Mapping basic recursive structures to runtime reconfigurable hardware

Hossam ElGindy and George Ferizis
School of Computer Science & Engineering
The University of New South Wales
Sydney, NSW, Australia
{hossam,gferizis}@cse.unsw.edu.au
FAX: +61 2 9385 5995

UNSW CSE-TR-0419

July 2, 2004

Abstract

Recursion is a powerful method that is used to describe many algorithms in computer science. Processing of recursion is traditionally done using a stack, which can act as a bottleneck for parallelising and pipelining different stages of recursion.

In this paper we propose a method for mapping recursive algorithms, without the use of a stack structure, into hardware by pipelining the stages of recursion. The use of runtime reconfigurable hardware to minimise the amount of required hardware resources, and the related issues to be resolved, are addressed.

1 Introduction

Recursion is a powerful tool that is heavily used for the development of programs today, that allows a programmer to compactly and easily design code that would be much more complex in design if it is done using iteration. Recursion makes it easier to develop programs and makes it inherently easier to debug due to much simpler code being present for reviewing. Recursive descriptions can be seen in many elegant algorithms such as tree traversals and "divide-and-conquer" geometrical and mathematical problems in multi-dimensions [3]. Iterative equivalents for such algorithms are nowhere near as elegant. Furthermore recursion is fundamental to functional language paradigms which rely on recursion due to an absence of common iterative operators.

On a general-purpose processor, recursive descriptions are implemented by the use of a stack that is used to temporarily store arguments and results between stages of a recursive function. This solution could be implemented easily on an FPGA-based system. However the ability of such systems to provide customised pipelining and parallelism, which software implementations on a general-purpose processor cannot provide, will not be possible with such an implementation.

A solution for this problem is not provided by the majority of high level language development tools such as Handel-C [1], which do not support recursive procedures. The exclusion of recursive constructs from such tools is a testimony to the difficulty of the process of mapping them into hardware.

Previous work into mapping recursive functions into FPGAs without the use of a stack has relied on transforming the function into a loop [**?**]. However it makes no attempt to parallelise the recursive calls that are made in instances where a function calls itself multiple times.

In this paper we present a method for mapping *"basic"* recursive functions into reconfigurable hardware that unrolls the recursion with the use of runtime reconfiguration, and hence does not use a stack.

Our approach builds on the previous work on unrolling iterative loops for the purpose of mapping them into runtime reconfigurable hardware [4] and addresses the additional difficulties that are unique to recursion. These difficulties include hardware allocation which does not have the constant growth rate as in iterative loops, and minimising the cost of runtime reconfiguration. In this paper we allude to the use of special-purpose logic to predict hardware requirements for the function being unrolled at the earliest time possible. More information can be found in the full technical report [**?**].

We begin by defining what is meant by *"basic"* recursive functions. A *"basic"* recursive function is a function that, calls itself zero or a constant number of times, at any depth of recursion all instances of the function call themselves the same number of times, and when given initial arguments the maximum depth of recursion can be accurately calculated, or can be calculated at any time during the processing of an input stream of arguments.

We begin by presenting the communication model that is used, followed by a description of our method for mapping this conceptual model of recursion into hardware solutions on FPGAs that minimises the effect of runtime reconfiguration delay. We finally present two case studies: merge sort [9] and Strassen's matrix multiplication algorithm [11]. The two algorithms rely on recursion and both of which have been implemented using this method.

2 General recursive problem

We shall now describe a model of recursive processes that illustrates our proposed solution. We model a basic recursive function as a tree. This is shown in figure 1, which has an example of a function that calls itself twice.

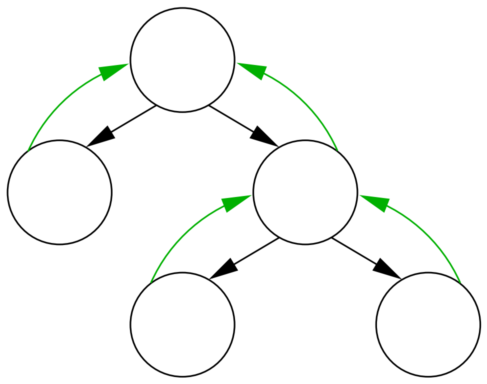


Figure 1: Tree model of a recursive function

Our implementation is based on pipelining the operation between consecutive levels with an area on the chip dedicated to processing the nodes contained in each level. To allocate the minimum amount of hardware the number of levels in the recursive tree must be estimated accurately at runtime. The necessary logic to meet this estimate can then be configured at runtime. This requires runtime reconfiguration. Therefore an effective implementation of this process requires the ability to hide the delay that is produced by runtime reconfiguration.

