Estimation and Optimization of the Performance of Polyhedral Process Networks

Haastregt, S. van

CHAPTER 6

INDUSTRIAL CASE STUDY

In this chapter, we study the design process of an industrially relevant sphere decoder application used in wireless mobile communications. We take a sequential C specification of this application as a starting point. Our goal is to automatically obtain an RTL implementation in VHDL or Verilog from the sequential C specification. We compare two tool flows to achieve this goal: the commercial AutoESL high-level synthesis tool [Xil11] [1] and the open-source Daedalus system-level design tool flow [Lei08]. AutoESL is a state-of-the-art high-level synthesis environment that combines heuristics with designer input to obtain design points that satisfy design constraints. We want to compare the Daedalus-based approach discussed in this dissertation with the AutoESL approach to gain insight in the effectiveness of our approach.

We introduce the application in Section 6.1. We review a reference implementation of the application in Section 6.2. We describe an implementation using AutoESL in Section 6.3, and describe an implementation using Daedalus in Section 6.4. We compare the different implementations in Section 6.5 and conclude in Section 6.6.

6.1 Sphere Decoding

The application that we study in this chapter implements part of the *WiMAX* standard [FK08]. *WiMAX* (*Worldwide Interoperability for Microwave Access*), based on the IEEE 802.16e-2005 standard, refers to a new generation of (mobile) wireless broadband access networks. WiMAX employs *Multiple Input, Multiple Output*

[1] AutoESL is currently known as Xilinx Vivado HLS [Xil13].

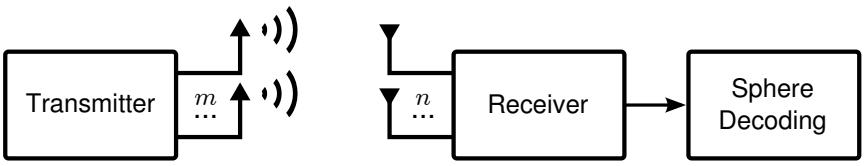

Figure 6.1: An $m \times n$ MIMO system that uses sphere decoding to reconstruct the transmitted symbols.

(*MIMO*) antenna configurations, meaning that both the transmitter and the receiver use multiple antennas, as illustrated in Figure 6.1. All transmitter antennas transmit at the same frequency, but each antenna transmits data from a different data stream. This results in multiple parallel data streams that share the same frequency channel, referred to as *spatial multiplexing*. Spatial multiplexing increases bandwidth efficiency, but comes at the cost of increased computational demands at the receiver side, where advanced techniques are required to separate the different data streams.

Different techniques exist to separate data streams at the receiver side. Decoding the data from the different antennas using a *Maximum Likelihood* (*ML*) detector yields the optimal *Bit Error Rate* (*BER*) performance [BBW+05]. However, the computional complexity of an ML detector grows exponentially with the number of antennas and the choice of modulation scheme, making an ML detector implementation cost-prohibitive for high-data rate systems with large numbers of antennas. Alternatively, channel decoding can be realized using a *sphere decoder*, whose implementation is less expensive while still achieving a BER performance comparable to that of an ML detector [ACDR09]. The actual sphere decoding step is preceded by a channel preprocessing step, which prepares a *channel matrix* that characterizes the MIMO antenna system. In this chapter, we focus on the channel preprocessor of the sphere decoder system that was described in [DTD+09]. The considered sphere decoder system implements a receiver for the most demanding case of the IEEE 802.16e-2005 standard, namely a 64-QAM system with 4 transmitter and 4 receiver antennas.

In Figure 6.2, we show the block diagram of the sphere decoder system that we consider. Before the actual sphere detecting takes place, the channel matrix preprocessor prepares the channel matrix. Inside the channel matrix preprocessor, channel estimation [BSE04] is used to determine the complex-valued 4×4 channel matrix. To improve BER performance, channel reordering is applied to this matrix. The resulting matrix is reorganized into an 8×8 real-valued matrix by the Modified Real-Valued Decomposition (*M-RVD*) block. This real-valued matrix is then converted to an upper-triangular matrix using QR Decomposition (*QRD*). Next, the sphere detector is applied to produce a stream of detected QAM symbols. Subsequent decoding

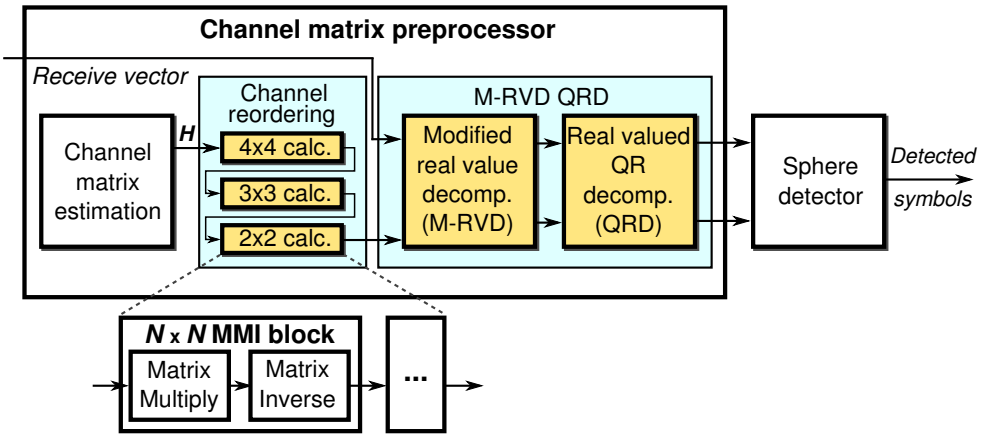

Figure 6.2: Sphere decoder block diagram.

of these symbols then yields the original transmitted bits.

