in Int. Conf. on Compiler, Architecture and Synthesis for Embedded Systems (CASES 2001), Atlanta, GA, November 2001.

A Compiler Framework for Mapping Applications to a Coarse-grained Reconfigurable Computer Architecture

Girish Venkataramani
Walid Najjar
University of California
Riverside
{girish, najjar}@cs.ucr.edu

Fadi Kurdahi
Nader Bagherzadeh
University of California
Irvine
{kurdahi, nader}@ece.uci.edu
bohm@cs.colostate.edu
Wim Bohm
Colorado State University
Fort Collins
bohm@cs.colostate.edu

ABSTRACT
The rapid growth of silicon densities has made it feasible to deploy reconfigurable hardware as a highly parallel computing platform. However, in most cases, the application needs to be programmed in hardware description or assembly languages, whereas most application programmers are familiar with the algorithmic programming paradigm. SA-C has been proposed as an expression-oriented language designed to implicitly express data parallel operations. Morphosys is a reconfigurable computer architecture that supports a data-parallel, SIMD computational model. This paper describes a compiler framework to analyze SA-C programs, perform optimizations, and map the application onto the Morphosys architecture. The mapping process involves operation scheduling, resource allocation and binding and register allocation in the context of the Morphosys architecture. The execution times of certain compiled image-processing kernels is comparable to the hand-coded assembly version, and the speed-ups compared to Pentium III range from 3x to 42x.

1. INTRODUCTION
Dehon [1] shows that computational density is a strong argument for FPGA-based reconfigurable computing systems over processor-based alternatives for data-parallel applications. However, for applications where the data path is coarse-grained (8 bits or more), the performance and power consumption on FPGAs are handled inefficiently. Also, the compilation time (kernel's synthesis, placement and routing) for and reconfiguration time on FPGAs are typically long. Coarser grained reconfigurable architectures [2], [15], [16], [17], [18] have been proposed as an alternative between FPGA-based systems and fixed logic CPUs.

The Morphosys architecture [2], [3], [4], [5], [7] is an example of a coarse-grained reconfigurable architecture. It is an integrated system-on-chip targeted at data-parallel applications with high throughput requirements. The reconfigurable element of Morphosys is an 8x8 array of processing elements that support a SIMD computational model. SA-C [9], [12], [13] is a highly expressive, algorithmic language that has been designed, primarily, to bridge the gap between algorithms and hardware circuits on FPGAs.

This paper describes a compiler framework for mapping applications written in SA-C for execution on the Morphosys architecture. The compiler's focus is on mapping SA-C loops, which expose data parallelism, onto the reconfigurable element. During analysis, the compiler performs loop optimizations and structural transformations in the context of the target architecture. Algorithms that perform operation scheduling, resource allocation and binding, and register allocation in the context of the Morphosys computational model are applied to the source program to produce a complete execution schedule.

2. RELATED WORK
Compiling applications written in a high-level language to coarse-grained reconfigurable platforms has been an active field of research in the recent past.

In Garp [17], the reconfigurable hardware is an array of computing elements. The compiler draws heavily from techniques used in compilers for VLIW architectures to identify Instruction Level Parallelism (ILP) in the source program, and then schedule code partitions for execution on the array of computing elements.

In CHIMAERA [19], the reconfigurable hardware is a collection of programmable logic blocks organized as interconnected rows. The focus of the compiler is to identify frequently executed instruction sequences and map them into a *Reconfigurable Functional Unit Operation* (RFUOP) that will execute on the reconfigurable hardware.

PipeRench [18] is an interconnection network of configurable logic and storage elements. The approach is to analyze the application's *virtual* pipeline, which is mapped onto *physical* pipe stages to maximize execution throughput. The compiler uses a greedy place-and-route algorithm to map these pipe stages onto the reconfigurable fabric.

The RAW micro-architecture [16] is a set of inter-connected tiles, each of which contains its own program and data memories, ALUs, registers, configurable logic and a programmable switch that can support both static and dynamic routing. The compiler partitions the program into multiple, coarse-grained parallel threads, each of which is then mapped onto a set of tiles.

The RaPiD architecture [15] is a field-programmable architecture that allows pipelined computational structures to be created from a linear array of ALUs, registers and memories. These are interconnected and controlled using a combination of static and dynamic control.

Some research efforts [8], [20] have focused on generic issues and problems in compilation like optimal code partitioning, and optimal scheduling of computation kernels for maximum throughput. While [20] proposes dynamic programming to generate an optimal kernel schedule, [8] proposes an exploration algorithm to produce the optimal linear schedule of kernels in

order to minimize reconfiguration overhead and maximize data reuse.

Our work concentrates on producing an instruction schedule that identifies and exploits parallelism at a fine- and coarse-grained level. The focus of the compiler is to build a framework to map a *single kernel* onto the reconfigurable hardware for efficient execution. This objective is orthogonal to those addressed in [8], [20], where the focus is on optimal *inter*-kernel scheduling. Also, the techniques proposed in [15], [18] can be used to optimally pipeline the schedule generated by our compiler.