The example in figure 3(a) is not tail recursive, with values being returned back up the recursive tree. This communication pattern presents problems in pipelining if only a single area of logic is dedicated per level, as this area will be blocking while waiting for subsequent logic to return data. A solution to this is presented later in the paper.

We also point out that the recursion in figure 3(a) is balanced, whereas not all recursive trees are balanced. Unbalanced recursion presents problem in the scheduling of data being sent through the pipeline, as well as introduces an increased complexity in hardware allocation. This is due to the difficulty in predicting the node population of a level of recursion due to the irregular growth rate that is a result of unbalanced recursion. The problem presented by unbalanced recursion is not discussed in this paper and has been left for future work.

3 General solution

The tree in figure 3(a) models a balanced recursive function with two recursive calls. As indicated by the arrows directed from the children nodes back to their parent nodes, the recursive calls return values that are processed by the parent nodes.

Our approach is similar to previous approaches [2, 8] that transform general recursive calls into two tail recursive calls. Such transformations have been suggested in the past as software compiler level optimisations [2, 8]. These earlier techniques rely on transforming a recursive call into two tail recursive calls and using a stack to hold arguments generated in the first tail recursive call for the second tail recursive call. Such methodology does not attempt to make use of parallelism and pipelining opportunities that modern hardware offers. Our approach eliminates the need for a stack by taking advantage of multiple processing elements and networking that current reconfigurable hardware makes possible to implement and utilise on-demand.

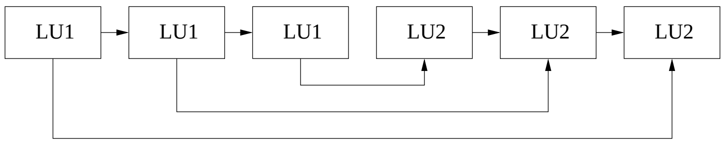


Figure 2: Logic unit allocation

We begin by looking at the recursive call modelled in the tree shown in figure 3(a).

The related problem of unrolling iterative loops and their incremental mapping into runtime reconfiguration has already been previously addressed [4]. The hardware produced by the incremental mapping of an iterative loop follows a linear function in the number of iterations, as the amount of computations per loop is constant. The unbalanced nature of recursion and the variable amount of computations per level raise few difficulties in the mapping if we want to maintain an interrupted throughput.

Our mapping begins by removing upward returns in the tree by adding another tree that shares the leaves with the original tree. The result is a DAG as shown in figure 3(b).

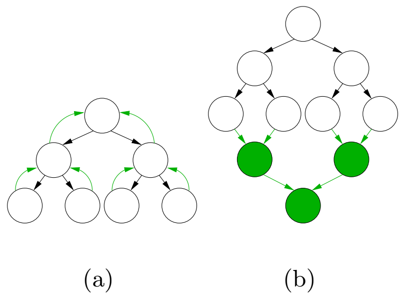


Figure 3: A recursive tree, and the corresponding DAG produced.

Construction of the new graph involves the splitting of the statements in a recursive function into two disjoint sets. The first set contains the statements that occur before the recursive calls. These statements correspond to the unshaded nodes in the graph. The second set contains the statements that occur after the recursive calls. These statements correspond to the shaded nodes in the graph. For each set we define a different logic type as follows:

1. *PreRecursion*: This corresponds to logic that can compute during the expansion portion up to and including the point of truncation. This is in effect the first tail recursive function.
2. *PostRecursion*: This corresponds to logic that can compute during the collapsing of the tree. This is in effect the second tail recursive function.

An instance of a *PreRecursion* unit and a *PostRecursion* unit, creates a complete instance of the original function. As can be seen in figure 3(b), the shaded nodes which correspond to units of type *PostRecursion* are created by mirroring the unshaded nodes. The nodes that are related in this mirroring are named twins. Thus a twin corresponds to a complete function instance.

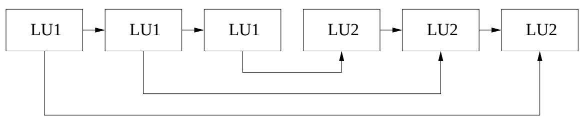


Figure 4: Logic unit allocation

All instances of each unit communicate between levels as shown in figure 3(b), with extra communication between twin units. They all take in the necessary arguments to compute the values they are to output.

