Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2008, Article ID 674340, 14 pages
doi:10.1155/2008/674340

Research Article

Area Optimisation for Field-Programmable Gate Arrays in SystemC Hardware Compilation

Johan Ditmar,[1] Steve McKeever,[2] and Alex Wilson[3]

[1] *Kellogg College, University of Oxford, 62 Banbury Road, Oxford OX2 6PN, UK*
[2] *Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK*
[3] *Celoxica Ltd., 66 Milton Park, Abingdon, Oxfordshire OX14 4RX, UK*

Correspondence should be addressed to Johan Ditmar, johan.ditmar@kellogg.ox.ac.uk

Received 21 June 2008; Accepted 18 August 2008

Recommended by Gustavo Sutter

This paper discusses a pair of synthesis algorithms that optimise a SystemC design to minimise area when targeting FPGAs. Each can significantly improve the synthesis of a high-level language construct, thus allowing a designer to concentrate more on an algorithm description and less on hardware-specific implementation details. The first algorithm is a source-level transformation implementing function exlining—where a separate block of hardware implements a function and is shared between multiple calls to the function. The second is a novel algorithm for mapping arrays to memories which involves assigning array accesses to memory ports such that no port is ever accessed more than once in a clock cycle. This algorithm assigns accesses to read/write only ports and read-write ports concurrently, solving the assignment problem more efficiently for a wider range of memories compared to existing methods. Both optimisations operate on a high-level program representation and have been implemented in a commercial SystemC compiler. Experiments show that in suitable circumstances these techniques result in significant reductions in logic utilisation for FPGAs.

1. Introduction

Hardware compilation translates a program written in a high-level language into a description of a hardware circuit. The ultimate aim is to take software code and produce an efficient digital system design. SystemC [1] is a language designed for this purpose, allowing modelling of hardware in C++ syntax. This capability allows a designer to work at a higher level of abstraction compared to RTL design. Furthermore, SystemC offers faster simulation, enabling rapid prototyping, and effective design exploration [2]. These benefits can result in a significant boost in productivity. SystemC was originally designed as a modelling language but there are now several hardware compilers for this language, one of which being the agility compiler [3].

This paper focusses on methods for area optimisation in hardware compilation. For ASICs, this can significantly reduce the chip area and thus the production costs involved. For FPGAs, improving logic usage may be a necessity, given that these devices have limited resources. There are a variety of ways to improve the logic usage of a design. Most of these are optimisation techniques that are known for a long time, well understood, and described in, for example, [4]. These techniques are part of the domain of logic synthesis and are performed on a gate-level description. At this level, they can be applied to both RTL synthesis and hardware compilation. This paper investigates two area optimisation methods that are specific to hardware compilation and are performed on a high-level program representation such as an abstract syntax tree or a control and data flow graph, rather than on a gate-level description. The first method implements function exlining, which is the task of mapping a function to a dedicated piece of hardware that is shared between calls. Our method implements exlining as a source-level transformation that can be supported in existing compiler frameworks with relatively little effort. The second optimisation technique automatically maps arrays in SystemC to multiport memories in hardware. This involves a novel procedure for automatically assigning concurrent array accesses to memory ports whilst avoiding resource conflicts.

Both optimisations have been implemented in the agility compiler, which is discussed in Section 2. Sections 3 and 4 describe the two optimisation methods and their implementations and demonstrate their benefits in minimising logic utilisation for FPGAs. Section 5 summarises the main findings of this study. Finally, Section 6 discusses the current limitations of the two implementations and explores possible avenues for future research.

2. Agility SystemC Compiler

The agility compiler [3] is a commercial hardware compiler intended for the compilation of a SystemC program to a hardware description. It provides facilities for creation, compilation and synthesis of a large subset of the SystemC language. In addition to support for an extensive range of input language constructs, the compiler back end can target a wide range of architecture-specific functionality for a variety of technologies, enabling efficient synthesis. Agility is a timed synthesis tool, accepting designs composed either of SystemC threads punctuated by wait statements, or fully synchronous or fully asynchronous SystemC methods. As a result, the cycle timing of synthesised output exactly matches that of an input design, significantly aiding functional verification.

2.1. Language Support

The agility compiler language support is extensive, including all of the synthesisable subset defined by the Open SystemC Initiative (OSCI) Synthesis Working Group [5]. This includes most C++ constructs, such as

(i) conditional statements — **if**, **switch**;

(ii) loop statements — **while**, **do** ... **while**, **for**;

(iii) control flow — **break**, **continue**, **return**.

In addition, agility supports C++ templates for generic programming in SystemC as well as object-oriented constructs such as (abstract) classes, inheritance, and polymorphism. Exceptions, dynamic (run-time) recursion, and dynamic pointer synthesis (including dynamic dispatch of virtual functions) are not supported, as their synthesis is either impossible on many devices (dynamic recursion) or would result in very inefficient hardware.

2.2. Synthesis

The agility compiler allows a designer to compile SystemC source code and produce different output formats: EDIF, VHDL, and Verilog. Figure 1 shows this design flow. When targeting FPGAs, agility can directly produce an EDIF netlist for Xilinx and Altera architectures. The EDIF is optimised and technology mapped and can be passed directly to the vendor's place and route tools. Alternatively, agility can produce RTL VHDL or Verilog for use with a third-party RTL synthesis or simulation tool.

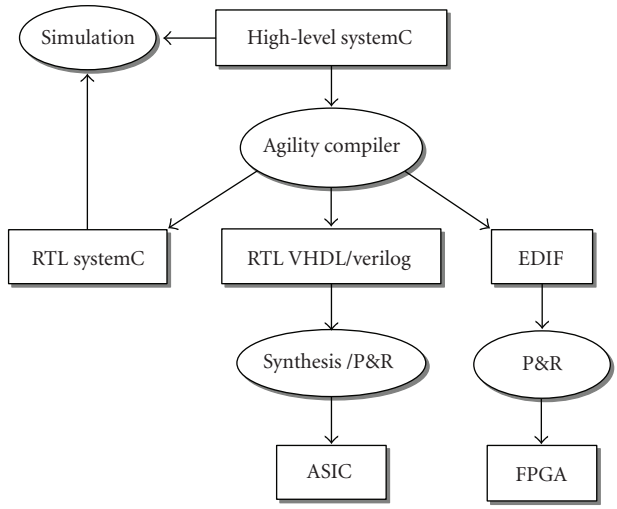


Figure 1: Agility design flow.

2.3. Verification Support

In addition to the aforementioned synthesis outputs, the compiler also supports the output of RTL SystemC for verification purposes. This output has exactly the same external interface as the synthesised input design, allowing the input design's test bench to be reused for functional verification. In addition, by design, this SystemC output is structurally identical to the RTL VHDL and Verilog output, allowing functional verification of the HDL output without requiring the use of an HDL simulator.