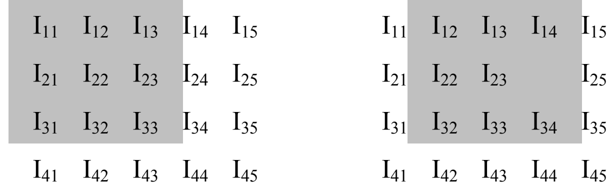
3. THE SA-C LANGUAGE

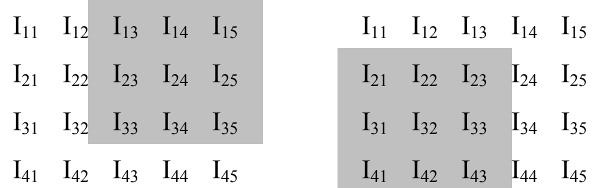
```
Int8[8,8] f(int8[8,8] A) {
  Int8[8,8] R =
    For window w[3,3] in A {
      Int8 x = For e in w
      return (sum(e));
    } return (array(x));
} return R;
```

(a)

```
For (I=0; I<M; I++) {
 For (J=0; J<N; J++) {
  For (X=I; X<(I+3); X++) {
   For (Y=J; Y<(J+3); Y++) {
    R[I][J] += A[X][Y];
   }
  }
 }
}
```

(b)





(c)

Figure 1: SA-C Loop Example that performs the following function:

$$R[x][y] = \sum_{a=x}^{x+2}\sum_{b=y}^{y+2} A[a][b]$$

(a) The SA-C source code
(b) equivalent C code
(c) The source array, *A*, and the windows (shaded areas) produced in different iterations.

SA-C [9], [10], [11], [12], [13] is an expression-oriented, single assignment language. The data types in SA-C support variable bit-width precision for integer and fixed point numbers. The language does not support recursion or pointers.

SA-C supports true multi-dimensional arrays. Thus, the language allows the programmer to access sub-arrays like elements, rows, columns, windows, slices, and planes. In addition, a number of common image processing operations like histogram and median are built into the language.

Every loop in SA-C has three components to it – the loop generator, the loop body and the loop collector. A loop generator specifies what values are generated in each iteration, and how many iterations the loop will perform. The loop collector generates a return value for the loop expression, by combining, in various ways, values that are produced within the loop.

There are 2 main types of loop generators – array-*element* and w*indow* generators. An *element generator* produces a scalar value from the source array per iteration. A loop with a *window generator* allows a window to "slide" over the source array producing sub-arrays of the same rank (dimensions) as the source array. Figure 1(a) shows an example of a SA-C loop, and Figure 1(b) shows the equivalent C code. The code computes the resultant array, *R*, as a function of the input array, *A*, as follows:

$$R[x][y] = \sum_{a=x}^{x+2}\sum_{b=y}^{y+2} A[a][b]$$

The SA-C program has 2 loops. The outer loop contains a window generator that produces 3x3 windows from the source array, *A*. Figure 1(c) shows snapshots of *A*, and the shaded areas represent the windows that are generated. The inner loop contains an element generator, producing scalar values from the generated window. Essentially, the inner loop computes the sum total of each generated window. The outer loop creates an array whose elements are the summation values produced by the inner loop.

Hierarchical Data flow graphs (HDFG) are used as intermediate representation in the compiler. An HDFG is an acyclic, directed, data flow graph, where some nodes can have sub-graphs within them. These graph representations are similar in structure to the data dependence control flow (DDCF) graphs [2]. While the DDCF graphs are geared toward translation to VHDL, the HDFG

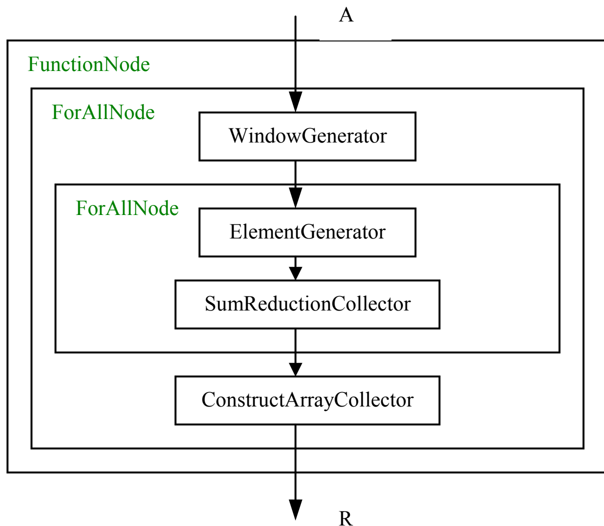


Figure 2: HDFG Representation

is more suited for Morphosys related optimizations and analysis. Figure 2 shows the equivalent HDFG representation of the example in Figure 1.

4. THE MORPHOSYS ARCHITECTURE

Morphosys [7-11, 22, 24] is a reconfigurable, integrated system-on-chip targeted at applications with inherent data parallelism, high regularity and high throughput requirements.

Figure 3 shows the organization of the Morphosys architecture. It consists of five main components:

- Tiny RISC processor: is a MIPS-like core with a 4-stage pipeline. It has 16 32-bit registers and three functional units – a 32-bit ALU, a 32-bit shift unit and a memory unit. The Tiny RISC processor handles general-purpose operations and controls the execution of the RC Array through special instructions in its ISA [7].
- Reconfigurable Cell Array (RC Array): is the reconfigurable computing element of the architecture. It consists of an 8x8 matrix of processing elements called the reconfigurable cells. Each RC cell consists of an ALU-Multiplier, a shift unit, input multiplexers, and the context register. The context register provides control signals for the RC components. All RC cells in the same row/column share the same configuration word (perform the same operation), while different rows/columns may receive different context words.
- Context memory: stores the configuration program (the contexts) for the RC Array. It is logically organized into two blocks, each of which is further subdivided into eight sets.
- Frame buffer: is a streaming buffer that contains two sets with two banks in each set. It enables streamlined data transfers between the RC Array and main memory, by overlapping computation with data load and store, alternating using the two sets.