One possible layout in hardware for this DAG is an array as shown in figure 4, with the communication between logic units set so that the concept of a twin can be seen. The logic to compute the results of a node may be replicated in proportion to the node population in that level of the graph to maintain a constant throughput. However there are instances where the communication between different levels of recursion incurs more cost than the actual computation. In this case throughput is bounded by the communication between logic units, and thus throughput remains constant irrespective of the amount of logic configured. Following this observation a minimum amount of logic is configured, which is the amount needed to compute the resulting computation of a single node.

As the depth of recursion increases, our mapping dedicates the minimum amount of logic needed as dictated by the maximum depth reached. This is achieved by the use of runtime reconfiguration to configure logic on demand. Rruntime reconfiguration is a task that requires time orders of magnitude longer than the time for performing computation. It will not be desirable to have the system stall and wait for more logic to be reconfigured when logic for a new recursive level is required.

To combat this problem, we implement a prediction mechanism that monitors input into the system and detects the need for more logic before it is actually needed. All input items pass through this prediction circuitry, before being placed into the compute logic. The collected information is used to hide as much of the performance penalty introduced by runtime reconfiguration as possible. The prediction logic will also be responsible for deciding how much hardware should be allocated when reconfiguring new logic, in respect to the node population at that depth of recursion.

When the prediction logic of the pipeline decided that more hardware is needed when a new item enters the system it has time related to the length of the currently configured pipeline to configure new logic. Prior configuration of a suitable pipeline depth will make it possible to hide this cost completely.

4 Hardware Model

Implementation of this methodology into a reconfigurable chip has not been created as we are still in early stages of development. However a simulator based on the reconfigurable multiple bus model [7] has been developed. Implementation and testing of the validity of the mapping process have been performed using this simulator. Logic utilisation and time performance results are not available yet. However our simulations show that the implemented algorithms are correct, that throughput is constant and that the need for run-time reconfiguration is predicted at the earliest time possible.

4.1 System layout

The hardware system being proposed for mapping recursive structures is an array of basic modules connected by a network using the reconfigurable multiple bus (RMB) model [7] with some modifications. Each basic module consists of a processing logic and a memory module. The RMB was chosen for its ability to have simultaneous communication between processors without any contention, providing there are sufficient physical bandwidth, in a self-administered manner. The bus system establishes connections using the bottommost bus, and "compacts" the connections towards the top bus. The bus system is synchronous with the time to travel from one controller to an adjacent one being a single clock cycle.

The bus is organised in a series of network controllers that have multiple buses running through them. Each controller that has a processing element or memory module attached to it. The system has the following properties:

- The buses are toroidal in design with communication flowing only in one direction.
- The buses are 4-bit wide, with the system word being 4-bits. This bus width was chosen to offset the area used in having multiple bus lines and to allow for smaller units of configuration.
- The buses are synchronous with the time to send a signal from one controller to an adjacent controller being a single cycle.
- Memory and processors are placed regularly with processors and memory modules organised in pairs named "groups". This is illustrated in Figure 5.

