A parallel for loop memory template for a high level synthesis compiler

Craig Moore, Wim Meeus, Harald Devos, and Dirk Stroobandt
Hardware and Embedded Systems Group
Electronics and Information Systems Department
Ghent University, Belgium
phone +32 9 264 33 66 | fax +32 9 264 35 94
{craig.moore, wim.meeus, harald.devos, dirk.stroobandt}@elis.ugent.be

Abstract—We propose a parametrized memory template for applications with parallel for loops. The template's parameters reflect important trade-offs made during system design. The template is incorporated in our high level synthesis (HLS) compiler, where the template's parameters are adjusted to the application. The template fits parallel for loops with no loop dependencies and sequential bodies. We found two alternative template implementations using our compiler. In the future, we will develop templates for other types of for loops. These will be added to the compiler and it will identify the template that works best for the application it is compiling. Once a template is selected, the compiler will use design space exploration to select the best combination of template parameters for the targeted hardware and application.

I. INTRODUCTION

When developers decide to accelerate their sequential algorithm by running it on parallel hardware (like ASICs or FPGAs), they have to explore different architectures to find the one that works best. Many designers still manually re-write their sequential algorithms in an HDL. However, more and more designers are using high level synthesis (HLS) compilers to generate HDL. For highly parallel architectures, the memory design often is a performance bottle-neck because the parallel architectures must interface with (largely) sequential external memory. We have designed our own basic C to VHDL compiler and our goal is that it can explore different predefined memory templates to select or suggest the best one based on the C source code and resource constraints.

Most high level synthesis tools do not explicitly allow for architectural exploration of different types of memory designs. They often allow the designer to specify the interface, but it is up to the designer to create the appropriate memory design for coping with the memory gap. Manual HDL designers will usually create the memory based on an existing design which was chosen because of the way the application accesses values from memory. However, users of HLS synthesis tools may not have the necessary background to manually design such an architecture. We therefore believe compilers should also have a collection of known memory templates which can be adapted (automatically) to suit the application. Using these templates the HLS compiler should be able to systematically try out each template to find the one that makes the best trade off between resource allocation and performance (in terms of clock speed and execution cycles). As a first step, we are developing a number of parametrized memory templates that are used by the compiler. In this paper, we introduce our template for a streaming buffer interface for a parallel for loop.

The developed templates fits parallel for loops that have no intra-loop data dependencies and have a sequential loop body, like the loop shown in Fig. 1. The proposed template can be used by any HLS (C to VHDL) compiler. The template shown in Fig. 3 is composed of a loop controller, external memory ar- biter, input buffer, data path and output buffer with parameters that can be adjusted at compile time. The parameters control the size of the various components used in the design in order to adapt them to suite the application and targeted hardware. The template will interface with external memory using burst memory transfers where each memory request (read or write) will handle multiple consecutive transactions in memory. The length of the burst transfer (number of values returned) will be a parameter to the memory template.

Our paper is structured as follows. In section II, we will discuss the background concepts which shed some light on the decisions we made when designing our template. This is followed, in section III, by an overview of our memory template and the reasons why we designed it the way we did. In section IV, we give examples of two types of memory designs which were built using our template. Next we discuss the future work that will logically follow from this research in section V. Finally, section VI discusses how other HLS compilers handle the interface with external memory.

II. BACKGROUND

A. Parallel for Loops

A parallel for loop is a for loop where each iteration of the loop body can be executed in parallel. Of course, the result after executing a loop body in parallel can be different from
the sequential execution of the same loop if there are data dependencies, where values used in one iteration depend on the results of an earlier iteration. This difference is illustrated in Table I. As can be seen by this simple example, the results differ because of loop dependencies where iterations of $i > 1$ depend on values of $a(i)$ calculated in earlier iterations. We assume such dependencies are not present in our implementation. This is sometimes referred to as a dopar loop as defined by Wolfe [1]. In some cases, loop dependencies can be removed from for loops by first performing loop transformations [2], [3].