DMA controller: The DMA controller performs data transfers between the Frame Buffer and the main memory. The Tiny RISC core processor uses DMA instructions to specify the necessary data/context transfer parameters for the DMA controller.

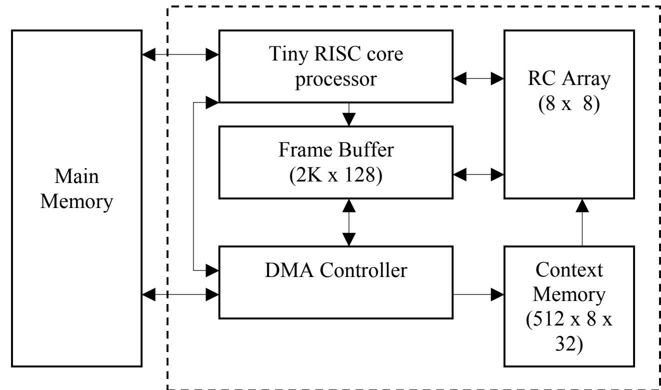
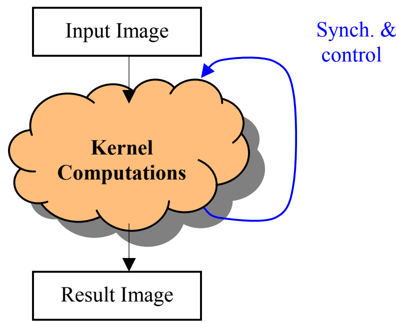


Figure 3: The Morphosys Architecture

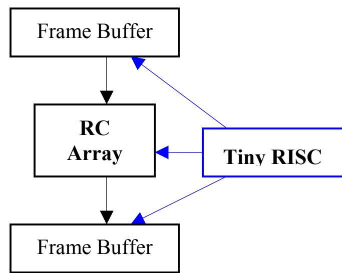
5. COMPILER FRAMEWORK

Code partitioning determines which segments of the program will execute on the RC Array and which will execute on the Tiny RISC processor. A typical kernel of an image-processing application (Figure 4(a)) consists of set of computation intensive operations that are performed in a loop. These kernels[1] (loops) are data parallel operations and are executed on the RC Array, while the sequential code (outside loops), and the necessary synchronization and control code are mapped onto the Tiny RISC (Figure 4(b)).

[1] The terms *kernel* and *loop* are used interchangeably throughout this document.



(a)



(b)

Figure 4: Mapping kernels onto Morphosys
(a) A typical image processing kernel
(b) Implementation on Morphosys

The main objective of the compiler is to translate each kernel into an instruction schedule for execution on the RC Array. The process of generating this schedule is referred to as "loop synthesis" throughout this document and is described in the next section. The compiler, first, performs a number of tasks that prepare the program graph for loop synthesis.

Figure 5 shows the flow of compilation. The right-side branch of compilation after code partitioning (Figure 5) represents the compilation of code that is not within loops. This phase of code generation is, essentially, similar to that of traditional compilers.

Code segments embedded within loops go through a series of transformations before they are synthesized[2]. In the *Transformation to Context Codes* phase, the compiler annotates every node in the HDFG with the equivalent sequence of RC Array context codes that performs the function of the node. Next, some conventional optimizations like constant folding, constant propagation, copy propagation, and operation strength reduction are performed on the program graph.

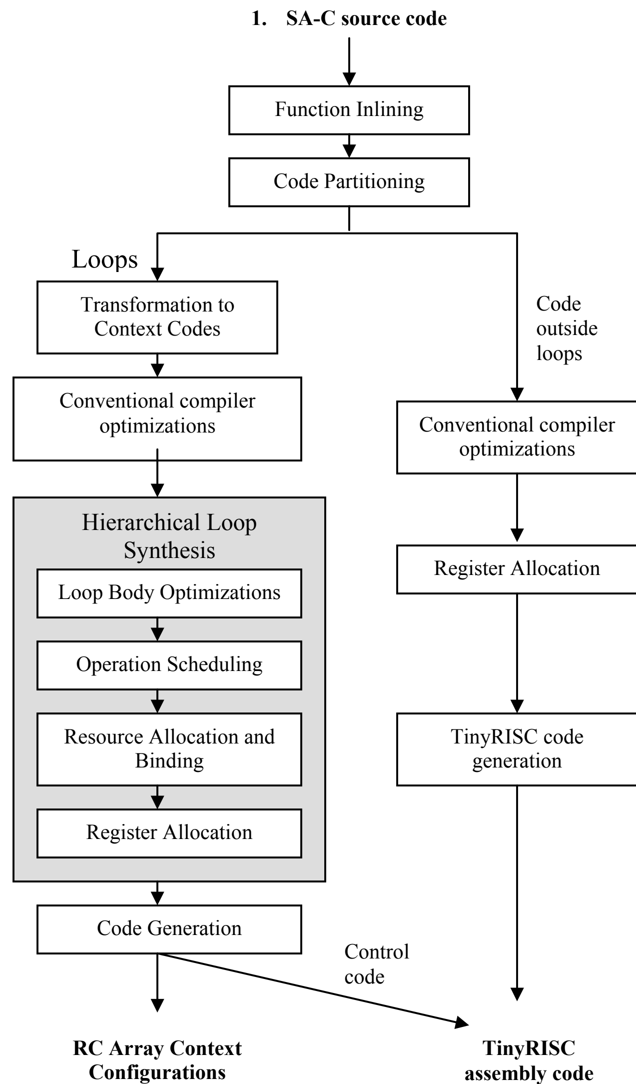


Figure 5: Flow of Compilation

6. HIERARCHICAL LOOP SYNTHESIS

Loops are synthesized individually based on their relative position in the *loop hierarchy*. The innermost loop is defined to be at the bottom of the loop hierarchy. The compiler's approach is to synthesize the inner most loop, and then recursively move up the hierarchy until the outermost loop is synthesized. The compiler framework defines different execution models based on the loop's generator.