In this chapter, we focus on the Modified Real Value Decomposition (M-RVD) and QR Decomposition (QRD) blocks of the sphere decoder. These two blocks are combined into a single block that we refer to as the *M-RVD QRD* block. Implementing these blocks to meet the application throughput requirements, while minimizing resource usage and latency through the receiver is a challenging design task because of the presence of recurrences in the application.

6.2 Reference Implementation

As a reference implementation, we consider the sphere detector described by Dick et al. [DTD+09]. This reference implementation has been implemented in Xilinx System Generator which is a high-level block-based design tool. The reference implementation is essentially a manually built structural RTL design, containing explicit instantiation of memory and computation primitives and explicit control structures.

The reference implementation targets a mid-speed grade Xilinx Virtex-5 FPGA with a clock frequency of 225 MHz. To conform to the WiMAX throughput targets, the design processes 360 data subcarriers in 102.9 μs. The channel matrix is recomputed for every data subcarrier, which implies the channel matrix preprocessor needs to process a new matrix every

$$\frac{102.9\ \mu\text{s}\ /\ 360}{1/225\ \text{MHz}} \approx 64 \text{ clock cycles.} \tag{6.1}$$

```
1  for (j=0; j<8; j++)
2    for (m=0; m<8; m++)
3      for (t=0; t<15; t++) {
4        X[j][0][t] = diagonal(X[j][0][t], ...);
5        for (n=1; n<8; n++) {
6          if (n < 7-m)
7            R[m][n-1][t] = offdiagonal(R[m][n][t], ...);
8        }
9      }
```

Figure 6.3: Top-level structure of the 8×8 M-RVD QRD C code. Additional code for the time division multiplexing refactoring is underlined.

To meet this high throughput requirement, all blocks in Figure 6.2 operate in a pipeline fashion, which is common for wireless receiver applications. The matrix elements are represented using 18-bit fixed point data types throughout the design. Data is communicated from one block to the next using FIFO buffers and double buffered and dual-ported memories, implementing a streaming system [NV08]. Each block operates on only a few kilobytes of data at a time, which means the sizes of the communication memories are relatively small. Therefore, all memories are implemented using on-chip block memory primitives, that is, no external memory is required for inter-block communication.

The QR decompositions used in the M-RVD QRD block are based on Givens Rotations [SM93]. This method consists of two stages, which we refer to as the *diagonal* and *off-diagonal* cells. The diagonal cell computes an angle such that the leading matrix element is rotated to zero. That angle is subsequently used by the off-diagonal cells to apply the rotation to the remaining nonzero elements of the same matrix row. The top-level structure of the M-RVD QRD C code is shown in Figure 6.3.

6.3 AutoESL

In this section, we describe implementing the M-RVD QRD block of the sphere decoder using the *AutoESL* tool. AutoESL (formerly known as AutoPilot) has been developed since 2006 by AutoESL Design Technologies, Inc. as a commercialization of the xPilot tool from UCLA [CFH+06], and was acquired by Xilinx in 2011 [Xil11]. AutoESL accepts code written in a synthesizable subset of the C, C++, or SystemC language as input. We focus on C++ design entry, with the goal of leveraging C++ template classes to represent arbitrary precision integer types and template functions to represent parameterized components. For the remainder of this

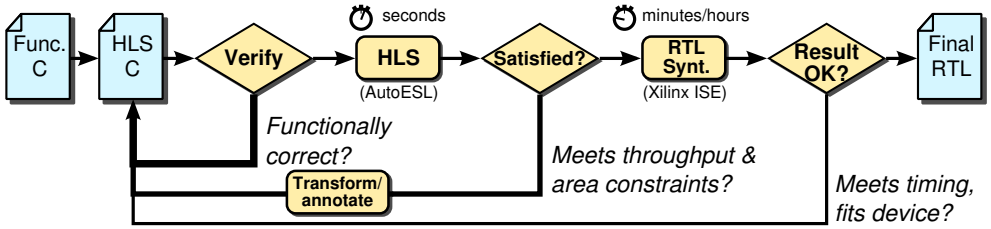

Figure 6.4: High-Level Synthesis design flow.

section, we refer to the high-level language code as "C code" without elaborating on these details.

6.3.1 Design Flow

The overall design process we have followed is shown in Figure 6.4. We start from a functional specification in the C language and a corresponding test bench. The C specification is a reimplementation of the MATLAB model that was used for the reference implementation made with System Generator. By using the test bench and a representative set of test vectors, the C specification is then repeatedly refactored to reflect the desired architecture, while preserving the functionality. This refactoring process makes use of two different interpretations of the C specification. The *functional* interpretation represents the conventional semantics of the C code, describing the sequential and functional behavior. The *architectural* interpretation represents the HLS semantics of the C code, describing the RTL architecture at a high level. The designer makes sure that the functional interpretation of the refactored C code is still identical to that of the original C code, while the architectural interpretation is changed to satisfy non-functional requirements like resource cost and throughput. Manipulation of the architectural interpretation focuses on the coarse-grained architectural aspects, such as memory porting, parallelism, and resource sharing. Fine-grained architectural aspects, such as RTL pipelining details, are handled automatically by the HLS tool by means of predefined characterization data of the target FPGA device.

The throughput resulting from the architectural interpretation can be analyzed statically or dynamically as an output of the HLS compilation. Resource cost estimates are reported after HLS compilation as well. If the various cost and performance metrics satisfy the design requirements, the resulting RTL is synthesized using platform-specific low level synthesis tools. Since HLS tools do not have precise knowledge about e.g. routing delays, metrics reported by the HLS tool typically differ to some

extent from the actual timing characteristics and resource costs obtained after RTL synthesis.