After synthesis through agility and then through target-specific place and route tools, final timing-level verification of fully synthesised designs can be achieved. This can be accomplished using the original SystemC test bench, a timing back-annotation of the HDL output and one of the several available mixed-language cycle-accurate simulators such as Aldec's Active-HDL [6] or Mentor Graphics's ModelSim [7].

3. Function Call Optimisation

Functions are commonly used in SystemC to divide a system up into tasks. Traditionally there are two methods for handling function calls in hardware compilation. One method, *inlining*, replaces each function call with the body of the function. Another method builds a single-hardware module for the function, which is subsequently shared between calls. This is called function *exlining*. Function exlining can potentially improve logic usage by reuse of the hardware associated with the function.

Function exlining has been implemented in several hardware compilers, such as [8, 9]. For these tools however, there is no description of how this optimisation is performed. This paper investigates the benefits of function exlining in hardware and describes a method for implementing this optimisation in SystemC compilation. It is shown that exlining can be adequately described in SystemC with the addition of asynchronous channels. This approach makes it possible

to implement the method as a source transformation in existing compiler frameworks with relatively little effort. A further benefit of this method is that it allows arguments to be passed by value as well as by reference without relying on run-time pointer resolution, a feature not supported by many hardware compilers.

The effects of function exlining on the efficiency of the produced hardware are discussed in the next section. Then in Section 3.2, the mentioned method for function exlining in SystemC is explained. Finally, Section 3.3 presents results that demonstrate function exlining in the agility compiler for various SystemC programs.

3.1. Function Calls in Hardware

This section describes function inlining and exlining in hardware compilation and the effect that these methods have on the size and speed of hardware designs. The benefits of function exlining are discussed, as well as the design restrictions that apply when using this function call method.

3.1.1. Inlining Versus Exlining

In software, a call to a function causes execution to jump to a new part of the code. Assuming the typical execution environment for a C++ program with registers and a stack, the registers and parameters get written to the stack just before the function call, then the parameters get read from the stack inside the function and read again to restore the registers when the function returns. These operations can add a significant time overhead, in particular for functions that take little time to execute. An inline function call is expanded without causing a function call. That is, the compiler inserts the complete body of the function in every context where that function is used. Inline expansion is typically used to eliminate the transfer of control overhead that occurs in calling a function. However, because inline calls are replaced with a copy of the function body, they can result in a significant increase in code size.

The notion of inline and exline functions applies similarly to hardware compilation. Here, exline functions are synthesised to separate modules that are shared between calls. Alternatively, inlining replaces function calls with the bodies of the called functions. Figure 2 shows a SystemC thread calling two functions `f` and `g` that have been defined elsewhere. Each wait statement represents the end of a clock cycle, except for the first wait, which marks the end of the reset cycle.

Function `f` has a single adder and a single multiplier in its datapath and has two states. States essentially correspond to wait statements in SystemC and their number largely determines how large the control logic for this function is in hardware. Function `g` is larger, both in terms of datapath logic and number of states.

Figure 3(a) describes the structure of the hardware synthesised from this program by exlining `f` and `g`. In this case, one hardware module is synthesised from one function. Therefore, only a single module is synthesised from function `f` despite having been called twice. After inlining, however, the function accessor will contain multiple instances of the function and the resulting hardware is larger. This is illustrated in Figure 3(b), which shows the hardware structure that is generated by inlining calls to `f` and `g`. From this example, it follows that function exlining results in smaller logic compared to inlining as hardware is being shared. However, this view is not the whole picture and there are other factors involved that affect the results when exlining.

(1) Exlined Functions Require Additional Multiplexers

If arguments are passed to an exlined function, and the function is called multiple times, multiplexers must be created in hardware to switch between arguments from different calls. The logic depth of these multiplexers and thus the delay through them increase with the number of function calls and so do their sizes. Function exlining can therefore potentially decrease the maximum frequency $f_{\max}$ of a design, if these multiplexers are in the critical path. Furthermore, the size of multiplexers can be significant, in particular, for FPGAs where they are implemented in general-purpose lookup tables [10]. If the function that is exlined is small, this means that the overhead of multiplexers could outweigh the benefits of exlining the function.

(2) Function Exlining May Hinder Resource Sharing

Resource sharing is an optimisation that automatically shares hardware resources between arithmetic operations in a program and is performed by many hardware compilers. Resource sharing is generally only performed on resources within the same module and those in different modules cannot be shared due to the difficulty of determining exclusive access to a resource from multiple threads of execution. This means that the hardware produced by function inlining, in which all hardware resources associated with functions become part of the same module, is more suited to resource sharing than exlining, in which a separate module is produced for each function. As a result, the size of the data path after exlining functions may be larger than the size after inlining [11]. Similarly, memory port sharing, as described in the second part of this paper, may also be hindered by function exlining.

When making the decision on whether to inline or exline calls to a function, it is therefore necessary to balance the circuit area saved by exlining against the added overhead associated with exlining.

3.1.2. Restrictions of Exlined Functions

The SystemC standard [1] does not specify when function calls should be inlined or exlined. In C++, functions are shared or exlined by default. By analogy, one could assume that exlining is a suitable default implementation of functions in SystemC. However, exlined functions are more restrictive in their use than inlined functions.

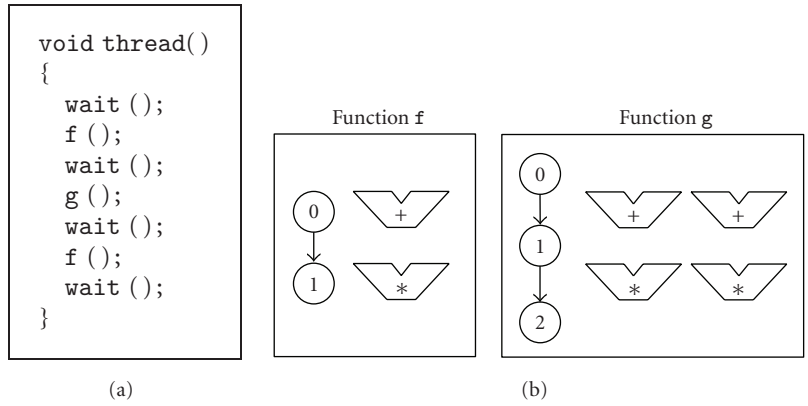


Figure 2: SystemC program calling two functions f and g.

