# Mapping Loops on Coarse-Grain Reconfigurable Architectures Using Memory Operation Sharing

Jong-eun Lee
jelee@poppy.snu.ac.kr

Kiyoung Choi
kchoi@azalea.snu.ac.kr

Nikil Dutt
dutt@cecs.uci.edu

Architectures and Compilers for Embedded Systems (ACES)
Center for Embedded Computer Systems, University of California, Irvine, CA 92697

Technical Report #02-34
Center for Embedded Computer Systems
University of California, Irvine, CA 92697, USA

September 2002

## Abstract

*Recently many coarse-grain reconfigurable architectures have emerged as programmable co-processors, considerably relieving the burden of the main processors in many multimedia applications. While their very high degree of parallelism enables high performance in compute-intensive loops, their shared memory interface between several processing elements often becomes a bottleneck in many multimedia and DSP applications. In this report we present a technique that ameliorates this memory bottleneck through the sharing of different loop iteration executions using a novel organization of the loop pipeline. We develop the conditions for sharing memory operations on a generic reconfigurable architecture template and propose a heuristic method to generate the pipelines accordingly within a general mapping flow. Experimental results using our technique on a typical coarse-grain reconfigurable architecture show improvement of up to 3 times.*

## Contents

## List of Figures



## List of Tables

## Abstract

*Recently many coarse-grain reconfigurable architectures have emerged as programmable coprocessors, considerably relieving the burden of the main processors in many multimedia applications. While their very high degree of parallelism enables high performance in compute-intensive loops, their shared memory interface between several processing elements often becomes a bottleneck in many multimedia and DSP applications. In this report we present a technique that ameliorates this memory bottleneck through the sharing of different loop iteration executions using a novel organization of the loop pipeline. We develop the conditions for sharing memory operations on a generic reconfigurable architecture template and propose a heuristic method to generate the pipelines accordingly within a general mapping flow. Experimental results using our technique on a typical coarse-grain reconfigurable architecture show improvement of up to 3 times.*

## 1 Introduction

Recently many coarse-grain reconfigurable architectures, typically used as coprocessors for compute-intensive loops, have shown their capability in boosting the performance of the general purpose processors [1, 2, 3, 4, 5]. Their abundant parallelism, high computational density and also flexibility in terms of changing the behavior during run-time all contribute to making them better alternatives to traditionally used DSPs (Digital Signal Processors) or ASICs (Application-Specific Integrated Circuits), especially in the platforms for complex SOC's (Systems-On-a-Chip).

Typically coarse-grain reconfigurable architectures have identical processing elements (PEs), each of which contains functional units (e.g., ALU, multiplier) and a small number of storage units (e.g., register file, small local memory), even though there is a wide variance in the number and functionality of components as well as the interconnections between them. Those PEs are connected through programmable interconnects that support rich communication between neighboring PEs, typically to form a 2D array. To achieve high performance in many data-dominated applications, they often have specialized memory interfaces (e.g., *Frame Buffer* in MorphoSys [1] and *global control unit* in REMARC [2]). One of the common features of coarse-grain reconfigurable architectures is that the memory interface is shared by a group of PEs, namely a row or a column, for it would be too costly if every PE had its own memory interface or load/store units. Consequently, one critical performance bottleneck is the sharing of the memory interface by memory operations within loops mapped to the reconfigurable architecture; thus optimizing memory operations will have a great impact on the quality of mapping and the resulting performance.

Our technique presented in this report exploits the opportunity of the memory interface being shared by memory operations appearing in different iterations of a loop. Particularly, if a data array is used in a loop, it is often the case that successive iterations of the loop refer to the overlapping segments of the array, so that parts of data being read in each iteration have already been read in previous iterations. This redundant memory access can be reduced if the iterations are executed in a pipelined fashion [6], by organizing the pipeline in such a way that the related pipe stages share the memory operation and save the memory interface resource. Since this redundancy can be seen only when the multiple iterations as well as the pipelining execution are taken into account,

this kind of *memory operation sharing* is not addressed by conventional loop optimization techniques. We develop the conditions for sharing memory operations using a generic reconfigurable architecture template. We also propose a heuristic method to generate the pipelines with memory operation sharing within a general mapping flow. Experimental results using our technique on a typical coarse-grain reconfigurable architecture show performance improvements of up to 3 times in throughput, for loops showing inter-iteration data reuse patterns.

This report is organized as follows. In Section 2 we outline some of the related work and in Section 3 we introduce a generic reconfigurable architecture template, which is based on the common features found in many published coarse-grain reconfigurable architectures. The mapping flow based on the high performance pipelining technique is illustrated in Section 4. In Section 5 we present our memory operation sharing technique, which can effectively reduce the memory operations, increasing the resource usage efficiency. We demonstrate the effectiveness of the techniques through the experiments in Section 6 and conclude the report in Section 7.

## 2 Related Work

There is little work published on mapping techniques for coarse-grain reconfigurable architectures. Although there is extensive research and even commercial tools for FPGA (Field-Programmable Gate Array) and fine-grain reconfigurable architectures, the techniques developed for them are not directly applicable to coarse-grain reconfigurable architectures (a PE contains at least an ALU), because of the substantial differences in PEs and interconnection architectures among others. Reconfigurable architectures of high granularity (e.g., MorphoSys [1], MATRIX [7], REMARC [2]) are reported to have only assembly-level programming environments [3] and are typically lacking a compiler environment.

As a granularity-neutral technique for reconfigurable architecures, Bondalapati *et al.* [6] described algorithmic techniques for mapping loops onto reconfigurable hardware. They aim to minimize the run-time reconfiguration when the resources (e.g., PEs) in the architecture are less than what is needed to pipeline all the computations in the loop. They have similar concerns in that they try to find a better pipeline organization during mapping applications onto the architecture. However, the only architectural feature they consider is the reconfiguration resource. We consider many architectural features such as the memory interface, the interconnections between the PEs, etc. as well as the fast reconfiguration cache.

As another technique for reconfigurable architectures of no specific granularity, Bondalapati [8] proposed *data context switch* technique to maximize the throughput of IIR (Infinite Impulse Response) type algorithms. The technique can hide the delay from the loop-carried dependency when there is an outer loop with no loop-carried dependency, by utilizing the independent data sets of the outer loop. Even though this technique was applied to a high-granularity reconfigurable architecture, Chameleon [9], as well as an FPGA, it requires an outer loop with no loop-carried dependency to find independent data sets. Moreover, it assumes that each PE has access to an ample local memory to store the data context when the outer loop is executing other iterations, which is not valid for many coarse-grain reconfigurable architectures.