### B. Connecting to External Memory

In FPGAs, values stored in an array are almost always kept in block random access memory (BRAM). Each clock cycle, BRAM will only be able to return one value per address, either because of the size of the values or the size of the bus. This can present a problem for parallel architectures where multiple values may need to be accessed in parallel in the same clock cycle. A second issue to consider is the fact that different manufactures have different types of interface protocols when connecting to external memory. This puts an extra burden on the developer because he will have to manually design the interface to external memory based on the manufacturer’s protocol. HLS tools have some support for external memory but they leave the interface largely to the developer to create, which we will discuss in section VI. To handle memory transfers efficiently, we assume the external BRAM we connect to can use burst transfers and byte enabling.

1) **Burst Transfers:** In a burst transfer each memory request deals with a series of transactions rather than a single transaction. For a read burst, the memory controller is told the starting address and the number of values to return (length of the burst). If the memory controller is busy it will assert a wait request causing the read burst to hold its current values until the wait request is deasserted. After several cycles, the value at the starting address is returned and subsequent values will be returned each clock cycle until the final value at $address + burstsize - 1$ is returned. For write burst transfers, the memory is provided with the starting address, number of transfers, and the starting value. As soon as the wait request is deasserted, the burst will continue with a new value for each clock cycle until the final value at $address + burstsize - 1$ is presented.

2) **Byte Enable:** Byte enable signals are useful for writing values to external memory which have a smaller bit width than the memory bus. It also allows the values used in the data path to be smaller than the values stored in external memory. The byte enable signal is used to specify which byte lanes in a memory write transaction are enabled. For example if the word values are 8 bits wide (1 byte) and the memory bus is 16 bits wide (2 bytes) as seen in Fig. 2, then it would be possible to read/write two values from memory during each memory transaction. For this example, the byte enable signal would be 2 bits wide, one bit for each byte on the memory bus. If a transaction involved only one data value, then the byte enable signal would be “01” where only the lower byte is enabled. This tells the external memory controller to only change the lower byte stored in memory while preserving the existing byte at that address.

### III. A Parallel for Loop Memory Design

As with most applications, programmers must adapt their algorithm to their target architecture(s). However, in hardware design, it is equally important to adapt your hardware architecture to the application. One very important issue in this respect is the number of parallel loop body instantiations used to increase execution speed. This number must be balanced with the available resources on the FPGA and the capacity of the memory system. Obviously if the FPGA has limited number of resources, then the number of parallel loop body
static void parfor(int start, int end) {
    unsigned char a[100];
    unsigned char b[100];
    int i;
    parfor (i=start; i<end; i++)
    { 
        b[i] = a[i]*a[i]+4*a[i];
    }
}

Fig. 4. Source code and architecture

instatnations will be limited. Furthermore, if the buffers can only support a limited number of memory requests in parallel, then this also will limit the number of parallel loop body instantiations.

As seen in Fig. 3, the buffers hold values transmitted/received from external memory until they are read/written by the data path. Each loop iteration body in the data path is independent so the buffers must supply or receive values in parallel so that no loop iteration body must wait on another. To do this, we matched each loop iteration body with one FIFO in the input buffer and one FIFO in the output buffer. As we will discuss later, the length of each FIFO must be large enough to handle one burst transfer from memory.

We decided to make use of the byte enable signal for a number of reasons. The most important one being that it ensures that adjacent values in memory are preserved if the data values are smaller than the values in main memory as shown in Fig. 2. Of course, doing this requires that the width of the data values must be a multiple of eight bit. It also increases the size of the values stored in the FIFOs by one bit, but this trade off allows for control over the size of the data values. The byte enable signal is unnecessary however if the width of the data values is the same as the width of the memory values read from external memory.

A. Manual Implementation

We first designed a manual implementation of the memory system which can be parametrized and used with our HLS compiler. We used the simple example shown in Fig. 4 (top) using the keyword parfor, which will be used later to tell the compiler that iterations of the loop body can be executed in parallel. Two multiplication operations were included in the loop body since they are resource intensive which limits the number of parallel iterations that can be instantiated. The array elements are unsigned char values or 1 byte so that we can use the byte enable signal discussed previously. As shown in the figure, we chose a 32 bit wide memory bus so that we can read and write up to 4 data values for each memory transaction. It was decided that the number of FIFOs would match the number of data paths, but there could be fewer in the input buffer depending on the resource constraints. The size of the bursts is determined at compile time by the designer bearing in mind that it must be balanced with the bandwidth for the data path, external memory and the size of the FIFOs selected. We will discuss these trade-offs further in Section IV.

