Rapid Functional Modelling and Simulation of Coarse Grained Reconfigurable Array Architectures

Kunjan Patel, Séamas McGettrick, C. J. Bleakley

UCD Complex and Adaptive Systems Laboratory,
UCD School of Computer Science and Informatics,
University College Dublin, Ireland

Abstract

Increases in the complexity of Coarse Grained Reconfigurable Array (CGRA) architectures have made implementation of new architectures difficult and time consuming. Due to the large number of design options available, it is difficult for designers to make optimal design decisions in the early stages of the design cycle. This paper proposes a novel functional modelling framework for CGRA architectures which makes the design space exploration process easier and faster. The framework allows architecture modelling, application mapping and simulation in a single environment, avoiding development of a complex tool set. The proposed approach provides flexibility which allows users to quickly investigate many design options without remodelling. The usefulness and extensibility of the framework is illustrated by presentation of a case study and associated design metrics.

Key words: Functional modelling, simulation, Java, CGRA

1. Introduction

Reconfigurable architectures have become popular in recent years because of their advantages over Application Specific Integrated Circuits (ASIC) and

Email addresses: kunjan.patel@ucd.ie (Kunjan Patel), seamas.mcgettrick@ucd.ie (Séamas McGettrick), chris.bleakley@ucd.ie (C. J. Bleakley)

URL: www.kunjanpatel.co.nr (Kunjan Patel), www.csi.ucd.ie/users/chris-bleakley (C. J. Bleakley)

microprocessors, i.e., flexibility, high throughput and low Non-Recurring Engineering (NRE) cost. In particular, CGRA architectures are emerging as high performance, low power design options [1]. A CGRA architecture consists of a grid of interconnected reconfigurable processing units which can perform logical or arithmetic operations. Design of CRGAs using Hardware Description Languages (HDLs) is a time consuming and costly process. Due to the the low level of abstraction in HDLs and to the complexity of CGRA systems, small changes in the architecture can lead to large changes in the implementation. As a consequence, it is common practice to develop a functional model of an new CGRA architecture in software early in the design process so that its performance can be checked prior to HDL implementation [2][3].

In this paper, a novel framework, Rapid Configurable Array Modeller and Simulator (RaCAMS), is proposed for rapid modelling, cycle accurate functional simulation and application driven design space exploration of CGRA architectures. Previously CGRA simulators have been designed for functional simulation of a fixed CGRA architecture, rather than for design space exploration. RaCAMS is designed to be flexible so that the developer can quickly and easily assess the impact of CGRA architectural changes. The RaCAMS approach improves on previously described CGRA simulation approaches in four ways.

Firstly, RaCAMS uses a single software model for architecture description, application mapping and simulation. Conventional simulators read the CRGA configuration from an input file and parse it. For ease of programming, the configuration file is typically textual and human readable. The parser component of the simulator is complex and sensitive to changes in the architecture. In RaCAMS, the core Simulation Engine provides an Application Programming Interface (API) which enables CGRA configuration by an application module which calls methods and passes parameters to the simulation engine via the API. In RaCAMS, the configuration is part of the simulator source code and, as such, is syntax and type checked by the compiler. Hence a parser module or program need not be developed and maintained. This approach does lead to modular re-compilation when simulating new configurations but, given the speed of modern compilers, this is a small fraction of the overall development time.

Secondly, since conventional simulators are designed for a fixed architecture, modification of the architecture always requires source code changes. In RaCAMS, the architecture parameters which are more commonly altered,

such as bit width and number of array elements, are parameterized and can be altered at run-time. This accelerates design space exploration. As in conventional simulators, more complex architecture changes, such as support for new operations, are implemented by modification of the source code.

Thirdly, the RaCAMS software framework has been designed in an object-oriented manner so as to facilitate later modification and extension. Hence, base classes are provided for user interaction (Graphical User Interface) and for various CGRA design options, for example, migration from fixed to floating point.

Fourthly, unlike previous simulators, RaCAMS has been written in Java. Java was chosen because programming in Java has been found to be three times less prone to errors and two times more productive than C++ [4]. In addition, unlike C++, Java has built-in support for dynamic class instantiation. This feature allows RaCAMS more flexibility, enabling exploration of a greater variety of architectural design options at run-time.

Since RaCAMS is bit-true and cycle-exact, once design space exploration is complete, the final version of the simulator can be used for verification of the hardware by co-simulation, for development, verification and performance assessment of new CGRA configurations, and for host processor software development.

This paper is structured as follows. Section 2 presents a survey of relevant CGRA modelling and simulation methodologies. A description of CGRA architectures is given in Section 3. In Section 4, the proposed approach is described. Section 5 describes a case study of architecture exploration for a matrix multiplication algorithm. Finally, the paper is concluded in Section 6.

2. Related Work

Developing a CGRA is a complex and time consuming process. In recent years, a number of researchers have tackled this problem. Table 1 shows a comparison of the various software modelling approaches employed, including RaCAMS.

As can be seen, most previous CGRA simulators are focused on verification of a fixed architecture and on providing the programmer with support for development of new configuration settings. Most use conventional programming languages, such as C++ or C, for implementation of the simulator.

Table 1: Comparison of CRGA software simulation approaches (mod=modification).

