Rapid Implementation of Floating-point Computations Using Phase-Coherent Dynamically Configurable Pipelines

D. Rutishauser and R. Shuler
Avionic Systems Division, NASA Johnson Space Center
Houston, Texas, U.S.A.

Abstract - *The Phase-Coherent Dynamically Configurable Pipeline is a concept for the rapid implementation of pipelined computational algorithms in configurable hardware. The approach allows a high level of sharing of floating-point resources among multiple computations. The concept features a simple tag-based control scheme and a sparse-pipeline allocation approach that enables all the stages of an arithmetic pipeline to be processing simultaneously, with multiple computations allocated to the same pipeline. Thus the approach increases hardware resource utilization and reduces power consumption. A framework is presented that implements the concept. The current framework targets a Field-Programmable Gate Array (FPGA), and simplifies the coding phase of the algorithm and troubleshooting. The framework is demonstrated on a technology currently under development by NASA to provide automatic hazard detection and avoidance for spacecraft landing systems.*

Keywords: reconfigurable computing, floating-point, automatic landing, pipeline

1 Introduction

The implementation of computational algorithms is governed by the requirements of the application. In meeting these requirements the designer is also concerned with development time and maintainability of the implementation in the face of changes to the algorithm. Frequent algorithm design changes may occur due to parallel development of the algorithm and its realization.

For 3-dimensional, real-time, dynamic computational applications such as found in space systems, requirements for fast but low power processing often direct the search for implementation options to custom configurable solutions. Field-Programmable Gate Array (FPGA) implementations have been shown to have up to a factor of ten less power consumption compared to microprocessors [1].

This work investigates applying reconfigurable technologies in support of the Automated Landing and Hazard Avoidance Technology (ALHAT) project [2]. ALHAT is developing a system for the automatic detection and avoidance of landing hazards to spacecraft. The system is required to process large amounts of terrain data from a Light Detection and Ranging (LIDAR) sensor, within strict power constraints. Current design environments for configurable hardware development require substantial knowledge and expertise in hardware design, and these are not traditional skills of algorithm designers. Development times for custom hardware solutions are also significantly higher than for a software implementation. These two characteristics can cause projects to prefer software and microprocessor-based solutions despite the performance potential of configurable hardware [3].

The wide dynamic range associated with 3-dimensional transformations in applications such as ALHAT is best suited to floating-point arithmetic, a primary driver of the complexity of configurable hardware design. Operators in High Level Design Languages (HDLs) do not support floating-point arithmetic, as found in most software languages. Many efforts in the field of configurable computing research focus on developing compilers and frameworks to allow the design entry phase to have a similar complexity level as traditional software design [4]. If a framework with the capabilities required is not available, design alternatives include systolic array implementation or the development of a custom processor. In a systolic array, the operations and data path are designed specifically for the desired operation [5], [6]. This approach typically produces the highest performance when compared to other options, but the design is not flexible and changes to the high level algorithm require new iterations of a potentially time-consuming design effort. Often particular computations occur only a small fraction of the time, but the systolic array computational resources are wired for a specific computation. HDL tools will take advantage of if-then-else topology to recognize when in-line fixed point resources can be re-used, but not module based floating-point resources. In addition, real-time dynamic applications that interface with numerous sensor systems are characterized by sparse data arrivals from those systems. In this sparse data environment fully-pipelined designs are not used to their full capability.

Custom soft processors provide more flexibility to algorithm changes, and better accommodate re-use of resources, but have a more substantial initial design effort. These processors may use instructions tailored to the application in order to perform competitively with general purpose Application Specific Integrated Circuit (ASIC) processors [7], [8].

This work addresses the issues of development time for floating-point arithmetic algorithms in configurable hardware and ease of design modification with a framework that is simple in comparison to a software-to-hardware compilation system. The framework enables the definition of dynamically-configured floating-point pipelines in HDL that follow the flow of a software implementation more closely than a systolic array, and is suitable for straightforward translation from an executable software implementation that can be used for verification of the design. The pipelines use a data tag control scheme that avoids the complexity of centralized pipeline control logic. The tag approach allows dynamic configuration of pipeline components on a cycle by cycle basis, and supports traditional fully pipelined data path configurations, and several schemes for re-use of unallocated cycles in sparsely filled pipeline configurations. Re-use of unallocated cycles requires definite knowledge of where those cycles are. In one method of particular interest, resource constraints are addressed with a phase-coherent allocation approach that overloads pipeline stages with multiple operations, without the need for a scheduling algorithm or complicated control logic.