[2] *Synthesizing* a loop implies generating an execution schedule in terms of context codes and Tiny RISC control codes. This will specify the temporal ordering of and the resources used by all operations.

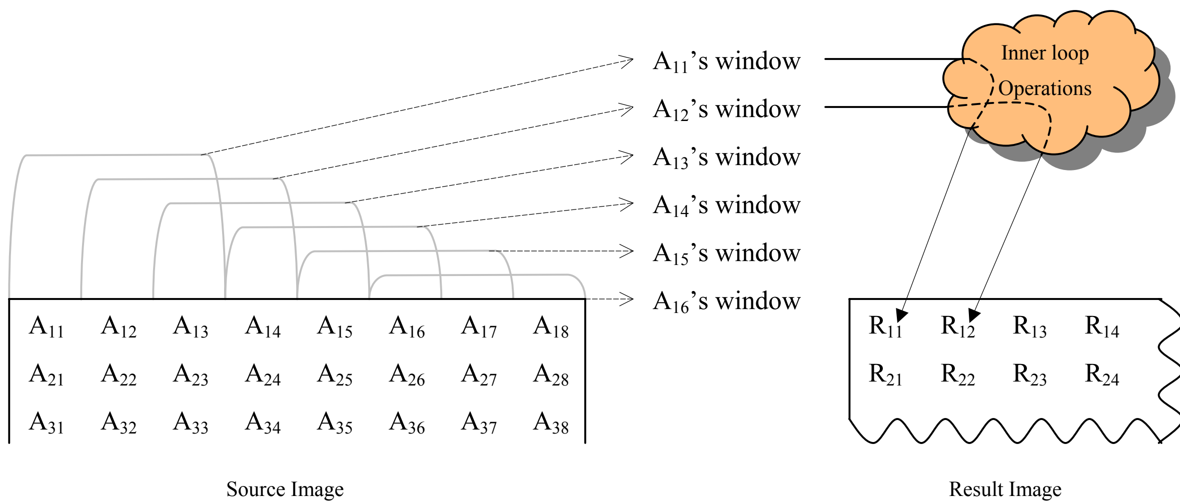


Figure 6: Snapshot of windowing loop

Figure 7: Runtime snapshot of Windowing Loop; (a), (b), (c), (d) are windowing computations that are executed one after the other.

6.1 Element-Generating Loops

An element-generating loop's body is a function of a particular element of the source array. For such a loop that is not nested within another, there are no data dependencies and no common computations between iterations. The loop is unrolled in both dimensions so as to process 64 loop iterations concurrently. Execution of every loop iteration is performed on a single RC Array cell. Hence, the resource-binding problem is trivial and is obviated.

The operation-scheduling problem reduces to scheduling a data flow graph onto a single, sequential processor. The scheduling algorithm first identifies the *ready* operations (whose data dependencies have been satisfied), and randomly picks a *ready* operation and schedules it in the next available processor cycle. The same schedule is executed on every RC cell, but on different data items.

The register allocation strategy keeps track of free registers and live operations, and allocates registers to intermediate results, as and when required. Register spills are handled by writing the values to the frame buffer.

6.2 Window-Generating Loops

Figure 6 shows a snapshot of the windowing loop example from Figure 2. The loop generates a 3x3 window in each iteration of the loop. The figure shows the iteration windows in the first row of the image. Each iteration window is transformed into a single pixel of the resultant image.

In spite of the SIMD computational model of the RC Array, all the iteration windows present in the RC Array cannot be computed concurrently. This is because some of the elements are part of multiple iteration windows. For example, element A_{13} is a member of 3 iteration windows – A_{11}, A_{12} and A_{13}. However, execution of non-overlapping iteration windows like A_{11} and A_{14} can be performed concurrently.

The framework for executing windowing loops is shown in Figure 7. It shows a snapshot of the elements of the source array that are placed on the RC Array. Each shaded region in the figure corresponds to a separate iteration window in the source image. Each sub-figure (Figures 7 (a), (b), (c) and (d)) represents an execution snapshot showing the iteration windows that are executed concurrently. However, the execution of the iteration windows in one sub-figure is never executed concurrently with the iteration windows of another sub-figure. Hence, the loop's execution schedule is a sequential ordering comprised of the execution schedules of the sub-figures. There are a total of 36 iteration windows in the RC Array, and sets of 4 iterations can be executed concurrently.

This framework can be generalized for any loop generating windows of size MxN. The RC Array processes (8-M+1) iterations in the horizontal dimension and (8-N+1) iterations in the vertical dimension, for a total of [(8-N+1) x (8-M+1)] iterations between successive data fetches. The following sections describe how this strip-mined version of the loop is synthesized. In the current implementation of the compiler, all windows are assumed to be smaller than or equal to an 8x8 window in size. Since most standard image-processing applications work within this constraint, this is a reasonable assumption to make.