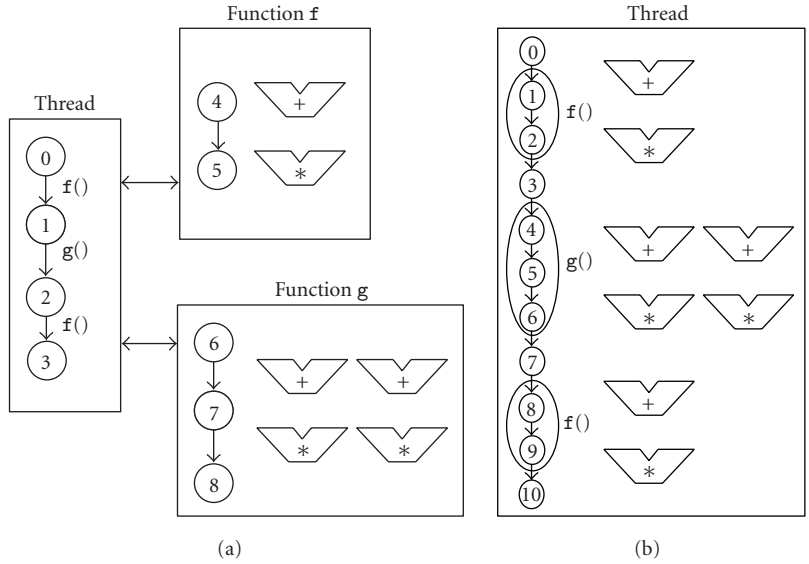


Figure 3: Function call methods in hardware. (a) Function exlining. (b) Function inlining.

(1) A Single Instance of an Exlined Functions Cannot Be Called in Parallel

Exlined functions cannot be called simultaneously from different threads as there is only one instance of each function to perform the task. Inlined functions do not have this restriction, as function instances are not shared in this case.

(2) Exlined Functions Cannot Be Called Recursively

Exlined functions cannot be called recursively as there is no stack in hardware. Functions labelled as inline can be called recursively without the use of a stack by means of recursive instantiation of the function, provided that the maximum recursion depth can be determined at compile time.

(3) Calls to a Particular Exlined Function Must Be in the Same Clock Domain

An exlined function must be in the same clock domain as its callers to avoid cross-clock domain synchronisation issues. Resynchronisation logic that is commonly used to resolve such issues would break SystemC timing semantics. Exlined functions can not therefore be called from multiple clock domains.

Despite these restrictions, exline functions are useful in hardware. They can greatly reduce the hardware size by sharing resources between calls. Unlike automatic resource sharing, exline functions allow sharing resources between threads and allow sharing of control path as well as data path logic. A hardware compiler will therefore benefit from supporting exlining as a method for synthesising function calls.

```
int f( int x )
{
  // function body
}

class Module : public sc_module
{
  sc_in < bool > clock;
  sc_in < int > input1, input2;
  sc_out < int > result;

  void thread( )
  {
    wait ( );   // end of reset cycle

    result = f( input1 );   // inlined call 1
    wait ( );
    result = f( input2 );   // inlined call 2
    wait ( );
  }

  Module(sc_module_name name) : sc_module(name)
  {
    SC_CTHREAD( thread, clock.pos( ) );
  }
};
```

Listing 1: SystemC program with function calls.

3.2. Synthesising Function Calls

This section describes a method for exlining function calls in SystemC synthesis. It explains how function exlining can be modelled in the SystemC language and how it has been implemented in the agility compiler.

3.2.1. Exlining in SystemC

In order to synthesise exline functions, it is useful to manually describe exlining in SystemC. This way, it can be evaluated and tested before implementation in a compiler. Furthermore, if function exlining can be modelled in SystemC, it can be conveniently implemented as a source transformation in a hardware compiler, rather than treating function exlining as a special case requiring significant additional functionality. To manually exline a function in SystemC, the following steps can be taken.

(1) The body of the function is moved to a newly created thread, inside an infinite loop.

(2) Handshaking is added between the function calls and the new thread to signal the start and end of function execution.

(3) Communication channels are added between the function calls and the new thread to transfer arguments and return value.

Step (1) is straightforward and involves creating a new thread that runs on the same clock as the calling thread or

```
class Module : public sc_module
{
  sc_in < bool > clock;
  sc_in < int > input1, input2;
  sc_out < int > result;

  // handshaking
  sc_async_signal < bool > start, done;
  // argument and return value
  sc_async_signal < int > arg_chan, rtn_chan;
  void thread( )
  {
    wait ( );   // end of reset cycle

    // exlined call 1 :
    arg_chan.write( input1 );   // send argument
    start.write( true );        // start execution
    while( !done.read( ) )   { wait( ); }
    result = rtn_chan.read( ); // recv return value

    wait ( );   // end of clock cycle

    // exlined call 2 :
    arg_chan.write( input2 );   // send argument
    start.write( true );        // start execution
    while( !done.read( ) )   { wait( ); }
    result = rtn_chan.read( ); // recv return value

    wait ( );   // end of clock cycle
  }

  // newly created thread for function f
  void f_thread( )
  {
    wait ( );   // end of reset cycle

    while(1)
    {
      while( !start.read( ) )   { wait( ); }
      int arg = arg_chan.read( ); // recv argument
      int rtn = f( arg );       // inlined call to f
      rtn_chan.write( rtn );   // send return value
      done.write( true );      // end execution
    }
  }

  Module(sc_module_name name) : sc_module(name)
  {
    SC_CTHREAD( thread, clock.pos( ) );
    SC_CTHREAD( f_thread, clock.pos( ) );
  }
};
```

Listing 2: SystemC program modelling function exlining.

accessor. Steps (2) and (3) require asynchronous communication between two clocked threads for which SystemC has no facilities. To communicate between threads, SystemC uses synchronous `sc_signal` channels that introduce a clock cycle latency. This means that if they were used for exlined function calls, there would be an overhead of several clock cycles in calling a function. While this is perhaps

```
void f( int * x, int * y )
{
  (*x)++;
  (*y)--;
}

void thread()
{
  int x = 1;    //initialise x

  wait();       // end of reset cycle

  f( &x, &x );   // x remains unchanged
  wait();

  output = x;   // output = 1
  wait();
}
```

Listing 3: SystemC program illustrating pointer aliasing.

```
int f( int arg )
{
  // function body
}

ag_share_routine( f ); // exline all calls to f
```

Listing 4: Agility directive for exlining a function.

acceptable in untimed synthesis, it would break the timing semantics of SystemC in which only `wait` statements take clock cycles. For the purpose of exlining, a new channel type is therefore introduced, called `sc_async_signal`. A channel of this type has the same interface as `sc_signal`, but implements asynchronous communication between synchronous threads. This channel type is used both for handshaking and transferring arguments.

3.2.2. Example

Listing 1 shows a SystemC program with two (inlined) calls to a function `f`.

In order to exline calls to `f`, a new thread `f_thread` is created, as shown in Listing 2. Two asynchronous channels, `start` and `done`, are introduced to signal the start and end of function execution. Two additional channels, `arg_chan` and `rtn_chan`, transfer argument and return value between the callers and the function.

The result is that only one instance of function `f` is created, rather than the two instances in the original program.

3.2.3. Passing Arguments by Reference

Function arguments in SystemC can be passed either by value or by reference. If an argument is passed by reference, a pointer to the argument is passed to the function. This pointer may then be derefenced inside the function which allows the argument to be modified. If the function is exlined, it can be accessed by multiple callers and pointers to arguments that need to be resolved during execution inside the function. This feature relies on a hardware compiler being able to synthesise pointers. Although pointer synthesis is possible, it tends towards producing inefficient hardware in terms of area and speed and at the same time offers little modelling benefit in the absence of dynamic memory allocation [12]. For this reason, not many hardware compilers support this feature, including agility. As a consequence, it would not be possible to modify arguments within a function.