Architecture	MorphoSys	RICA	Montium	CGADL	PACT XPP	RaCAMS
Reference	[5]	[6]	[7]	[8]	[9]	n/a
Granularity	CGRA	CGRA	Mixed	CGRA	CGRA	CGRA
Simulation approach	software simulator	software simulator	HW/SW co-simulation	generated software simulator	software simulator	software simulator
Programming language(s)	C++	C++	VHDL and Matlab	SystemC	C	Java
Tool(s)	open source	open source	commercial	custom proprietary	open source	open source
Architeture Exploration	source code mod	source code mod	source code mod	architecture mod and simulator source regeneration and recompilation	source code mod	run-time parameter mod or source code mod
Parser	User written	User written	User written	Generated	User written	Part of compiler
Run-time architecture exploration	no	no	no	no	no	yes

A hardware-software co-simulation approach was used for simulation in Montium. The Montium software was implemented in Matlab while the hardware was modelled in VHDL and simulated in ModelSim [10]. This approach allows interchange of software and hardware counterparts of the components. However, a small change in architecture can lead to major changes in the software and hardware models [11].

The RICA team proposed a high speed software simulator for CGRAs [12]. The simulator was implemented in C++ and achieves simulation speeds close to those of FGPA based emulators. Similarly to the work described herein, an object-oriented approach is used to ensure that the simulator is extensible. However, the focus of the work is on rapid simulation, rather than rapid modelling. The model does not support integrated configuration or run-time architecture modification.

The PACT XPP project uses a Native Mapping Language (NML) to describe the configuration settings for their architecture. The NML requires a dedicated compiler, xmap, for converting the NML source files to configuration binaries. The behaviour of the xmap compiler is tightly coupled with the architecture and will require modifications which the CGRA architecture is modified.

An Array Description Language (ADL) was presented in [13]. This language is used to describe CGRAs at a high level of abstraction. A simulator for and HDL description of the array is generated based on this architecture description. A present, it is unclear how restrictive ADLs are in the variety

of architectures and functionality that they support - source code changes may still be required. Furthermore, run-time architecture exploration is not supported. After an architecture change the simulator, must be re-generated and re-compiled. ADLs rely on a complex simulator generation tool which requires a significant development effort. To the authors' knowledge, no ADL tool set is currently publicly available, making the approach impractical for small development teams.

3. Preliminaries

Figure 1 shows the general architecture of a CGRA. The main processor performs control operations and irregular tasks while computationally intensive tasks are performed by the CGRA. Input data is injected by a Direct Memory Access (DMA) unit controlled by the main processor. When the CGRA computations are finished, output data is transferred to main memory by the DMA. Each CFU in the array can perform a Set of Operations (SoO) such as addition, multiplication, logical AND etc. The operation that a CFU performs is controlled by its configuration. Similarly, the transfer of data between the CFUs is determined by the configuration. CGRA architectures provide higher performance than a processor because regular algorithms can be divided into a number of basic operations which run in parallel on the array. Since CFUs are reconfigurable, the same architecture can be used for execution of various algorithms.

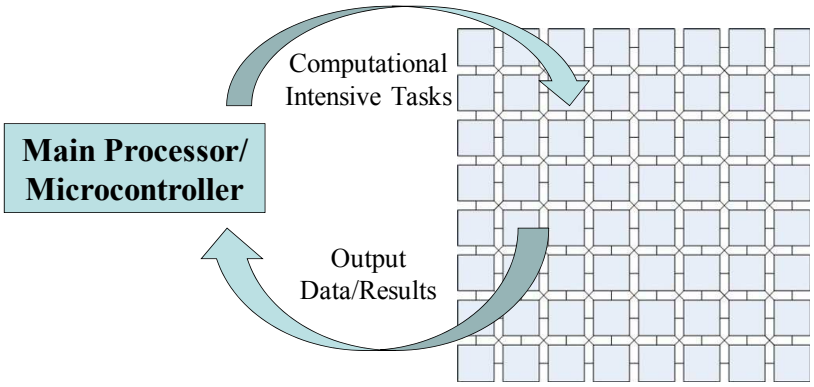

Figure 1: General CGRA architecture

4. Proposed Approach

The proposed approach uses a functional model to predict the behavior and performance of a CGRA architecture. The design flow when using RaCAMS is shown in Figure 2. After deciding the architecture's application scope, a primitive SoO can be decided and the corresponding RaCAMS model can be designed. The performance of the architecture is then assessed and the architecture design is tuned until the performance goals are met.

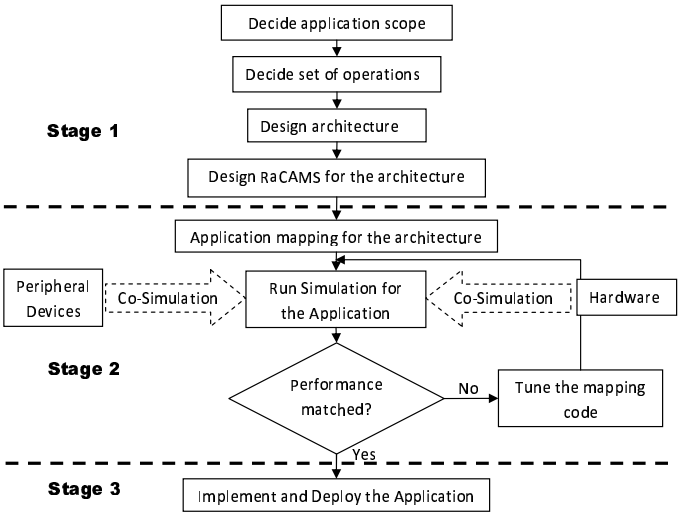

Figure 2: CGRA architecture design flow using RaCAMS

4.1. RaCAMS Architecture

Figure 3 shows an high level view of the RaCAMS architecture. RaCAMS is divided into four modules.