This paper is organized as follows. Section 2 describes research related to the approach described in this work, Section 3 provides details of the dynamically configurable phase-coherent pipeline design, the prototype test application and experimental platform are discussed in Section 4, test results are discussed in Section 5, and a description of plans for future work is provided in Section 6.

2 Related Work

Several examples exist in the literature of research frameworks for the implementation of floating-point computations on configurable hardware. In [9], the Trident compiler for floating-point algorithms written in C is described. The framework represents a substantial development effort with all the functionality of a traditional compiler: parsing, scheduling, and resource allocation. A custom synthesizer that produces Very High Level Design Language (VHDL) code and custom floating-point libraries are also included. The phase-coherent pipeline approach does not require the complex components found in the Trident system, and is suitable for straightforward translation from C code to VHDL defining the pipeline design.

The authors in [10] present a VHDL auto-coder to reduce the development time of floating-point pipelines. A computation is defined in a custom HDL-like pipeline description file, and the code for a single pipeline implementing the computation is produced. The approach requires a user to learn the author's custom HDL, and does not attempt to share resources. A C++ to VHDL generation framework is presented in [11], using an object-oriented approach to VHDL auto-coding of arithmetic operations. All computations are subclasses of a class "operator", and a method of the class "operator" produces VHDL to implement a pipeline for the computation. Again a user must learn the author's syntax to define computations and resource constraints are not addressed.

The resource sharing approach for phase-coherent pipelines has some similarities to those developed for ASIC synthesis algorithms. Hwang et al. [12] define the Data Introduction Interval (DII) as the period in clock cycles between new data arrivals at the input of a pipeline. A DII equal to one represents fully pipelined operations. As discussed in Section 3, a DII greater than one is required for phase-coherent resource sharing. Resource sharing approaches are described in [13] and more recently for FPGAs in [14], where analysis of a data flow graph of the computation and heuristics must be used. The phase-coherent allocation approach is governed by simple algebraic relationships and does not require complex analysis or heuristics. Phase-coherent allocation does not perform dynamic scheduling, and does not require any scoreboarding method [15] or hardware to check for structural or data hazards.

The association of tag values with data for control discussed in Section 3 is similar to the tagged-token dataflow computational model used in the Manchester Dataflow Machine [16]. The local pipeline control in our method is simpler, and does not require a matching unit, token queue, or overflow unit. Tagged-token dataflow concepts have also been used more recently in a configurable hardware implementation for parallel execution of multiple loop iterations [17].

A hand-coded systolic array and MATLAB®-based FPGA implementation of the coordinate conversion stage of processing LIDAR scan data for an automatic landing hazard detection system is compared in [18]. Fixed-point arithmetic is used. As previously discussed, a hand-coded systolic array is not easily adaptable to algorithm design changes. The MATLAB® solution is more easily developed and adapted, but is most suitable to the processing of streaming data and would not be an effective approach for other computations, such as the hazard detection stage of the ALHAT algorithm.

3 Implementation Framework

In this section, the key elements of the phase-coherent, dynamically configurable pipeline framework are described. An example design is used to illustrate how the concepts work together to provide the benefits of the approach.

3.1 Dynamically Configurable Pipeline

In the context of this work, a dynamically configurable pipeline is a pipelined operation with an interconnect configuration that is selectable during circuit operation. This selectable input/output configuration between floating-point operations, temporary storage, and input/output devices is achieved with multiplexers on each port of the pipeline. As stated in [19] the resource overhead of using multiplexers to share computing resources is balanced by the reduction of resources achieved. An additional concern, particularly for reconfigurable computing, is development time [20]. Dynamically configurable pipelines provide a flexible data path that is easier to modify than the fixed data path of a systolic array, reducing overall development and maintenance time.

3.2 Phase Tag Control

A dynamically configurable data path allows the inputs of an operation to be consuming operands and results to be produced for different computations potentially every clock cycle. Further, operations such as floating-point arithmetic typically require latencies greater than one to function at high enough clock frequencies to meet the performance requirements of applications. A control scheme is required to route operands and results between pipelines and other resources with the correct timing. In contrast to using a scheduling algorithm and global control approach, a distributed phase tag control method is used.