Fortunately it turns out that if the value of a pointer argument is known at compile time for every caller of an exline function, then the call-by-reference can be replaced by a call-by-value without the need for pointers in hardware. This is achieved by dereferencing the pointer at the point of call rather than inside the function, and passing the result over an asynchronous channel to the function. The function receives this value and may modify it. After the function finishes execution, the modified value is sent back to the caller on a second, different channel and the call finishes.

Although this method allows arguments to be modified inside an exlined function, it has some limitations as well. By sending arguments over a channel rather than passing them by reference, the function will operate on a copy of the argument rather than the original. This requires that the argument must be of a copyable type that can be sent over a channel. Furthermore, the copy requires extra storage in the function and potentially increases sequential logic. Fortunately, this usually does not lead to an overall increase in logic area in FPGAs, except for register-rich designs such as those containing large register files.

Another potential issue arises when several arguments are passed by reference to a SystemC function where two or more pointers refer to the same object. In this case, changing one of the arguments inside the function may have an indirect effect on another argument. This effect is called *pointer aliasing* and is illustrated in Listing 3.

In this example, two pointers are passed to `f` that both point at the same integer `x`. The result is that `x` will first be incremented and then decremented and the effect is that `x` remains unchanged. When function `f` is exlined using the method described in this section, then `x` is sent on two different channels and the function will operate on two distinct copies of `x`. This would remove any pointer aliasing and cause a mismatch in behaviour between SystemC and hardware implementation: depending on the order in which channel communication happens, `x` will either be incremented or decremented. Fortunately, given the restriction that pointers must be resolved at compile time, pointer aliasing can be detected by the compiler.

```
int f( int arg )
{
  // function body
}

int f1( int arg )
{
    return f( arg ); // create instance of f
}

int f2( int arg )
{
    return f( arg ); // create instance of f
}

ag_share_routine( f1 );  // exline all calls to f1
ag_share_routine( f2 );  // exline all calls to f2
```

Listing 5: Creating multiple shared instances of a function.

3.2.4. Function Calls in Agility

Early versions of Agility only supported inlining as a method for synthesising function calls. In order to achieve better synthesis results, function exlining was added based on the method described in this section. Together with the addition of asynchronous channels to Agility, the method was implemented as a source-to-source transformation on the abstract syntax tree (AST), the compiler's internal representation of a SystemC program.

The obvious way to control function call expansion in Agility is to use the `inline` keyword and other C++ rules set out in [13]. This approach however has several disadvantages. Firstly, it would change the behaviour of existing designs that rely on functions being inlined rather than exlined. Exlining function calls that were previously inlined would not only affect the hardware that is produced, but would potentially break the design due to the restrictions of exline functions that were mentioned in Section 3.1.2. Furthermore, the rules for inlining in C++ are not strict and merely hint to the compiler that inlining is preferred. This would not provide many users the control they desire. For these reasons, a new synthesis directive, `ag_share_routine`, was added to Agility to exline a function and automatically perform the described source-to-source transformation. This directive takes the function to be exlined, which is illustrated in Listing 4.

In this example, all calls to function `f` are exlined and a single instance is created in hardware. In order to decrease multiplexer depth and improve clock frequency, it is sometimes beneficial to map a function to multiple shared instances instead. This can be achieved in Agility by creating several exline functions for each call `f` as is illustrated in Listing 5.

In this example, if `f` is always called via `f1` or `f2`, no more than two shared instances of `f` are created in hardware.

3.3. Results

Experiments were performed to demonstrate the effect of function exlining and inlining on the efficiency of hardware produced by Agility. For this purpose, three designs in SystemC were used: an inverse discrete cosine transform (IDCT), calculating the determinant of a 3×3 matrix (DET), and multiplying two 3×3 matrices (MULT). These designs all contain a function that is called multiple times and can be exlined. Each design was compiled to EDIF and implemented on an Xilinx Virtex-4 device in two versions: one inlining and another exlining the function. Post-implementation simulations were performed to verify that both versions are equivalent. Table 1 shows the number of slices and maximum clock frequency $f_{\max}$ for each design for the two function call methods as reported by the Xilinx tools. For each design, the table also lists the size of the function that is exlined as well as the number of calls to this function and the number of arguments. All arguments are 32-bit wide.

From these results, it follows that function exlining reduces the size of all designs, in particular those containing a large function such as MULT. For smaller functions and those with many arguments, the overhead of multiplexers that are created to switch between arguments from different calls becomes noticable. This is true for the IDCT example, where exlining only has marginal effect. The same multiplexers also add to the logic delay and can reduce $f_{\max}$ if they become part of the critical path. This is the case for DET and MULT, where the the maximum frequency is significantly reduced by exlining.

4. Array Optimisation

The array is a commonly used data structure in SystemC and can be mapped in different ways to hardware. Normally, arrays are mapped to register files. This implementation matches the behaviour of arrays in SystemC, but is not very efficient in terms of performance and logic area. ASIC libraries generally include efficient RAM components and modern FPGAs typically contain a large number of RAM blocks which can be used to implement arrays instead. Memories have a limited number of ports, and part of the process of mapping arrays to memories is assigning each memory access to a port such that contention is prevented. Many RTL synthesis tools can infer RAMs from arrays, but they require that the designer assigns access to ports manually. High-level languages do not offer this kind of control and a hardware compiler must therefore be able to automatically assign each array access to a memory port such that no port is accessed multiple times in parallel.

The problem of automatically assigning memory accesses to ports has received little attention by itself. The reason is that this problem has traditionally been solved using general resource sharing methods such as described in [14]. As we shall show, these methods cannot be used for all types of memories and thus a different approach must be taken. This

Table 1: Comparison between function call methods for various designs targeting an XC4VLX40 FPGA.

Example	Func size (slices)	Calls	Args	Method	Size (slices)	f_{max} (MHz)
IDCT	1,008	2	9	inline	6,213	69
				exline	6,165	69
DET	541	3	4	inline	2,418	74
				exline	1,715	61
MULT	2,424	3	3	inline	7,218	78
				exline	2,747	64

paper proposes an algorithm to solve this problem. The algorithm has been implemented in the agility compiler.

The effects of mapping arrays in SystemC to memories in hardware are discussed in the next section. Then in Section 4.2, existing research in this field is examined. Our proposed method for assigning array accesses to memory ports is discussed in Section 4.3. Finally, Section 4.4 presents results that show the benefits of using this method in minimising logic utilisation for FPGAs.

