A Domain Specific Language for Reconfigurable Path-based Monte Carlo Simulations[*]

David B. Thomas and Wayne Luk
Imperial College London
United Kingdom
{dt10,wl}@doc.ic.ac.uk

Abstract

FPGAs have been successfully used to accelerate many computationally bound applications, such as high-performance Monte-Carlo simulations, but the amount of programmer effort required in development, testing, and tuning is also very high, requiring a new custom design for each application. This paper presents Contessa, a pure-functional continuation-based language for describing path-based Monte-Carlo simulations, and a completely automated method for turning platform-independent Contessa programs into high-performance hardware implementations. Our approach exploits the large degree of thread-based parallelism available in Monte-Carlo simulations, allowing data-dependent control-flow and loop-carried dependencies to be expressed, while retaining high-performance. The Contessa toolchain is evaluated using five different simulation kernels, in comparison to both software and manually described hardware. When compared to an existing FPGA implementation, Contessa requires a quarter of the Handel-C source-code length, and doubles the clock rate to over 300MHz while requiring a similar number of resources, and also provides a 35 times speedup over a C++ implementation using an Opteron 2.2GHz.

1 Introduction

FPGAs have been demonstrated to be an effective platform for computationally intensive applications, such as Monte-Carlo simulations for financial applications. However, although FPGA accelerators can provide a significant speedup over software, they also take much longer to develop: a hardware engineer must analyse each application, develop an efficient hardware architecture, then test the resulting combination of hardware and management software. It is also unlikely that the solution is directly portable to alternate hardware platforms or newer FPGA architectures, so it is difficult to automatically scale across heterogeneous clusters, or achieve forwards compatibility.

This paper presents Contessa, a Domain Specific Language (DSL) for describing simulations architectures, which attempts to address these problems. Applications are described using a platform-independent language, using a pure-functional continuation-passing style, which can then be automatically turned into efficient pipelined hardware for multiple platforms and architectures. The language and implementation strategy exploit the huge amount of thread-based parallelism available in a Monte-Carlo simulation, allowing simulations to contain complex data-dependent branching and iteration without degrading performance.

Our main contributions are:

- The Contessa language, a pure-functional, continuation based language for describing path-based simulations. Each Contessa program describes a complete simulation application, that can be automatically compiled, loaded, and executed.
- An automatic mapping technique for converting Contessa descriptions into high-performance pipelined designs, requiring no manual annotations or platform-specific hints.
- A performance evaluation of the language and mapping strategy, using five simulations written in Contessa, Handel-C, and C++.

In Chapter 2 the Contessa language is presented, by first introducing the core language structure, then the features designed specifically for simulations. Chapter 4 then explains the automatic process for converting Contessa programs into pipelined hardware, followed by an evaluation of the system in Chapter 5.

[*]The support of UK Engineering and Physical Sciences Research Council (Grant reference EP/D062322/1 and EP/C549481/1), Celoxica and Xilinx is gratefully acknowledged.

2 Overview of the Contessa Language

The Contessa language is a pure-functional language, having only a few basic types of declarations and statements. It is a pure-functional language (i.e. variables cannot be re-assigned), and like many pure-functional languages it uses tail-recursion to express iteration without side-effects. However, unlike most functional languages, Contessa does not allow recursive functions to return values: a function can either return a value, or it can use recursion, but it cannot do both. While somewhat unconventional, this approach allows the automatic mapping strategy described in Section 4, providing a high-performance reconfigurable architecture from a implementation independent description;

There are actually two semantically equivalent syntaxes of Contessa: a minimal lisp-like syntax and prefix expressions, making the functional nature of the language apparent; and a C-based notation, using infix expressions and curly brackets. In this paper we use the C syntax, and only provide an informal overview of the language features, focusing on an overview of the language and route to hardware. A more formal description of the language and its semantics will appear in a future paper.

Contessa programs use a pure-functional proper subset of C, with the main consequence being that there are no assignment statements and loop statements are not allowed. Data type support is also limited to statically-sized types, such as scalars, structs, and fixed-length arrays; there are no pointers, or dynamic memory allocation. However, this subset still includes features such as procedures, conditional statements, and expressions: using a set of simple class libraries and some pre-processor macros, it is actually possible to directly compile and execute a Contessa program using a standard C compiler.