In the phase tag control method, a tag word is associated with a set of inputs. The tag is assigned to a buffer that has the same latency as the operation consuming the inputs. The buffer is called a phase keeper. The output of the phase keeper is tested to determine when the outputs of the operation are valid for the inputs associated with the tag. One tag can be used to control the input/output configuration of many floating-point units, providing all units have the same latency. For coding convenience, and to handle occasional units with different latency, phase keepers were built in to all our floating point units. If they are not used, the HDL tool flow removes them automatically. The functional units output two phase tag words. The ready tag (.r), usually with a latency of one clock cycle, is used to signal that a pipeline is ready to accept data. The completion tag (.p) indicates that an operation is finished. The tags are used to control the inputs and outputs of a pipeline. The content of the tag is used by an algorithm developer to indicate which step of the computation is associated with the operands or results. The phase tag control approach supports both dense and sparsely allocated pipelines. Examples of several design cases follow.

Figure 1 is a VHDL code sample of the dynamically configurable pipeline and phase tag control approach. The design is a fully pipelined implementation of the computation A+B+C+D=sum. In this and the remaining code examples, the signal AI(n) is an array of records that bundles data, phase tag, and function (addition or subtraction) for the input of adder unit n. AO(n) is an analogous signal for the output of the unit. The signals PI(n) and PO(n) are arrays of phase tag signals for the input and output, respectively, of phase keeper unit n.

In this example, three adders are connected such that the first unit (0) adds inputs A and B in parallel with the second unit (1) that adds inputs C and D. The outputs of the first two adders are wired to the inputs of the third adder (2), typical of a systolic array. Instead of an explicit state machine to control the output of the pipeline when the result of the computation is valid, the phase tag appearing at the output of the third adder, AO(2).p, is tested to determine when to store the result in registered signal sum1.

```
…
ad0: FADDP port map (clk, AI(0), AO(0));    --adder unit instantiations
ad1: FADDP port map (clk, AI(1), AO(1));
ad2: FADDP port map (clk, AI(2), AO(2));

process (clk) begin
if rising_edge(clk) then
…
 AI(0) <= (ZERO, ZERO, NOPH, ADD);
 AI(1) <= (ZERO, ZERO, NOPH, ADD);
 AI(2) <= (ZERO, ZERO, NOPH, ADD);

 if example1 then                           -- *** static full rate pipe with 3 adders ***
  if data_strobe then                       -- initiate first two adds on data strobe
   AI(0) <= (data_a, data_b, x"10", ADD);
   AI(1) <= (data_c, data_d, x"10", ADD);
  end if;
  AI(2) <= (AO(0).o, AO(1).o, AO(0).p, ADD); -- intermediate sums always wired to final adder
  if AO(2).p = x"10" then
   sum1 <= AO(2).o;                          -- consume result when tag appears at output
  end if;
 end if;

…

end if;
```

Figure 1. Example of dynamically configurable phase-tag control design of a fully pipelined implementation of the computation A+B+C+D=sum.

```
…
process (clk) begin
if rising_edge(clk) then
…
 AI(0) <= (ZERO, ZERO, NOPH, ADD);
 AI(1) <= (ZERO, ZERO, NOPH, ADD);
 AI(2) <= (ZERO, ZERO, NOPH, ADD);

 if example2 then                          -- *** sparse pipe with only one adder and max rate ***
  if data_strobe then
   AI(0) <= (data_a, data_b, x"10", ADD);   -- initiate first add on data strobe
  end if;
  if AO(0).r = x"10" then
   AI(0) <= (data_c, data_d, x"11", ADD);   -- initiate second add as soon as adder ready for input
  end if;
  if AO(0).p = x"10" then
   sum_ab <= AO(0).o;                       -- save sum a+b for one clock cycle
  end if;
  if AO(0).p = x"11" then
   AI(0) <= (sum_ab, AO(0).o, x"21", ADD);  -- initiate final sum
  end if;
  if AO(0).p = x"21" then
   sum2 <= AO(0).o;                         -- consume final sum
  end if;
 end if;
…
end if;
end process;
```

Figure 2. Example of dynamically configurable phase-tag control design of a sparsely pipelined implementation of the computation A+B+C+D=sum.