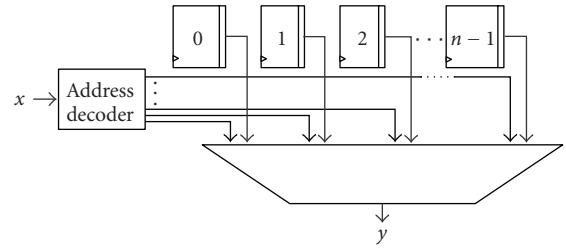


Figure 4: Array read access in hardware.

4.1. Arrays in Hardware

In C++, an array is represented by a continuous memory segment containing all array elements in a representation corresponding to their type. By analogy, one could assume that a memory is a suitable hardware implementation of an array in SystemC, both being multi-dimensional representations of bits. However, arrays in SystemC have different semantics from memories.

(1) SystemC Arrays Offer Parallel Access to Elements

In SystemC, a design can access multiple array elements in the same clock cycle and there are no restrictions to the number of parallel accesses. A memory on the other hand has a limited number of ports which means that only a limited number of simultaneous accesses is allowed.

(2) SystemC Arrays are Accessed in One Cycle

In timed SystemC threads, only wait statements take clock cycles and nothing else. An array access must therefore finish within one cycle. Many architectures however support synchronous memories, where a read operation is controlled by the system clock and takes two cycles: one to setup the address and one to read the data. To match the SystemC timing semantics, memory accesses could be pipelined in an attempt to establish the address one cycle ahead. However, this is only possible in certain program contexts.

(3) SystemC Arrays Have Write-Before-Read Semantics

In SystemC, when an array write is followed by an array read in the same cycle from the same address, the value that is read is the value that has just been written in the same cycle. In hardware, many multi-port memories have read-before-write behaviour, which means that a value that is written does not become available until the next clock cycle. Consequently, any value that is read has always been written in an earlier cycle. This behaviour can cause a mismatch between SystemC model and implementation.

Because of the difference in behaviour between arrays in SystemC and memories in hardware, not all arrays can be implemented in memory. For this reason, Agility implements an array as a register file by default, consisting of registers and combinational logic. This implementation however may use considerable logic resources. This is illustrated in Figure 4, showing the hardware that is built for a read operation `y = Array[x]` from a register array with `n` elements. Each array element requires a register in hardware. An address decoder translates address `x` into a bit vector which controls the output multiplexer. This multiplexer selects the output of the particular element that is indexed by `x`. If an array is read several times, several address decoders and multiplexers are required.

Figure 5 shows the hardware that is built for a write operation `Array[x] = y` to a register array with `n` elements. In this case, the output lines of the address decoder are connected to the write enables of the registers to select which element to write to. If an array is written to multiple times, multiplexers are required on the inputs of the registers to select which data to write.

With more complex systems being developed onFPGA platforms, the need for storage in these devices is increasing. Thus modern FPGAs contain a large amount of on-chip memory. This memory can be targeted automatically from arrays in a SystemC program. Mapping arrays to memory rather than general purpose logic can significantly reduce the logic usage of a design.

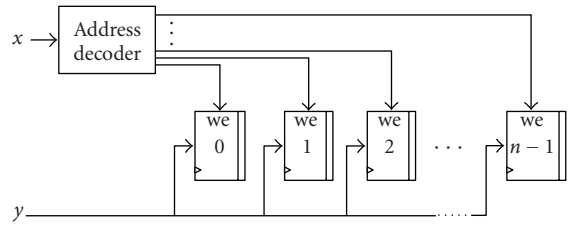


Figure 5: Array write access in hardware.

4.2. Related Research

The problem of synthesising memories from arrays has received attention in the past, though most of this research has focussed on efficient mapping of arrays to physical memory blocks [15, 16]. The problem of assigning memory accesses to ports has not been investigated by itself. The reason is that memory accesses are treated as normal data path operations and are covered by general sharing methods such as described in [14, 17]. In these methods, a compatibility graph is built where two operations are compatible when they can be assigned to the same resource, independent of the type of resource. This is not always the case for memory accesses. For example, suppose a particular memory has one read only port and one read-write port. Whilst read operations can use either port, write operations can only be assigned to the read-write port. The compatibility between a read access and a write access thus depends on which port the read access will be assigned to. Consequently, a global clique partitioning algorithm operating on a compatibility graph cannot be applied to solve this problem.

Assigning array accesses to memory ports can also be performed using a constructive approach, in which operations are assigned to functional units in a step-by-step fashion [18]. For each memory access, such an algorithm attempts to find a memory port that is capable of executing the read or write operation and that has not been assigned yet in the current clock cycle. In the case where there are two or more memory ports that meet these conditions, the one which results in minimum multiplexer depth is chosen. Whilst this method is simple, it is based on local information only and therefore often leads to suboptimal results. This is true particularly in the presence of exclusive branches, where an efficient assignment of accesses in one branch depends on the accesses present in the other branches.

Another paper that addresses the problem of assigning operations to functional units is [19]. This method builds, for each clock cycle, a bipartite graph containing the operations that are executed in this cycle together with functional units that the operations can be assigned to. All edges in the graph run between operations and functional units and specify whether an operation can be performed on a certain unit. As with clique partitioning, weights can be associated with these edges, representing the costs associated with particular assignments. The problem of assigning each operation to a unique functional unit, such that the sum of all edge weights is minimal, is called *weighted bipartite matching*.

The bipartite graph does not contain information regarding compatibility between operations and it is therefore not possible to assign operations that are executed in mutually exclusive branches to the same functional unit. To overcome this limitation, the method proposed in this paper uses a transformed bipartite graph which, instead of nodes representing operations, contains nodes representing sets of operations that are executed in mutually exclusive branches. Each of these sets can then be assigned to functional units using weighted bipartite matching. To build this type of bipartite graph for assigning memory accesses, the algorithm must analyse the program to gather all memory accesses that are executed in a particular cycle and merge those that occur in mutually exclusive branches. An algorithm for performing this analysis is presented in the next section.

4.3. Proposed Method

To assign memory accesses to ports, the algorithm needs to determine which accesses may occur simultaneously and which are independent. If two accesses are erroneously determined to be independent, incorrect hardware will be produced that suffers from memory port contention. On the other hand, it is acceptable if the algorithm is conservative and determines that two accesses can occur at the same time, when in fact they cannot. The proposed method is divided into two parts: access analysis and port assignment. The first part analyses the semantic structure of the program to determine which memory accesses are independent. This is the case if they are separated by a wait statement or are in different branches of an if/switch statement. The information that is gathered by access analysis is then used by the port assignment algorithm in order to assign accesses to ports.

4.3.1. Control Flow Representation

In order to describe the algorithm, a SystemC program is represented in a control flow graph (CFG). A CFG is a representation, using graph notation, of all paths that might be traversed through a program during its execution. The nodes represent operations and directed edges are used to represent jumps in the control flow. For the purpose of assigning memory accesses, a CFG is presented in which there are four node types: *conditional forks*, *conditional joins*, *waits*, and *basic blocks*. A basic block is a sequence of operations that is always entered at the beginning and exited at the end. Without loss of generality, it is assumed here that a basic block contains at most a single-memory access.