Huang and Malik [10] proposed a design methodology for their own dynamically reconfigurable

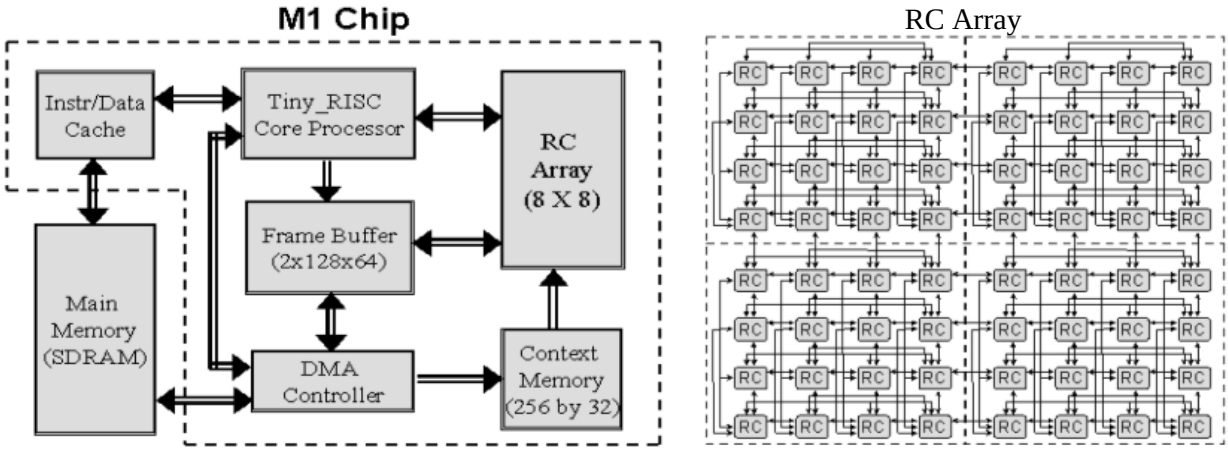



**Figure 1. MorphoSys architecture [1].**

datapath architecture, which is used as an accelerating co-processor. Even though this work targets coarse-grain reconfigurable hardware, it is significantly different from our approach. The main difference is that their reconfigurable datapath is designed for a specific application and reconfiguration is used only to switch between the loops for which the reconfigurable datapath is designed. In constrast, we employ a generic architecture template that is representative of a large class of coarse-grain reconfigurable architectures; thus our architecture is designed independently from applications and can be used for many different applications by reconfiguration.

# 3 Generic Reconfigurable Architecture Template

We first review two typical examples of coarse-grain reconfigurable architecture: MorphoSys and REMARC. Then, we present our generic reconfigurable architecture template, which is being developed for the purpose of application mapping (compilation), simulation, and design space exploration, among others.

## 3.1 Review of the MorphoSys architecture

MorphoSys [1] is comprised of a core RISC processor, a reconfigurable cell array (RC Array), and a high-bandwidth memory interface as shown in Figure 1. The RC Array consists of an $8 \times 8$ array of identical PEs (also called RCs). The architecture of MorphoSys can be described at two levels: intra-cell and inter-cell levels.

At intra-cell level, each RC is similar to a very simple microprocessor except that an instruction is replaced with a *context word*[1] and there is no instruction decoder or program counter. The datapath of an RC is centered around an ALU-multiplier and a shifter connected in series. The output of the shifter is temporarily stored in an output register and then goes back to the ALU-multiplier, to a register file (of size 4), or to other cells via buses and inter-cell connections. Finally

[1]Each RC is the basic unit of reconfiguration [1] and configuration data for each RC is called *context word* and stored in the *context register*.

for the inputs of the ALU-multiplier are there muxes, which select the input from several possible sources such as the register file, neighboring cells, or the memory buses. The bitwidth of the functional or storage units is at least 16 bits except the multiplier, which supports multiplication of $16 \times 12$ bits. The function of an RC is instructed by a context word, which defines the opcode and an optional constant (for functional units) and the control signals (for the muxes and the register file).

At inter-cell level, there are two major components: the interconnection network and the memory interface. The interconnection exists only between the cells of either the same row or the same column. Since the interconnection network is symmetrical and every row (column) has an identical interconnection with other rows (columns), it is enough to define only interconnections between the cells of one row. Now for a row, there are two kinds of connections. (1) One is dedicated interconnection between two cells of the row. This is defined between neighboring cells and between cells of every 4-cell group (so that every cell in that group can have an interconnection with any other). (2) The other kind of connection is called *express lane* and provides a direct path from any one of each group to any one in the other group.

MorphoSys' memory interface consists of Frame Buffer (with a controller) and memory buses. Memory buses connect between Frame Buffer and the RCs with the width of $8 \times 2 \times 8$ bits such that for each of the eight rows there are two 8-bit buses shared by all the cells of the row. Frame Buffer is located between the RC Array and the main memory, providing the RC Array with the necessary bandwidth of 128 bits per cycle. To support such a high bandwidth, the MorphoSys architecture has (1) DMA (Dynamic Memory Access) to the main memory and (2) the *two-set structure* overlapping the data transfer with computation [1].

Context Memory, where the configuration is stored, has 32 *context planes*, with a context plane being a set of context words to program the entire RC Array for one cycle. The dynamic reloading of any of the 32 context planes incurs effectively zero run-time overhead [1] in MorphoSys, as the Context Memory can be updated concurrently with the RC Array execution. However, MorphoSys supports only row-parallel or column-parallel SIMD (Single Instruction-stream Multiple Data-stream) operations. Therefore, only eight context words are needed for a context plane and each context word is broadcast to a row or column of RCs.

### 3.2 Review of the REMARC architecture

REMARC (Reconfigurable Multimedia Array Coprocessor) [2] consists of a *global control unit* and an $8 \times 8$ array of *nano processors* as shown in Figure 2.

A PE or a nano processor is similar to a simple microprocessor, but here the configuration of a PE is called *instruction*,[2] which is essentially the same as the context word in MorphoSys. The datapath of a nano processor consists of an ALU, a 16-entry data RAM, an 8-entry register file, and special purpose registers such as DIR (Data Input Registers) and DOR (Data Output Register). All the functional or storage units are 16 bit wide. Compared to MorphoSys' RCs, nano processors

[2]What is called instruction in REMARC can be seen as configuration because there is no instruction decoder and the connections between the PEs are also defined by the "instructions". Moreover, there is nothing else corresponding to configuration in REMARC.