- *The Simulation Engine* consists of an complete array model including models of the CFUs to be simulated. Each CFU executes operations and passes data to other CFUs according to the mapping and configuration setup. The Simulation Engine records usage statistics during simulation.

- *The Application Mapping Unit* consists of modules which configure the Simulation Engine according to the requirements of the application.

- *The Data Exchange Unit* consists of a RAM model. The module reads the input data from a file and writes the output data to file.

- *The Graphical User Interface (GUI)* is designed to make RaCAMS user friendly. There are two main GUIs. One GUI takes input parameters and the second GUI displays simulation statistics. Command line control is also available.

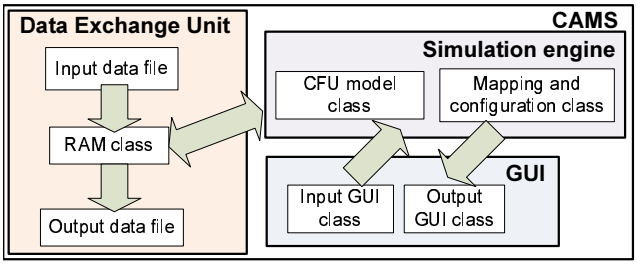

Figure 3: Overview of RaCAMS structure

A Unified Modelling Language (UML) diagram showing the details of the RaCAMS architecture is shown in Figure 4. The digram shows the most important classes, the data that they contain and the methods included in them, as well as the relationship between classes.

Every CFU is modelled as a class (CFU); registers and I/O ports are modelled as arrays of variables; decoders and multiplexers as switch-case statements and the operations supported by the CFU are described using methods. An array of CFUs is modelled by instantiating multiple instances of the CFU class (Mapping Main). The array hardware is described by passing the row and column indices of the CFUs as arguments. The configuration of the array is set up by the application mapping class (Application mapping). This includes the operation to be performed by each CFU and how data is transferred between the functional units, the CFU register files and the CFU input and output ports. A class is used to model the behaviour of RAM and transfer data to and from file (RAM). This basic architecture can be extended by adding classes to implement new applications (<interface> Application) or specialized functionality (e.g. Data Rounding).

Once simulation starts, the CFU registers are updated on a cycle by cycle basis. In this way, support is provided for modelling the parallelism inherent in the final array.

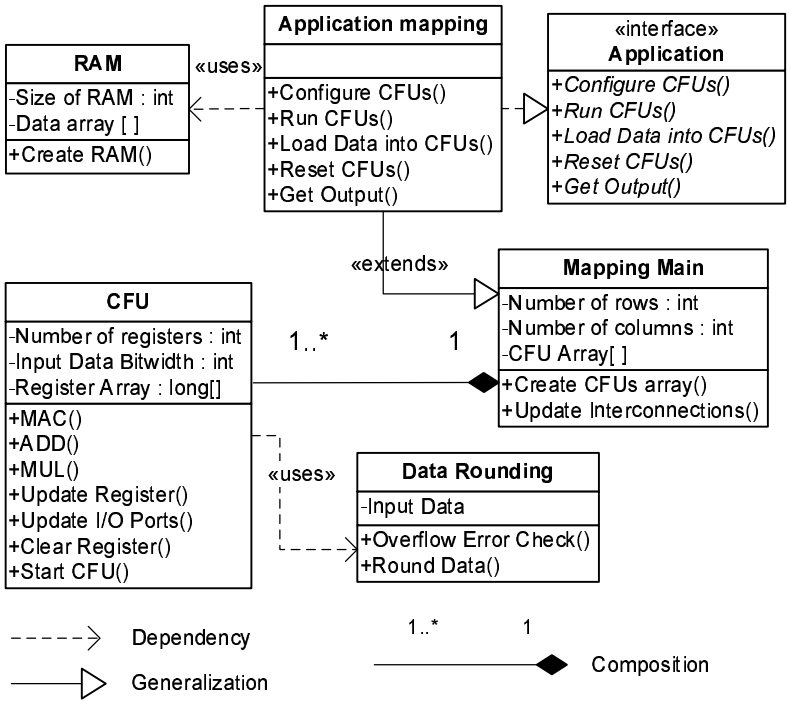

Figure 4: UML diagram of RaCAMS structure

Application mapping is a crucial phase of system design and the performance of the system depends on it [14]. Automatic application mapping is a substantial problem and is beyond the scope of this paper. Automatic mapping can be achieved after architecture is focused and so manual application mapping was performed in this work.

To map a new algorithm, a new class is created which is inherited from the Application Mapping class. The class sets the array configuration by means of the method calls and parameter passing via the Simulation Engine API.