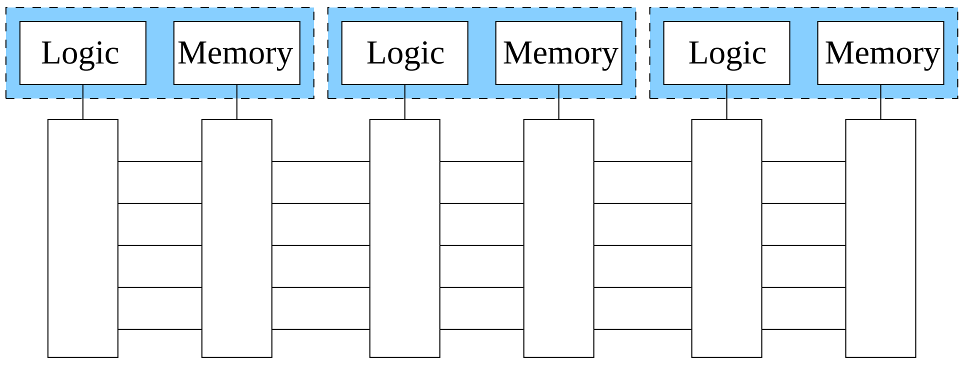


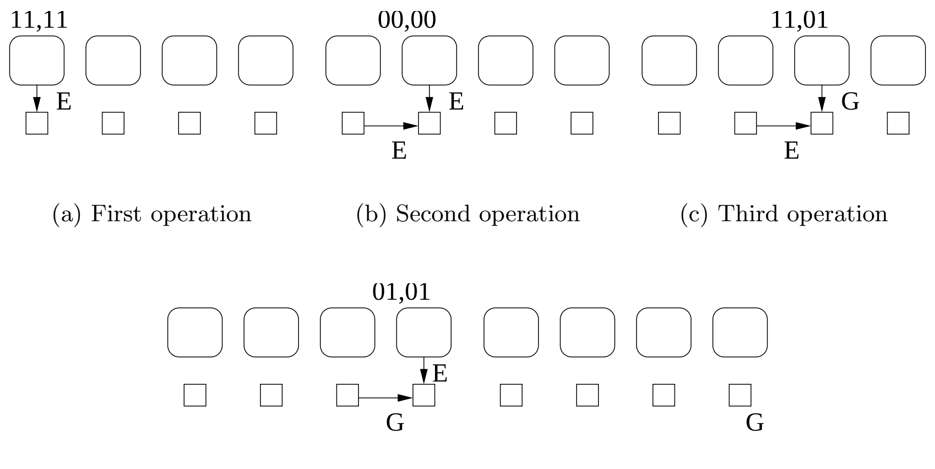
Figure 5: Logic unit and memory layout on the bus

The top most bus in each controller is a special purpose communication bus. We associate a small LUT that process the packets transmitted on the top bus before sending them to the next controller. The LUT accepts input from the group connected to the controller and from the bus and outputs the result back to the group and onto the bus. This bus can be used to perform carry-type operations without requiring the use of processor modules. The processor module in the group associated with each controller is used to program the LUT at run-time.

The availability of multiple bus lines allows for parallel communication between different groups of nodes without contention.

4.2 Pipelining of instructions

The special-purpose top bus of the RMB with embedded LUTs is typically used to simulate operations on words greater than 4 bits without the need to access the processor modules. The LUT associated with each controller is programmed appropriately for the operation in progress. In an addition operation the least significant bits are processed first and the carry bit is propagated to the controller with higher significant bits, while in a comparison operation the bits are processed in the reverse order. The example in Figure 6 shows an 8-bit greater-than comparison is made between two numbers 11001101 and 11001001 using only 4 processors instead of the 7 needed if a tree was to be built. The smaller squares represent the LUTs and the larger squares represent processors. G is the greater than result value, E the equal result value.



(a) First operation (b) Second operation (c) Third operation

(d) Fourth operation (e) Final result

Figure 6: Pipelining of a greater than instruction.

In the second stage of pipelined comparison, shown in Figure 6b., the first processor is now available for comparing the next two data values, there by allowing for pipelining of instructions. Pipelining instructions can be used for streaming data through the same instruction, and we use in the merge sort algorithm where most processors only perform comparison and write operations.

The correct operation of a pipelined instruction is achieved by sending fragments of a complete word into different controllers at different times. We use the knowledge of the distance between controllers, and therefore the time to travel between them, to ensure that processors send data into the bus at the same time that the previous controller sends data in. This can be seen in Figure 6.

This model was chosen for its high level of connectivity. The RMB is a simple network without complex routing protocols. The system also provides parallelism that makes it suitable for hiding the latency of run-time reconfiguration of logic units.

5 Merge Sort Algorithm

Merge sort is a well known recursive algorithm. Starting with a set of inputs the algorithm reduces the set into smaller problem sets recursively until a set of suitable size is reached which it then sorts. After this the resulting sorted sets are merged together recursively until a single sorted set is obtained.

Conceptually the recursion moves in two directions. Starting at a single point it recursively divides the set into smaller partitions until a truncation point is reached. These sets are then merged together. It is observed that while dividing the sets no actual operations are done on the values contained in these sets. With this observation it is possible to remove the top half of the recursive graph that divides the sets and input values into each of the the nodes that are left.

Thus for an input set that contains 8 values we have a recursive tree as shown in figure 7.