clock	trigger	logic	adder pipe stage 1	adder pipe stage 2
1	**DATA STROBE**	**A1 & B1 to adder, tag #10**		
2	**ready #10 tag**	**C1 & D1 to adder, tag #11**	**process A1+B1**	
3			**process C1+D1**	**process A1+B1**
4	**done #10 tag**	**save A1+B1**		**process C1+D1**
5	**done #11 tag**	**C+D, A+B to adder, tag #21**		
6	DATA STROBE	A2 & B2 to adder, tag #10	**process final sum**	
7	ready #10 tag	C2 & D2 to adder, tag #11	process A2+B2	**process final sum**
8	**done #21 tag**	**store or forward result**	process C2+D2	process A2+B2

Figure 3. Configuration logic events, event triggers, and processes allocated to adder unit pipeline stages for example of a sparsely pipelined implementation of the computation A+B+C+D=sum.

Figure 2 shows the same computation implemented with only a single adder unit, and assuming a DII that results in sparse data arrival strobes compared to the length of the computation. In this example, the ready tag is tested to determine when the adder unit can accept a second input. This approach allows the single adder to process the first two add operations as quickly as possible without a conflict. Note that intermediate results must be saved for the first addition because the result cannot be consumed immediately. Intermediate results can be stored up to the register resource limit of a particular part.

Using a two stage floating-point adder module, Figure 3 shows the processes allocated to each adder stage, the configuration logic for each clock cycle, and the event which triggers the logic. Data point 1 is shown in bold. A second data point is in gray. The minimum DII is 5 clock cycles, at which time the previous point is finished with the adder input. The pipe is still processing the previous point, but the correct actions will be triggered by tags as they emerge from the pipe, as each carries configuration "knowledge" of what should be done with its associated data.

The phase tag scheme makes call and return logic, similar to subroutine calls in software programs, possible in a configurable hardware design. The re-used adder has effectively become a call and return module. The tag associated with data input indicates where the control logic should resume when the adder is finished.

In addition to tags, an application can use mode variables to control the configuration of the pipeline units. Mode variables can allow use of fewer distinct tags. But the pipe has to be completely empty before changing a mode variable. The examples do not have mode variables, but they were used in our prototype application.

Note that the tags are strobes. Initialization code gives them the default value "NOPH" (no phase) unless they are specifically assigned. If a group of functional units will be re-used in a different module, then this initialization should be contingent upon a mode variable. The functional units are set up with tri-state input signals so that they can be shared between modules. All that is required is to pass the input and output signals for the functional units to the active module, and enable initialization defaults only in the active module. If the DII is not regular, then buffers should be used to ensure that the minimum DII is met.

The example in Figure 2 uses intra-pipeline stage sharing to process more than one computation stage on a single adder unit. Effective resource sharing using this method can be a complicated problem [14]. In the next section, the concept of phase-coherent resource allocation is presented as a straightforward means of accomplishing intra-pipeline sharing. The allocation is constrained by simple relationships based on the DII and minimum pipeline unit latency. These criteria are not restrictive in real-time data processing systems that interface with various sensor subsystems with different latencies. In such systems, fully pipelined computations are not generally required.

3.3 Phase-Coherent Resource Allocation

Phase-coherent pipeline allocation is a simple means to allow pipeline stage sharing that enables different computations to be allocated to the same functional units. The method requires that results associated with a particular computational sequence all emerge at a constant phase, that is, at a constant multiple of a minimum unit latency L. If a unit does not naturally have this latency, it must be padded with enough empty pipeline stages. The multiplex stage is included in the latency value. The DII should be equal to or greater than L[1]. The pipe can be said to have L independent phases. For maximum re-use, successive data inputs are allocated to different phases, until all phases are used, and only then are conflicts possible. Under these conditions, a simple algebraic relationship can be used to compute the period of time that units can be re-used as follows.