6.2.1 Windowing Loop Optimizations

Figure 8 shows a program that computes the resultant array, *R*, for any two input arrays, *A* and *B*. The program can be summarized by the following function:

$$R[x][y] = \sum_{a=x}^{x+2} \sum_{b=y}^{y+2} (A[a][b] * B[a][b])$$

```
Int8[:,:] R =
  For window wa[3,3] in A
       dot window wb[3,3] in B {
    Int8 asum =
      For a in wa dot b in wb
      return (sum(a * b));
  } return (array(asum));
```

(a)

```
For (I=0; I<M; I++) {
 For (J=0; J<N; J++) {
  R[I][J] = 0;
  For (X=I; J<(I+3); X++) {
   For (Y=J; Y<(J+3); Y++) {
    R[I][J] += A[X][Y] * B[X][Y];
   }
  }
 }
}
```

(b)

Figure 8: Windowing Loop example
(a) SA-C code
(b) C code

The windows generated in separate iterations of this loop have some common sub-expressions. Figure 9 shows the pictorial view of two iterations of this loop. For example, the computations "$A_{12} * B_{12}$" and "$A_{13} * B_{13}$" are performed in both iterations.

In general, whenever a particular element of the source array appears in multiple iteration windows, there could potentially be common sub-expressions. For compiler analysis, the windowing loop must be unrolled so as to expose them. The number of iterations of the windowing loop that need to be unrolled is equal to the number of overlapping iterations. For a loop generating MxN window with steps of *sh* and *sv* in the horizontal and vertical dimensions respectively, the number of overlapping iterations, *NI*, is given by:

NI = `ceil(N/sh) * ceil(M/sv)`
where ceil(n) returns the largest integer lesser than or equal to n.

However, for window sizes greater than 4 in either direction, it is not possible to fetch all the *NI* windows into the RC Array. Consider a window size of 5x5. The first window in each row begins on column 1, and the last window begins on column 5 and ends on column 9. Hence, this requires a total of 9x9 elements, whereas the RC Array is a matrix of 8x8 RC cells. For such windows, there will be a total of (8-N+1) iterations that need to be analyzed. However, if the source array is smaller than the RC Array itself, then the number of windows is equivalent to (w-N+1),

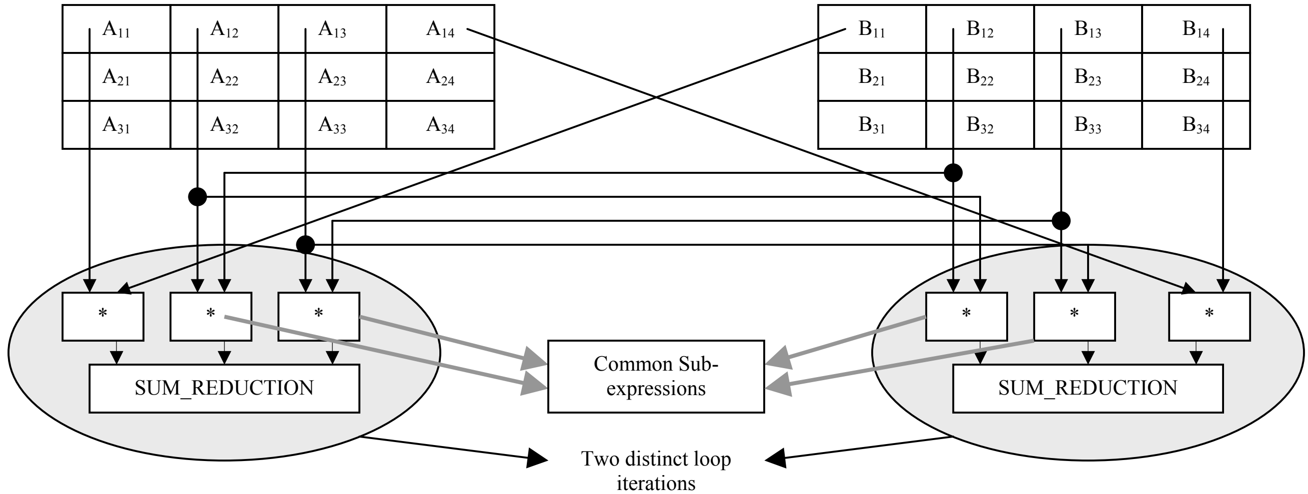


Figure 9: Snapshot of loop iterations

where *w* is the width of the source array. Hence, the number of iterations, *NI*, is modified as follows:

```
X = MIN(M, 8)
Y = MIN(N, 8)
H = ceil[{MIN(N, Y – N + 1)}/sh]
V = ceil[{MIN(M, X – M + 1)}/sv]
NI = H * V
```

The compiler analyzes these *NI* iteration windows and eliminates all redundant sub-expressions. This gives rise to dead code, which is eliminated as well. At the end of this optimization pass, there will be *NI* distinct data flow graphs corresponding to each iteration. However, there may be some *cross-edges* between these data flow graphs that represent the re-use of computation. These edges are synthesized into registers during the register allocation phase.

6.2.2 Loop Synthesis

To synthesize a windowing loop, the synthesis techniques discussed in the following sections will be applied to **each** of the *NI* iteration graphs. The final schedule is a linear ordering of each iteration's schedule.

In the context of the RC Array, a *resource* is defined to be a single row. Before synthesis, resource requirement numbers are assigned to all loop nodes. The top-most loop in the loop hierarchy is always assigned a resource requirement of 8 in order to maximize RC Cell utilization. The resource requirement for inner loops is defined to be "the vertical dimension of the window generated by its parent loop". Figure 10 shows an example program (a), and its HDFG representation (b). Each loop in the HDFG is annotated with its resource-requirement (RR) assignment.