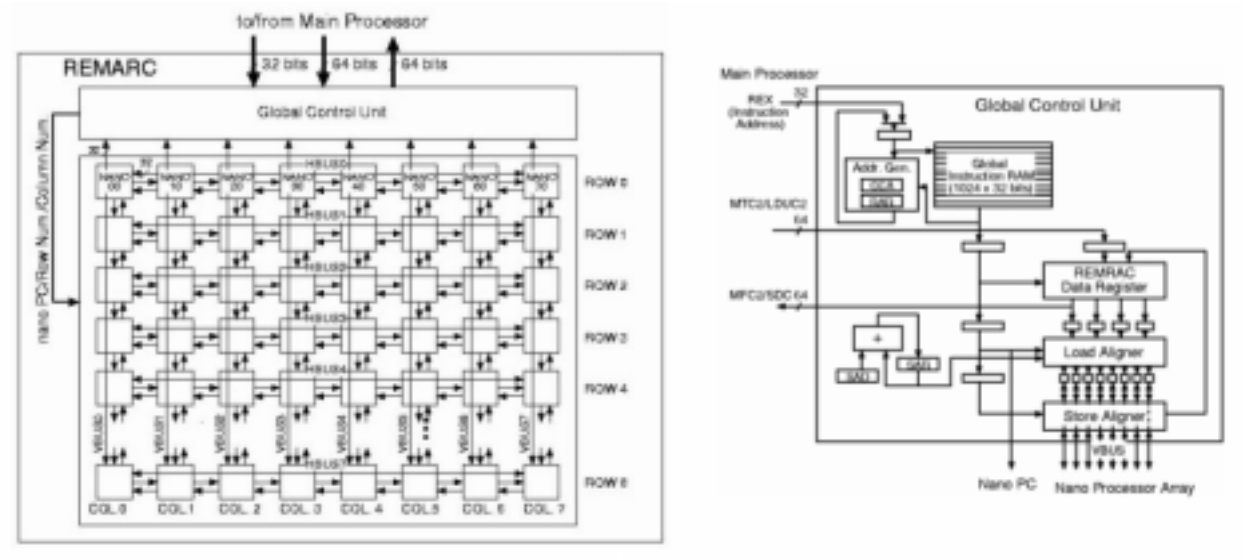



**Figure 2. REMARC architecture [2].**

have more storage units but do not have multipliers.

The interconnection network of REMARC is simpler than that of MorphoSys. REMARC has only nearest neighbor network additionally with a global bus for every row and column. The global buses, which are called HBUS (in row direction) and VBUS (in column direction), are 32 bit wide. However, VBUS'es can also be used as memory buses while both VBUS and HBUS can be used for broadcasting configuration in SIMD mode.

The memory access in REMARC is controlled by the global control unit, which has a 32-entry 64-bit data register file and load/store *aligners* as well as a global instruction RAM and *nano PC*-related circuitry. The global control unit transfers data between the data register file and nano processors using VBUS'es. But since the data register is only 64-bit-wide,[3] the global control unit uses the load/store aligners to arrange the data in multiple registers to transfer to/from VBUS'es.

The configuration for each nano processor is stored in the 32-entry instruction RAM, so that it can easily switch to any of the 32 configurations indicated by the nano PC (generated by the global control unit). SIMD mode configurations (using the VBUS or HBUS as broadcast channels) are also supported to utilize the instruction RAMs more effectively.

### 3.3 Generic reconfigurable architecture template

We now describe a generic reconfigurable architecture template, which is based on the common features found in many coarse-grain reconfigurable architectures including MorphoSys and REMARC. This generic architecture template, named *DRAA* (Dynamically Reconfigurable ALU Array), is developed as a model for compilation (application mapping), simulation as well as the design space exploration of such architectures.

[3]This seems to be so that it can easily interface with the main processor via already-established 64-bit buses

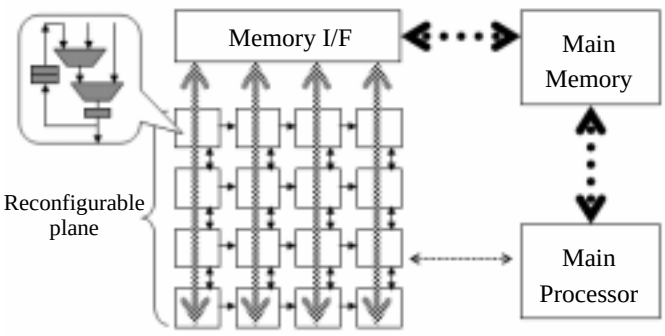



**Figure 3. DRAA: Generic reconfigurable architecture template.**

The DRAA consists of identical PEs placed in a 2D array, regular interconnections between them, and a high-speed memory interface as illustrated in Figure 3. The 2D array of PEs connected via interconnections is called the *reconfigurable plane*. A direct data transfer path between the main processor and the reconfigurable plane may be provided to facilitate quick transfer of a few variables such as parameters or computation results (e.g., scalar product). The DRAA can be defined at three levels: PE microarchitecture, line architecture, and reconfigurable plane architecture. Table 1 lists the parameters for an example DRAA architecture.

### 3.3.1 PE microarchitecture

We can describe the PE microarchitecture in similar ways as microprocessor architectures are described in ADLs (Architecture Description Languages) such as EXPRESSION [11]. For instance, the datapath can be described as a netlist of the components comprising the PE with relevant attributes such as supported opcodes, timing, etc. Alternatively, a set of supported functionalities as corresponding to an instruction set may be used as a PE microarchitecture description.[4] We assume each PE's latency remains constant regardless of its function, configuration, or even data, since variability in PE's latency would significantly diminish the architecture's regularity.

### 3.3.2 Line architecture

We assume that there are only row and column interconnections in the reconfigurable plane. For instance, interconnections in the diagonal direction are excluded. Also, all rows (columns) are assumed to have the same interconnections as the other rows (columns). Therefore, only the interconnections of a row and a column need to be defined. These 1D interconnections may be described as a list of dedicated connection pairs, the number of global buses, and if any, specialized interconnections.

In addition to the row/column interconnections, the line architecture includes memory access resource, which is a part of the memory interface. Similar to the interconnections, the memory access resources are assumed to be equally distributed along the rows or columns, which are referred to as *lines*. For example, REMARC has a 32 bit wide bus for each column, which can be used for

[4] In this work, the latter method was used to implement a PE-level mapping tool.

**Table 1. An example reconfigurable architecture**

| Category | Parameter | Value |
|---|---|---|
| PE micro-architecture | ALU | 1 |
| | MUL | 1 |
| | Reg files | 1 (4x16bit) |
| | Netlist (data transfer path) | {...} |
| | Functionality set | {...} |
| | Latency | 1 |
| Row (=line) interconnect | # of PEs in a line | 8 |
| | # of lines | 8 |
| | Pairwise interconnections | {...} |
| | Bus | 2 (16bit/each) |
| Memory access resource per line (=row) | Bus (shared w/ row global buses) | 2 (16bit/each) |
| | Latency | 1 |
| | Buffer depth per bus | 256 |
| Configuration | # of configurations | 8 |
| | # of configurations in SIMD mode | 64 |
| | Fast configuration reloading overhead | 0 cycle |
| Misc. | Direct communication channel w/ main processor | none |

transferring four 8-bit or two 16-bit data. In this case, a line is a column and the memory access resource per line is a 32 bit wide bus.

### 3.3.3 Reconfigurable plane architecture

Reconfiguration-related parameters are among the most critical ones in the plane level architecture description. In many architectures where distributed configuration memory (cache) is supported for fast run-time configuration switching, parameters such as configuration cache size and dynamic reloading overhead can make a big difference in the performance. Compilers also need to be aware of the architectural features to avoid the huge penalty when, for example, the generated configurations do not fit in the cache size.

## 4 Pipelined Mapping of Loops

Programming frameworks for coarse-grain reconfigurable architectures involve many subtasks. For instance, kernel[5] extraction and selection, kernel mapping onto the architecture, configuration memory management [12], and data memory management can all have much impact on the performance of the implementaion. Also if it is used as a soft core in an SOC, the customization of the reconfigurable architecture itself in such areas as the reconfigurable plane dimension, intercon-

[5]Here kernels mean highly repetitive loops of the application to be implemented on the DRAA.

nection scheme, the PE microarchitecture, the bitwidth, etc. might be necessary to optimize the performance as well as to reduce the power consumption.