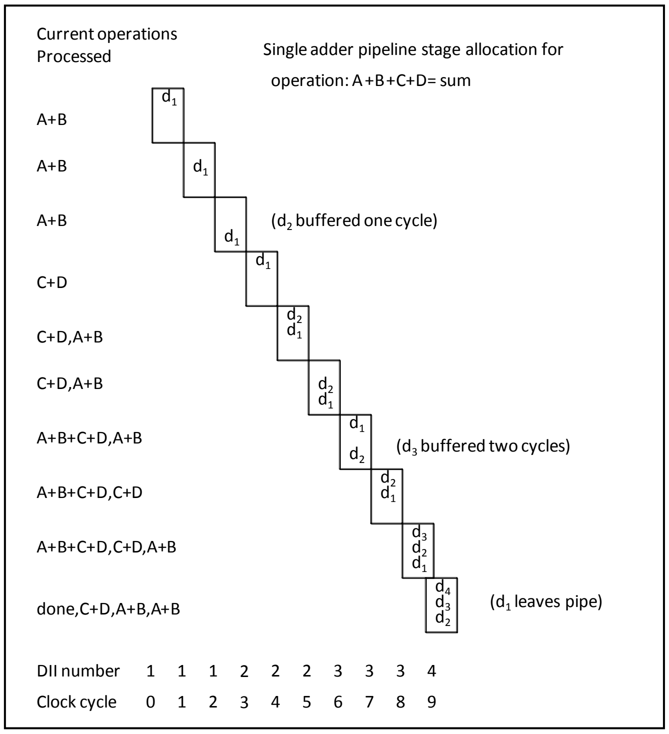
Given a dynamically configurable pipelined functional unit with a latency of L, each pipeline stage, $S_p(n)$, can process a datum of an independent computation. The reuse interval, I_R, is defined as the number of clock cycles in which units can be reused freely. This interval is computed as shown in (1).

$$I_R = DII \cdot L \tag{1}$$

For maximum re-use, the interval can be applied separately to each functional unit. The reuse interval may be applied manually if hand-coding or incorporated into a translator. Figure 4 shows a diagram of phase-coherent pipeline allocation for the computation of the prior code examples. In Figure 4, the DII=3, L=3, and I_R=9. The d_n variables represent a set of input operands for the four add operations at DII=n. As shown, the phase offsets for allocation are implemented with a one or two cycle store of the incoming data value at the input of the unit, as shown in clock cycles 3, 6, and 7. This is required because the input stage of the unit is busy processing prior input data at these cycles. A latency L phase keeper buffer tracks the allocation of available

[1] In real systems, the DII is never regular due to crossing clock domains between the sensor systems and application. First-In First-Out (FIFO) buffering can be used to force an average DII allowing the use of the phase-coherent allocation.

computation phases and is used to control the assignment of inputs to functional units. Also shown in Figure 4 is that by cycle 8 when the third input data set is consumed, each pipeline stage of the unit is processing data. Intermediate results or temporary state variables, for example the intermediate addition result A+B, do not benefit from the phasing scheme and must either be used within the DII or copied every DII clocks. Alternately they could be retained in no-op pipe units.



Current operations Processed
Single adder pipeline stage allocation for operation: A+B+C+D= sum
A+B
A+B
A+B
C+D
C+D,A+B
C+D,A+B
A+B+C+D,A+B
A+B+C+D,C+D
A+B+C+D,C+D,A+B
done,C+D,A+B,A+B
d_1
d_1
d_1
d_1
d_2 d_1
d_2 d_1
d_1 d_2
d_2 d_1
d_3 d_2 d_1
d_4 d_3 d_2
(d_2 buffered one cycle)
(d_3 buffered two cycles)
(d_1 leaves pipe)
DII number 1 1 1 2 2 2 3 3 3 4
Clock cycle 0 1 2 3 4 5 6 7 8 9

Figure 4. Phase-coherent allocation of a single adder unit performing the computation A+B+C+D=sum on three input data sets, d. The parameters of the allocation are DII=3, L=3, and I_R=9, all in design clock cycles.

The VHDL that implements the example of Figure 4 is shown in Figure 5. As shown, the code implementing the phase-coherent allocation method is straightforward and suitable for generation by an auto-coder. The phase tags control each stage of the computation as well as the cycle the adder unit is free to accept new data. The dynamically configurable inputs and outputs allow the same unit to process each computation stage within the reuse interval.

