAM and FPGA Scrach Pad Memory (SPM) is 2.85 times slower than transferring between Synchronous dynamic random access memory (SDRAM) and SDRAM.

Experiments have shown that the bandwidth has the same value for both larger and smaller memory block sizes.

Solution: There are two solutions to solve this problem:

- adapt the mapping algorithm to map computations to the FPGA only if a speedup would be obtained. In order to be able to predict the speedup, the mapping algorithm needs as input the memory needed for each function. If possible the decision is taken at compile time, but, for the other cases a runtime solution was developed. The runtime solutions relies on runtime profiling information to assess which is the best mapping, and it is described in [9].

- reduce the transfer between SDRAM and the SPM. This can be accomplished by determining the type of the memory (write only, read only, read write), but also by improving the allocation algorithm, as some data might be accessed only from the FPGA. If this is the case, the data could reside directly in the SPM.

Problem: In order to support floating point operations, a floating point library of cores has to be available to the DWARV hardware compiler.

Solution: Xilinx provides floating point cores, but they can not be used directly by DWARV. The reason is that there are several differences between the Xilinx FP cores and the standard ones supported by C. One example of such difference is the rounding mode - truncation is used by C standard, rounding is used by Xilinx tools. VHDL wrappers were developed for each floating point operation, that made the Xilinx cores comply to the C standard.

5.2 Toolchain related issues

By studying the applications presented in Section 4, we identified mismatches between the assumptions made when Molen was designed and the existing applications. These mismatches are summarized in Table 7.

We give below more extensive description of the problems and solutions.

hArtes implementation of the Molen programming paradigm: While the Molen machine can be seen as an ideal machine organization, when building a real platform,
choices have to be made resulting in a less than optimal physical implementation. Besides the functionality described in Section 2, other features are needed for a full fledged implementation, like profiling, automatic memory transfer, debug functionalities.

**Solution:** We extended the existing primitives and created the Molen Abstraction Layer (MAL). Its role is to abstract even further the details of the platform. Given the organization of the FPGA in the hHP the SET primitive will perform a full reconfiguration of the reconfigurable part.

As the processor is not tightly coupled with the FPGA (in fact, it is physically on another board) the EXECUTE and BREAK primitives are implemented using memory mapped control registers of the Molen controller mapped on the FPGA.

The debug and profiling facilities are offered at the level of the primitives. MOVET, MOVEF, EXECUTE and BREAK primitives have included facilities to profile the time spent in each of them. At the end of the program execution, all the information is dumped in a special profile result file.

For debugging purposes, MOVET and MOVEF dump the memory contents used by the kernel, before and after its invocation. This allows the developer to check the correct functioning of the processing element, by comparing execution results on the GPP with the results obtained from executing on the FPGA.

**Memory transfer and allocation problem:** One of the objectives of hArtes was to facilitate an automated mapping to different processing elements using the shared memory paradigm. In the hArtes implementation, the GPP has access to all the memory of each processing element, but the processing elements only have access to their local memory. In case the local memory of each processing element is not enough for all the data it needs to process, transfers have to be performed to and from the local memory.

**Solution:** As the hardware configuration was given, we will discuss only the software solutions developed to address this problem:

- the memory is allocated to the local memory processing element memory. This solution is described in [9].
- the runtime library will manage the transfer automatically, when certain data are needed by a specific processing element.

We will describe the extensions implemented for the second solution. Even if it is a more straightforward, unoptimized approach, that does not perform any significant analysis or optimization, it is a good starting point that enables an application to be executed on our platform. Also, based on this implementation various optimizations can be performed in later stages.

Two new primitives are introduced, to differentiate the MOVET or MOVEF primitives for the special case of a pointer:

- molen_MOVETXaddr - used for each pointer parameter sent to a processing element, instead of the normal molen_MOVET
- molen_MOVEFXaddr - used for each pointer parameter sent to a processing element, after the kernel finished the execution

These two primitives work together with the support of the runtime system. All the dynamic memory allocation will be made by special 'wrapper' functions that keep track of allocations. The replacement of the 'malloc' functions by the wrapper function 'hmalloc' (and similar functions, i.e. 'realloc' and 'calloc') is made by the source to source transformation tool. Then, at a later point in the execution of the program, for each address that is sent as a parameter to the CCU, the runtime system can determine the size of the block to which that address belongs.