Procedures within a Contessa program are divided into two types: functions, which are procedures that return values; and blocks, which are procedures that do not return values (i.e. they have a void return type). A function can contain conditional statements and function calls, but it cannot contain any recursive calls either directly, or indirectly through another function. Functions are essentially a macro facility, and these conditions are to ensure that a function can be statically expanded at compile-time.

Blocks are quite different to functions, as they cannot return values, but instead they can contain an arbitrary amount of recursion, both directly and indirectly. However, this is recursion without returning a value: when a block makes a call to another block, that call will never return. This means that once a call is made, there is no reason to retain the dynamic environment (such as local variables) of the calling function, so no stack is needed.

A consequence of throwing away the environment is that the parameters passed from one block to another must contain the entire thread state: the environment of the calling block is discarded, so any information needed later in the thread's lifetime must be passed as a parameter to the called block. This is known as continuation-passing style, where the combination of the target block and block parameters represents the thread continuation, and completely captures all information about the thread [1].

```
parameter(float,VOLATILE_ENTER);
parameter(int, MAX_D); // Remaining parameters elided.

accumulator(float,price);   // Price at end of simulations.

 // This is a function, and is expanded in place.
float lognrnd(float mu, float sigma)
{ return exp(normrnd()*sigma+mu); }

// This block is the arity-0 entry point for all threads.
void init()
{ stable(0, S_INIT); } // Start threads in stable block.

// Stable market: step price forward for each day in simulation.
void stable(int d, float s)
{
 if(d==MAX_D){
   price += s; // Accumulate final price of simulation.
   return;      // Exit thread with nullary return.
 }
 if(unifrnd()>VOLATILE_ENTER){
   volatile(d+1,0,VOL_INIT,s);  // Simulate volatile day.
 }else{
   float ns=s*lognrnd(STABLE_MU, STABLE_SIGMA);
   stable(d+1, ns);    // Simulate stable day in one step
}}

// Volatile market: step price in small increments through day.
void volatile(int dinc, float t, float v, float s)
{
if(t>MAX_T){        // End of day, so ...
 stable(dinc, s); // ... return to stable phase.
}else{
 float nt=t+exprnd();       // Advance intra-daily time.
 float nv=sqrt(v+unifrnd());        // New volatility.
 float ns=s*lognrnd(VOL_MU, VOL_SIGMA*nv);
 volatile(dinc, nt, nv, ns);   // Next volatile step.
}}
```

Figure 1. Example of a Contessa simulation, using three blocks, and five continuation points (shown underlined).

3 Contessa Program Example

Figure 1 gives an example of a simple simulation written in Contessa. The aim is to simulate some hypothetical asset's price path, taking into account two types of market conditions: stable, where a whole day's trading can be sim-

[1]This differs slightly from the Functional Programming (FP) definition, although the central idea of forward computation without function returns is the same. In FP a continuation passed to function X usually contains both data and a reference to another function Y, and when X completes execution it will use function Y to continue processing. We intend to add the ability to pass dynamic continuations to the next iteration of Contessa.

ulated in a single simple step; and volatile, where multiple smaller steps are required, using a more complex algorithm for each step.

The code starts with a number of simulation parameters, which are read-only variables visible throughout the program. At run-time these parameters will be bound to values estimated from the latest market data, so these parameters are the high-level inputs to the program. The run-time system guarantees that parameters will not change within the lifetime of each thread, so from within the Contessa program they are constant.

Below the parameters is an accumulator, "price". Accumulators act as outputs from the program, collecting statistical properties derived from the aggregate behaviour of many simulations. Accumulators are globally visible, but unlike parameters they are write-only.

Incorporating write-only constructs into a supposedly pure-functional language may appear contradictory, but there is a sound theoretical justification: all the statistical estimators supported in Contessa are associative and commutative. A vector of numbers will always have the same arithmetic mean, no matter how the elements in the vector are permuted, and similarly, an accumulator in Contessa will always produce the same statistical estimate, independent of thread execution order. Programs are also still side-effect free, as threads cannot read from accumulators, so the interaction of any other thread with an accumulator is undetectable.

Below the accumulator is a function *lognrnd*, which establishes a helper for generating the log-normal distribution, using the *normrnd* and *exp* functions. At compile-time it will be expanded within any expression where it is used.