4.2. Flexibility

RaCAMS provides flexibility to the designer to test various design options prior to hardware implementation. A list of architecture design options is provided in Table 2. With the exception of the SoO, all of these options can be altered at run-time without recompilation or source code modification. This flexibility is due to the object-oriented design of the RaCAMS framework and to the dynamic instantiation features of Java. Due to their

complexity SoO changes require source code modification. However, the framework provides a library of pre-defined operations on which the SoO can be based.

Table 2: Flexibility levels in RaCAMS

Level	**Design option**	**Run-time modification**
Simulation	Number of iterations	yes
	Size of RAM	yes
	Data rounding	yes
CFU	Number of registers per CFU	yes
	Set of Operations (SoO)	no
	CFU bitwidth	yes
Array	Number of columns	yes
	Number of rows	yes
	Interconnections	yes

4.3. Resource Usage Statistics

At the end of the simulation, the user receives a report with following resource usage statistics:

- Number of times the registers are accessed
- Number of times the RAM is accessed
- Number of times each operation is performed

The user can add more statistics by modifying the source code. By default, the above statistics are calculated across all CFUs but the user can obtain reports for individual CFUs.

5. Modelling and Simulation using RaCAMS - A Case Study

This section describes how a CGRA architectures can be modelled, simulated and assessed using the proposed approach. Three architectures are assessed for the purposes of matrix multiplication.

5.1. Selection of SoO

The three CFU architectures are shown in Figure 5 and are summarized in Table 4. CGRA-A has a cross interconnect and can execute one of two operations: NOP (No Operation) or MAC (Multiply Accumulate). CGRA-B has star and cross interconnects and each CFU can execute one of three operations: NOP (No Operation), ADD (Addition) or MUL (Multiplication). CGRA-C, has a double cross interconnect with the same SoO as CGRA-A. In all three architectures it is assumed that data is injected by a DMA unit according to a data access pattern set by the host processor.

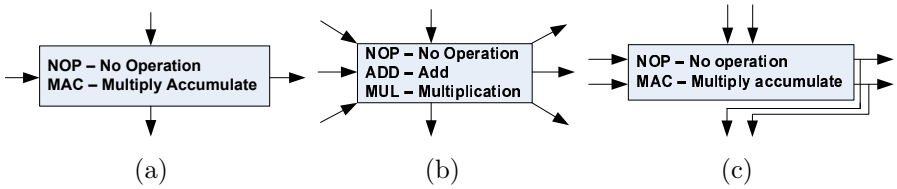

Figure 5: Three types of CFU: a) CFU-A; b) CFU-B and c) CFU-C

Table 3: Specification of CFUs

Specification	**CFU-A**	**CFU-B**	**CFU-C**
SoOs	NOP, MAC	NOP, ADD, MUL	NOP, MAC
Registers	3 Internal, 1 Output	3 Internal, 1 Output	3 Internal, 1 Output
Input ports Output ports	2 2	4 4	4 2
Interconnection Scheme	Cross	Cross+Star	Double cross
Data Flow Direction	E→W, N→S	E→W, N→S, NE→SE, SW→NE	E→W, N→S

5.2. Application Mapping

Matrix multiplication is a common function used in DSP and multimedia applications [15]. To illustrate how attributes such as SoO and interconnection affect algorithm mapping, matrix multiplication of two 4x4 matrices was mapped to the CGRA-A, CGRA-B and CGRA-C architectures. The basic equation for matrix multiplication is given in (1). The resultant matrix (C)

elements are given by the sum of the products of elements in A and B matrix. In the 4x4 matrix example, four multiplications and three additions are required to calculate each element of the result matrix.

$$C_{ij} = \sum_{k=0}^{n} A_{ik} B_{kj} \tag{1}$$

Figure 6(a) shows the mapping for matrix multiplication on CGRA-A. For simplicity, only the mapping for calculating one output element, C_{11},where C is the output matrix is shown. As CFU-A can perform a MAC operation, each CFU performs a single iteration of (1). This partial answer is then passed to the next CFU where it is summed with the next partial answer until the solution is calculated. This mapping can be replicated on the array to calculate all of the elements of the output matrix.

Likewise, Figure 6(b) shows the mapping of matrix multiplication to CGRA-B. CFU-B is unable to compute a full iteration of 1 since it can only perform either a single multiply or a single addition in one clock cycle. Therefore the elements of matrix A and B are first multiplied and then the results are passed to adjoining CFUs to be added together, to calculate the resulting C element. It can be seen that the use of ADD and MUL operations requires extra star interconnections (SW→NE) in order to calculate matrix multiplication. This requires more area than cross interconnects. Thus a small change in the SoO can lead to a large change in the overall architecture.