Figure 6(a) shows a SystemC program with conditional constructs in which an array is accessed. Figure 6(b) shows the corresponding CFG, containing three basic blocks.

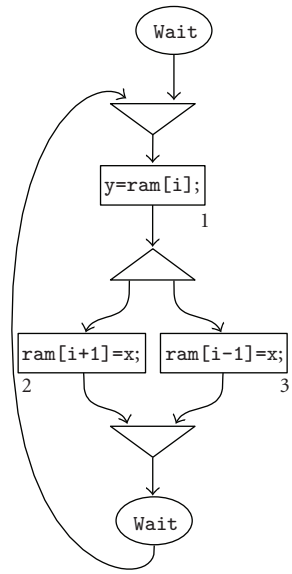
Cycles in the CFG are created by loops in the SystemC program. It is assumed that all combinational loops in SystemC will have been unrolled by the compiler at this stage. Consequently, cycles in the CFG always contain at least one wait node, as they cannot otherwise be implemented in synchronous hardware.

```
void thread( )
{
  sc_uint ⟨8⟩ i = 0;
  wait ( );
  while (1)
  {
   y = ram[i];
   if (i < 128}
   {
    ram[i + 1] = x;
   }
   else
   {
    ram[i − 1] = x;
   }

   i ++;

   wait ( );
   }
}
```

(a)



(b)

Figure 6: Control flow representation. (a) SystemC description, (b) control flow graph.

4.3.2. Access Analysis

The access analysis algorithm gathers, for each clock cycle, sets of independent memory accesses that are assigned to different memory ports. It performs this process independent of the number of memory ports available and the access types of these ports. It attempts to combine memory accesses in such a way that the final number of sets, and thus the number of required memory ports, is minimal. Some sets may contain both read and write accesses and must be mapped to read-write ports. As not all memories have these ports, these sets may later have to be split into sets that can be assigned to simple ports. This increases the number of required memory ports and the algorithm therefore attempts to minimise the number of sets with mixed accesses.

The algorithm takes the CFG as the input. It then splits it into directed acyclic subgraphs (sub-DAGs) corresponding to individual clock cycles and processes them separately. One way of doing this is to remove all wait nodes and edges incident upon them from the CFG. Then the graph can be split into subgraphs by temporarily regarding all edges as undirected, and finding all connected nodes (e.g., using depth-first search). As all cycles in the CFG contain at least one wait node, all directed subgraphs thus obtained will be acyclic. To assign accesses to memory ports, the basic blocks in each sub-DAG corresponding to a clock cycle are traversed in topological order, starting with accesses early in the cycle. During traversal, the algorithm gathers sets of independent memory accesses that can be mapped to the same memory port. This information is represented as a triple list of sets (r, w, rw) of accesses as follows:

(i) r contains sets of independent read accesses;

(ii) w contains sets of independent write accesses; and

(iii) rw contains sets of independent read and write accesses.

Accesses in the same set can be mapped to the same memory port and accesses in different sets must be mapped to different ports to prevent resource conflicts. Consequently, the minimum number of memory ports required to assign all accesses in the triple to ports is equal to the total number of sets in the triple. For example, suppose the triple is equal to

$$(r, w, rw) = ([\{r_1, r_2\}], [\{w_1\}, \{w_2\}], [\{r_3, w_3\}]). \quad (1)$$

In this case, at least four memory ports are required to assign all accesses: one port capable of reading, two ports capable of writing, and one port capable of both reading and writing.

Pseudocode for the algorithm that gathers all accesses to a particular memory in a program is shown in Listing 6.

The map `accesses` store the access triple at each basic block and are used for temporary storage. `pred` and `succ`, respectively, return the parents and children of a basic block. When a particular basic block is encountered, the access triples of its predecessors are merged to combine accesses from mutually exclusive branches. Then, the memory access in the current basic block is added to the triple. After all basic blocks in a clock cycle have been visited, the final access triple is calculated by merging those basic blocks without successors. Function `effect` models a memory access in a basic block and appends the access as a singleton set to the end of the appropriate list in the access triple

```
Assign(CFG)
{
  result := ∅
  for each g ∈ subDAGs(CFG)
  {
    accesses := ∅
    for each blk ∈ topsort (basic blocks in g)
    {
      // merge control flows into blk
      merged := merge {accesses [b] | b ∈ pred[blk]}

      // add the effect of a memory access in blk
      accesses [blk] := effect blk merged
    }
    result[g] := merge {accesses [b] | succ[b] =∅}
  }
  return result
}
```

Listing 6: Memory access analysis algorithm.

```
mergetwo : triple -> triple -> triple
mergetwo (r1,w1,rw1) (r2,w2,rw2) = (fr, fw, frw)
 where (r1r2, r1', r2')      = combine r1 r2
       (w1w2, w1', w2')      = combine w1 w2
       (rw1rw2, rw1', rw2')  = combine rw1 rw2
       (r1rw2, r1'', rw2'')  = combine r1' rw2'
       (w1rw2, w1'', frw2)   = combine w1' rw2''
       (rw1r2, rw1'', r2'')  = combine rw1' r1''
       (rw1w2, frw1, w2'')   = combine rw1'' w1''
       (r1w2, fr1, fw2)      = combine r1'' w2''
       (w1r2, fw1, fr2)      = combine w1'' r2''

       -- take sets of accesses from two lists
       -- xs and ys and combine them into zw
       combine xs ys = (zw,drop n xs,drop n ys)
        where zw = zipwith (∪) xs ys
              n  = length zw

       -- the merged triple is created from the
       -- combined sets plus any remaining sets
       fr  = r1r2 ++ fr1 ++ fr2
       fw  = w1w2 ++ fw1 ++ fw2
       frw = rw1rw2 ++ r1rw2 ++ w1rw2 ++
             rw1r2 ++ rw1w2 ++ r1w2 ++
             w1r2 ++ frw1 ++ frw2
```

Listing 7: Function for merging two control flows.

(r, w, rw). Read accesses are added to r and write accesses to w. Function `merge` combines the memory acesses in a number of mutually exlusive branches, for example at the end of an if statement. The aim of `merge` is to combine accesses in the most efficient way as to minimise the number of memory ports required. The algorithm for merging two triples is shown in Listing 7 in functional programming notation.