At all times in the development process, the source code of the design is fully functional and can be verified using the C test bench using a regular C compiler and debugger. This is very different from a traditional RTL design flow, where a fully functional version of the design source code becomes available only after weeks or even months of labor. This RTL source code is developed independently of the original reference code, thereby requiring an extensive validation phase. In contrast, obtaining functionally correct design source code that successfully passes through the HLS tool is only a matter of days or even hours. This means an early functionally correct RTL implementation can be obtained quickly, although it is unlikely to already meet resource cost and throughput constraints.

6.3.2 Design Entry

Modern HLS tools like AutoESL and PICO (semi-)automatically leverage a wide range of compiler optimization techniques such as common subexpression elimination and loop unrolling, and computer architecture techniques such as pipelining and resource sharing to improve cost and performance aspects of a design. For some of these techniques, the effectiveness is highly dependent on the structure of the application. Therefore, the decision when and how to apply a particular technique often has to be made by the designer. Some techniques can be applied or controlled with a tool pragma, while other techniques must be reflected in the way the algorithm is described. In this section we describe the techniques applied for the M-RVD QRD block of the sphere decoder application. In particular, we have applied a combination of time division multiplexing, loop unrolling, array partitioning, and case-specific optimizations. All of these techniques have been applied by modifying C code only such that a different architectural interpretation is obtained, while the functional interpretation is preserved.

Time Division Multiplexing

For designs without feedback, an HLS tool is generally able to instantiate registers freely to increase clock frequency and throughput. However, in pipelines that are part of a feedback loop, registers cannot be inserted freely without introducing pipeline stalls. Hence, feedback loops, also known as *recurrences*, in a design are the key limiter of throughput [Pap91]. For example, Figure 6.3 shows the high level structure of the 8×8 M-RVD QRD loop nest. Although there are several recurrences in the application, the critical recurrence in this code occurs when the result

`X[j][0]` of the `diagonal` function call is used as an argument to the next `diagonal` call. Synthesis of the `diagonal` function results in a 14-stage pipeline. As a result, each `diagonal` call has to wait 14 cycles until the result of the previous call becomes available, which means the pipeline is highly underutilized. To accommodate the recurrence without introducing pipeline stalls, we use the wait cycles to process independent data streams, by applying *Time Division Multiplexing* (*TDM*) over 15 datasets. The underlined parts in Figure 6.3 explicitly reflect time division multiplexing or *c-slowing* [LRS93] over separate datasets through the inner `t`-loop.

We observe several characteristics of this design. First, the code accurately reflects the order in which data is processed in the reference design. Second, the TDM refactoring is expressed entirely at the C code level. This means it can be seamlessly ported to any HLS tool that supports the used C constructs, such as multi-dimensional arrays. Third, the number of datasets to iterate over, that is, the TDM depth, cannot be determined without knowing the sizes of the critical recurrences. Although AutoESL does not compute the number of datasets automatically, the HLS process does analyze the source code for recurrences and reports to the designer where recurrences are not satisfied. The designer can use this information in a subsequent AutoESL run. In the sphere decoder application, since 360 *independent* data subcarriers have to be processed for each frame, TDM is a straightforward way to handle the critical recurrence while incurring small increases in resource cost and latency. The resource cost increase stems from additional buffering for the fifteen time multiplexed data subcarriers. The processing latency of the M-RVD QRD block for a single data subcarrier is 945 clock cycles, or $4.2\mu s$.

Loop Unrolling

Application throughput constraints translate directly or indirectly into parallelism requirements on the RTL architecture. For example, the code in Figure 6.3 processes a block of 15 subcarriers. As shown in Equation (6.1), every 64 cycles a new subcarrier must be processed to meet application throughput requirements. As a result, the loop nest in Figure 6.3 must start executing a new block of 15 subcarriers every $15 \times 64 = 960$ cycles. Because the outer loops together comprise 960 iterations, this implies that the body of the `t` loop must be pipelined with an initiation interval II_t of 1. As a result, the inner `n` loop must be unrolled to perform all off-diagonal computations in parallel, which is possible in this application since the calls to `offdiagonal` in the inner loop are independent. We specify the pipelining and unrolling as `pragma` directives to AutoESL, thereby minimizing rewriting of the code and preserving code readability and maintainability. These pragmas are shown in Figure 6.5. AutoESL currently requires unrolled loops to have constant loop bounds, hence the need to ex-

```
1  #pragma AP ARRAY_PARTITION variable=R complete dim=2 partition
2  for (j=0; j<8; j++)
3    for (m=0; m<8; m++)
4      for (t=0; t<15; t++) {
5        #pragma AP PIPELINE ii=1
6        X[j][0][t] = diagonal(X[j][0][t], ...);
7        for (n=1; n<8; n++) {
8          #pragma AP UNROLL
9          if (n < 7-m)
10           R[m][n-1][t] = offdiagonal(R[m][n][t], ...);
11       }
12     }
```

Figure 6.5: Applying loop unrolling (line 8), pipelining (line 5), and array partitioning (line 1) to the M-RVD QRD C code.

plicitly move the conditional statement into the loop body. During the HLS process, AutoESL automatically attempts to compute the number of cycles the loop nest takes to execute, taking into account constant loop bounds and pipeline latencies. This enables a designer to quickly interpret the achieved throughput.