6.2.2.1 Operation Scheduling

The operation-scheduling problem for a windowing loop is defined as finding a schedule that executes in minimum time under two constraints - the availability of resources and the RC Array execution mode. There are two modes of execution on the RC Array – row mode and column mode. In any given clock cycle, only one mode of operation can be active. Concurrent operations must all execute in the same modes throughout each operation's lifetimes.

The operation scheduling algorithm itself is known to be NP-complete. One popular heuristic is the *List Scheduling* algorithm. The compiler uses an extension of this algorithm that takes into accounts the constraints of the RC Array.

The schedules thus generated (for each of the *NI* iterations) are then linearly ordered to complete the execution of all the iterations that are present in the RC Array. Then, the next set of data is fetched into the RC Array and the same execution schedules are repeated. For a windowing loop generating *M*x*N* windows, the total execution time, *T*, of the loop over an image of size, [h, w], is given by:

Let S = Size of the source image in any dimension
Let Dt = Distance between first element of two successive data fetches
= NW * st
where
NW = Number of windows in that dimension
st = Window step in that dimension

The number of data fetches in that dimension = S/Dt
Number of windows in any dimension,
NW = ceil[(X – W + 1)/st]
where
X = MIN(Wp, 8)
W = window size in that dimension
Wp = source image size in that dimension
(= 8 if outermost loop)

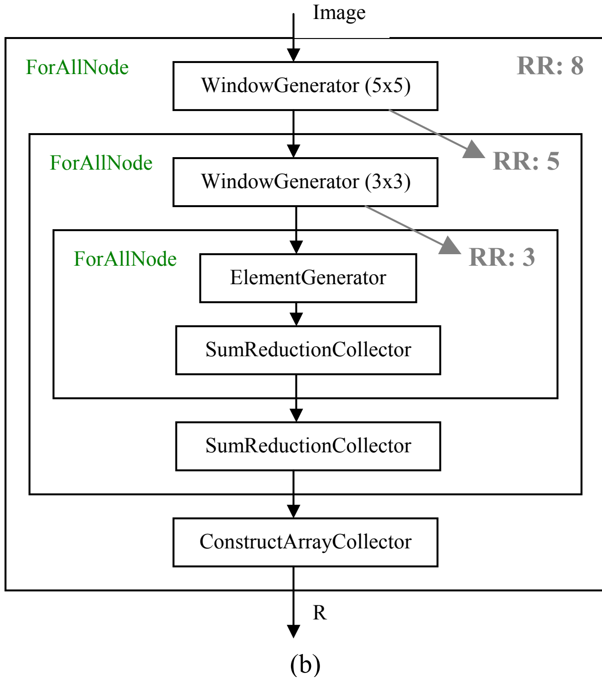
Hence, total Data fetches, D = Dh * Dv
where
Dh = Number of data fetches in horizontal dimension
Dv= Number of data fetches in vertical dimension

Execution time of window loop, T = $D * \sum_{i=1}^{NI} ki$

where k_i = Latency of the ith iteration's schedule

```
Int8[:,:] R =
  For window win[5, 5] in Image {
    Int8 res =
      For window w[3, 3] in win {
        Int8 x =
          For elem in w
          Return (sum(elem));
      } return (sum(x));
  } return (array(res));
```

(a)



(b)

Figure 10: Resource Allocation: for Loop Hierarchies is based on the vertical dimension of the window generated by the parent loop (is 8 for outer-most loops. Example of: (a) SA-C program, and (b) its equivalent HDFG representation

6.2.2.2 Resource Allocation and Binding

The RC Array is divided into four quadrants each of size 4x4. A given RC cell can directly access only the cells in the same row and column as itself. Further, these cells need to be in the same quadrant as itself. In Figure 11, for example, cell R_{22} can directly access (in the same clock cycle) cells R_{12}, R_{32}, and R_{42} in the vertical dimension, and cells R_{21}, R_{23}, and R_{24} in the horizontal dimension. Accessing any other cell would incur a communication penalty.

The objective of resource allocation is to minimize these communication latencies. To solve this problem, a graph is created, where the nodes are operations and edges between nodes indicate “affinity to sharing a resource” between the two nodes. These edges, called *shareable* edges, are added as follows:

- Concurrently executing nodes don’t share any edge
- Starting from a base node, all other nodes are assigned an edge between them
- Two nodes that have a direct data-dependence (i.e. an edge in the data flow graph) are assigned a higher weight (say k) than all other nodes (default weight is 1). This is because the result of one operation is the input operand of the other. There will be no communication penalty if the two operations share the same resource. Hence, a weight on an edge gives more importance to it.

R_{11}	R_{12}	R_{13}	R_{14}	R_{15}	R_{16}	R_{17}	R_{18}
R_{21}	R_{22}	R_{23}	R_{24}	R_{25}	R_{26}	R_{27}	R_{28}
R_{31}	R_{32}	R_{33}	R_{34}	R_{35}	R_{36}	R_{37}	R_{38}
R_{41}	R_{42}	R_{43}	R_{44}	R_{45}	R_{46}	R_{47}	R_{48}
R_{51}	R_{52}	R_{53}	R_{54}	R_{55}	R_{56}	R_{57}	R_{58}
R_{61}	R_{62}	R_{63}	R_{64}	R_{65}	R_{66}	R_{67}	R_{68}
R_{71}	R_{72}	R_{73}	R_{74}	R_{75}	R_{76}	R_{77}	R_{78}
R_{81}	R_{82}	R_{83}	R_{84}	R_{85}	R_{86}	R_{87}	R_{88}

Figure 11: RC Array Connectivity

Another type of edges, called *closeness* edges, is also added to the graph. These edges reflect the condition when two nodes are assigned to different resources; however, these resources must be as close to each other as possible. Consider an operation, *op*, which needs two operands that are produced as results of operations, *op1* and *op2*. Then, *op1* and *op2* must be scheduled as close to each other as possible in order to avoid the communication penalty. These edges are added as follows:

- If the operands of a node are produced by two different operations, then these two operations will share a *closeness* edge between them
- The weight on this *closeness* edge is accumulated if more *closeness* edges are generated between the same two nodes.