Finally come the three blocks in the program, *init*, *stable*, and *volatile*. *init* simply provides a nullary thread-entry block, and it immediately transfers control to *stable*. Note that when *init* calls *stable*, the entire state of the thread is captured by the continuation (*stable*,0,S_INIT).

The *stable* block is the main loop in the simulation, with each iteration advancing the simulation by one day, until the loop exit test on line 17 indicates that the final day (specified as a program input using the parameter *MAX_D*) is reached. If the loop has finished the final price of the simulation is added to the accumulator *price* on line 18, which acts as the output from the thread. The thread is then ended using a return statement with no parameters.

If the loop exit has not been met, the code then continues to the conditional at line 21, which randomly decides whether the current day is stable or volatile. If the condition is true, control is transferred to the *volatile* block, which performs a detailed simulation of the day. Otherwise the entire day's price movement is simulated in line 25, using the function *lognrnd* which will be expanded inline, followed by a recursive call to *stable*, advancing to the next day.

The *volatile* block implements another loop, which is nested within the *stable* block; this nesting is most apparent in the parameter *dinc*, which is never modified or used by *volatile*, but is passed forward unchanged until the loop exits at line 32. The body of the *volatile* loop models asset price changes at exponentially distributed time offsets during the day, so the number of iterations is not known in advance, and varies for each thread.

Together the *stable* and *volatile* blocks advance a thread through an entire simulation path, using the simulation input parameters *S_INIT*, *MAX_D*, etc., and recording the result of the simulation thread in *price*. A Contessa program requires many such threads to be executed, all storing their results in *price*, until a sufficiently accurate aggregate mean asset price has been determined. This process is handled by a run-time environment, which is responsible for binding the simulation parameters to program inputs, initiating and managing threads, and monitoring the gradual convergence of accumulators.

There are currently two implementation paths for Contessa programs, using an FPGA (described in the next section), or using software. We mentioned earlier that it was possible to directly compile programs using C, but this is not an efficient solution, due to the deep levels of recursion involved. In simple cases the C compiler will convert this to tail-recursion, but in more complex mutually recursive cases the C program will use the stack, which is likely to overflow. Instead, Contessa programs are first translated to a constant-space C version, which can then be compiled with a standard C compiler, and linked with the runtime environment.

4 FPGA Implementation Strategy

The aim of Contessa is to allow programs to be expressed in a way that is high-level, efficient, and platform independent. In particular, we are interested in targeting highly concurrent platforms, such as modern FPGAs, that have significant computational power, but are difficult to program using conventional high-level languages such as C. In this section we describe the compilation strategy from a Contessa program to a high performance FPGA-based implementation.

It is well-known that FPGAs can offer a huge amount of computational power, even when using floating-point, but the nature of the architecture imposes a number of important constraints. First, one must aggressively pipeline all operations if a high clock-rate is to be achieved. The latency of Xilinx single-precision floating-point cores varies from 9 cycles for a multiply, up to 27 cycles for division. This latency often causes problems when scheduling threads of computation, particularly when there are loop-carried dependencies.

The second problem is that of accessing global shared state: although FPGAs are rich in fast local storage in the form of block RAMs, global state must usually be stored in slower off-chip RAMs. Mapping logical data-structures to available physical RAMs in a way that allows efficient access is a difficult task, and even with an optimal mapping there may still be significant contention when threads attempt to access shared state.

Contessa was designed to have an implementation strategy that eliminates these two problems. First, the problem domain of Monte-Carlo simulations allows a huge amount of thread-level concurrency: by using one logical thread per simulation we immediately have anywhere from a thousand to a billion threads to schedule for each program execution. These threads are strictly independent, as there are absolutely no side-effects, so threads (and the operations comprising each thread) can be scheduled in any possible order. This allows us to adopt a C-Slow strategy when implementing the thread processors: in the presence of highly pipelined functional units, we simply start as many threads as are needed to fill every stage of the functional unit.