Array Partitioning

After unrolling, the seven off-diagonal cells need to be fed with new data every clock cycle. One of the data sources is a three-dimensional array `R` that is mapped onto a block memory primitive of the FPGA. These block memory primitives have only two memory access ports, which means at most two accesses to array `R` can take place every clock cycle. However, every clock cycle seven different elements need to be read from `R`, since the loop iterator `n` of the unrolled loop appears in the array index expression. This means shortage of memory ports now limits throughput. To overcome this problem, we apply array partitioning to partition the array into subarrays [CJLZ09], again directed by `pragma` directives. We show such a pragma on line 1 of Figure 6.5 to partition the second dimension of array `R`. Each subarray is then mapped onto a separate block memory primitive, effectively providing two memory ports dedicated to each subarray and thereby solving the array bandwidth limitation. Again, memory port limitations are analyzed during the HLS process and AutoESL reports when shortage of memory ports prevents achieving the requested pipelining.

```
1 template <int Wa, int Wb, int Wc>
2 ap_int<Wa+Wb> MADD(ap_int<Wa> a, ap_int<Wb> b, ap_int<Wc> c) {
3   #pragma AP INLINE self off
4   #pragma AP LATENCY max=3
5   #pragma AP INTERFACE ap_none port=return register
6   return a*b+c;
7 }
```

Figure 6.6: C++ code for the MADD function.

Case-specific Optimizations

As an example of a case-specific optimization we consider a non-obvious source of multiplications in the C language, namely multi-dimensional array accesses. Since an array is eventually mapped to a memory with a single-dimensional address space, the multi-dimensional array index has to be converted into a linear address. For example, consider an $M \times N$ array defined in C as `a[M][N]`. The address of array element `a[i][j]` is computed with the expression $i \cdot N + j$. The cost of evaluating this expression varies greatly with the value of N. For example, when $M = 8 \wedge N = 15$, computing the address requires a multiplication by fifteen, which cannot be implemented using only a single shift operation because it is not a power of two. When the array dimensions are interchanged, thus $M = 15 \wedge N = 8$, the multiplication by fifteen is replaced by a multiplication by eight which can be implemented using a single shift operation.

Function and Class Templates

The M-RVD QRD block is specified entirely in the C++ language. To illustrate how function and class templates from C++ can be used, we show the code of the Multiply/Add (MADD) function which is part of a library used by the diagonal cells of the M-RVD QRD block. We provide the C++ code of this function in Figure 6.6. Throughout the design we use arbitrary precision integer (`ap_int`) data types. To allow effective use of library functions, we have designed these functions to support different argument bit widths using C++ templates, as illustrated in line 1 of Figure 6.6 for the MADD function.

Resource Sharing

In many embedded signal processing applications, maximizing throughput is often not as important as minimizing resource usage for a given throughput. In these cases,

effective resource sharing is an important design goal. Some resource sharing is implicit when a loop is pipelined rather than unrolled, since consecutive iterations of the loop execute on the same datapath generated from the body of the loop. In this section, we focus on achieving *additional* resource sharing.

AutoESL employs heuristics to decide which function calls are inlined. The MADD in Figure 6.6 was inlined in our design study. When an inlined function is called in two different places, the entire implementation of this function appears twice in the RTL. To enable sharing of resources in such cases by AutoESL, we disable inlining using the pragma in line 3 of Figure 6.6.

User-Influence on the Generated RTL

AutoESL provides means to influence aspects of the RTL at the source code level. The use of such means turned out to be inevitable to obtain a design competitive with hand-written RTL for the M-RVD QRD block. Because AutoESL's default timing characterization prevented timing closure of RTL resulting from multiplications in the C code, we have enforced the correct characterization by means of the pragmas shown in lines 4 and 5 in Figure 6.6. Line 4 enforces a latency of three clock cycles and line 5 enforces an output register of the MADD RTL block. Such pragmas allow a designer to "correct" suboptimal decisions of the HLS tool for a particular part of the design. The need for such manual corrections should diminish over time as HLS tools are further improved.

6.3.3 Design Productivity

To compare design times of the HLS and reference implementations, we have reconstructed the approximate amount of working time on the designs. Design times for the reference implementation have been estimated by the original implementors [DTD+09] as 4.5 weeks. Design times for the HLS implementation have been extracted from source code version control logs as 5 weeks. We observe that the design times to reach an optimized implementation are approximately the same for the HLS implementation and the reference implementation. However, the RTL design flow yields only a single design point, while the HLS design flow yields many design points with different performance and cost tradeoffs.

The effects of the refactoring-based design process for the M-RVD QRD block can be seen in Figure 6.7. On the left vertical axis, we show the overall application throughput determined from static clock cycle count analysis of AutoESL, combined with post-place and route timing closure information. On the right vertical axis, we show the corresponding post-place and route LUT and flipflop usage. For comparison, the

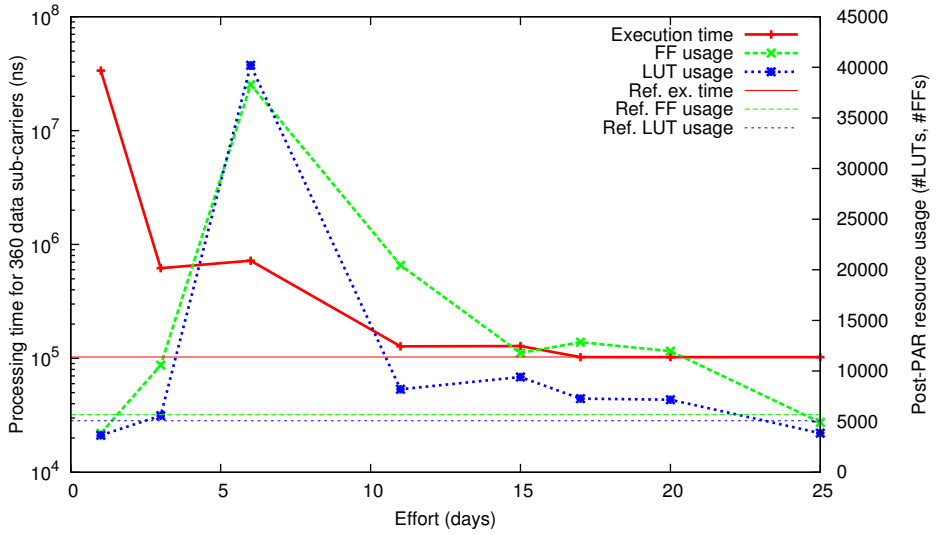

Figure 6.7: Performance and resource usage of the M-RVD QRD block plotted as a function of development time.

horizontal lines represent the target application throughput and resource usage of the reference implementation. After obtaining a "clean" algorithmic C model, it took about a day to get the code through AutoESL for the first time. This required rewriting of several nonsynthesizable constructs such as non-analyzable pointers. This first implementation exploits little parallelism as it executes almost entirely sequentially. By performing continual refactoring, the throughput and cost are improved with full functional verification at each refactoring step. The bulk of the architectural refactoring was completed in about ten working days. The remaining time has been spent tuning the design to reduce resource usage and to improve timing closure in place and route. Below we summarize the design process for the M-RVD QRD block.