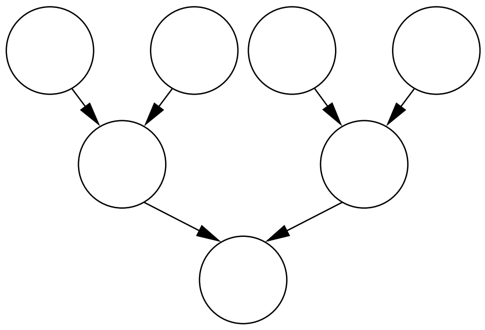


Figure 7: Operation of the merge sort algorithm

Similar to Orenstein et al [9] we allocate a single logic unit per level of this tree. This is shown in figure 8, with the node that the logic unit controlling that level is currently working on being shaded. As can be seen each logic unit is not working at the beginning but begins working as values filter towards it.

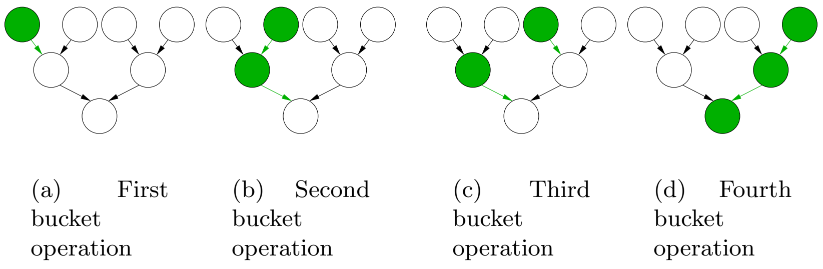

(a) First bucket operation (b) Second bucket operation (c) Third bucket operation (d) Fourth bucket operation

Figure 8: Merge sort operation over time

For this mapping we will assume the size of the set at the truncation point is 1 and hence there is no sort operation done at all.

5.1 Mapping

The structure of this problem is such that we only need logic units of type 2. Each logic unit will buffer a certain number of values n before beginning to merge what has been buffered with the next n values that it receives.

The first logic unit will buffer 1 value and then merge this with the next value. The next logic unit will buffer 2 values and so on. Logic unit k in the array will buffer 2^k values.

5.2 Hardware Allocation

The recursive tree is largest at the top most level of the tree and then collapses into a single node. As shown in figure 8, none of these nodes will be operating in parallel. Furthermore it can be shown that no nodes in the tree will ever be operating in parallel. Thus no extra hardware per level is required to maintain constant throughput. For this mapping one logic unit will be allocated per level of the graph.

5.3 Prediction Mechanism

If the total size of the input stream is known it is trivial to find the depth of recursion for the entire operation, however for this implementation we assume that the size of the input stream is not known. Therefore at a given point in the input stream we require a way to predict a change in the depth of recursion to enable an area to be reconfigured.

We will assign the following variables:

$D(t)$: The depth of the recursion at time t

$N(t)$: The number of elements inputted into the system at time t

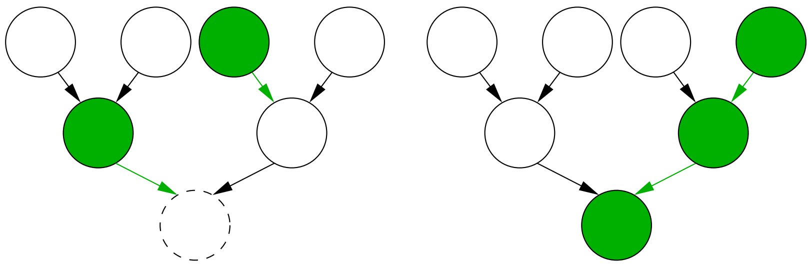
Looking at the structure of the graph and therefore nature of the recursive call we can see that the following equation is true:

$$D(t) = \lceil \log_2 N(t) \rceil$$

By calculating the value of $D(t)$ at the input of every value, and then comparing it with the previous value of $D(t)$ the need for reconfiguration can be detected.

A value will enter the system that will cause a need for reconfiguration just as the previously last configured logic unit begins outputting results. This is shown in figure 9(a) at a time t. These outputted values pass into a logic unit that buffers them until it knows these are the final result values, or is reconfigured to operate the merge sort algorithm.

As shown in figure 9(b), this area will not be needed until the second stream to be merged arrives at this unconfigured area of logic. This will occur at a time that is no less than $\frac{3t}{2}$. By configuring a high enough number of logic units such that the initial value of $\frac{t}{2}$ is less than reconfiguration time the cost of reconfiguration can be completely hidden.



(a) System at time t. The dashed node corresponds to unconfigured logic.

(b) System at time t_2 where $t_2 = \frac{3t}{2}$

Figure 9: Stages of system operation during execution.