In this section we describe a general flow of mapping loops onto the DRAA architectures defined in Section 3. First the loop pipelining in the context of reconfigurable architectures is introduced, then the mapping flow based upon the pipelining is described.

### 4.1 Loop pipelining

When mapping loops onto reconfigurable architectures, one common way to achieve high performance is to build a pipeline for the operations comprising the loop body (*spatial mapping*). Let's assume there is no loop-carried dependency or control flow within the loop. Then if the pipeline is built with $N_p$ stages and every pipe stage takes at most $n$ cycles, the execution time of one iteration or the latency is $Latency = N_p \cdot n$ cycles and *initiation interval*, the interval by which successive iterations start execution, is $n$. Therefore, the total execution time of $N$ iterations is

$$Exec\_time = Latency + (N-1) \cdot n. \tag{1}$$

This pipelined execution has the following advantages:

- It parallelizes $N_p$ iterations using the abundant resources available on the reconfigurable architecture.
- Since each stage has simple, fixed functionality, it does not need reconfiguration during execution.

On the other hand, its disadvantages are

- spreading all the operations of the loop body on the limited reconfigurable architecture may require too many resources,
- data dependencies between the operations should be taken care of by allocating interconnect resources to them, and
- it may require delay registers[6] to make every stage to have the same latency in number of cycles.

In summary, pipelined execution is efficient in terms of performance once the pipeline is constructed and filled, but because operations and their dependencies have to mapped with the limited resources, it may be difficult to build the pipelines.

Another way of mapping independent loops efficiently would be mapping every iteration to one PE and having it perform necessary operations by changing the configurations (*temporal mapping*). While this method has some advantages such as that direct application of conventional programming/compilation techniques is possible, no need to worry about interconnection, and that even

[6]Delay registers do not increase the throughput but can increase the latency. More importantly, implementing a delay register may use a whole PE in DRAAs.

the best performance may be obtained for some cases, it does not perform well as the loop becomes complex. It is because each PE's storage resources such as registers and quickly-reloadable configurations will be too limited for long sequences of operations in longer loops.

*Temporal mapping* (changing the configuration), however, does relieve the difficulty of spatial mapping (constructing the pipeline). It can also be used in combination with spatial mapping in such cases as more lines are needed than are avaiable. Also, because of the equivalence of using multiple lines to using one line with multiple configurations from the performance point of view (assuming zero configuration switch overhead), we initially consider using as many lines as needed (each line with a fixed configuration). Then, in a later phase each line can be time-multiplexed with multiple configurations.

Throughput is decreased if temporal mapping is used in such cases as more lines are needed than are available. On the other hand, throughput may be increased if more resources are available than the pipeline requires and the loop iterations can be split into independent groups. Simply replicating the pipeline on the unused lines of the plane and having the pipelines execute independent iterations in parallel will do.

Consider the example shown in Figure 4. If the mapping algorithm has generated a pipeline of upto two rows (=lines) $\times$ seven columns and the architecture has a $4\times4$ array of PEs, the array can have two pipelines in parallel while overflowing columns will drop the performance due to two configurations used. If the iterations are independent, the total execution time will be $Latency + (\lceil N/2 \rceil - 1) \cdot 2$, assuming the latency of each PE is one and configuration switch overhead is zero. Generally, assuming each PE has single cycle latency, the total execution time can be written as[7]

$$Exec\_time = Latency + (\lceil N/n_G \rceil - 1) \cdot S_{cs}, \tag{2}$$

where $n_G$ is the number of independent iterations running in parallel and $S_{cs}$ is the slowdown factor due to the configuration switch. $S_{cs}$ is 1 if a single configuration is used, or otherwise $N_{conf} * (1 + C_{sw})$, with $N_{conf}$ and $C_{sw}$ being the number of configurations and dynamic reloading overhead, respectively.

[7]Note that configurations need to be switched only when the pipeline actually needs more than one physical plane.

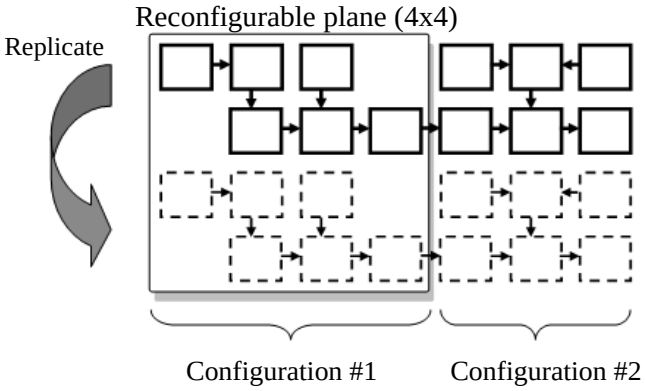



**Figure 4. A pipeline bigger than plane's physical dimension.**

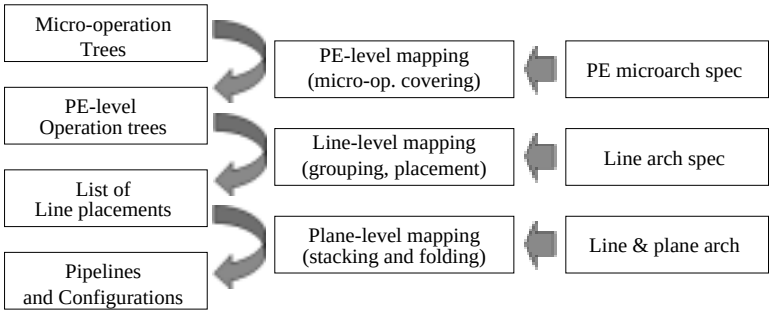


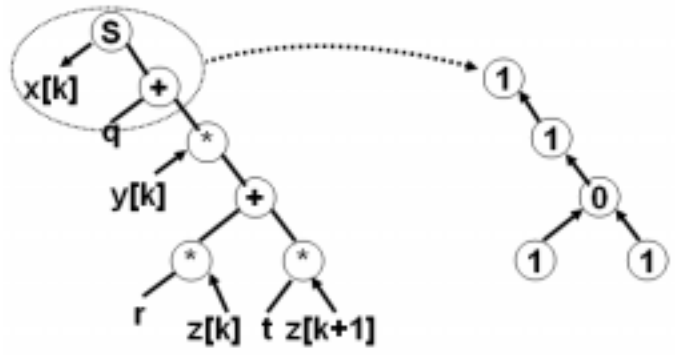

**Figure 5. Three major steps in the mapping flow.**

### 4.2 Mapping flow

We now illustrate a flow for the mapping of applications represented as loops to a typical class of DRAA architectures. To achieve maximal throughput, our approach generates high performance pipelines that are executed on the DRAA architecture. The mapping flow, shown in Figure 5, has three steps: micro-operation covering (PE-level), operation grouping and placement (line-level), and stacking of line placements followed by folding with time-multiplexing (plane-level). We explain these steps using the example loop in Figure 6.

```
for ( k=0; k<100; k++ )
   x[k]=q+y[k]*(r*z[k]+t*z[k+1]);
```