In CGRA-C, each of the CFUs is set to multiply two of the inputs and accumulate the result in an internal register. The input data is also routed to the next CFU to the right and below. In this way the whole matrix multiplication result can be calculated with maximum data reuse and minimal time. In Figure 7(a) the first clock cycle of the matrix multiplciation on CGRA-C is shown. The data is fed by the DMA to the inputs of the top left CFU where it is multiplied. The second clock cycle is shown in Figure 7(b). Here the second partial answer is calculated and accumulated in the top left CFU. Meanwhile the data elements used in the top left CFU in the first clock cycle are now used by the two adjoining CFUs which also calculate the partial answer needed for another element of the result matrix. The process continues in cycle three when partial answers are calculated in six of the sixteen CFUs, see Figure 7(c). Finally in Figure 7(d), the top left CFU has calculated its final result for element C_{11}. The other CFUs continue to be fed data from

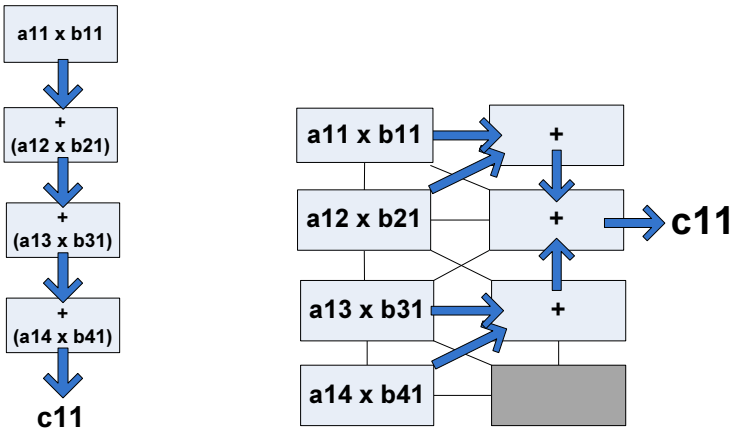

Figure 6: Matrix multiplication mapping of two 4x4 matrices on a) CGRA-A and b) CGRA-B

the DMA and pass data to the right and down until the full matrix product is calculated. The DMA feds the data as follows:

$$A\left(t,c\right)=\begin{bmatrix} a11 & 0 & 0 & 0 \\ a12 & a21 & 0 & 0 \\ a13 & a22 & a31 & 0 \\ a14 & a23 & a32 & a41 \\ 0 & a24 & a33 & a42 \\ 0 & 0 & a34 & a43 \\ 0 & 0 & 0 & a44 \end{bmatrix}, B\left(t,c\right)=\begin{bmatrix} b11 & 0 & 0 & 0 \\ b21 & b12 & 0 & 0 \\ b31 & b22 & b13 & 0 \\ b41 & b32 & b23 & b14 \\ 0 & a42 & a33 & a24 \\ 0 & 0 & a43 & a34 \\ 0 & 0 & 0 & a44 \end{bmatrix}$$

where, row t is injected into c^{th} CFU of CGRA-C in the t^{th} execution cycle.

5.3. Design Metrics

The architectures described above were implemented using the RaCAMS framework. Table 4 shows the results of multiplication of two 4×4 matrices on the CGRA-A, CGRA-B and CGRA-C architectures. All results are for single channel data processing on a 8×8 CFU array. It can be seen in Table 4 that mapping to CGRA-A requires less register accesses than mapping to CGRA-B and CGRA-C. But CGRA-C requires only half of the RAM accesses than CGRA-A and CGRA-B. So, CGRA-A or CGRA-B can be chosen for high performance applications and CGRA-C can be chosen for low power