The second problem, that of access to shared mutable state, is solved by removing all read-write shared state. The only mutable state accessible to threads is the current block environment, which is both conceptually, and (as is detailed below) physically, local to the functional units transforming the state. The only kinds of shared global state are parameters, which are read-only and can hence be safely cached, and accumulators, which are write-only and can be viewed by each thread as their own personal black-hole.

```
void step(int i, float c)
{
   float nc=c+normrnd(MU,1);
   if(i==T)
      aC += nc;       // accumulation
   else
      step(i+1, nc); // block transfer
}
```

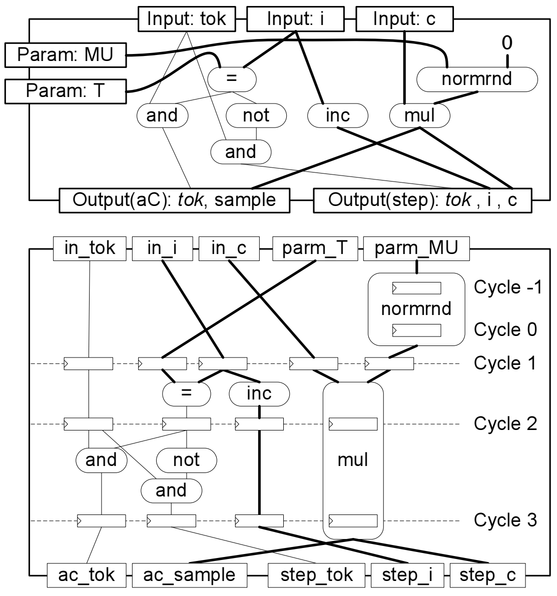


Figure 2. Conversion process from block specification to abstract data-flow graph, then to scheduled pipeline.

4.1 Blocks

Blocks can be viewed as pure (albeit non-deterministic) functions, with a domain defined by the block inputs, and the program's global parameters. The codomain of the function is more complicated, as it consists of zero or one continuations (comprising the name of the target block, and the parameters to that block), plus zero or more accumulation operations (comprising the name of the target accumulator and the value to be accumulated). The function's mapping is defined both by the data-flow derived from expressions, and the control-flow implied by statements.

The first step in the conversion process is to convert the block's statements and expressions into an abstract Directed Acyclic Graph (DAG). This is shown in Figure 2, where the source code for *step* is converted to the DAG in the middle of the figure. The heavy lines between nodes show data-flow within the block, derived from expressions, while the thin lines in the DAG show the control-flow, extracted from the statements.

Notice that all control-flow depends on an input called "tok". This is an implicit block parameter, not accessible in the original source code, and automatically inserted while building the DAG. The token identifies whether a set of block input parameters represents a valid thread state, and is used to identify "bubble" states. The "if" statement in the source code is implemented by creating a chain of boolean logic that depends on both the condition and the value of "tok": if the input token is invalid then neither output group will be activated, otherwise exactly one of the groups will be activated.

A block can have multiple output groups, each of which is associated with accumulate and block transfer statements in the block. Each accumulate statement results in a new output group, even if the same accumulator is named as the target, as a block may send two or more samples to the same accumulator. However, the number of block transfer groups is dependent only on the number of distinct blocks called. Because exactly one or zero transfer statements may execute in a block, any block calls referring to the same target block will be multiplexed to a shared output group.

The DAG represents a hardware independent description of the block, but to use it in hardware the abstract nodes must be bound to concrete components, and scheduled within the pipeline. In the current compiler the binder simply chooses the highest performance component available for each node, using a database of internal and external IP cores. The scheduler then uses the latency information in the component database to produce a pipeline schedule.

The scheduler uses a combination of ASAP (As Soon As Possible) followed by ALAP (As Late As Possible) scheduling. This two-stage strategy is needed because random number generators have no inputs, and so have no well defined scheduled position under ASAP. During the first pass all nodes on the paths between inputs and outputs are scheduled using ASAP, establishing the critical scheduling paths through the DAG, then in the second pass all remaining nodes are scheduled using ALAP. This often has the effect of scheduling random number generators at negative times (as seen in the bottom of Figure 2), and can significantly reduce the buffering needed when compared with a naive strategy of scheduling such blocks at time zero.

The scheduled and bound DAG is then rendered into an HDL description (currently VHDL), providing a concrete implementation of the block's entire transition function. Each concrete block is fully pipelined, and can accept one new thread on each clock cycle.

4.2 Parameters

Parameters are simply values which remain constant for the lifetime of a set of threads, but which may be changed between threads, so parameters naturally map into registers. New constants are loaded over a data-bus shared between all parameter registers, with the clock-enable controlled by a decoder to select the specific parameter. This data-bus is mapped into the main bus described in Section 4.4, allowing the controlling software to update parameters.