**Figure 6. An example loop.**



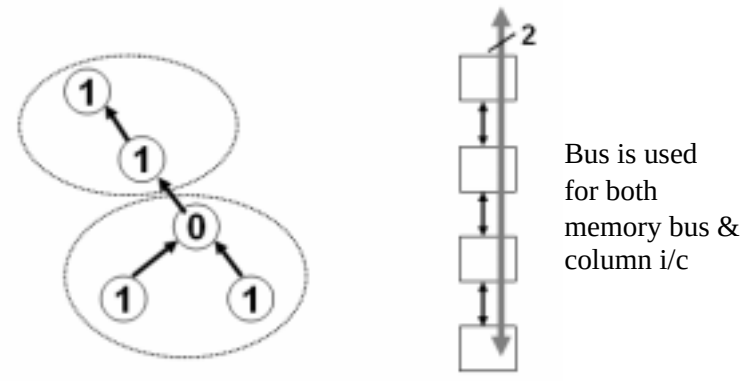

(a) Expression tree (b) PE-level operation tree

**Figure 7. PE-level mapping.**

#### 4.2.1 PE-level mapping

In the PE-level mapping, the loop body is represented as expression trees of micro-operations. The micro-operation trees are covered with patterns that can be implemented with a single configuration

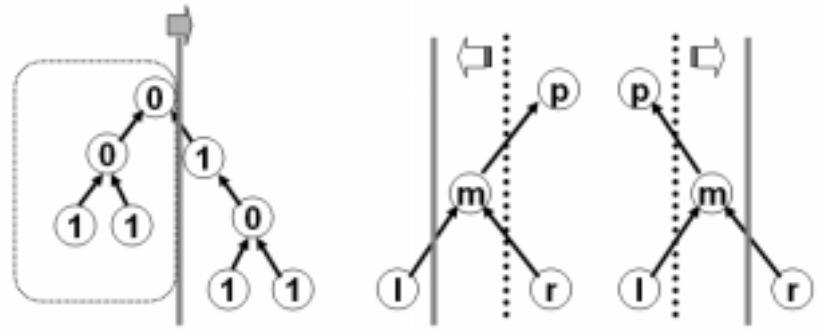
of a PE, producing *PE-level operation* trees.[8] A PE-level operation is an abstraction for a pattern of micro-operations that can be implemented with one configuration of a PE. If a series of ADD and STORE with no more than two memory operations can be implemented with one configuration, for example, the first two operations in Figure 7 (a) can become one node in (b). The numbers in Figure 7 (b) represents the number of memory operations contained in the node, which is necessary information for the next step.




(a) Grouping PE-level operations

(b) Line architecture

**Figure 8. Line-level mapping.**




(a) Sliding cut

(b) Local change of cuts

**Figure 9. Sliding cut algorithm.**

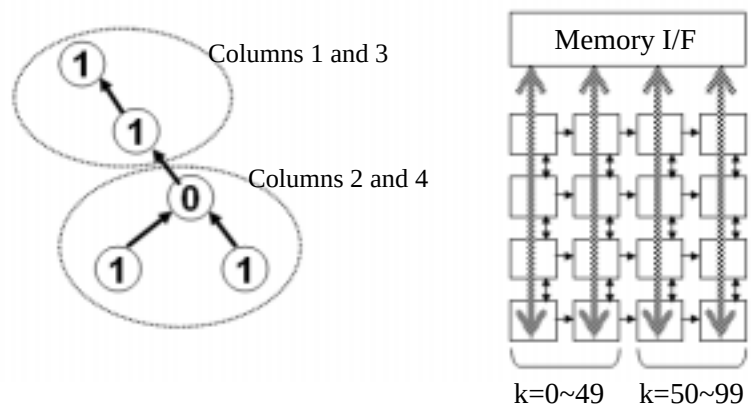
### 4.2.2 Line-level mapping

Following the PE-level mapping is the line-level mapping, where the PE-level operation nodes are grouped and placed on each line. Let's suppose the line architecture is given as in Figure 8 (b).

[8]Though the behavior of PE-level operations may be complex, the resulting graph of PE-level operations is constrained to be a tree if the microarchitecture of the PE has only one output port.

That is, each column of PEs shares a memory bus (which is capable of transferring two operands at each cycle) and an interconnection exists between neighboring PEs. Then, two conditions should be met for the nodes to be placed together. First, they should have no more than two memory operations in total, since the capacity of the memory bus is two. Second, it should be possible to allocate an interconnection resource for each edge (data dependency) between the nodes. One possible grouping is shown in Figure 8 (a).

In performing the node grouping, one of the major concerns is to ensure the line placements can be stacked together satisfying the data dependency between the line groups with existing interconnections, while preferably finding large groups for area efficiency. We use a heuristic algorithm called *sliding cut* algorithm[9] developed for these architectures: the PE-level operation has at most two children (PEs have at most two input ports) and only nearest neighbor connections and global buses are supported for row/column interconnections. The algorithm tries to group nodes in the order found as a 'sliding cut' sweeps the tree horizontally from left to right (Figure 9 (a)). Groups generated by the sliding cut[10] are tested for line placement until the group contains more operations than the architecture supports for one line. Then, the last (also largest) group that passed the test is selected as a line placement. In this algorithm, groups that will be placed near each other are likely to have more edges between them, which matches well with the interconnection architecture. To connect data-dependent PEs in different groups, the PEs should be on the same row (assuming here line is column), which is accommodated by expanding the line placements. Expanding line placement can be realized easily by inserting dummy nodes (functioning as delay registers), which only increases the latency but not decrease the throughput.




(a) Line placements (b) Plane architecture

**Figure 10. Plane-level mapping.**

[9]This algorithm does not guarantee that the interconnections can always be allocated for all inter-group edges; if that happens it resorts to the plane-level mapping, where multiple configurations are generated to resolve the resource shortage.

[10]To have better cuts, it is checked if there are such cases as in Figure 9 (b). Since *m* can have no memory operation (all its input/output are being used), it is possible and clearly advantageous to ex(in)clude node *m* from the group.

### 4.2.3 Plane-level mapping

In the plane-level mapping, the line placements generated in the previous step are stitched together on the 2D plane of PEs. In this example the line placements can be put together nicely, resulting in a pipeline within a bounding box of 3 rows $\times$ 2 columns. If the physical plane size is 4 $\times$ 4, the pipeline can be replicated in the other two unused columns as shown in Figure 10, resulting in about 2 times in throughput. Note that in this example the generated mapping achieves the maximum throughput on the DRAA architecture even though only 10 out of the 16 PEs are used, since all the memory I/O resources are used at every cycle. (This illustrates the need to pay critical attention to the memory I/O resource.)

There are two factors of potential performance decrease in the plane-level mapping stage. First, in stacking the line placements there may be the case where edges (representing data dependency) cannot be allocated interconnections such as buses or dedicated connections due to the scarcity of the resource. Second, the pipeline obtained from stacking the line placements may have a bigger dimension than the physically available plane size. In those cases, multiple configurations are used with time-multiplexing (dynamic reconfiguration). If the number of configurations to be multiplexed is greater than the number of "quickly reloadable configurations" supported by the architecture, the overhead will be severe. In this case other levels of techniques (e.g., loop fission [13]) should be used together or the loop would be decided to be implemented on the main processsor.