Naturally this approach has several limits:

- except in parameters, addresses should not be present in the memory blocks used by the kernels. This means, for example, structures like linked list will not be supported. Although this is a limitation, none of the kernels analysed used such complex data structures.
- without further analysis or information from the developer the transfer can be inefficient. For example, some memory blocks are only written by the kernel while others are only read. The C language provides some information about this (example: const keyword), but it is incomplete (ex: there is no way to specify write only locations).

**Molen parallelism problem:** When targeting a multicore heterogeneous system, one crucial aspect that has to be taken into account is expressing and using parallelism. For a shared memory system, one solution is to use OpenMP. The Molen programming paradigm is complementary to OpenMP in the sense that OpenMP annotations can be used by the compiler to generate the appropriate Molen primitives.

**Solution:** There are two solutions related to the use of parallelism:

- use Molen parallelism instead of the thread parallelism present in OpenMP, where possible. The advantage of this approach is that the overhead of thread management is eliminated (Molen overhead is much less than thread creation/switching overhead)
- use Molen primitives inside each thread. As Molen was developed for a single threaded environment, modification of the Molen primitive implementations are needed, in order for them to support the thread concept.

**Generating Molen from OpenMP**

This transformation can be done in almost all cases in which the code that has to be executed in parallel, also has to run on another processing element as in Figure 6. For this case, the compiler will understand the parallelism
structure, and instead creating and synchronizing threads it will generate and schedule Molen primitives. The results are presented in Figure 7. Similar examples can be constructed for other OpenMP constructs.

```c
#pragma omp sections nowait
{
#pragma omp section
{
#pragma map call_hw VIRTEX4 1
fft(p, n);
}
#pragma omp section
{
#pragma map call_hw VIRTEX4 4
value = sad(d, l);
}
}
```

Fig. 6: Molen pragma in the context of OpenMP sections

```
SET(1);
SET(4)
MOVTX_ADDR(1, p, n);
MOVTX(1, n);
EXECUTE(1);
MOVTX_ADDR(4, p, n);
MOVTX(4, n);
EXECUTE(4);
BREAK();
```

Fig. 7: Molen primitives generated for the OpenMP sections example

**Using Molen Primitives in OpenMP Threads**

When dealing with parallelism, for some cases, the scheduling of Molen primitives described earlier is not possible. One such case is shown in 8. Another option is to use the Molen primitives in the OpenMP generated code without the compiler understanding the semantic of the OpenMP pragmas. Then another problem arises: multiple Molen primitives will be invoked with the same identifier. To solve this, we proposed the following extension of the Molen primitives:

- use as identifier not only the identifier provided in the code, but the tuple (thread-identifier, identifier).
- provide an internal mechanism that the same CCU can be instantiated multiple times (for example, using different positions on the FPGA).

With these two additions, no other modification to the compiler or OpenMP runtime are needed, and Molen can be used like in Figure 8.

```c
#pragma omp for schedule(dynamic,chunk)
for (i=0; i<N; i++)
{
prepare(p[i]);
#pragma map call_hw VIRTEX4 1
fft(p[i],1024);
}
```

Fig. 8: Molen pragma in the context of OpenMP threads

incorrect results without any warning or error present during the compilation process.

Examples of such parameters are the CCU-frequency which affects the number of cycles needed for various operations, the endianess and the number of cycles to read from memory.

**Solution:** As the development went ahead, it was obvious a mechanism had to be provided to make sure the CCUs were generated for the system in which they were integrated. The identified parameters are:

- endianess
- the number of cycles needed to read/write the memory and eXchange Registers (XREG) memory area.
- the number of pipeline stages of each FP element.
- the frequency at which a specific CCU could run.

The solution depends on the moment at which the check can be performed. For the first three parameters, the check is performed at compile-time. The obvious choice was to add dummy signals to the VHDL code of the CCU. The names of these signals, encode the parameters for which the kernel was generated. In case the CCU was generated with a different configuration, the VHDL compiler would not have been able to match the signals and would report an error.