5.4 Implementation Layout

During regular system operation, we have the logic units arranged in an array as shown in figure 10. The shaded node is the logic unit controlling reconfiguration. It should be noted that unlike figure 8 there is only a single logic unit type on the array. This is due to the fact that the graph does not expand and then collapse but only collapses as shown in figure 8.

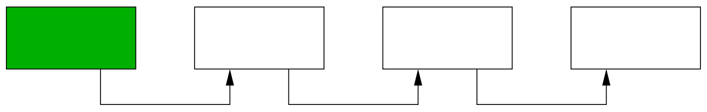


Figure 10: System during operation

When the need for reconfiguration arises the reconfiguration unit begins reconfiguring a new logic unit to compute a further depth of the tree. This is shown in figure 11. As seen in this diagram, the reconfiguration unit(the shaded node) reconfigures the next logic unit(shown at the end of the array). The communication network that the RMB provides allows for this to occur easily.

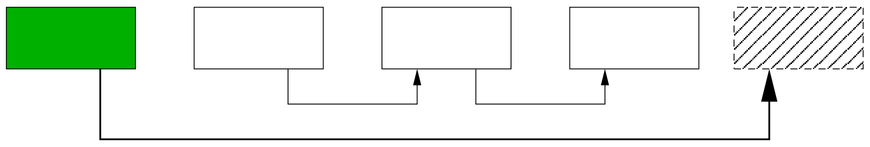


Figure 11: System during reconfiguration

6 Strassen's Matrix Multiplication Algorithm

Strassen's Matrix Multiplication algorithm [11] is a recursive matrix multiplication algorithm that reduces the complexity of matrix multiplication to $O(N^{log_2(7)})$ from $O(N^3)$ which is the complexity of conventional matrix multiplication algorithms.

The algorithm begins by reducing the larger original multiplication recursively into 7 smaller matrix multiplications until a matrices of a certain cut off size are reached. From here the matrices are multiplied and then arithmetic operations are done on the results of these multiplications until a single result matrix is obtained. It is a simple divide and conquer algorithm.

Here are the arithmetic operations that are done on the matrices, with the following matrix multiplication with a, b, c, d, e, f, g and h being quadrants in the initial matrices to be multiplied.

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix} \tag{1}$$

It can be observed from this that the following equations are true:

$$\begin{aligned} r &= ae + bf \\ s &= ag + bh \\ t &= ce + df \\ u &= cg + dh \end{aligned} \tag{2}$$

From these equations the following has been derived.

$$\begin{aligned} r &= P_5 + P_4 - P_2 + P_6 \\ s &= P_1 + P_2 \\ t &= P_3 + P_4 \\ u &= P_5 + P_1 - P_3 - P_7 \end{aligned} \tag{3}$$

where

$$\begin{aligned} P_1 &= A_1 \cdot B_1 \\ P_2 &= A_2 \cdot B_2 \\ P_3 &= A_3 \cdot B_3 \\ P_4 &= A_4 \cdot B_4 \\ P_5 &= A_5 \cdot B_5 \\ P_6 &= A_6 \cdot B_6 \\ P_7 &= A_7 \cdot B_7 \end{aligned} \tag{4}$$

where

$$\begin{array}{ll} A_1 = a & B_1 = g - h \\ A_2 = a + b & B_2 = h \\ A_3 = c + d & B_3 = e \\ A_4 = d & B_4 = f - e \\ A_5 = a + d & B_5 = e + h \\ A_6 = b - d & B_6 = f + h \\ A_7 = a - c & B_7 = e + g \end{array} \tag{5}$$

We observe the following features of the algorithm in respect to it's recursive properties:

- The recursive depth is known at the beginning of execution as the size of the matrices must be known
- It is not tail recursive (as operations occur while splitting into smaller multiplications and in combining the results)

By knowing the depth of recursion at the beginning of execution we can dismiss the need for any prediction mechanisms for runtime reconfiguration. The correct amount of area will be configured as the execution begins and once it has been configured it need not be reconfigured.

The tail recursion is dealt with by the application of two separate logic types as described earlier. One logic type LT_1 will control the dividing of the matrices (equations (5)) and the actual multiplication at the truncation point, and the other logic type LT_2 will control the merging of results from these multiplications(equations (3)).