## 5 Memory Operation Sharing

We now motivate the need for memory operation sharing with an example and present a methodology to achieve it within the mapping flow described. We also propose a heuristic placement technique to automate the memory operation sharing, for a special class of input trees.

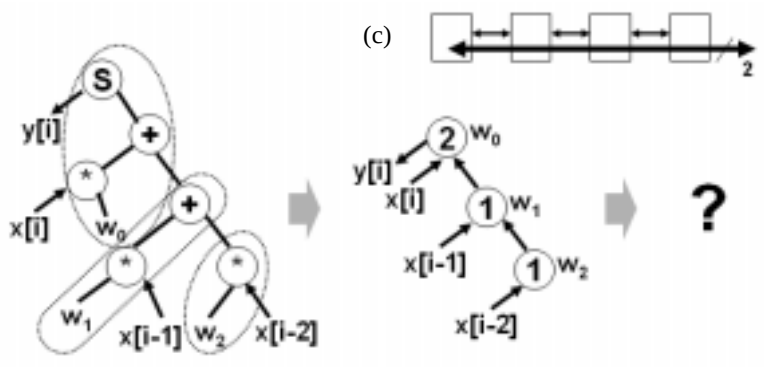
### 5.1 Motivational example

Since the loops implemented on DRAAs are often memory operation-bounded as seen in the previous example, reducing the effective number of memory operations will have a great effect on the performance. One such opportunity comes from the data reuse pattern in loops of DSP algorithms.

Let's take an example of an FIR (Finite Impulse Response) filter algorithm defined as

$$y[i] = \sum_{j=0}^{2} w_j \cdot x[i-j], \tag{3}$$

where $w_i$'s are constants and $i = 0, 1, 2, \cdots$. Figure 11 shows (a) the micro-operation tree representation of the input algorithm, (b) a PE-level operation tree after PE-level mapping, and (c) the line architecture of the target DRAA, which supports at most two memory operations per row at each cycle. In PE-level mapping, a PE is assumed to support a series of MULTIPLY and ADD operations with one configuration if one of the MULTIPLY's operands is constant.




(a) Covering expression tree (b) PE-level op. tree

**Figure 11. FIR example.**

Since there are four memory operations in the PE-level operation tree, at least two rows are needed to implement the tree with a single configuration. The optimal mapping in this example, however, is to use only one row as follows. Figure 12 depicts a pipeline implementation using one row. The three nodes are placed on three PEs in a row and the edges (data dependency) between them are implemented with existing interconnections between the PEs. One of the two memory buses is used by the write operation of $w_0$ node and the other is shared by all the three nodes.

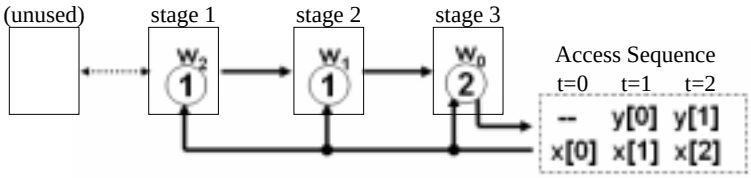



**Figure 12. An optimal placement for the FIR example.**

The reason why they can share one memory bus is that the three PEs are different stages of a pipeline. When $w_0$ node is processing the iteration $i = 0$, for instance, $w_1$ node is $i = 1$ and $w_2$ node is $i = 2$. And since those nodes access consecutive data in an array, $x[i]$, $x[i-1]$, and $x[i-2]$, respectively, what they are actually accessing is the same data, namely $x[0]$. Thus, the three memory operations are indeed the same and the three PEs can share a memory bus or one memory operation, using half the resource or generating twice the performance in this example.

## 5.2 Memory operation sharing

As seen in the example, the nodes[11] in the PE-level operation tree may share a memory operation by placing (*aligning*) them on the same line. Here, we develop the conditions for the nodes to share the memory operations and address how to handle the conditions within the mapping flow.

[11]Hereafter, we use 'nodes' to mean the PE-level operations in the PE-level operation tree.

### 5.2.1 Conditions for memory operation sharing

The memory operation sharing is a technique that exploits the redundancy of memory operations over different iterations of a loop. We assume the loop is already optimized, so that common subexpressions (e.g., memory operations reading the same address in the same iteration) are already eliminated. But even with an optimized loop, there may be memory operations that read the same address in different iterations. We distinguish those memory operations that read the same address in iterations differing by a constant number. We call those memory operations (and also the nodes containing[12] them) *alignable*.

Alignable operations can easily be detected from their memory access indexes. Suppose that the loop iterator is `i` and it is incremented by `c` at every iteration. Then two memory read accesses `A[a*i+s]` and `A[a*i+t]` are alignable if the difference `s–t` is divided by `a*c`. They will access the same address in iterations differing by `(s–t)/(a*c)`, which will be called *iteration difference* of the two memory operations. More than two operations are alignable if they are pairwise-alignable.

Now, to make the alignable operations actually access the same address at every cycle, the nodes containing them have to be placed in the right stages of the pipeline. So the second condition for memory operation sharing is that the pipe stage difference of two alignable nodes should equal the iteration difference of their memory operations, while the first condition is that the alignable nodes should be placed on the same line.

### 5.2.2 Handling the conditions in the mapping flow

The conditions for memory operation sharing provide useful guidelines for a better placement of the PE-level operation tree. If a 2D placement technique were used for the mapping of the tree onto the reconfigurable plane, it might be possible simply to use those conditions as additional constraints of the placement. But because our mapping flow utilizes 1D placement (line placement) instead of computationally very expensive 2D placement, it needs a special care to handle the conditions in the mapping flow.

Since the alignable nodes can be found anywhere in the tree, simply placing the alignable nodes first and then doing the placement for the rest of the tree may bring about many problems in the plane-level combining process. Moreover, the alignable nodes may not be connected directly as in the FIR example. Rather, they may have only common ancestors up in the tree. In this case, enforcing the second condition means placing all the nodes from the alignable ones up to their first common ancestor. Then, the remaining tree (unplaced portion of the tree) may have a more irregular interface with the placed portion, further complicating the combining process later.

One reasonable way of satisfying the conditions while not complicating the rest of the problem is to identify a subtree including all the alignable nodes[13] and do placement for it separately. More precisely, the root of the subtree is defined as the first common ancestor of all the alignable nodes,

[12] PE-level operations may contain other operations including another memory operation as long as the PE microarchitecture permits.

[13] Note that there may be multiple sets of alignable nodes. In this case, a subtree is constructed for each set of alignable nodes.