For example, a dummy signal could be named "check_fp_sp_mult_top_6". This means the CCU uses a floating point (fp) single precision (sp) multiplication unit (mult) that has a pipeline of 6 cycles. For each parameter that needs to be tested a corresponding signal is added. In order to ensure the same frequency is used, a dynamic solution was chosen. As an FPGA has a complex Digital Clock Module (DCM) that can generate different frequencies, the compiler will set a special configuration register with the correct value, before running the CCU.

**Synthesis times and maximum frequencies problem:** For FPGA development, the time needed for synthesis can become a bottleneck in the development process. This happens because the synthesis time for a single VHDL kernel varies between 10 minutes to almost 2 hours, as we can see from Table 8. As the compilation flow involves a source to source transformation tool, each synthesis will regenerate all the files that are mapped to different processing elements. Traditional build system will detect a change in the timestamps of the VHDLs and restart the synthesis process.

**Solution:** The build system was adapted to check not only the timestamp but also the contents of the files before restarting the synthesis process.
Table 8: Compilation times using FPGA flow, at different frequency constraints

<table>
<thead>
<tr>
<th>Kernel</th>
<th>DW ARV time (s)</th>
<th>Xilinx time(s)</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>sad</td>
<td>678</td>
<td>200</td>
<td>166</td>
</tr>
<tr>
<td>SATD</td>
<td>876</td>
<td>1529</td>
<td>166</td>
</tr>
<tr>
<td>SATD unrolled</td>
<td>1529</td>
<td>7038</td>
<td>125</td>
</tr>
<tr>
<td>FFD</td>
<td>10</td>
<td>900</td>
<td>125</td>
</tr>
<tr>
<td>FRACSHIFT</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Toolchain retargetability problem:** Developing a toolchain for a heterogeneous architecture is very time and resource consuming task. To provide the highest degree of retargetability, the design of the framework must be done with care, to allow easy integration with 3rd party toolchains and changes in the architecture supported.

**Solution:** There are several aspects that have to be taken into account:

- use an open, generic standard for communication between tools. In the context of the hArtes project, these standards are Ansi C and XML annotations.
- keep the annotations (either in C and XML) independent from the platform. A good example for such design is the OpenMP standard, which abstracts away most of the details (like number of threads) from the users.
- keep platform dependent logic in libraries as opposed to hardcoding it in tools.
- provide an open build system. As specific compilers have to be used to build the final system, each with its own specificities, developing an open build system, will make it easier for the developer to control the whole process.

### 5.3 Applications related issues

**Problem I:** One problem is that the speedup was not always achieved for the application, given different input sets. This is illustrated in Table 3.

**Solution:** adapt the mapping, at runtime, based on the value of the parameters and on the speedup. A description of this solution is given in [8].

**Problem II:** Determine the optimal balance between several parallel kernels on a reconfigurable device. For the wavefield synthesis application, finding a balance between the number of kernel instances that compute the output waves and the number of kernels that compute the coefficients when needed, does not have an obvious solution.

**Solution:** at compile time Integer Linear Programming (ILP) can be used to determine the best balance. An efficient solution is given in [10].

**Problem III:** Determine the best parameters for specific optimizations, to take advantage of the reconfigurable device. The two cases for this are the loop unroll number and the number of variables that are made local. Although a known optimization, in the context of the reconfigurable device, this has different implications, like reducing the frequency by making routing harder.

**Solution** Work has been done to identify the optimal unroll factor, taking into account profiling information, memory transfers and area utilization [3].

### 6. Conclusion

In this paper, we presented the work done in the context of hArtes project. We analysed the applications, and we highlighted the main challenges that had to be addressed before obtaining an efficient implementation on a Molen system. Several comments about the development flow were made, which, provide insights into what a complex toolchain for heterogeneous platforms has to provide. The hArtes tool chain is now being completely redesigned for later commercial release by a spin off that was created at the end of the project.

### Acknowledgment

The authors would like to thank all the partners from the hArtes consortium. This research has been funded by the hArtes project (EU-IST-035143).

### References