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{pmatrix}$$

Figure 12: Example 4x4 matrix

$$\begin{pmatrix} 0 & 2 & 8 & 10 & 1 & 3 & 9 & 11 & 4 & 6 & 12 & 14 & 5 & 7 & 13 & 15 \end{pmatrix}$$

Figure 13: Array of values reordered

6.1 Stream Reordering

Examining the algorithm in the expansion phase of the recursion, we observe that the values needed for operations are not in continuous areas of the stream if the matrix is inputted in row major or column major order. This creates large numbers of values that need to be buffered. As can be observed in a $N \times N$ matrix A, the first set of operations during the expansion phase of Strassen's algorithm would involve arithmetic on the values $A_{(0,0)}$, $A_{(0,\frac{N}{2})}$, $A_{(\frac{N}{2},0)}$ and $A_{(\frac{N}{2},\frac{N}{2})}$. The positions of these values in the stream of size N^2 for a row-major ordering would be 0, $\frac{N}{2}$, $\frac{N^2}{2}$ and $\frac{N^2+N}{2}$. Operating on these values would clearly involve the buffering of over half the stream. As N increases, the detrimental performance impact of this would grow rapidly. We propose a new ordering scheme to overcome this problem.

As can be seen Strassen's algorithm breaks a matrix into 4 sub-matrices which then are operated on to produce 7 matrices. The process is repeated with these sub-matrices until the truncation point of the recursion is reached.

Given 4 sub matrices A, B, C, D, all the operations between the matrices occur on elements in the same position in each matrix, ie. any operation will involve only $(A_{i,j}, B_{i,j}, C_{i,j}, D_{i,j}) \forall (i, j)$. Our reordering scheme attempts to place these 4 values as close to each other as possible, while still maintaining the matrices that will be finally multiplied in contiguous memory blocks.

This reordering is achieved by taking a standard level-ordering such as Ahnentafel indexing and reversing the order in which the numbers per level are put together.

As can be seen the higher order portion of the final index produced is based on the lowest depth of the tree, not the highest depth as is usually the case. An example of this for the matrix in figure 12, is given in figure 13.

By placing the values in this order it is possible to execute the matrix multiplication in a streaming manner, and with enough processors to execute the entire procedure with only a small and constant amount of buffering.

Examination of the reordered array shows that $A_{(0,0)}$, $B_{(0,0)}$, $C_{(0,0)}$ and $D_{(0,0)}$, are the first 4 values in the array. They are then followed by $A_{(0,1)}$, $B_{(0,1)}$, $C_{(0,1)}$ and $D_{(0,1)}$.

The first 4 values will produce the values for $A_{1:(0,0)}$, $A_{2:(0,0)}$ and so on until $A_{7:(0,0)}$ with the appropriate arithmetic equations. The next 4 will create $A_{1:(0,1)}$, $A_{2:(0,1)}$ and so on until $A_{7:(0,1)}$.

When the next stage of recursion attempts to compute the values for the sub matrices produced by $A1$, it will find they are not in contiguous blocks of memory as was the case in the first recursion, but are separated by a distance of 7 values.

This distance increases with the number of recursion and conforms to the geometric sequence of 7^i where i is the depth of recursion, starting at 0.

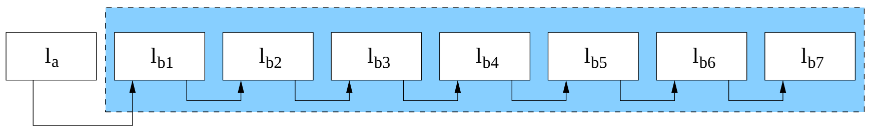


Figure 14: Farm expansion and layout

6.2 Hardware Allocation

This distance presents another problem. A logic unit will be required to buffer a significant number of values at a greater depth of recursion, but once it reaches a certain point it is capable of outputting 7 result values for each value that is inputted. This shows that the bottleneck in the computation shifts from the communication system to the processing hardware during computation. This distance problem is removed by the allocation of more than one processor per level.

As discussed earlier, the distance between related values increases 7 fold between each level of recursion. A logic unit l_a will be sending values to a single unit l_b normally. However if it sent values to 7 different units

$l_{b1}, \cdots, l_{b7}$ using a round robin type scheduling for the destination of each value outputted, this distance problem would be removed. This results in a layout as shown in figure 14. The processor l_a sends all values to the processor l_{b1}. This value is placed onto the LUT communication row that was described earlier until the end of these 7 processors, with each one taking only the value it is scheduled to take. The arrangement of logic units is named a farm, and a logic unit sending to the farm is named a farm expansion. The value being passed through the LUT row never leaves the farm. A similar system is used for collapsing the tree, but instead the size of a farm is 4 units.