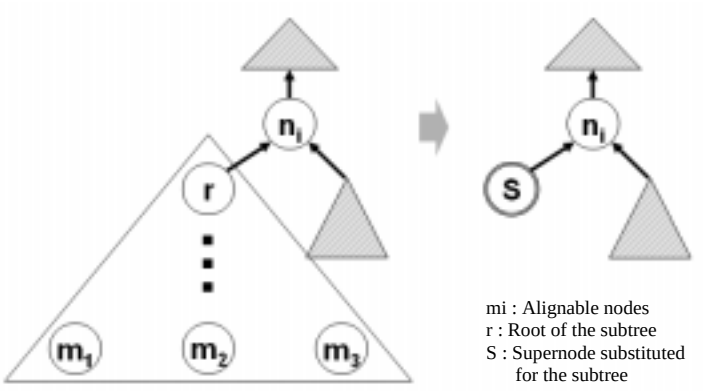



**Figure 13. Identifying and replacing a subtree.**

and the subtree includes the root and all the nodes below the root (Figure 13). If the placement of this subtree can be done separately[14] generating a group of line placements, a supernode is substituted for the subtree and this new tree is fed into the line-level mapping process. Finally, the plane-level mapping process stacks the line placements from both the subtree and the new tree.

### 5.3 Placement heuristic

Even though it is difficult to do 2D placement for a general subtree satisfying the conditions for memory operation sharing, we can devise a placement heuristic for regularly structured subtrees found in many DSP and multimedia algorithms.

This heuristic assumes two architectural features. First, the PE microarchitecture allows the PE-level operations to have at most two inputs. This assumption guarantees that the PE-level operation tree as well as the subtree is a binary tree. Second, the line architecture supports at least nearest neighbor interconnections. Both of them hold true in many architectures including MorphoSys and REMARC.

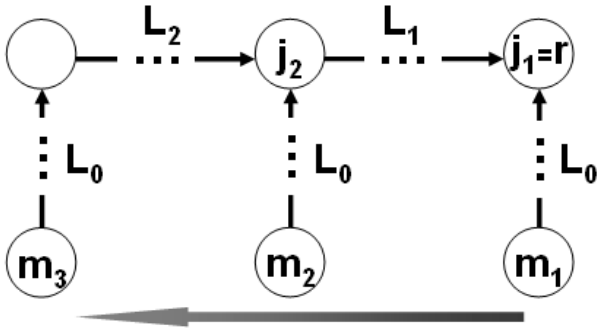



**Figure 14. Placement heuristic for memory operation sharing.**

[14]We assume manual placement for general subtrees.

By regularly structured trees, we mean the trees that can be structured into something like Figure 14. Suppose the subtree has the root $r$ and $n$ alignable nodes. Alignable nodes contain memory read operations of the form of `A[a*i+s]`, which are different only in `s`. Let's call this `s` *index offset* of the node.

Let $m_i(i = 1, \cdots, n)$ be the alignable nodes and assume that $m_i$ has greater index offset than $m_{i+1}$, as shown in Figure 14. We want to map the subtree directly on the reconfigurable plane, placing the alignable nodes on the same line to satisfy the first condition for memory operation sharing. Let the path from each $m_i$ to $r$ be denoted by $P_i$ and the *join node* where $P_i$ and $P_{i+1}$ meet first be denoted by $j_i$. Since the join nodes are also placed on the same line as shown in the Figure 14, the distance ($L_0$) between $j_i$ and $m_i$ is the same for all $i = 1, \cdots, n-1$. To satisfy this, we insert dummy nodes as needed. Now, to satisfy the second condition for memory operation sharing, we try to make the pipe stage differences be the same as the iteration differences. It is done by inserting dummy nodes between $j_i$ and $j_{i+1}$ such that the distance ($L_i$) is equal to the iteration difference of $m_i$ and $m_{i+1}$. If the distance is already larger than the iteration difference, then the heuristic gives up finding placement for this subtree. In practical examples, giving up well-structured subtrees doesn't happen very often. Typically, the join nodes come from a series of ADD operations, so that the distance between neighboring join nodes is initially 1.

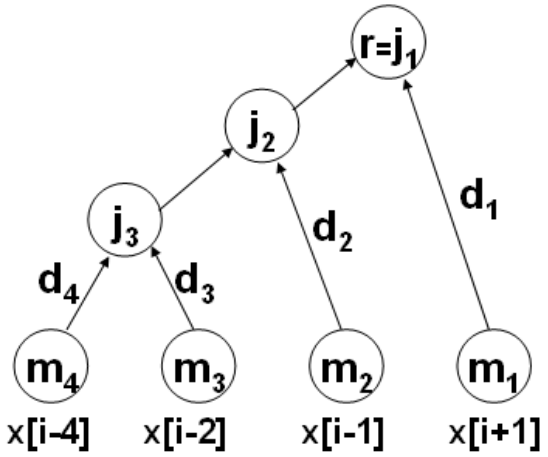



**Figure 15. Four paths joining at three join nodes.**

Figure 15 shows a general form of regularly structured trees, which are found in many digital signal processing applications. There may be nodes between $m_i$ and $j_i$ or between $j_i$ and $j_{i+1}$. But, no nodes other than join nodes are allowed to have more than one child.[15] Once a subtree is found to have this form, it is straight forward, as explained above, to transform it into what looks like Figure 14. To find whether a subtree has this form, we check the following properties.

- All the nodes in the subtree are covered by the set of paths $P_i, (i = 1, \cdots, n-1)$; in other words, every node is on at least one of the paths.

[15]Only the alignable nodes may be the leaves of the subtree. But, an aligable node may not be a leaf, as in the FIR example of Figure 11 (b).

**Table 2. The loops used in the experiments**

| Loop | Description (LL means Livermore Loops) | #mem. op. | #repetition |
|---|---|---|---|
| hydro | Hydrodynamic excerpt from LL | 4 | 40 |
| ICCG | Incomplete Cholesky-conjugate gradient from LL | 6 | 40 |
| banded | Banded linear equations (unrolled) from LL | 14 | 3 |
| state | equations of state from LL | 10 | 12 |
| ADI | Alternating direction, implicit integration (innermost, part) from LL | 11 | 7 |
| diff | First difference from LL | 3 | 98 |
| wavelet | A wavelet filter implementation (innermost) | 5 | 24 |
| ME | Motion estimation kernel (unrolled) from MPEG encoder | 128 | 30 |

- The set of join nodes in $P_i$ is a subset of the set of join nodes in $P_{i+1}$.[16]

# 6 Experiments

For our experiments, we used the example architecture described in Table 1. For the line interconnections, nearest neighbor connections and global buses are used, where global buses may also be used as memory buses. For the PE microarchitecture, a PE supports arithmetic operations as well as a series of MULTIPLY and ADD operations with one configuration if one of the MULTIPLY's operands is constant.

Table 2 shows the eight loops used in the experiments. Six loops are from the Livermore Loops benchmark suite and the other two are from a wavelet filter and the motion estimation kernel of an MPEG encoder. In the experiments, only the loops exhibiting inter-iteration data reuse patterns are used. The degree of the data reuse varies; some (hydro, ICCG, diff) have only 2 alignable memory operations in a loop while others (banded, state) have three to five sets of alignable operations with two to three operations per set. ME has 8 sets of 16 alignalbe operations.

For each of the benchmarks, we selected an appropriate loop level. Since the current mapping flow can handle only one level of loop at a time, when the benchmark has nested loops, either the inner loops were unrolled or only the inner-most loop was used. Then, the loops were fed into the mapping flow, with and without the memory operation sharing optimization.