In principle the parameter registers could be a performance problem, as a program could use the same parameter in many different expressions, requiring the output of the register to be propagated to many locations in the FPGA. The current approach is to simply use a slave register, driven by the master register, allowing the synthesis or place-and-route tools to replicate the register if necessary. A second approach (not yet tried), would be to simply declare any signals using the parameter register as false-paths, as many cycles will elapse between the parameter register being changed and any blocks using the value of the register.

4.3 Accumulators

Accumulators are treated as black-box components that read samples from a channel, updating their internal state in some unspecified way in response to each sample. When an accumulator instance is required, the compiler searches it's database of accumulator IP cores for an accumulator using the correct data-type.

In the current implementation one accumulator component is instantiated for each accumulate statement that occurs in a block. This means that for each named accumulator in a program, there might exist multiple accumulator instances in the actual design. The ability to freely replicate accumulators is a consequence of the associativity and commutativity of accumulation, and allows all accumulate statements in the program to execute in every cycle without losing samples.

The accumulator state is exposed through a bus connected to each accumulator instance, allowing the current state to be retrieved (even while simulations are running), and to be reset. During compilation any replication of accumulator instances is recorded in the design's meta-data, allowing the run-time environment to present a unified view of each accumulator's aggregate state.

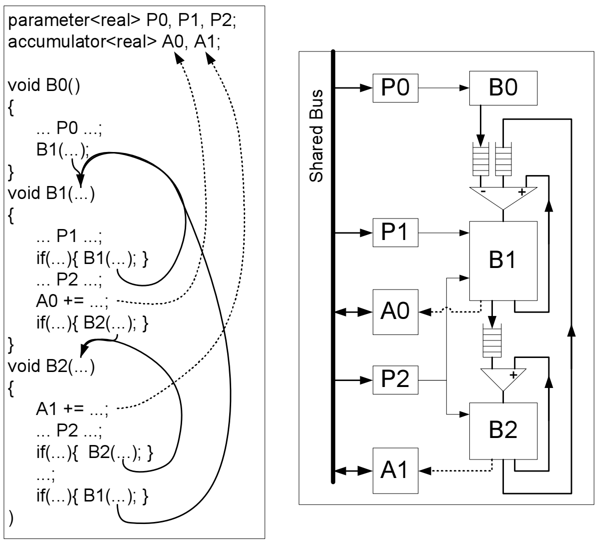
4.4 Program Assembly

After creating the concrete pipelines for all blocks, instantiating parameter registers, and selecting the appropriate accumulator components, we are left with a collection of component instances, but no connections between them. We now need to establish five different types of connections: Parameter to block; block to accumulator; bus to parameter; bus to accumulator; and block to block.

The first of these, connecting parameters to blocks, simply wires the output of the parameter registers to the appropriate parameter inputs of block instances. The block to accumulator connections are also just wires, as there is an accumulator instance for each accumulator output port on a block. The bus connections are also very simple, as the shared bus is a relatively low-performance hierarchical memory-mapped bus, requiring a small amount of arbitration logic to map the accumulators and parameters into the bus hierarchy.

The final connections, from block to block, are more complex, as a block may be the target of multiple block calls, requiring arbitration. Figure 3 gives an abstract example of a program that contains multiple transfer statements. Block B2 contains just one transfer statement, which transfers back to itself (i.e. it is a tail-recursive block). However, block B1 also needs to transfer to block B2, so there are two sources of threads at the input to B2.

We assume that all threads in the program will eventually exit (there are no infinite loops in well-formed programs), so it is more important to finish threads that are already executing in block B2, rather than incoming threads from B1. This is enforced using a biased multiplexor at the input to block B2: if the thread state from the recurrent connection is valid (indicated by the token of the transfer output group)



```
parameter<real> P0, P1, P2;
accumulator<real> A0, A1;

void B0()
{
    ... P0 ...;
    B1(...);
}
void B1(...)
{
    ... P1 ...;
    if(...){ B1(...); }
    ... P2 ...;
    A0 += ...;
    if(...){ B2(...); }
}
void B2(...)
{
    A1 += ...;
    ... P2 ...;
    if(...){  B2(...); }
    ...;
    if(...){ B1(...); }
)
```

Figure 3. Assembly of blocks, parameters and accumulators to implement a program;

the multiplexor will always select that thread, only taking the thread from B1 if the feedback thread is not valid.