4 Prototype Application

Details of the algorithms supporting ALHAT for landing hazard detection and avoidance are provided in [21]. The general approach is to produce a regular grid of surface elevation data in the coordinate frame of the landing site from the LIDAR range samples. This elevation map is then analyzed for surface slope and roughness and compared to thresholds for these parameters to identify hazards. The processing stages for LIDAR scan data are coordinate conversion, re-gridding, and hazard detection. The first two stages are currently demonstrated in the prototype design. The computations implemented are summarized in this section.

```
pk0 : PKEEP  port map (clk, PI(0), PO(0));     --keeper unit instantiations
pk1 : PKEEP  port map (clk, PI(1), PO(1));
...
process (clk) begin
if rising_edge(clk) then
PI(0) <= RDYPH;                                --constant indicating units ready for data
waiting <= FALSE;
...
AI(0) <= (ZERO, ZERO, NOPH, ADD);
AI(1) <= (ZERO, ZERO, NOPH, ADD);
AI(2) <= (ZERO, ZERO, NOPH, ADD);

if example3 then                              -- *** phase coherent pipe with DII*L re-use interval ***
 PI(0) <= PO(0);                              -- default to a re-circulating phase tag
 if data_strobe or waiting then
  if PO(0) = RDYPH then                       -- wait for an available pipe phase
   PI(0) <= x"10";                            -- phase tag indicates 1st operation this re-use interval
   AI(0) <= (data_a, data_b, x"10", ADD);     -- send a and b to adder inputs
  else
   waiting <= TRUE;                           -- if pipe not available, come back and try again
  end if;
 end if;
 if PO(0) = x"10"  then                       -- when a+b completes, save and start next
  PI(0) <= x"20";
  AI(0) <= (data_c, data_d, x"20", ADD);
  sum_ab <= AO(0).o;                          -- this state variable valid  for L clocks or DII/L stages
 end if;
 if PO(0) = x"20"  then                       -- when c+d completes, initiate final sum
  PI(0) <= x"30";
  AI(0) <= (sum_ab, AO(0).o, x"30", ADD);
 end if;
 if PO(0) = x"30"  then                       -- when final sum completes ...
  PI(0) <= RDYPH;                             -- vacate this phase
  sum3 <= AO(0).o;                            -- consume result
 end if;
end if; -- end example3

...
end if;
end process;
```

Figure 5. Example of dynamically configurable phase-tag control design of a phase-coherent pipeline implementation of the computation A+B+C+D=sum.

4.1 Coordinate Conversion and Re-gridding

As described in [21], the coordinate conversion stage converts each LIDAR range sample from scanner angle and range coordinates to Cartesian coordinates. The computation is shown in (2), where $pr_{x,}$ $pr_{y,}$ $pr_{,z}$ are components of the converted point, $t_{x,}$ $t_{y,}$ $t_{,z}$ are the components of the sensor position vector, $p_{x,}$ $p_{y,}$ $p_{,z}$ are the components of the range sample, and $q_{1,}$ $q_{2,}$ $q_{3,}$ q_4 are components of a quaternion vector for the coordinate rotation.

$$pr_{x,y,z} = 2\begin{bmatrix} (q_3q_3 + q_0q_0 - 0.5)p_x + \\ (q_0q_1 + q_3q_2)p_y + \\ (q_0q_2 + q_3q_1)p_z \end{bmatrix} + t_{x,y,z} \tag{2}$$

In the re-gridding stage of the computation, converted range samples are projected into a grid cell of the elevation map, and a bilinear interpolation scheme is used to update the elevation of each vertex of the cell containing the projected point. The elevation of the projected point weighted by the distance from the point to the vertex is added to the current weighted elevation for that vertex. Updates to the weighted elevations and the weights for each vertex of the grid cell containing a projected point are made using the computation shown in (3). In (3) *r* and *c* are the row and column numbers of the elevation map grid cell vertices, respectively.

$$
\begin{aligned}
&u = r - \lfloor r \rfloor; \\
&v = c - \lfloor c \rfloor; \\
&W(r,c) += (1-u)(1-v); \\
&E(r,c) += (1-u)(1-v)pr_z; \\
&W(r+1,c) += u(1-v); \\
&E(r+1,c) += u(1-v)pr_z; \\
&W(r,c+1) += (1-u)v; \\
&E(r,c+1) += ((1-u)v)pr_z; \\
&W(r+1,c+1) += uv; \\
&E(r+1,c+1) += (uv)pr_z;
\end{aligned}
\tag{3}
$$

4.2 Experimental Setup

The prototype design is tested on a Xilinx® Virtex™-5 FX130T FPGA hosted on an Alpha Data ADM-XRC-5TZ mezzanine card. The Virtex™-5 family has a radiation tolerant version, providing a path to space flight certification of the design. The ADM-XRC-5TZ board has 48 megabytes of SRAM across six banks, used for storing the elevation and weight data for the elevation map. The prototype design uses two SRAM interfaces that bundle two SRAM banks each. The interfaces are designed to use one bank for even addresses and one for odd. This approach makes it possible to run the SRAM interface at twice the design rate to reduce memory latency. Currently all interfaces operate at the same clock frequency.