Table 3 compares the mapping results in the following terms: (a) the number of lines used for one instance of the loop pipeline, (b) the number of configurations, (c) the latency of the pipeline, (d) the throughput of the entire reconfigurable plane, which is equal to the number of pipelines on the plane divided by the number of configurations, and (e) the total number of cycles for the whole iterations, which depends on the repetition count of the loop. For the comparison of the two cases — without and with memory operation sharing — the rightmost columns show the reduction of the latency and the ratio of the throughputs.

[16] Assuming `a*c` in Section 5.2.1 is positive.

**Table 3. Comparison of mapping results**

| Loop | Without Memory Operation Sharing | | | | | With Memory Operation Sharing | | | | | T.Cyc | Thrpt |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | #Ln | #Cnf | Lat | Thrpt | T.Cyc | #Ln | #Cnf | Lat | Thrpt | T.Cyc | Rdc(%) | ratio |
| hydro | 2 | 1 | 4 | 4 | 13 | 2 | 1 | 5 | 4 | 14 | -7 | **1** |
| ICCG | 3 | 1 | 3 | 2 | 22 | 3 | 1 | 3 | 2 | 22 | 0 | **1** |
| banded | 7 | 2 | 11 | 0.5 | 15 | 10(8) | 2(1) | 13(11) | 0.5(1) | 17(13) | -13(13) | **1(2)** |
| state | 5 | 1 | 9 | 1 | 20 | 4 | 1 | 11 | 2 | 16 | 20 | **2** |
| ADI | 6 | 1 | 7 | 1 | 13 | 5 | 1 | 7 | 1 | 13 | 0 | **1** |
| diff | 2 | 1 | 3 | 4 | 27 | 1 | 1 | 3 | 8 | 15 | 44 | **2** |
| wavelet | 3 | 1 | 4 | 2 | 15 | 2 | 1 | 4 | 4 | 9 | 40 | **2** |
| ME | 65 | 18 | 66 | 0.056 | 588[a] | 23 | 6 | 36 | 0.167 | 210 | 63 | **3** |

[a]The actual number of cycles may be larger in this case because this mapping uses more configurations than the configuration cache can hold.

In the results, the throughput is determined by *the number of configurations* and *the number of pipelines*, the latter of which is not shown in the table but can easily be calculated by dividing the architecture's number of lines (=8) by the number of lines used. First, the number of configurations increases due to the two factors: the need for more PE resources (i.e., more than 8 rows or columns are used) and the need for more interconnection resources (i.e., when plane-level mapping fails to connect line placements with a single configuration). Second, the number of pipelines is reversely proportional to the number of lines used.

The number of lines used was decreased in many loops by using the memory operation sharing technique, although there were cases where reducing the number of memory operations didn't make any difference in the use of lines (hydro) or the input tree didn't allow the heuristic technique to find a placement for it (ICCG).

Table 3 shows that in general the pipelines generated using the memory operation sharing technique tend to have slightly longer latency[17] but take less number of lines and often increase the throughput considerably, up to 3 times in the ME example. While the loop banded was mapped with fewer lines when memory operation sharing was not used, it was neutralized by multiple configurations, which were used because the line placements couldn't be connected within a single configuration.

The numbers in parentheses (in the loop banded) are when the technique was applied selectively on a subtree basis. Not all the subtrees for which the heuristic could find the placement contributed for throughput; some subtrees placed by the heuristic used more lines than would be by the mapping flow algorithm, typically when $L_0$ (in Figure 14) is large and the number of memory operations in the subtree is small. By selectively applying the technique for each subtree, a better result could be obtained.

Table 3 shows that the memory operation sharing technique also helps reducing the number of configurations.[18] It is because the memory operation sharing technique makes the input tree

[17]This is is due to the dummy PE-nodes inserted to make the pipeline regular during the heuristic placement.

[18]The number of columns used (not shown), however, tends to increase slightly when the memory operation sharing

smaller for the mapping flow so that the mapping flow can handle it more easily espeically in terms of the inter-line interconnections (in the case of banded) as well as it generates very efficient pipeline structures for a certain class of input trees (in the case of ME).

## 7 Conclusion

We presented a novel memory operation sharing technique for the loops with inter-iteration data reuse patterns. During the pipelined mapping onto reconfigurable architectures, the technique exploits the opportunity of sharing memory interface resources between executions of different iterations. We developed the conditions for sharing memory operations in a loop, using a generic reconfigurable architecture template, and proposed an efficient heuristic method to generate the pipelines accordingly within a mapping flow. Our experimental results demonstrates that the technique can generate performance improvement of 3 times on a typical coarse-grain reconfigurable architecture.

The mapping of applications onto reconfigurable architectures requires much research. For instance, in current work we assumed that iterations of a loop are executed in pipeline, to develop a mapping flow that works reasonably for many applications. However, some loops may be best mapped when iterations are executed in parallel. Therefore the mapping style could be one dimension for optimizing mapping of different applications. Finally, the current mapping flow has several constraints in architectures and application loops. Our future research will investigate mapping techniques for more general classes of architectures as well as other types of loops.

## 8 Acknowledgements

This research was supported by grants from Motorola Corporation and Hitachi Ltd. We also thank members of the EXPRESS compiler team, especially Mehrdad Reshadi for their assistance.

## References

[1] H. Singh et al. MorphoSys: An integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. Computers*, 2000.

[2] T. Miyamori and K. Olukotun. REMARC: Reconfigurable multimedia array coprocessor. *Proc. ACM/SIGDA FPGA '98, Monterey*, 1998.

[3] R. Hartenstein. A decade of reconfigurable computing: A visionary retrospective. *Proc. DATE*, 2001.

[4] P. Schaumont et al. A quick safari through the reconfigurable jungle. *Proc. DAC, Las Vegas*, 2001.

technique is used, although in all the examples except ME, they came within 8 columns. In the case of ME, 16 and 11 columns were used respectively, when the memory operation sharing technique was used, not used.

[5] S. Goldstein et al. PipeRench: A coprocessor for streaming multimedia acceleration. *Proc. ISCA '99, Atlanta*, 1999.

[6] K. Bondalapati et al. Loop pipelining and optimization for run-time reconfiguration. *Proc. RAW*, 2000.

[7] E. Mirsky and A. DeHon. MATRIX: A reconfigurable computing architecture with configurable instruction distribution and deployable resources. *Proc. IEEE FCCM*, 1996.

[8] K. Bondalapati. Parallelizing DSP nested loops on reconfigurable architectures using data context switching. *Proc. DAC*, 2001.

[9] *Chameleon Systems*. http://www.chameleonsystems.com/.

[10] Z. Huang and S. Malik. Exploiting operation level parallelism through dynamically reconfigurable datapaths. *Proc. DAC, New Orleans*, 2002.

[11] A. Halambi et al. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. *Proc. DATE*, 1999.

[12] R. Maestre et al. A framework for reconfigurable computing: Task scheduling and context management. *IEEE Trans. VLSI Systems*, 2001.

[13] M. Kaul et al. An automated temporal partitioning and loop fission approach for FPGA based reconfigurable synthesis of DSP applications. *Proc. DAC*, 1999.