- **Day 1:** The C code is accepted by the HLS tool, and a functional hardware implementation is already available. However, the total processing time is off by two orders of magnitude.
- **Day 3:** After becoming more familiar with the tool and applying basic refactoring techniques such as enabling pipelining using a single C preprocessor pragma, the processing time is reduced significantly.
- **Day 6:** Because of code restructuring such as loop unrolling, the resource usage increases considerably. This limits the achievable clock frequency, effectively increasing the processing time again. However, the design source code

is now in a shape that enables further optimizations.

- **Day 11:** C integer data types are replaced by fixed point data types and optimized primitive blocks (e.g., the MADD function) are introduced in the design. This significantly reduces resource usage. A pipeline *II* of one is now feasible. However, during RTL synthesis the design does not achieve timing closure, which means throughput constraints can still not be met.
- **Day 17:** By optimizing the code of the data paths (e.g., by applying the case-specific optimization described in Section 6.3.2), latencies and resource usage are further reduced. As a result, the RTL now achieves timing closure, so at this point an implementation meeting throughput requirements is available.
- **Day 25:** Further optimizations including algorithmic optimizations have been applied to reduce the resource usage of the design.

In this work we had the advantage that the reference implementation was already available to us. Thus, we knew the high-level application architecture that had to be constructed to meet throughput requirements. Hence, we have been fully concentrating on getting a similar architecture out of the HLS tool initially. After obtaining a design point meeting throughput requirements, the goal was changed to reducing resource cost to the level of the reference implementation. Again, we had the advantage of knowing detailed resource cost statistics for the reference implementation, thus by comparing with the HLS implementation we knew what parts could be optimized further. Many different design points can be implemented using HLS in a short amount of time, as each design point in Figure 6.7 is a fully functional design with different performance and cost aspects. On the other hand, the RTL design process has yielded only one design point in approximately the same amount of time.

6.4 Daedalus

We have implemented the M-RVD QRD block of the sphere decoder also using the Daedalus tool flow. We have followed the same refactoring-based design process as with the AutoESL design, with the only difference that the "HLS" step of Figure 6.4 now consists of running Daedalus instead of AutoESL. We have started from the same C code that was also used as starting point for the AutoESL design. Obtaining an RTL implementation from C code consists of two steps, as depicted in Figure 1.1. The first step is to convert the M-RVD QRD C code into a PPN specification using PNGEN, which we describe in Section 6.4.1. The second step is to synthesize an RTL implementation from the PPN specification using ESPAM, which we describe in Section 6.4.2.

6.4.1 Design Entry

We distinguish between two classes of refactorings in a Daedalus design flow. Refactorings of the first class transform the source code into a form suitable for Daedalus. Refactorings of second class are similar to the architectural refactorings in the AutoESL design flow. That is, they serve to alter the architectural interpretation of the source code. The first two refactorings discussed below are of the first class, whereas the remaining refactorings are of the second class.

Compatibility Restructurings

PNGEN requires that the sequential C code is a static affine nested loop program (cf. Section 2.3). The sequential C code for the M-RVD QRD block already conforms to this requirement, such that meeting the SANLP requirement requires no effort. To ease integration of IP cores in LAURA, we rewrite the sequential code such that besides for- and if-statements, only function calls and plain copy assignment statements are exposed in the top level function given to PNGEN. This means other statements containing arithmetic operations have to be embedded into function calls. For example, the following statement

```
in_diag = -mat_im[i];
```

is rewritten as

```
negate(mat_im[i], &in_diag);
```

with `negate` being a new function that writes the negation of its first argument to its second argument.

Introduction of Source and Sink Processes

Each PPN should have at least one source and one sink process that represent the input and output interfaces of the system. In a physical implementation, these source and sink processes exclusively communicate with the environment. For example, a source process may represent a video capture device, whereas a sink process may represent a display device. The remaining non-source and non-sink processes perform the actual data processing.

In the current implementation of PNGEN, source and sink processes have to be explicitly specified in the C input, using function calls that have only output arguments and only input arguments, respectively. Any arguments to the top level function, such as `im` on line 1 of Figure 6.8, are currently ignored by PNGEN. The M-RVD QRD reference code communicates input and output data via array arguments of the

```
1  void mrvdqrd(int im[4][4][15], ...) {
2    #pragma AP ARRAY_PARTITION variable=im complete dim=1 partition
3    ...
4  }
```

Figure 6.8: Input arguments to the top-level M-RVD QRD function.