A Gigabit Ethernet interface is included for command and data input and output. The prototype is designed to run at the Ethernet clock speed of 125 MHz. To avoid buffering of the input data, the prototype is designed with a DII of 12 clock cycles. The 12 cycles is derived from each LIDAR sample consisting of three single-precision floating-point components of four bytes each.

With this prototype platform, the coordinate conversion and re-gridding computations were implemented within a few days using the phase-coherent pipeline approach. A full-rate elevation map computation is verified (input LIDAR samples are converted and re-gridded within 12 clock cycles) on this prototype. These results show that the LIDAR data can be processed in real time, or faster than real time.

5 Results

A comparison of resources used between various implementations of the example computation presented in Section 3 is shown in Table I. The resource values are reported from the Xilinx® synthesis tools. The static implementation is a direct wiring of the adders and data path to realize the computation. The dynamically configurable/phase tag control implementations are designed as presented in Figures 1, 2, and 4. The phase-coherent implementations are designs applying the phase-coherent method to each case represented by Figures 1, 2, and 4. The floating-point units are implemented using the Xilinx® CORE Generator™ tool. DSP slice resources are used in the multiplier units but not shown in Table I. Comparing the Lookup Table (LUT) resources between the static and phase-coherent implementations shows the phase-coherent pipeline method yields an 85% reduction in resources. This means a given FPGA can hold the equivalent of about seven times as many source lines of floating point application equivalent code using the phase-coherent method, as using traditional data path methods. If a particular design does not approach resource limits, phase-coherent reuse reduces design size resulting in faster place and route.

TABLE I. RESOURCE COMPARISON BETWEEN DIFFERENT IMPLEMENTATIONS OF COMPUTATION SUM=A+B+C+D.

Resource	**Implementation- 3 Instantiations of sum=a+b+c+d**		
	Static	***Dynamically Configurable/Phase Tag Control***	***Phase-Coherent***
Slice Registers	1198	753	342
LUTs	4768	1755	705
Slices	1653	694	309
LUT/FF Pairs	4706	2010	834
Min. Period	6.6ns	6.6ns	6.6ns

6 Conclusions

The method described in this work achieves substantial improvements in the ease of both development and resource reuse for pipelined computations on configurable hardware. Using this HDL method, declarations and wiring are simplified, and operand/result assignments are easily mixed with other synchronous code. The HDL reads like and corresponds closely to a software specified algorithm. This allowed rapid design of the prototype, and should allow fast response to algorithm changes. The HDL is suitable for straightforward translation from an executable software definition that can be used for algorithm verification. This reduces the gap between the expertise required to design configurable implementations and that of typical algorithm designers.

The difficulty of resource reuse is reduced with a data tag control scheme and phase-coherent allocation method that replace the need for complex global scheduling, heuristics or cycle dependent logic. Sparse data arrival in real-time is efficiently allocated to pipeline stages, reducing design size and place and route times.

Further examination of the utility of the approach is planned with the full implementation of the initial hazard detection algorithms in the ALHAT project [21]. Unlike coordinate conversion and re-gridding, hazard detection is computation-bound with high potential parallelism, exercising the generality of the approach. The algorithm has been updated several times since this initial version [3] providing a relevant case to test the difficulty of design modification using the phase-coherent framework. The approach will also be considered for application to dynamically configurable ASICs.

7 References

[1] G. Govindu, L. Zhuo, S. Choi, P. Gundala, V. Prasanna, “Area, and Power Performance Analysis of a Floating-point based Application on FPGAs”, In Proceedings of the Seventh Annual Workshop on High Performance Embedded Computing (HPEC 2003), http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.157.9977.

[2] C. Epp, E. Robertson, T. Brady, "Autonomous Landing and Hazard Avoidance Technology (ALHAT)," Aerospace Conference, 2008 IEEE,pp.1-7,March. 2008,doi: 10.1109/AERO.2008.4526297.

[3] C. Villalpando, A. Johnson, R. Some, J. Oberlin, S. Goldberg, "Investigation of the Tilera processor for real time hazard detection and avoidance on the Altair Lunar Lander," Aerospace Conference, 2010 IEEE, pp.1-9, March, 2010, doi: 10.1109/AERO.2010.5447023.