Now we describe how we designed each component in Fig. 4.

1) Loop Controller: This controls the execution of the loop by telling the input and output buffers when to start collecting data and how many iterations to execute. When the output buffer signals its execution is finished the loop controller notifies the top level with a last cycle signal.

2) Loop Iterations in the data path: For this memory template, we assume that the loop bodies are not pipe-lined. This assumption has an impact on the system we designed. It means that each loop iteration can only process a new value after it finishes processing the old value. This is sometimes referred to as the latency, or the amount of time between when it starts and when the result is produced [4]. The latency is not always strictly obeyed by each iteration so some may take longer than others to complete so we designed the system with this in mind. This latency has an impact on the length of the FIFOs in the input buffer, which we will discuss in section IV.

3) Input Buffer: The input buffer was designed with the same number of FIFOs as the number of values that can be read in parallel from external memory. It also uses an address generator that calculates the starting address for each burst, and when values arrive from external memory, it will mark the bytes that are enabled before storing them in the appropriate FIFO. Each FIFO therefore stores values which are 1 bit wider than the data values themselves to signify if the value is enabled. This bit is then used downstream for generating the byte enable signal by the output buffer.

Each FIFO in the input buffer controls the flow of values to the parallel loop iteration body it is paired with by instructing it to start as soon as it has a new value in it. Once its loop iteration body has finished processing its result, it notifies the FIFO that it is completing its last cycle and will be ready for another value in the following cycle. If there is another value in the FIFO, then in the following cycle it will instruct the loop iteration body to start processing the new value. All of this activity is monitored by the input buffer controller so that it knows when the FIFOs have enough capacity for a new burst from external memory.

The input buffer is constantly monitoring the amount of free space that it has available and will request a new read burst from the memory arbiter before all the values in the input buffer are depleted. This ensures a continuous stream
of data from the input buffer to the data path. If the buffer does not have enough capacity for a burst, then it must wait until it has enough free space for another burst by allowing the downstream components to catch up. The buffer may also have to wait if the output buffer runs low on free space. The output buffer will notify the input buffer of this by asserting a wait request signal. When this happens, the input buffer will stop sending values to the data path until the wait request is deasserted by the output buffer.

4) Output Buffer: The output buffer is largely the mirror image of the input buffer. Its address generator is the same as the input buffer’s because in our example shown in Fig. 4(top) values for the output array are written in the same order as the values are read with only a different offset value. Like the input buffer, the output buffer has a controller that keeps track of the amount of free space it has and notifies the input buffer with a wait request whenever it is almost out of space. However, it must wait until it has enough values in its buffer before it can start a write burst whereas the input buffer, can begin as soon as the first value arrives. Therefore, the output buffer monitors the number of values it has buffered, and whenever it has enough to complete a write burst or has reached the end of the loop it notifies the memory arbiter that it is ready to perform a write burst.

5) Memory Arbiter: The memory arbiter regulates access between the input and output buffers since external memory can only process one burst request at a time\(^1\). Priority is always given to the downstream output buffer since it could potentially fill up before the input buffer completes its read burst, which could lead to dead lock. During our work we connected the memory arbiter to external memory via a PCI bus which was designed using the Quartus’ SOPC builder. In future versions, we plan to build targeted interfaces into the compiler so that we can bypass SOPC builder. The external memory for this work is assumed to be DDR memory which supports burst memory transactions with byte enable signals.

IV. PARAMETRIZED MEMORY TEMPLATE

As we have shown, the memory template we have proposed has a number of parameters that influence the resource utilization. If the application requires a significant amount of buffering, the memory design could dominate and affect the available space for the data path. Many of these considerations were encountered during our first implementation and we tried to build them into our HLS compiler. We have identified the key parameters shown in Fig. 5, and added them as command line parameters to the compiler. This allows the programmer to adjust the size of the different components in the template at compile time. Each parameter in the template influences the size and can influence the value for other parameters in the design.