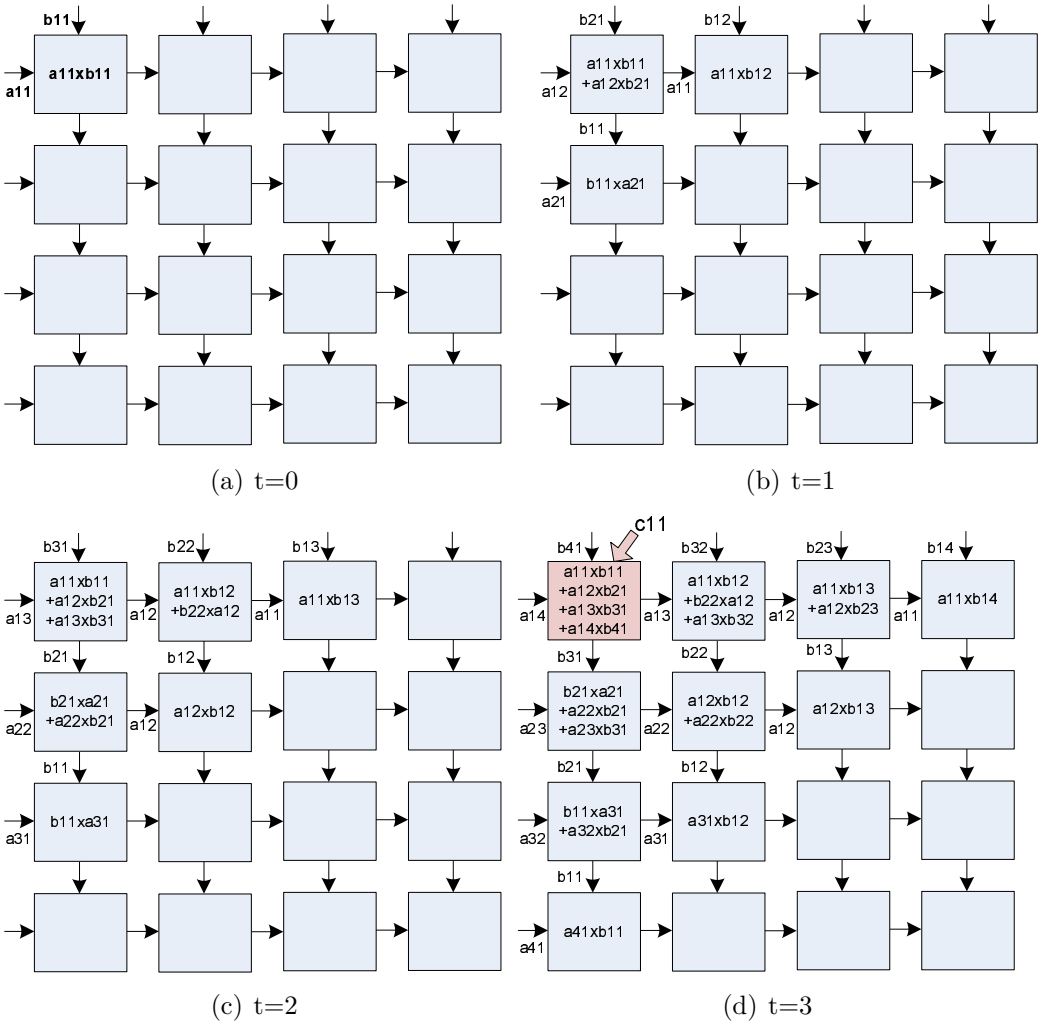

Figure 7: Matrix multiplication mapping of two 4x4 matrices on CGRA-C

applications. The CFU utilisation factor (CUF), i.e. the ratio of the number of CFUs used to the number of CFUs idle, for CGRA-A and CGRA-C is greater than for CGRA-B. The greater the CUF the more channels can be processed in parallel, increasing performance.

5.4. Speed vs Number of CFUs:

The effect of the number of CFUs on performance was investigated using RaCAMS. The number of CFU objects instantiated was varied per row and

Table 4: Matrix Multiplication (8x8 array) Results

Statistics	**CGRA-A**	**CGRA-B**	**CGRA-C**
CFU utilization factor	1	0.64	1
Maximum number of channels can be processed in parallel	16	8	4
Number of cycles required when maximum number of channels processed	64	77	10
Register accesses	192	224	208
RAM accesses	80	80	32
Number of CFUs configured per iteration for single channel	16	28	16
Addition operations	-	48	-
Multiplication Operations	-	64	-
MAC operations	64	-	64

per column and the same mapping was replicated over the array of CFUs. In CGRA-C, performance does not vary with the number of CFUs for a single channel since $m \times m$ CFUs are required to perform matrix multiplication of two $m \times m$ matrices. Figure 8 shows the number of cycles required to perform matrix multiplication of two 4x4 matrices for a single channel. It can be seen that after a certain point (64 CFUs), there is no improvement in the speed of execution because of data dependency. However, this breakpoint varies according to the particular application. Hence the designer needs to assess the appropriate trade-offs between performance and the number of CFUs for each application.

5.5. Speed vs Number of CFUs vs Number of Channels:

In the case of multi-channel data streaming applications, parallelism can be increased by processing multiple channels simultaneously. This can be carried out in two ways. One way to do this is to increase the number of CFUs in the array and perform the whole operation for all of the channels in parallel. The second way is to divide an operation into smaller operations and execute these small operations for multiple channels in parallel. For example, using CGRA-A, all of the elements of the output matrix can be calculated in parallel which allows fewer channels to be processed in parallel. Alternatively, all of the elements of the output matrix can be calculated one by one over multiple channels. In other words, the number of channels that

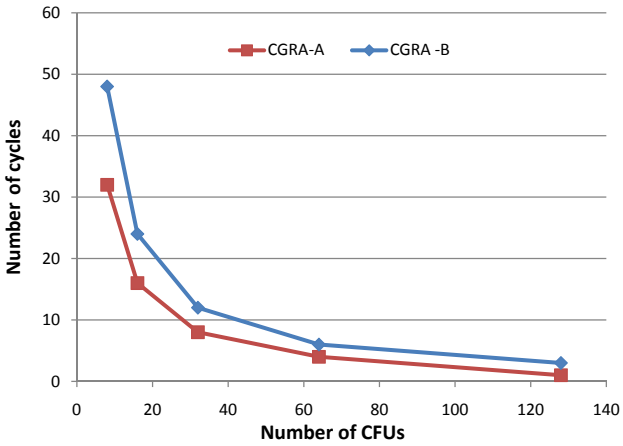

Figure 8: Trade off- Cycles vs Number of CFUs

can be processed in parallel depends on the number of CFUs as well. Thus, the problem requires 3-D design space exploration by the user.

This can be performed using the RaCAMS framework by varying the number of CFUs and making changes to the application mapping class. Three different architectures, with 16, 64 and 128 CFUs respectively, executing matrix multiplications of two 4x4 matrices over multiple channels of data were considered. Figure 9 shows that CGRA-A with 16, 64 and 128 CFUs can process a maximum of 4, 16 and 32 channels respectively in parallel. Whereas CGRA-B with 16, 64 and 128 CFUs can process a maximum of 2, 8 and 16 channels respectively in parallel.