The graph thus generated is subject to CLIQUE_PARTITIONING[3]. There are two different objectives that need to be satisfied during resource allocation – resource sharing (based on the *shareable* edges) and assignment of resources close to each other (*closeness* edges). To satisfy these seemingly orthogonal objectives, the compiler performs two levels of clique partitioning:

- Perform CLIQUE_PARTITIONING based on the *shareable* edges.
- Create a new graph by collapsing each clique into a single, unique node.
- Perform CLIQUE_PARTITIONING on this new graph based on the *closeness* edges.

One of the components of the CLIQUE_PARTITIONING problem is to find the maximal clique in the graph (MAX_CLIQUE). This problem is known to be NP-complete. The compiler uses a

---

[3] CLIQUE_PARTITIONING is a popular graph-partitioning algorithm. A *clique* is defined as a fully connected sub-graph.

heuristic to solve it – the clique containing the node with the maximum number of edges is assumed to be the best candidate for the maximal clique.

In the end, the graph is a set of "super-cliques", where each node in the super-clique represents a clique from the first level of clique partitioning. When every clique in the super-clique has been assigned a resource, all the operations within that clique will share this resource. The compiler uses a heuristic is used in assigning resources to the cliques within a super-clique. It tries to keep the node with largest "closeness requirements" (equal to the sum total of all weights on its *closeness* edges) as close as possible to every other node.

6.2.2.3 Register Allocation

Register Allocation strategy for windowing loops use the same strategies as used by element-generating loops. However, after performing common sub-expression elimination, values (represented by *cross-edges*) may be forwarded to other iterations. Register allocation is performed in two phases. First, the cross-edges are allocated to registers. These registers will be required throughout the entire loop execution between data fetches. Then, registers are allocated to each (of the *NI*) iteration. However, these registers are alive only during the particular iteration's execution.

7. PERFORMANCE MEASUREMENTS

To measure the efficiency of the compiler, certain sample image-processing kernels (Table 1) are compiled and their execution times over a sample data set are measured. These kernels are also written separately in native C code, which are compiled using the VC++ 6.0 compiler with the highest level of optimizations turned on and are executed under Windows 2000 on an 800 MHz Pentium III platform. The speed-ups achieved over Pentium III range from 3x for Convolution to 42x for Motion Estimation.

Table 1: Test Applications

Application Kernel	**Description**
Wavelet	Common program used for multi-scale analysis in computer vision, and image compression. This particular implementation works on 5x5 windows of the source image
Prewitt	An edge detection algorithm that uses 3x3 horizontal and vertical masks
2D Convolution	Linear convolution of every 3x3 window in the source image
Motion Estimation	A kernel used in MPEG-4 compression; identifies redundancies between frames in an MPEG video stream

In this work, we have focused on efficient mapping of image processing kernels in the application for execution on the RC Array. We have not addressed issues regarding optimized management of data movement and data caching (in the frame buffer). In the best case, all data movement between main memory and frame buffer can be overlapped with computation. In the worst case, there are no concurrent data movements. Figure 12 shows the amount of overhead that could be incurred in each of the test benches. In our experience, in almost all applications, at least 50% of data movement can be overlapped with computation. Figure 13 compares the performance of the compiled kernels (assuming 50% data overlapping) with the execution of equivalent codes on Pentium III.

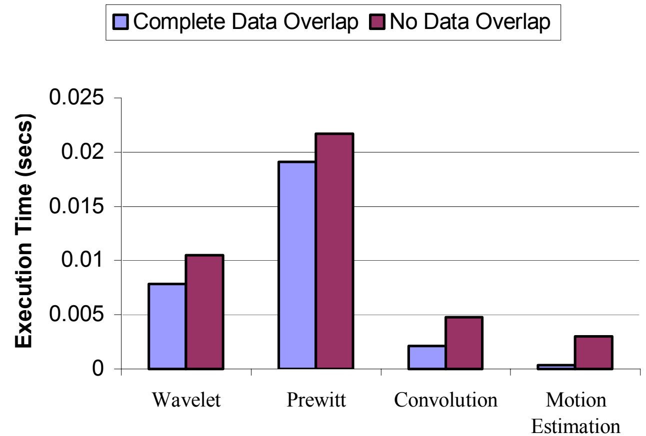


Figure 12: Effect of Data Overlap

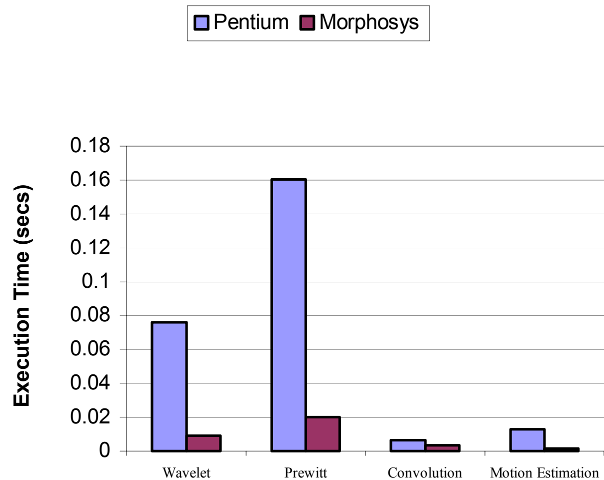


Figure 13: Performance Comparison

We have also compared our results with the execution of hand-coded versions of the same kernels. The hand-coded Motion Estimation kernel executes in 1 millisecond. The compiled version executes in 0.3 milliseconds with 100% data overlapping, and in 1.6 milliseconds with 50% data overlapping.