[4] M. Cardoso, P. Diniz, and M. Weinhardt, “Compiling for reconfigurable computing: A survey,” ACM Comput. Surv. 42, 4, Article 13, June 2010, pp. 1-65,doi:10.1145/1749603.1749604.

[5] K. Sano, T. Iizuka, S. Yamamoto, "Systolic architecture for computational fluid dynamics on FPGAs," Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on, pp.107-116, April 2007,doi: 10.1109/FCCM.2007.20.

[6] S. Qasim, S. Abbasi, B. Almashary, "A proposed FPGA-based parallel architecture for matrix multiplication," Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on, pp.1763-1766, Nov. 2008, doi: 10.1109/APCCAS.2008.4746382.

[7] D. Goodwin and D. Petkov, “Automatic generation of application specific processors,” Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '03). ACM, New York, NY, USA, pp. 137-147, doi:10.1145/951710.951730.

[8] J. Yu, C. Eagleston, C. Han-Yu Chou, M. Perreault, and G. Lemieux, “Vector processing as a soft processor accelerator,” ACM Trans. Reconfigurable Technol. Syst. 2, 2, Article 12, June 2009, pp. 1-34, doi:10.1145/1534916.1534922.

[9] J. Tripp, K. Peterson, C. Ahrens, D. Poznanovic, M. Gokhale, "Trident: an FPGA compiler framework for floating-point algorithms," Field Programmable Logic and Applications, 2005. International Conference on, pp. 317-322, Aug. 2005, doi: 10.1109/FPL.2005.1515741.

[10] G. Lienhart, A. Kugel, R. Manner, "Rapid development of high performance floating-point pipelines for scientific simulation," Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International, pp.8 April, 2006, doi: 10.1109/IPDPS.2006.1639439.

[11] F. de Dinechin, C. Klein, B. Pasca, "Generating high-performance custom floating-point pipelines," Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on , pp.59-64, Sept. 2009, doi: 10.1109/FPL.2009.5272553.

[12] K. S. Hwang, A. E. Casavant, C. Chang, M. d'Abreu, "Scheduling and hardware sharing in pipelined data paths," Computer-Aided Design, 1989. ICCAD-89. Digest of Technical Papers., 1989 IEEE International Conference on, pp.24-27, Nov. 1989, doi: 10.1109/ICCAD.1989.76897.

[13] S. Wakabayashi, N. Ohashi, J. Miyao, N. Yoshida, "A synthesis algorithm for pipelined data paths with conditional module sharing," Circuits and Systems, 1992. ISCAS '92. Proceedings., 1992 IEEE International Symposium on, vol.2, pp.677-680, May 1992, doi: 10.1109/ISCAS.1992.230161.

[14] S. Mondal, S. Memik, "Resource sharing in pipelined CDFG synthesis," Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific, vol.2, pp. 795- 798, Jan. 2005, doi: 10.1109/ASPDAC.2005.1466464.

[15] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, 3rd ed. San Francisco: Morgan Kaufmann Publishers, 2003.

[16] J. R Gurd, C. C Kirkham, and I. Watson, “The Manchester Prototype Dataflow Computer”, Commun. ACM 28, 1, January 1985, pp.34-52, DOI=10.1145/2465.2468.

[17] H. Styles, D.B. Thomas, W. Luk, "Pipelining Designs With Loop-Carried Dependencies," Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on, pp. 255- 262, 6-8 Dec.2004doi: 10.1109/FPT.2004.1393276.

[18] K. Shih, et al., “Fast real-time LIDAR processing on FPGAs,” http://www.informatik.uni-trier.de/~ley/db/conf/ersa/ersa2008.html (accessed 6/3/2011).

[19] W. Sun, M. Wirthlin, S. Neuendorffer, "FPGA pipeline synthesis design exploration using module selection and resource sharing," Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on,vol.26,no.2,pp.254-265,Feb.2007,doi: 10.1109/TCAD.2006.887923.

[20] J. Villarreal, A. Park, W. Najjar, R. Halstead, "Designing modular hardware accelerators in C with ROCCC 2.0," Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on, pp.127-134, May, 2010, doi: 10.1109/FCCM.2010.28.

[21] A. Johnson, A. Klumpp, J. Collier, A. Wolf, “Lidar-based hazard avoidance for safe landing on Mars,” http://trs-new.jpl.nasa.gov/dspace/ (accessed 6/3/2011).