The amount of buffering and therefore the throughput can be tuned by adjusting the frequency of farm expansions throughout the system. We will now examine the effects of different frequencies. With a farm expansion at every level starting at a level k, the amount of buffering can be kept constant throughout the operation. It should be observed that the minimum number of values that will be buffered is 8, 1 value for each quadrant of each matrix A and B.

We will begin by introducing a variable D, which is the maximum depth of recursion.

6.2.1 No farm expansions

With one processor per level we observe the following:

Number of processors used:

$$2D$$

Maximum amount of buffering needed:

$$8 \times 7^{D-1}$$

6.2.2 A farm expansion at every opportunity

Number of processors used:

$$\sum_{0 \leq i < D} \left(7^i + 4^i\right) = \frac{7^{D-1} - 1}{6} + \frac{4^{D-1} - 1}{3}$$

Maximum amount of buffering needed:

$$8$$

6.2.3 A farm expansion at every opportunity after a level k

Number of processors used:

$$\sum_{0\le i<D-k} (7^i + 4^i) = \frac{7^{D-k-1}-1}{6} + \frac{4^{D-k-1}-1}{3}$$

Maximum amount of buffering needed:

$$8 \times 7^k$$

6.2.4 A farm expansion at every j^{th} level

Number of processors used:

$$j \times \sum_{0\le i<\frac{D}{j}} (7^i + 4^i) = j \times \frac{7^{\frac{D}{j}}-1}{6} + \frac{4^{\frac{D}{j}}-1}{3}$$

Maximum amount of buffering needed:

$$8 \times 7^{D-\frac{D}{j}}$$

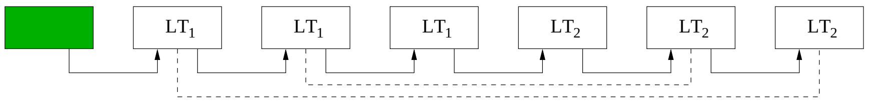


Figure 15: Strassen array layout

6.3 Implementation Layout

As with the merge operation, we have the logic units arranged in an array as shown in figure 15.

The two different logic units should be noted, as well as the dashed line connecting them. This is not a connection on the network, but is drawn to show which logic unit is coupled with the logic unit of the other type (ie. working on the same level of recursion).

7 Conclusion

A method for mapping basic recursive structures to run-time reconfigurable hardware has been demonstrated. Two case studies have been presented to show the methods validity and correctness, based on an ability to predict the need for run-time reconfiguration well before it is required. Whereas only basic recursive structures have been mapped to reconfigurable hardware it is believed that similar techniques can be used to map more complex and general recursive structures. The cases of unbalanced recursion and recursive problems where prediction may not be optimal, as in the case of merge-sort, as topics for future research.

References

[1] Handle-C language reference manual. *http://www.celoxica.com/.*

[2] J. Arsac and Y. Kodrato. Some techniques for recursion removal from recursive functions. pages 295–322, 1982.

[3] J. L. Bentley and M. I. Shamos. Divide-and-conquer in multidimensional space. In *Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 220–230. ACM Press, 1976.

[4] K. Bondalapati and V. K. Prasanna. Loop pipelining and optimization for run time reconfiguration. *Reconfigurable Architectures Workshop*, May 2000.

[5] H. ElGindy and G. Ferizis. On hiding latency in reconfigurable systems: the case of merge-sort fo r an fpga-based system. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 242–242. ACM Press, 2003.

[6] H. ElGindy and G. Ferizis. On Improving the Memory Access Patterns During The Execution of Stra ssen's Matrix Multiplication Algorithm. In *Twenty-Seventh Australasian Computer Science Conference*, volume 27, 2004.

[7] H. ElGindy, A. K. Somani, H. Schroder, H. Schmeck, and A. Spray. RMB - A Reconfigurable Multiple Bus Network. *Proceedings of the IEEE Symposium on High-Performance Computer Architecure*, pages 108–117, 1996.

[8] Y. A. Liu and S. D. Stoller. From Recursion to Iteration: What are the Optimizations? In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 73–82, 2000.

[9] J. Orenstein, T. Merret, and L. Devroye. Linear sorting with O(log N) processors. *BIT*, 23:170–180, 1983.

[10] P. Rondogiannis and W. W. Wadge. First-order functional languages and intensional logic. *Journal of Functional Programming*, 7(1):73–101, 1997.

[11] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.