The first parameter is the memory bus width which typically varies from 8 to 1024 depending on the architecture. It influences the number of data words that can be read in parallel from external memory which in turn dictates the width of the data words that are used.

\[
\text{word width} = \frac{\text{memory bus width}}{\text{number of words}}
\]

As we can see from this equation, the word width is dependent on the memory bus width and the number of words read in parallel. Obviously, the number of words should be a integral divisor of the memory bus width. The number of input FIFOs inside the input buffer should be a multiple of the number of words:

\[
\text{number of input FIFOs} = C_f \times \text{number of words}
\]

where the constant \(C_f\) is selected based on resource and performance constraints. The number of input FIFOs is a multiple of the number of words on the memory bus since each value must be read as soon as it becomes available on the input port. Finally, the number of loop iterations used must be a multiple of the number of FIFOs used by the input buffer as seen by this equation:

\[
\text{number of loop iterations} = C_i \times \text{number of input FIFOs}
\]

where the constant \(C_i\) is selected based on the resource and performance constraints so that each input FIFO supplies one or more loop iterations with data. A multiplexer is required when there are more loop iteration bodies than input FIFOs.

The number of loop body iterations that can be executed in parallel is resource dependent. If the loop body is resource intensive, then it will limit the number of iterations that can be instantiated and run in parallel. This could also affect the available space for the input and output buffers. The number of loop iterations and the number of output buffer FIFOs must
be equal since values produced by the iterations are written to the output buffer in parallel.

\[
\text{number of output FIFOs} = \text{number of loop iterations}
\]

Apart from the number of loop iterations, the length of each loop iteration body affects the length of the FIFOs in the input and output buffers.

The length [words] of each input FIFO is dependent on the size of the burst length [clock cycles] and the difference between the rate that words enter [words/clock cycle] (from external memory) and the rate at which words leave [words/clock cycle] (to the data path) where:

\[
\text{FIFO length} = \text{burst length} \times (\text{memory word rate} - \text{data word rate})
\]

As we can see from the equation if the data path can process words at the same rate (or faster) than they arrived from memory, then the FIFOs would be unnecessary. The memory word rate in the formula is architecturally dependent, but the data path word rate is dependent upon the length of the loop iteration’s body since each iteration can’t start a new word until the previous word has traveled the length of the data path. Once the word rate for the data path and external memory are set, then the length of each FIFO is proportional to the length of the burst. Furthermore, the output buffer’s FIFOs must have enough capacity for an entire memory burst plus additional capacity to compensate for the difference between the data path and memory word rates. Keeping these considerations in mind, we design two systems using our parametrized template.

---

2We assume that the memory and data paths are clocked at the same speed.
which should reduces the area and power consumption for the design. As we have shown in this design, the compiler can be used to reduce the number of input FIFOs, but we could have also adjusted other parameters of the template to explore different configurations. In the future, we hope to enable the compiler to perform these design explorations automatically when selecting (or optimizing) a template.

V. Future Work

We will be looking at developing memory templates for use with other types of parallel implementations of a for loop. For example streaming of parallel pipeline operations where there is a single loop body and new iterations are started on each clock cycle. Another type of loop would be a combination where there are multiple data paths each executing pipeline operations in parallel. Finally, we will look at adding pipe-lined for loops that take into consideration data reuse using a shift operation. A value that will be used in a later iteration is placed into a shift register which is shifted each clock cycle until it reaches the end of the register. At that point it is used by the iteration it was meant for, similar to the memory design in [6]. Once we have a number of memory templates to choose from, we will compare our compiler’s performance with other HLS compilers like ROCCC [7], AutoPilot [8], and Catapult-C [9].

At present we use the parfor keyword to tell the compiler when it is safe to execute the loop iterations in parallel, but in the future we will enable the compiler to analyze any for loop to see if it can be implemented as a parallel for loop, which would involve checking for loop dependencies. If the loop cannot be implemented as a parallel loop, then transformations should be carried out as in [10], [2], [11]. These transformations on the C-source code should be done by the user before compiling the source code to VHDL. Once the code is transformed, the compiler can select a memory template best suited for the application based on its data access patterns. The data access patterns should be analyzed using either data flow graphs like in [12] or make use of the polyhedral model as in [13]. If the compiler identifies that more than one templates is suitable for an application, then it must choose between them by estimating the resource utilization and performance of each.

The compiler should estimate resource utilization and performance when choosing which template to use. We envision the compiler would return an estimate of the resource utilization and performance for a each template chosen. However, as we saw in each of our automated designs, our template has parameters that control the number of loop iterations to instantiate, the number of FIFOs to use, and the length of each FIFO. At present each parameter is specified to the compiler by the user, but we hope to enable the compiler to find the value of each parameter which maximizes performance. In this way, the compiler can perform design exploration using each template and adjusting their parameters to find the one best suited for the application.