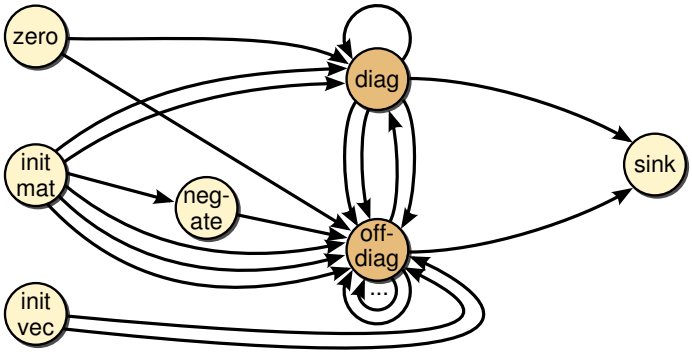

Figure 6.9: Initial PPN for the M-RVD QRD block.

top level function, as shown in Figure 6.8. This means we have to translate the top level function argument list into source and sink processes. For every input argument, we introduce a function and a loop nest such that all elements of the array are written exactly once. We illustrate this for the `im` input argument in Figure 6.11. The input argument is removed from the function header and defined as a local variable. The order in which the elements are written should match the order in which the elements are read for the first time by any subsequent processes. This ensures communication can be implemented using regular FIFO channels instead of more expensive reordering buffers. In a similar way, for every output argument, we introduce a function and a loop nest such that all elements of the array are read exactly once.

We are able to reuse the original test bench by making additional modifications to the C code. First, we modify the test bench to read and write test vectors from and to global variables. Next, we make the source and sink processes stateful by introducing an internal counter that is incremented upon every invocation of the particular function. Using this counter, the corresponding array elements are read from or written to the global test vectors. Although these changes assist us in verifying the functionality after each C code transformation, they have no implications for the final hardware implementation, since the source and sink processes are typically replaced by the interfaces they represent.

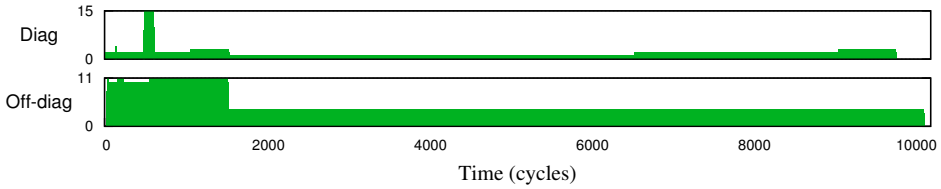

Figure 6.10: Flat execution profiles of diagonal and off-diagonal cell resources.

Initial PPN

After the source code refactorings discussed so far, a first PPN can be obtained using PNGEN, which is shown in Figure 6.9 This initial PPN contains one diagonal cell and one off-diagonal cell, since the diagonal and off-diagonal cell calls appear only once in the C code of Figure 6.3. An architecture with only one off-diagonal cell cannot achieve the throughput demanded by the application requirements. This can be observed using the flat execution profiles shown in Figure 6.10 obtained using cprof. The execution time for a single execution of the PPN is about 10000 clock cycles, which is more than ten times the desired execution time of 960 clock cycles.

Splitting

The AutoESL and reference implementations contain one diagonal cell and eight off-diagonal cells to meet the throughput requirements of the sphere decoder application. We have applied loop unrolling to the innermost loop to obtain eight off-diagonal cells in the AutoESL design. HLS tools such as AutoESL provide a pragma to unroll a loop while keeping the code compact and maintainable. To obtain the same architecture, we apply a plane cutting transformation to the PPN with a factor 8 on the innermost dimension n of the `offdiagonal` process, specified as:

$$planecut(\texttt{offdiagonal}, n, 8)$$

This results in eight `offdiagonal` processes, which resembles the architecture of the reference implementation.

After splitting the off-diagonal cell, the source processes also have to be split to ensure all eight off-diagonal cells receive data at a fast enough rate. This is similar to partitioning an input array in the HLS context, effectively increasing the bandwidth of that array. For example, for an input array `im`, representing the imaginary components of the complex-valued channel matrix, we apply a pragma in AutoESL to partition this array. This is shown by the pragma in Figure 6.8. The pragma splits `im` into four distinct subarrays `im_0[4][15]`, `im_1[4][15]`, `im_2[4][15]`, and `im_3[4][15]`. We have removed the input and output arguments to the top-level function by introducing

```
1  void mrvdqrd() {
2    int im[4][4][15];
3    for (j=0; j<4; j++)
4      for (m=0; m<4; m++)
5        for (c=0; c<15; c++)
6          initmatrix( &im[j][m][c] );
7  }
```

Figure 6.11: Source process for input argument `im` of Figure 6.8.

source and sink processes as discussed above. The code in Figure 6.11 implements a source process for input matrix `im`. After splitting the off-diagonal cell, we need to split the `initmatrix` process as well to make sure data from `mat_im` is delivered at a fast enough rate. This is done by applying a plane cutting transformation with a factor 4 on the `j` dimension of `initmatrix`. As a result, we obtain four `initmatrix` processes that deliver data to four out of eight off-diagonal cells. The other four off-diagonal cells require the real components of the complex-valued channel matrix, which is stored in an array `re`. We apply the same plane cutting transformation to the source process for this array.

A similar relation exists between sink process splitting in PPNs and output array partitioning in HLS, to ensure that data produced by the off-diagonal cells is consumed at a fast enough rate.

After the splitting transformations, we again use cprof to evaluate the performance of the new PPN. The technique described in Section 4.6.6 allows us to evaluate the splitting transformations at the sequential code level, without the need to apply the transformations to the sequential code. The resulting flat execution profiles are shown in Figure 6.12. The execution time is now reduced to about 960 clock cycles, which means the PPN meets the application throughput requirements.