The merging of two triples P and Q consists of repeatedly picking a set of accesses from P and a set from Q and taking the union until either P or Q is empty. If any sets remain in P or Q, these sets are just inserted into the merged triple. There are many ways in which sets in P or Q can be combined, where some combinations are more favourable than others. For example, suppose two access triples P and Q are defined as

$$P = ([\{r_1\}], [\{w_1\}], []),$$
$$Q = ([\{r_2\}], [\{w_2\}], []). \tag{2}$$

One possible way to merge P and Q is to combine read accesses with write accesses: $([], [], [\{r_1, w_2\}, \{r_2, w_1\}])$. Although this requires two memory ports, not all memories have read-write ports. A better way to merge P and Q is $([\{r_1, r_2\}], [\{w_1, w_2\}], [])$, which requires one read port and one write port. The merge function therefore favours combinations between accesses of the same type over accesses of different types. In addition, merge attempts to avoid combining sets that merely contain read accesses with those that merely contain write accesses. For example, if P and Q are defined as

$$P = ([\{r_1\}], [], []),$$
$$Q = ([], [\{w_1\}], [\{r_2, w_2\}]), \tag{3}$$

then P and Q can be merged by combining $\{r_1\}$ and $\{w_1\}$: $([], [], [\{r_2, w_2\}, \{r_1, w_1\}])$, or $\{r_1\}$ and $\{r_2, w_2\}$: $([\{r_1\}], [], [\{r_2, w_2, r_1\}])$. Both solutions require two memory ports. However, when read access $\{r_1\}$ and write access $\{w_1\}$ are combined, a set with mixed access types is created which requires an additional read-write port.

Figure 7 shows a fragment of a SystemC program, together with the control flow subgraph for the particular clock cycle.

The code contains two read accesses (1 and 3) and three write accesses (2, 4, and 5). Table 2 shows the progress of access analysis whilst traversing the CFG. The final access triple for this clock cycle is $([], [\{2, 5\}], [\{3, 4, 1\}])$, which can be assigned to a memory with one write port and one read-write port.

Table 2: Steps showing progress of access analysis.

Operation	Accesses (r, w, rw)
`a = ram[p];`	$([\{1\}], [], [])$
`ram[q] = b;`	$([\{1\}], [\{2\}], [])$
`c = ram[r];`	$([\{3\}], [], [])$
`ram[s] = d;`	$([], [\{4\}], [])$
`merge(3,4);`	$([], [], [\{3, 4\}])$
`ram[t] = e;`	$([], [\{5\}], [\{3, 4\}])$
`merge(2,5);`	$([], [\{2, 5\}], [\{3, 4, 1\}])$

4.3.3. Assigning Accesses to Ports

To assign accesses to ports, a bipartite graph is constructed for each clock cycle from the access triple (r, w, rw). The sets of accesses in r and w can be mapped to read/write only ports as well as read-write ports whilst the sets in rw can be mapped to read-write ports only. The latter therefore gives the algorithm less freedom in assigning accesses to ports optimally. Furthermore, not all memories have read-write ports. For this reason, the access triples are transformed as to minimise the size of rw. This transformation involves repeatedly splitting sets in rw into two sets: one containing the read and one containing the write accesses. These are then added to r and w respectively. This process increases the number of required memory ports and is repeated until the number of required ports is equal to the number of ports on the memory or until rw is empty. For example, the final access triple in the example of Figure 7 was $([], [\{2, 5\}], [\{3, 4, 1\}])$, requiring one write port and one read-write port. If these accesses are mapped to a memory with one read-only and two write only ports instead, the triple is transformed as $([\{3, 1\}], [\{2, 5\}, \{4\}], [])$.

After the transformation, the bipartite graph is built. Each edge in the graph has a weight associated with it. This is a measure of the cost of binding the accesses in a set to

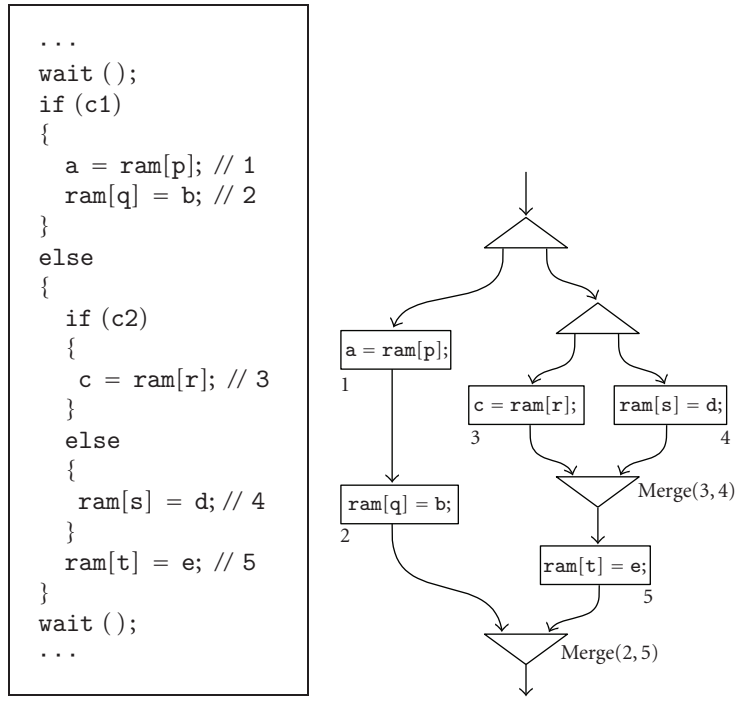


Figure 7: Memory access analysis. (a) SystemC fragment, (b) control flow subgraph corresponding to clock cycle.

a particular port. In the proposed algorithm, the weight on an edge (a_i, p_j) is based on the total number of accesses bound to port p_j if the set of accesses a_i is assigned to it. This weight is a global cost which takes into account accesses that were assigned in other cycles as well. This way, memory accesses will be balanced between ports and the depths of multiplexers on the input of ports is minimised. After building the graph, bipartite matching is performed to assign all accesses to ports.

4.4. Results

Experiments were performed to demonstrate the effect of mapping an array to memory. For this purpose, an inverse discrete cosine transform (IDCT) was used. An implementation of this algorithm was written in C by the MPEG Software Simulation Group (MSSG) [20] which contains an array requiring 1 kb of storage. This design was ported to SystemC and compiled to EDIF with and without mapping the array to memory. The generated EDIF for both array implementations was passed to the Xilinx design tools and implemented on a Virtex-4 device. Post-implementation simulations were performed to verify that both implementations are functionally correct. Table 3 shows the number of flip-flops and slices for the IDCT design for the two array implementations as reported by the Xilinx design tools.

From these results it follows that mapping the array to memory significantly reduces the size of the IDCT design. The proportion of used slices on the particular FPGA device is reduced from 34 percent to 15 percent, whilst only 1 out of 96 available RAM blocks is needed for implementing the array. As a result, the design can be implemented on a smaller, and thus cheaper, FPGA device.

Table 3: Comparison between array implementations for IDCT design targeting an XC4VLX40 FPGA.