The feedback connection is always accepted, so no buffering is needed, but when a block call from statement to B2 from B1 occurs, there is no guarantee that the thread can be accepted in that cycle. The solution used is to simply add a block-RAM based FIFO at the output of block B1, so threads are buffered until B2 is able to accept them.

Block B2 is the target of three block calls, so we now need a three-way select. As before, we bias the selection towards the direct recursive call from within the block, but there are then still two inputs to choose from. In such a situation we bias *against* the channel that is the shortest distance from the thread initiator block. Here the thread initiator block is B0 (as it has no inputs), so we bias against threads arriving from B0: only if the transfer groups from both itself and B2 are invalid, will block B1 accept a new thread from B0.

All connections between blocks are resolved in this way, eventually resulting a network of fully-pipelined blocks. At runtime the thread initiator block (B0) continually attempts to spawn new simulations, by transferring new threads to the downstream block B1. B0 will initially accept all new threads, which enter through the pipeline, and some number of cycles later may either attempt to transfer back to the start of B1, or to B2. These threads within the system take up capacity, either because they are in a block execution pipeline, or because they are waiting in a FIFO for their target block to become available. This clearly leads to the possibility of data-loss, as the network has finite storage, so a thread arriving at a full FIFO will have to be discarded.

To avoid data-loss we implement a simple back-pressure system using the FIFOs. Each FIFO fed by a block has a partially-full flag, which equals the depth of the FIFO minus the latency of the block pipeline feeding it. When the partially-full flag is asserted, the block will stop accepting all new threads, except those that arrive via feedback connections. This ensures that the capacity of the block cannot be overflowed. This blocking will eventually ripple backwards through the network, stopping new threads from entering the system; however, all threads using feedback connections continue to execute, so the system makes progress. This scheme has worked well in our initial experiments, but we have not yet examined properties such as freedom from dead-lock and live-lock.

5 Results

We now present some initial performance results gained from our compiler, using the five simulation kernels described and benchmarked in [7] All five use single-precision floating-point, for which we used the Xilinx CoreGen version 2 floating-point cores. Xilinx ISE 8.1 was used for all synthesis (via XST) and place-and-route tasks. Standard synthesis effort levels were used, the "fast_runtime" xflow options were used for place-and-route, and no clock constraints were applied.

Table 1 shows results for a Virtex-4 implementation in an xc4vlx100-12 part, in terms of performance, resource-usage, and lines of code. In the five tested simulations, the performance is directly proportional to the clock rate, as the main body of computation is performed in a single iterative block. Each iteration of this main block advances a simulation (thread) one time step, so the clock-rate in MHz directly determines the application performance in millions of simulations steps per second (MSteps/s). Clock rates in excess of 300MHz are achieved for all simulations, even though we applied no clock constraints and used fast-runtime settings for the place-and-route process.

The resource usage of the simulations is shown in the next set of columns, including the percentage of available xc4vlx100 resources used in parenthesis. The maximum resource utilisation of 25% of DSPs is seen in the GARCH simulation, due to the large number of multipliers used in the simulation's transition function. However, this suggests that the design can be replicated up to four times, potentially quadrupling performance (replication will be explored in a future paper).

One of the fixed resource overheads for all the simulations is the need for an accumulator: all the simulations have a single accumulate statement at the end, and so each require a single accumulator instance. The accumulator used is a high-performance floating-point component, and tracks four different statistics (maximum, minimum, mean,

	MHZ	Slices	(%)	LUTs	(%)	FFs	(%)	RAMs	(%)	DSPs	(%)	LoC
Random-Walk	345	3443	(7.0)	4723	(4.8)	4265	(4.3)	3	(1.3)	4	(4.2)	19
Random-Jump	324	5105	(10.4)	6649	(6.8)	6345	(6.5)	4	(1.7)	4	(4.2)	23
Scaling-Walk	338	3777	(7.7)	5064	(5.2)	4716	(4.8)	3	(1.3)	12	(12.5)	23
Bivariate-Walk	346	7022	(14.3)	9083	(9.2)	8813	(9.0)	7	(2.9)	16	(16.7)	30
GARCH	301	6523	(13.3)	8534	(8.7)	8345	(8.5)	5	(2.1)	24	(25.0)	24
Accumulator	361	2199	(4.5)	3203	(3.3)	2042	(2.1)	0	(0.0)	4	(4.2)	-