VI. Related Work

In this section we will look at how other HLS compilers allow the user to specify the interface with external memory. Many of them make use of FIFOs for addressing the memory gap, but do not allow for fine grained control over the interface with external memory which we hope to enable.

The Riverside Optimizing Compiler for Configurable Computing (ROCCC) is an open source C-to-VHDL compiler that was developed by the University of California at Riverside [7]. ROCCC expects the memory interface to support pipeline read requests using either synchronous or asynchronous valid and acknowledge (pop) signals. Its read interface works in the following way: upon receiving a series of read requests from the FPGA, the memory controller responds with the requested value and a valid signal. The FPGA will then send an asynchronous acknowledge (pop) signal which tells the memory controller to process the next value on the next clock cycle. If the memory controller is unable to support asynchronous valid/pop handshake, then the memory controller must pause before returning the next value after it sees the acknowledge signal on the rising clock edge. If this type of handshake between the memory controller and FPGA is used, then it reduces the maximum data transfer rate to the FPGA by half because the memory controller can only transfer values every other clock cycle [14]. Our system does not require asynchronous control signals since we use a burst memory protocol, which only requires a handshake when requesting a burst (see section II-B1 for a detailed description). Another problem we have found, is that the ROCCC compiler does not create the interface for writing to external memory, which is left to the user to create. Furthermore, it doesn’t handle cases where values are read and written from the same external memory. The ROCCC compiler also offers extensive loop transformation heuristics for optimizing data locality which is very useful. We also like the fact that it incorporates a “smart buffer” which allows for data reuses, and we hope to incorporate this into our compiler in the future.

AutoPilot, created by AutoESL, is a compiler that takes as input C, C++ or SystemC and generates RTL level VHDL/Verilog code [8]. According to an independent review [15], each input program must first be “restructured” so that it can run efficiently on FPGAs. Restructuring involves rewriting the source code so that it better emulates the parallel nature of the FPGA, which includes restructuring the code to use smaller buffers that can be used on the FPGA. Our compiler is similar in this respect where transformations or restructuring can improve the performance of the resulting HDL code. If the application makes use of external memory with AutoPilot, then additional C code must be included that describes the interface. This approach is useful, since the interface can be implemented in software, which could presumably be reused with other applications. We generate our interface at compile time using command line parameters to control its size and protocol. Additionally, our compiler can use external

3Read requests are made using address and address_ready signals.
macro files for describing the interface between functions and external components they connect to [16].

We also looked at Catapult-C’s compiler to see how it handles external memory interfaces. Its compiler is not meant for complete system design, rather it is useful for designing smaller components in a larger system. It is still up to the designer to generate the code for interfacing with external memory. Having said that, it does offer assistance for interfacing with on-chip memory such as FIFOs, single/dual port RAM, and other custom built-in I/O components [9]. This could allow the designer to adapt a component’s interface so it can connect with a manually designed on-chip memory controller or arbiter, which controls external memory. A benefit that we found using Catapult-C is that it will automatically generate a test-bench for verifying the design which was very useful.

VII. CONCLUSION

In this paper, we have introduced a parametrized memory template for applications with parallel for loops which have sequential loop bodies. As we have shown, this type of parametrized design allows the system designer to have fine grained control over the resulting memory structure without the burden of doing the entire HDL design manually. Furthermore, our design allows for burst memory transfers to and from external memory. The template also allows for data widths that are smaller than the memory bus width by using a byte enable signal. This instructs the external memory controller to only store the enabled byte lanes while ignoring the unenabled bytes and preserving the existing bytes at that memory address. We have shown two different implementations of a streaming buffer memory template that we have generated using our compiler. Each case carries trade-offs in terms of power, resources, and complexity and these trade-offs can be explored by changing the template’s parameter values. With further work, we hope to enable the compiler to automatically select the optimum parameter settings based on resource utilization and performance.

ACKNOWLEDGMENT

This research is supported by the I.W.T. grant 060068. Ghent University is a member of the HiPEAC Network of Excellence. The authors would like to thank Stephen Neuen-dorffer of Xilinx for his recommendations and suggestions.

REFERENCES