It can be observed from Figure 9, that for a given architecture, as the number of channels increases the number of cycles required to obtain the output also increases. So, depending on the performance requirement, a suitable trade-off can be made between the number of cycles required to execute the application and the number of CFUs.

5.6. Data Reuse

The mapping statistics generated by RaCAMS can be used to analyse the data reuse capability of the architecture. Figure 10 shows a comparison of RAM Data Reuse (RDR) for all three CGRA architectures. RDR is given by:

$$RDR = \frac{Number\ of\ unique\ RAM\ addresses\ accessed}{Number\ of\ RAM\ accesses} \tag{2}$$

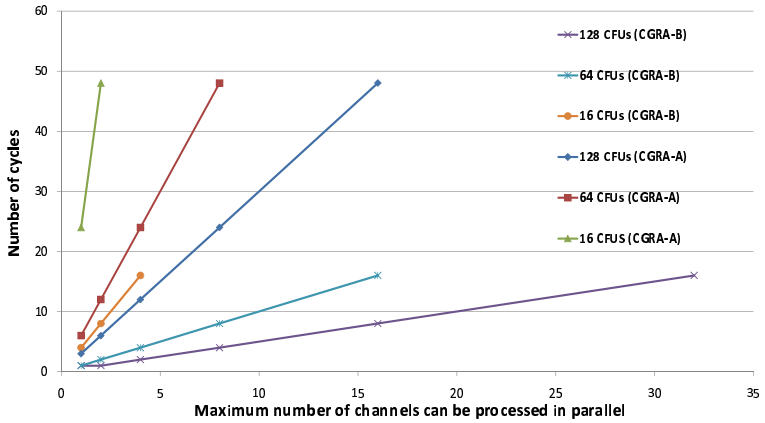

Figure 9: Trade off- Number of CFUs vs Number of Cycles vs Number of Channels in CGRA-A and CGRA-B

A RDR of 1 means that the CGRA only accesses input and output RAM locations and only accesses them once. A RDR of 0.2 means that 80% of RAM accesses are to locations that have already accessed or are used for temporary results. Ideally, to reduce energy consumption, RDR should be 1 meaning that input data is read only once, that no intermediate results are stored in memory and that output data is written only once. It can be seen that CGRA-C has the maximum RDR having less memory accesses than CGRA-A and CGRA-B. This illustrates the effectiveness of CGRA-C architecture for low power applications.

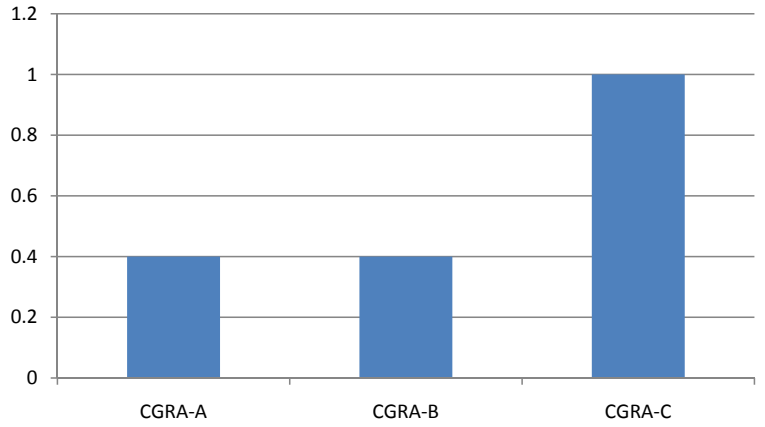

Figure 10: RDR comparison for three CGRA architectures

5.7. Productivity

To assess the efficiency of the RaCAMS approach, the number of lines of code (including comments) and the amount of programmer effort was measured. The results are shown in Table 5. Implementing RaCAMS for the purposes of modelling the three architectures in the case study took around 2 working weeks for design and around 5 working weeks for programming. Once the RaCAMS framework was available, modelling a Finite Impulse Response (FIR) filter took 2 more days and the Discrete Fourier Transform (DFT) took 3 more days.

The RaCAMS source code was found to be 44% more compact than an equivalent Verilog implementation. Configuration source code in RaCAMS and Verilog were found to be of similar size.

Table 5: Code development productivity for RaCAMS

Architecture	**Number of lines of Java code**				**Working weeks**
	Simulation Engine	**Application Mapping**	**Data Exchange Unit**	**Total**	
CGRA-A	354	234	165	753	3
CGRA-B	429	308	165	902	1
CGRA-C	354	339	165	858	1

5.8. Performance

A performance comparison was done to check the speed of execution of RaCAMS compared to a Verilog implementation of the CGRA. Both implementations were run on 2.5 GHz Intel Xeon quad-core processor. The RAM size was 3.2 GB and operating system was Linux, Fedora 13. RaCAMS was simulated and profiled using Netbeans IDE 6.9.1. The Verilog model was simulated and profiled using Synopsys VCS, version D-2010.06. The results are presented in Table 6. RaCAMS gives a performance improvement 2 to 13 times when compared to an equivalent implementation in Verilog. Compilation time for RaCAMS was approximately 1.2 seconds.