Table 1. Performance results of five different simulations described in Contessa and implemented in the xc4vlx100, measured by performance, resource usage, and Lines of Code (LoC).

and standard-deviation), so it contributes a significant base level of resource usage. The resource usage of the accumulator component is shown in the bottom row of Table 1, suggesting that, for the simpler simulations, over half the resources are actually being used in the accumulator.

The final column of the table attempts to summarise the brevity of Contessa, by including the Lines of Code (LoC) required. Although LoC is an imperfect measure of code expressiveness, it is worth observing that the Contessa program is a complete executable specification: these programs specify all inputs and outputs, and can be automatically compiled to a bitfile, loaded into software, and used to run the simulation, literally at the push of a button.

These performance figures mean little in isolation, so Table 1 provides a comparison with two other implementations. The first comparison is to the previous manual implementation of the five simulations, presented in [7]. These were created by manually scheduling the Xilinx floating point cores using Handel-C, and are listed as HC in the table. The second is to a software implementation of the simulations in C++, executed on a 2.2GHz Opteron. Although the code was not hand-optimised for maximum performance, it was written in a style that allowed for complete static analysis by the compiler, then compiled using g++ with full architecture specific optimisations. All numbers in the table represent ratios between the performance of the two implementations, with numbers exceeding one indicating that the Contessa implementation is better (shown in bold), and numbers less than one showing the other implementation is better (shown in italic).

In terms of performance the Contessa implementation easily beats that of the software, showing a maximum speedup of 62 times, and a (geometric) average speedup of 35 times. The speedup over the previous hardware implementation is lower, but is on average almost two times faster. In LoC the Contessa version also performs well in comparison to Handel-C, even though this comparison ignores the large amount of framework and support code used in the Handel-C version, measuring only the simulation-specific code.

One aspect where the Contessa implementation does not perform well in the comparison is in resource usage, as on average the Handel-C version uses approximately two-thirds of the resources used by Contessa. However, this can largely be explained by the much larger and more capable accumulation unit used by Contessa. The Handel-C version only calculates the arithmetic mean, using a fixed-point accumulator (with a scale that must be pre-determined at compile-time), which does not provide enough statistical information for most real-world applications. If the resources used by the accumulators are removed from both hardware versions (shown in the last row of the table), then the resource usage is much closer. The only significant difference is that the Contessa uses more RAMs to implement the FIFOs between blocks.

6 Related Work

FPGAs have previously been used for Monte-Carlo simulations [4, 5, 1, 8], but these were all developed from scratch using application-specific HDL. An automated method for compiling simulations into hardware was described in [7], but the actual implementation steps were performed by hand. The described method is also much more restrictive than that presented here, allowing only a small subset of possible simulations to be described.

There are a number of examples of other FPGA languages for stream-compilation [2, 6, 3] , but most have avoided or restricted loops due to the problems introduced by loop-carried-dependencies. By comparison Contessa offers arbitrarily complex control-flow, but places restrictions on the types and locality of data that can be used. The closest to Contessa in design and implementation is the tagged-token language presented in [6], which uses the same idea of pipelined data-flow graphs with internal feedback arcs for iteration. However, the language uses a traditional imperative approach, extracting parallelism via loops over source and sink memory arrays that must be explicitly identified, rather than using an explicit threading model.

	MSteps/s		Lines of Code	Resources					Place & Route
	HC	C++	(to HC)	Slices	LUTs	FFs	RAMs	DSPs	Time
Random-Walk	**1.84**	**16.58**	**4.74**	*0.51*	*0.52*	*0.47*	*0.67*	*0.00*	**1.62**
Random-Jump	**1.91**	**46.21**	**3.87**	*0.76*	*0.78*	*0.66*	*1.00*	*0.00*	**1.36**
Scaling-Walk	**1.72**	**24.50**	**4.87**	*0.59*	*0.55*	*0.56*	*0.67*	*0.67*	*0.86*
Bivariate-Walk	**2.25**	**45.74**	**3.87**	*0.67*	*0.68*	*0.60*	*0.57*	*0.75*	*0.67*
GARCH	**1.99**	**62.09**	**6.29**	*0.80*	*0.75*	*0.77*	*0.40*	*0.67*	**1.16**
Geometric Mean	**1.94**	**35.10**	**4.65**	*0.66*	*0.65*	*0.60*	*0.63*	*0.69*	**1.08**
Mean (No Acc.)	-	-	-	**1.26**	**1.36**	*0.93*	*0.63*	*0.93*	-