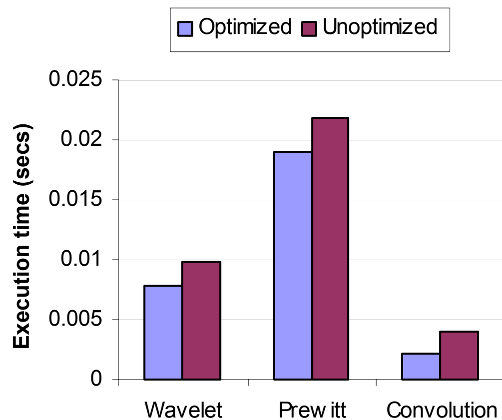


Figure 14: Compiler Optimizations

The effect of compiler optimizations is measured by comparing the execution times of optimized schedules with the un-optimized versions (Figure 14). In the Motion Estimation kernel, there are opportunities for optimizations. However, all other kernels experience significant benefits from compiler optimizations.

8. CONCLUSIONS

This paper presents a framework for efficient compilation of applications written in a high-level language to a reconfigurable computing architecture. In particular, the compiler aims at extracting the data parallelism at coarse- and fine-grained levels in a given application, and then produces an execution schedule that explicitly reflects a SIMD execution model.

It describes the synthesis approach of mapping loops, which performs operation scheduling, resource binding and register allocation, in order to produce a precise execution schedule. Also, different compiler optimizations are proposed that could potentially improve the execution time of applications on the target platform.

9. REFERENCES

[1] A. DeHon, "The Density Advantage of Configurable Computing". *IEEE Computer*, vol. 33, pp. 41-49, April 2000.

[2] H. Singh, et. al., "Morphosys: An Integrated Reconfigurable Architecture,". *NATO Symposium on Systems Concepts and Integration*, Monterey, CA, 1998.

[3] H. Singh, et. al., "Morphosys: Case study of a reconfigurable computing system trageting multimedia applications,". *37th Design Automation Conference*, Los Angeles, CA, 2000.

[4] H. Singh, et. al., "Morphosys: A Parallel Reconfigurable System,". *Euro-Par*, Toulouse, France, 1999.

[5] H. Singh, et. al., "Morphosys: A Reconfigurable Architecture for Multimedia Applications,". *Workshop on Reconfigurable Computing at PACT*, Paris, France, 1998.

[6] E. M. C. Filho, "The TinyRISC Instruction Set Architecture, Version 2," University of California, Irvine, Irvine, CA November 1998. http://www.eng.uci.edu/morphosys/docs/isa.pdf.

[7] M. Lee, et. al., "Design and Implementation of the Morphosys Reconfigurable Computing Processor,". *Journal of VLSI and Signal Processing-Systems for Signal, Image and Video Technology*, 2000.

[8] R. Maestre, et. al., "Kernel Scheduling in Reconfigurable Computing,". *DATE*, Munich, Germany, 1999.

[9] R. Rinker et. al., "An Automated Process for Compiling Dataflow Graphs into Hardware,". *IEEE Transactions on VLSI Systems*, vol. 9, 2001.

[10] R. Rinker, et. al., "Compiling Image Processing Applications to Reconfigurable Hardware," *IEEE International Conference on Application-specific Systems, Architectures and Processors*, Boston, MA, 2000.

[11] J. Hammes, et. al., "Cameron: High Level Language Compilation for Reconfigurable Systems," *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Newport Beach, CA, 1999.

[12] J. Hammes, et. al., "Compiling a High-level Language to Reconfigurable Systems," *Compiler and Architecture Support for Embedded Systems (CASES)*, Washington, DC, 1999.

[13] W. Bohm, "The SA-C Language - Version 1.0," Colorado State University, Fort Collins, CO, Technical Report June 2001. http://www.cs.colostate.edu/cameron/Documents/sassy.pdf.

[14] W. Bohm, "The SA-C Compiler Data-Dependence-Control-Flow (DDCF)," Colorado State University, Fort Collins, CO, Technical June 2001. http://www.cs.colostate.edu/cameron/Documents/ddcf.pdf.

[15] C. Ebeling, et. al., "Mapping Applications to the RaPiD Configurable Architecture," *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, CA, 1997.

[16] Elliot Waingold, et. al., "Baring it all to software: RAW machines," *IEEE Computer*, vol. 30, pp. 86-93, 1997.

[17] J. Wawrzynek and T.J. Callahan, "Instruction-level Parallelism for Reconfigurable Computing," *8th International Workshop on Field-Programmable Logic and Applications*, Berlin, 1998.

[18] S.C. Goldstein, et. al., "PipeRench: A Reconfigurable Architecture and Compiler," *IEEE Computer*, vol. 33, pp. 70-77, 2000.

[19] A. Ye, et. al., "CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit," *27th Annual International Symposium on Computer Architecture (ISCA)*, Vancouver, British Columbia, 2000.

[20] V. K. Prasanna et. al., "Mapping Applications onto Reconfigurable Architectures using Dynamic Programming," *Military and Aerospace Applications of Programmable Devices and Technologies*, Laurel, Maryland, 1999.