6. Conclusion and Future Work

This paper presented RaCAMS, a bit-true, cycle-accurate functional simulation framework for CGRA architectures. RaCAMS facilitates rapid de-

Table 6: Performance comparison of RaCAMS vs HDL model

Algorithm	**Type**	**CPU time (ms)**		**Performance**
		RaCAMS	**HDL**	**Increase factor**
FIR filter	5 taps	99	370	4
Matrix multiplication	4x4	513	1280	2
Wavelet	db2	246	390	2
Matrix determinant	3x3	306	510	2
DFT	8 point	213	2750	13
FFT	Radix 2 complex number	219	810	4

velopment of simulators for CGRA architectures. The framework allows fast exploration of the design space by means of run-time alteration of the design options. The framework integrates both application mapping and a Simulation Engine into a single software artifact, obviating the need for a separate parser. The framework is easily extensible, enabling modelling of a wide variety of CGRA architectures and applications. Java is used to improve programmer productivity and allow greater simulation flexibility by means of dynamic instantiation. A case study was provided which illustrates the use of RaCAMS for assessing architectural design options and in testing various application mappings. Productivity and performance metrics are provided which is show that RaCAMS is more compact and faster than an equivalent Verilog implementation. The authors plan to extend RaCAMS by providing a larger library of CFU operations and application mappings. In addition, an improved GUI will be developed. The authors plan to make the RaCAMS available to the research community.

Acknowledgments

This research was funded as a part of the Efficient Embedded Digital Signal Processing for Mobile Digital Health (EEDSP) cluster, 07/SRC/I1169, by Science Foundation Ireland (SFI).

References

[1] R. Hartenstein, "Coarse grain reconfigurable architecture (embedded tutorial)," in *Proceedings of the 2001 conference on Asia South Pacific design automation*. ACM New York, NY, USA, 2001, pp. 564–570.

[2] H. Parizi, A. Niktash, N. Bagherzadeh, and F. Kurdahi, "MorphoSys: A Coarse Grain Reconfigurable Architecture for Multimedia Applications," *Memory*, vol. 1024, p. 16.

[3] C. Fisher, K. Rennie, G. Xing, S. G. Berg, K. Bolding, J. H. Naegle, D. Parshall, D. Portnov, A. Sulejmanpasic, and C. Ebeling, "An emulator for exploring rapid configurable computing architectures," in *FPL '01: Proceedings of the 11th International Conference on Field-Programmable Logic and Applications*. London, UK: Springer-Verlag, 2001, pp. 17–26.

[4] G. Phipps, "Comparing observed bug and productivity rates for Java and C++," *Software: Practice and Experience*, vol. 29, no. 4, pp. 345–358, 1999.

[5] H. Singh, M. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves Filho, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 465–481, 2000.

[6] S. Khawam, I. Nousias, M. Milward, Y. Yi, M. Muir, and T. Arslan, "The reconfigurable instruction cell array," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 75–85, 2008.

[7] P. Heysters, G. Smit, and E. Molenkamp, "Montium-balancing between energy-efficiency, flexibility and performance," 2003.

[8] A. Chattopadhyay, X. Chen, H. Ishebabi, R. Leupers, G. Ascheid, and H. Meyr, "High-level modelling and exploration of coarse-grained reconfigurable architectures," in *DATE '08: Proceedings of the conference on Design, automation and test in Europe*. New York, NY, USA: ACM, 2008, pp. 1334–1339.

[9] V. Baumgarte, G. Ehlers, F. May, A. Nückel, M. Vorbach, and M. Weinhardt, "PACT XPP:A self-reconfigurable data processing architecture," *the Journal of Supercomputing*, vol. 26, no. 2, pp. 167–184, 2003.

[10] G. J. M. Smit and G. K. Rauwerda, "Reconfigurable architectures for adaptable mobile systems," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'05), Las Vegas, Nevada, USA*, T. P. Plaks, R. DeMara, M. Gokhale, S. Guccione, M. Platzner, G. J. M. Smit, and M. Wirthlin, Eds. CSREA Press, June 2005, pp. 17–25.

[11] G. J. M. Smit, P. M. Heysters, and E. Molenkamp, "The chameleon project in retrospective," in *5th PROGRESS Symposium on Embedded Systems, Nieuwegein, The Netherlands*, October 2004, pp. 181–184.

[12] M. Muir, I. Lindsay, T. Arslan, I. Nousias, S. Khawam, M. Milward, N. Aslam, and A. Major, "Extensible software emulator for reconfigurable instruction cell based processors," Sept. 2008, pp. 35–40.

[13] J. Filho, S. Masekowsky, T. Schweizer, and W. Rosenstiel, "CGADL: An Architecture Description Language for Coarse-Grained Reconfigurable Arrays," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 17, no. 9, pp. 1247–1259, Sept. 2009.

[14] B. Geelen, E. Brockmeyer, G. Lafruit, R. Lauwereins, and D. Verkest, "Exploration of system-level trade-offs for application mapping in multiprocessor system-on-chips," vol. 2, July 2005, pp. 222–225.

[15] H. Nguyen and L. John, "Exploiting SIMD parallelism in DSP and multimedia algorithms using the AltiVec technology," in *Proceedings of the 13th international conference on Supercomputing*. ACM New York, NY, USA, 1999, pp. 11–20.