Process Merging

Similar to the AutoESL and the reference implementations, the PPN implementation now consists of one processing resource for the diagonal and eight processing resources for the off-diagonal cell computations. This allows the PPN to meet the throughput demands of the application. We now ask ourselves if we can reduce the resource cost of the implementation while satisfying the throughput constraints. For this purpose, we analyze the utilization of the eight off-diagonal cell LAURA processors using the flat execution profiles obtained by cprof. The number of simultaneously active iterations on each off-diagonal cell LAURA processor over time is

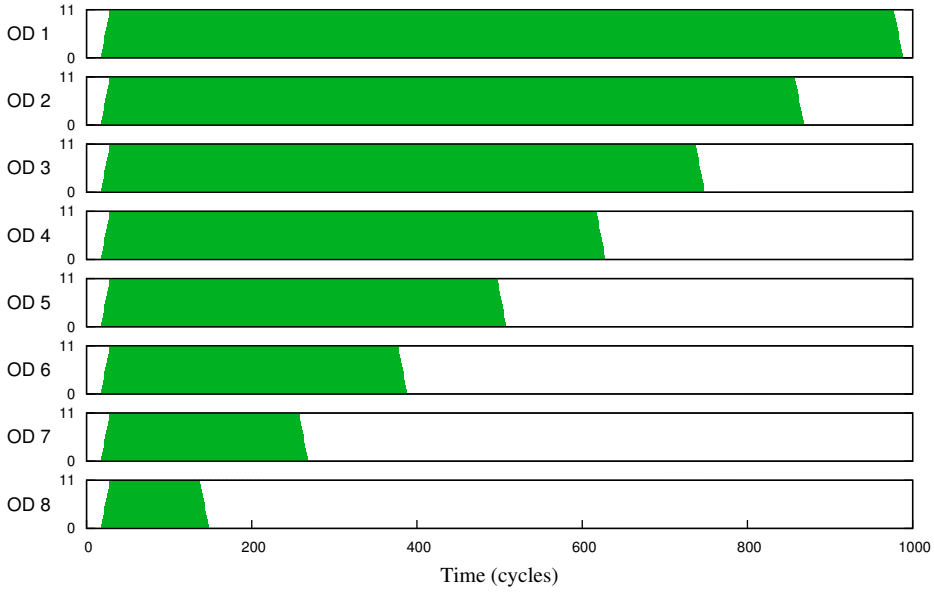

Figure 6.12: Flat execution profiles of off-diagonal cell resources after splitting by a factor eight.

```
1  for (n=1; n<8; n++) {
2    if (n < 7-m) {
3      if (n == 1)            pr$ = 1;
4      if (n == 2)            pr$ = 2;
5      if (n == 3 || n == 8) pr$ = 3;
6      if (n == 4 || n == 7) pr$ = 4;
7      if (n == 5 || n == 6) pr$ = 5;
8
9      // Offdiagonal cell profiling instrumentation ...
10   }
11 }
```

Figure 6.13: Evaluating merging transformations using cprof.

shown in Figure 6.12. The maximum number of simultaneously active iterations on a processor is given by the processor's pipeline depth, which is 11 cycles. From Figure 6.12, we observe that the utilization of the first off-diagonal cell processor (OD 1) is almost 100%, because the pipeline is fully occupied by eleven iterations for most of the time. On the other hand, the last off-diagonal cell processor (OD 8) is active for only $\frac{1}{8}$ of the time, which means the utilization is approximately 12%.

Our goal is to merge the processors with low utilization, such that resource cost is reduced while throughput is not affected. Off-diagonal cell 1 determines the overall throughput, as it has the longest execution time according to Figure 6.12. By looking at this figure, we expect that merging off-diagonal cells 3 and 8 should lead to a combined execution time that is still shorter than the execution time of OD 1. A similar expection holds for merging OD 4 and 7, and OD 5 and 6. This would lead to an implementation with only five off-diagonal cell processors instead of eight.

To evaluate whether this merging transformation is beneficial, we consider the two conditions for a merging transformation described in Section 5.2.2. The first condition is that the processes that are merged execute the same function. This condition is met, since each process executes the same off-diagonal cell function. The second condition is that the overall throughput should not be affected by the merging. We use cprof to assess if this condition is met. We leverage the technique of Section 4.6.6 to evaluate the merging transformation at the sequential code level. Recall that this technique employs a variable `pr$` which selects the processing resource on which an iteration executes. We assign iterations of the `n`-loop to `pr$` according to the merging transformation, as shown in Figure 6.13. For example, off-diagonal cell 1 is not merged with any other off-diagonal cell, and is therefore assigned exclusively to processing resource 1 on line 3. Offdiagonal cells 3 and 8 are merged, and are therefore both assigned to processing resource 3 on line 5. The resulting flat execution profiles

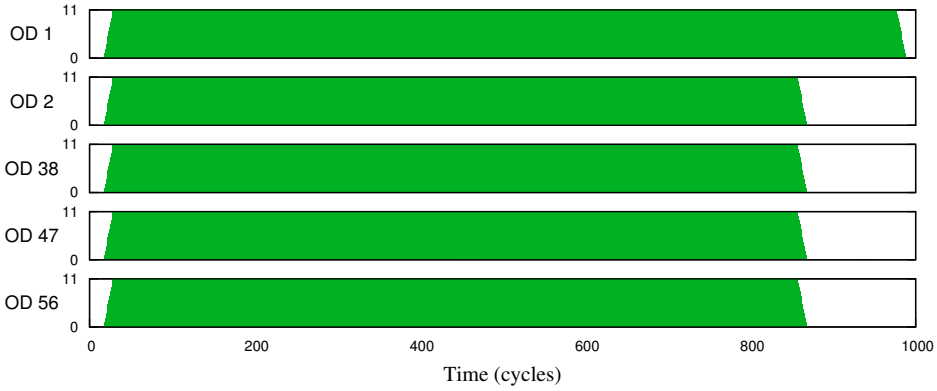

Figure 6.14: Flat execution profiles of off-diagonal cell resources after merging cells 3 & 8, 4 & 7, and 5 & 6.

are shown in Figure 6.14. The total execution time of the merged version is identical to the execution time of the unmerged version. We therefore conclude that the proposed merging transformation would be worthwhile to apply for further evaluation at the implementation level.