Array impl.	Flip-flops	Slices	Logic usage (%)
registers	1,687	6,213	34
memory	663	2,724	15

5. Conclusion

This paper has presented two area optimisation procedures for FPGAs in SystemC hardware compilation. The first is function exlining, which aims to reduce the logic size of a design by mapping a function to a separate piece of hardware that is shared between calls. It has been shown that function exlining can be described in SystemC with the addition of an asynchronous channel to the language. This method can be easily implemented in a hardware compiler as a source transformation and performed automatically. The second optimisation algorithm deals with mapping arrays in SystemC to memories in hardware. This method analyses the program and gathers sets of independent accesses that can be mapped to the same memory port whilst avoiding resource conflicts. Compared to previous methods, it solves the assignment problem more efficiently for a wider range of memories. Both optimisations can help to transform a

behavioural specification into an efficient implementation in hardware. This is in the spirit of hardware compilation, where designers should focus on the algorithm itself rather than on manually optimising code.

The proposed methods were implemented in the agility compiler and experiments were performed that showed the benefits of these methods in reducing logic utilisation in FPGAs. It was found that function exlining can greatly improve the logic usage of a design. However, sharing a function between different callers also introduces multiplexers to switch between arguments. The overhead of these multiplexers in terms of logic size and delay is potentially large for FPGAs, where they are implemented in general-purpose lookup tables. It is therefore necessary to balance the circuit area saved by exlining against the added overhead associated with exlining. Whilst function exlining saves resources by sharing them between tasks, mapping arrays to memories is based on choosing a different hardware implementation for a given task. Modern FPGAs contain a large amount of on-chip memory and this method allows a designer to target this abundant resource without significantly changing a design's specification. As experiments showed, this can significantly reduce logic area such that the design can be implemented on a smaller, and thus cheaper, FPGA device.

6. Limitations

Although the optimisation techniques described in this paper have shown promising results, there are several opportunities for improvement as well as for further research. Function exlining in agility is currently controlled by the user via the `ag_share_routine` directive. If a function is specified as shared, all calls to this function are exlined and care must be taken that no resource conflicts arise due to simultaneous calls to the function. The decision to exline a function could be made by the compiler instead. A method to achieve this is described in [21].

In the current implementation, all calls to an exline function share the same hardware module. In order to optimise multiplexer usage and avoid resource conflicts, a SystemC function could be mapped to multiple hardware modules instead. In this approach, function calls with common arguments could be detected through static analysis and combined in order to reduce multiplexer depth and thus improve clock frequency.

The current implementation supports arguments passed by reference without the need to resolve pointers during execution. As discussed, this not only poses restrictions on the type of arguments passed, but also increases sequential logic. A solution using run-time pointer resolution would avoid these issues, a method for which is presented in [22].

In the proposed algorithm for synthesising arrays, each array is mapped to a separate logical memory. This can be inefficient if a program contains several arrays that are smaller in size than the available memory components. In theory, those arrays could be implemented in a single memory thereby reducing memory cost. Schmit and Thomas [16] propose a method for grouping arrays of different sizes and dimensions and packing them into memories, which is suited to this purpose.

The bipartite matching algorithm that is used to assign memory accesses to ports is based on a cost function. In the current implementation, this function only takes multiplexer depth into account and attempts to balance accesses between ports. The algorithm could be improved by using true delay information instead.

Finally, this paper discussed the interaction between function exlining and other optimisation techniques. It was mentioned that function exlining may hinder other optimisations, such as resource sharing and memory port sharing, which are typically performed on each module individually rather than globally. More research is required to investigate how the proposed optimisation techniques can be extended to operate optimally together and in synergy with other optimisations.

References

[1] Open SystemC Initiative, "*SystemC 2.1 Reference Manual*," 2005, http://www.systemc.org/.

[2] T. Rissa, A. Donlin, and W. Luk, "Evaluation of systemC modelling of reconfigurable embedded systems," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '05)*, vol. 3, pp. 253–258, Munich, Germany, March 2005.

[3] Celoxica, "*Software Product Description for Agility Compiler v1.2*," 2006.

[4] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, New York, NY, USA, 1994.

[5] OSCI Synthesis Working Group, "*SystemC Synthesizable Subset, Draft 1.1.18*," December 2004, http://www.systemc.org/.

[6] Aldec, *Active-HDL*, http://www.aldec.com/.

[7] Mentor Graphics, *ModelSim*, http://www.model.com.

[8] S. Gupta, R. K. Gupta, N. D. Dutt, and A. Nicolau, "Coordinated parallelizing compiler optimizations and high-level synthesis," *ACM Transactions on Design Automation of Electronic Systems*, vol. 9, no. 4, pp. 441–470, 2004.

[9] Agility Design Solutions, "*Handel-C Language Reference Manual*," 2008, http://www.agilityds.com/literature/HandelC_Language_Reference_Manual.pdf.

[10] J. Ditmar, *Area optimisation in systemC hardware compilation*, M.S. thesis, University of Oxford, Oxford, UK, 2007.

[11] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Function call optimization in behavioral synthesis," in *Proceedings of the 9th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools (DSD '06)*, pp. 522–529, Dubrovnik, Yugoslavia, August-September 2006.

[12] E. Grimpe and F. Oppenheimer, "Extending the systemC synthesis subset by object-oriented features," in *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '03)*, pp. 25–30, Newport Beach, Calif, USA, October 2003.

[13] ISO/IEC 14882:1998, "*Programming languages—C++*," ISO/IEC, Geneva, Switzerland, 1998.

[14] S. Raje and R. A. Bergamaschi, "Generalized resource sharing," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '97)*, pp. 326–332, San Jose, Calif, USA, November 1997.

[15] D. Karchmer and J. Rose, "Definition and solution of the memory packing problem for field-programmable systems," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '94)*, pp. 20–26, San Jose, Calif, USA, November 1994.

[16] H. Schmit and D. E. Thomas, "Array mapping in behavioral synthesis," in *Proceedings of the 8th International Symposium on System Synthesis (ISSS '95)*, pp. 90–95, Cannes, France, September 1995.

[17] C.-J. Tseng and D. P. Siewiorek, "Automated synthesis of data paths in digital systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 5, no. 3, pp. 379–395, 1986.

[18] K. Kucukcakar and A. C. Parker, "Data path tradeoffs using MABAL," in *Proceedings of the 27th ACM/IEEE Design Automation Conference (DAC '90)*, pp. 511–516, Orlando, Fla, USA, June 1990.

[19] C.-Y. Huang, Y.-S. Chen, Y.-L. Lin, and Y.-C. Hsu, "Data path allocation based on bipartite weighted matching," in *Proceedings of the 27th ACM/IEEE Design Automation Conference (DAC '90)*, pp. 499–504, Orlando, Fla, USA, June 1990.

[20] MPEG Software Simulation Group, http://www.mpeg.org/.

[21] F. J. Kurdahi and A. C. Parker, "REAL: a program for REgister allocation," in *Proceedings of the 24th ACM/IEEE Design Automation Conference (DAC '87)*, pp. 210–215, Miami Beach, Fla, USA, June-July 1987.

[22] L. Semeria and G. De Micheli, "SpC: synthesis of pointers in C application of pointer analysis to the behavioral synthesis from C," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '98)*, pp. 340–346, San Jose, Calif, USA, November 1998.