Table 2. Performance of Contessa implementations relative to a manual Handel-C implementation on the same Virtex-4 part (shown as HC), and a C++ software implementation on a 2.2GHz Opteron. All numbers are ratios, with values exceeding one (in bold) indicating that Contessa performs better.

7 Conclusion

This paper presents the Contessa language, a pure-functional continuation based domain-specific language for describing path-based Monte-Carlo simulations. By exploiting the huge amount of thread-level parallelism available in simulation applications, the language is able to describe programs using complex control-flow and loop-carried dependencies, without sacrificing performance. The language also does not require architecture- or platform-specific annotations, allowing a single source description to target software and hardware, providing a true push-button route from description to run-time execution.

We compare the performance of Contessa across five simulations, against both a software and Handel-C hardware implementation. In terms of raw performance the Contessa implementations are on average twice as fast as the Handel-C implementation, and 35 times faster than a C++ implementation on a 2.2GHz Opteron. The Contessa source code is concise, requiring a quarter of the lines of code used in Handel-C. The Contessa versions require two thirds more area than the Handel-C area, but this is due to the more sophisticated accumulation logic it uses, collecting more statistics with greater accuracy. After normalising for area spent on accumulation logic the Contessa and Handel-C implementations use very similar numbers of resources.

The Contessa language is intended to form a platform, on which different strategies for Monte-Carlo and other financial applications can be tested. There is significant potential for automatic optimisations, such as storing loop variables in local RAMs (rather than propagating them through the register-based feedback loop), splitting and merging nodes to optimise communication, and applying constant-propagation techniques to parameters. We also plan to adapt the probabilistic ideas from [6], using both static analysis and profiling to identify the probability of each block call, allowing block resources to be tailored according to the computational load of the block. The FIFO communications links provide significant flexibility in block implementation, so the tradeoff between block resources and performance could be implemented using multiple clock-domains, partially sequential blocks, or even CPU blocks. Blocks can also be transparently replicated, allowing the most heavily loaded blocks to be partitioned. The intention is to make these options completely automatic, requiring no modifications or annotations to the Contessa source code.

References

[1] M. Gokhale, J. Frigo, C. Ahrens, J. L. Tripp, and R. Minnich. Monte Carlo radiative heat transfer simulation on a reconfigurable computer. In *Proc. Int. Conf. on Field Programmable Logic and Applications*, pages 95–104. Springer-Verlag, Berlin, 2004.

[2] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, pages 49–56, 2000.

[3] O. Mencer. ASC: A stream compiler for computing with FPGAs. *IEEE Transactions on CAD*, 25(9):1603–1617, 2006.

[4] S. Monaghan. A gate-level reconfigurable monte carlo processor. *J. VLSI Signal Process. Syst.*, 6(2):139–153, 1993.

[5] A. Negoi and J. Zimmermann. Monte Carlo hardware simulator for electron dynamics in semiconductors. In *International Annual Semiconductor Conference*, pages 557–560, Sinaia, Romania, 1996.

[6] H. Styles and W. Luk. Exploiting program branch probabilities in hardware computation. *IEEE Transactions on Computers*, 53(11):1408–1419, 2004.

[7] D. B. Thomas, J. A. Bower, and W. Luk. Automatic generation and optimisation of reconfigurable financial monte-carlo simulations. In *IEEE Int. Conf. on Application-specific Systems, Architectures and Processors*, 2007.

[8] G. L. Zhang, P. H. W. Leong, C. H. Ho, K. H. Tsoi, D.-U. Lee, R. C. C. Cheung, and W. Luk. Reconfigurable acceleration for Monte Carlo based financial simulation. In *Proc. Int. Conf. on Field-Programmable Technology*, pages 215–224. IEEE Computer Society Press, 2005.