6.4.2 Synthesis

After obtaining a process network with the desired throughput characteristics, we generate an RTL implementation using the ESPAM tool. Because the Daedalus tool flow does not provide means to synthesize data paths, we have reused the IP cores for the diagonal and off-diagonal cell functions from the AutoESL implementation. These IP cores can easily be integrated into the execute units of the generated LAURA processors that implement the processes. Since the source and sink processes represent interfaces of the application, we do not synthesize LAURA processors for these processes. The interface to the PPN consists of the FIFO buffers that connect the interior processes of the PPN to the source and sink processes.

To achieve the highest clock frequency currently possible for LAURA processors, we have used the optimization described in Section 3.5.1. Despite this optimization, the RTL for the eight-off-diagonal-cell implementation achieves a clock frequency of 176 MHz, whereas 225 MHz is required to meet the application throughput demands.

The alternative implementation with only five off-diagonal cells was not implementable by ESPAM, because merging of LAURA processors is not supported in the general case as explained in Section 5.1.2. An alternative with seven off-diagonal

Design	LUT	FF	DSP	BRAM	Fmax
SysGen [DTD+09]	5082	5699	30	19	225
AutoESL [CLN+11, NNH+11]	3862	4931	30	19	225
Daedalus-8OD	6506	5235	30	38	176
Daedalus-7OD	6672	5309	27	70	172
Daedalus-5OD			21		

Table 6.1: M-RVD QRD post-place-and-route implementation statistics.

cells, obtained by merging the last two off-diagonal cells, was implementable, because the compound process has a convex polyhedral set as its domain.

6.5 Comparison

In Table 6.5, we compare resource usage statistics for the M-RVD QRD block of a reference RTL design [DTD+09], the AutoESL design, and the Daedalus design. To obtain accurate comparisons, we have reimplemented the reference design using the Xilinx ISE 12.1 tools targeting a Virtex-5 VLX110T-2 FPGA. The AutoESL design has been developed using AutoESL AutoPilot 2010.07.ft and has also been implemented using ISE 12.1 targeting the same FPGA. The Daedalus designs have been developed using PNGEN 0.10-93-g73a41d1 and ESPAM 2011.10, and have been implemented using ISE 12.1 targeting the same FPGA. Verification of the RTL was performed using a manually written testbench in VHDL that used the same test vectors as the testbench for the SysGen and AutoESL designs.

The SysGen, AutoESL, and Daedalus-8OD designs all employ the same architecture containing one diagonal and eight off-diagonal cells. This is reflected in the DSP resource cost, which is the same for all three designs. The AutoESL design has lower LUT and FF cost mainly because the off-diagonal cell was more optimized than the off-diagonal cell of the SysGen design. The Daedalus-8OD design has higher LUT cost than the SysGen design because of the logic implementing the LAURA processors and channels. The Daedalus-8OD design has lower FF cost than the SysGen design, because the Daedalus-8OD design was not optimized for the target clock frequency of 225 MHz, and thus lacks careful insertion of more FF primitives to meet the target clock frequency. The Daedalus-8OD design requires twice the amount of block memory (BRAM) primitives as the SysGen and AutoESL design, to allow sufficiently large channel sizes that do not degrade the throughput. The Daedalus-7OD design contains only seven off-diagonal cells, which is reflected in a saving of three

DSP primitives. This comes at the expense of slightly higher LUT and FF cost, and almost a doubling in BRAM cost. The increase in BRAM cost is caused by larger buffer sizes needed to avoid blocking writes that degrade throughput. The Daedalus-5OD design was not implementable, as explained in Section 6.4.2. We know that an off-diagonal cell requires three DSP primitives. Eliminating three off-diagonal cells thus leads to a reduction of nine DSP primitives. Estimating the other cost characteristics of the Daedalus-5OD is not trivial, because these characteristics depend on the interplay of many factors. Therefore, these characteristics are left empty in Table 6.5. We did not succeed in obtaining an architecture with less than eight off-diagonal cells using AutoESL. Attempts to express such architectures in the C code resulted in implementations that did not satisfy the throughput requirements.

In Section 6.3.3, we have compared the design times of the SysGen and AutoESL designs. Comparing the design times of the Daedalus and AutoESL designs is difficult for the following three reasons. First, the AutoESL design time includes time needed to study the application and the SysGen reference design. Second, the blocks implementing the diagonal and off-diagonal cells were already available during the Daedalus design, whereas these had to be developed and optimized during the AutoESL design. Third, we needed to debug and adapt the Daedalus tools, as the application revealed corner cases that were not correctly handled by the tools. A design time estimate would thus be blurred because of these three reasons. Making an educated guess nonetheless, we expect that we could reproduce the architecture of the SysGen and AutoESL designs using Daedalus in about two weeks.

6.6 Conclusion and Summary

We were able to achieve an RTL implementation from sequential C code for an industrially relevant application using both the commercial AutoESL and academic Daedalus tools. The AutoESL design was competitive to the manually built reference implementation. The architecture employed by the AutoESL and reference designs could be replicated using Daedalus, although the Daedalus design did exhibit higher resource cost and a lower clock frequency. We attribute this to Daedalus being a primarily a research environment, in which the limited development power is invested in research aspects rather than competition with commercial products. We expect that more competitive designs can be obtained using Daedalus with additional engineering effort, as we do not see fundamental limitations.

The use of synthesis techniques and optimizations presented in Chapter 3, the cprof analysis technique presented in Chapter 4, and the transformations presented in Chapter 5 proved essential in obtaining the architecture of the sphere decoder reference

design. Moreover, the cprof technique allowed us to quickly evaluate performance of alternative application instances at the sequential code level. We therefore conclude that the work presented in this dissertation are essential contributions to handle industrially relevant applications in Daedalus.