A Register File and Scheduling Model for Application Specific Processor Synthesis*

E. Ercanli C. Papachristou

Computer Engineering Department
Case Western Reserve University
Cleveland, Ohio 44106

Abstract– In this paper, we outline general design steps of our synthesis tool to realize application specific co-processors such that for a given scientific application having intensive iterative computations especially with recurrences, a VLIW type of co-processor is synthesized and realized, and an accompanying parallel code is generated. We introduce a novel register file model, *Shifting Register File (SRF)*, based on cyclic regularity of register file accesses; and a simple method, *Expansion Scheduling*, for scheduling iterative computations, which is based on cyclic regularity of loops. We also present a variable-register file allocation method and show how simple logic units can be used to activate proper registers at run time through an example.

1. Introduction

Application Specific Processor (ASP) design concepts [1,7,8] gained attention after extensive developments have been done in two different fields: VLSI design automation and parallel code generation fields. Related advancements are made in layout compaction, logic synthesis, RTL and behavioral synthesis, software pipelining, and VLIW type of architectures.

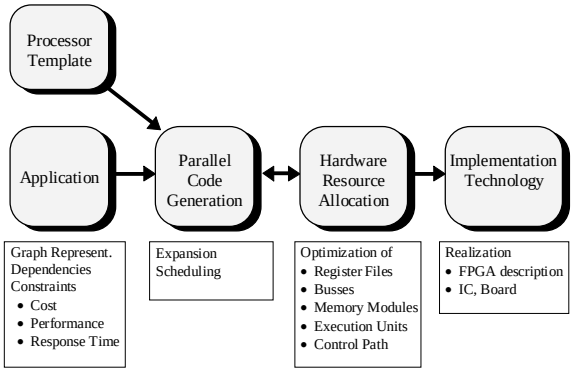
These recent developments offer high performance architectures and implementations for scientific applications. *Scientific applications* can be described as mostly iterative and recursive computations which contain explicit and implicit parallelism unlike symbolic applications such as system programs and operating systems. Parallelism in scientific applications can be attained automatically by complex compilers instead of by trained parallel programmers.

The objective of our research is to build an application specific processor synthesis tool such that for a given scientific application, a co-processor board (or chip) and accompanying parallel code will be designed, synthesized, and generated. The state-of-the-art techniques of *design automation* and *microarchitecture* fields, such as VLIW architecture, software pipelining, and programmable logic devices, are utilized; and new techniques, such as a scheduling method, a register file model and allocation, memory unit allocation, and implementation techniques are developed for high performance computation.

For ASP design, usually an architecture is selected first, then sequential code is parallelized to utilize the hardware efficiently. This parallel code generation process is called *instruction scheduling*, which must preserve the semantics of original sequential code while minimizing the execution time. In other words, scheduled code must ensure control and data dependencies, and forbid resource contentions.

*This work has been supported by the Semiconductor Research Corporation (SRC) under Contract DJ-527.

Figure 1: Design Steps



Software pipelining is a widely used static instruction scheduling technique to parallelize input code at compile time, which can be classified as global and local scheduling methods. In global scheduling techniques, also used in [1], many iterations are unrolled to form a large loop-body and operations are properly moved around to find a new but more condensed loop body. On the other hand, in local scheduling techniques, a single iteration (loop-body) is scheduled and successive iterations are properly started and overlapped to form a new condensed loop-body.

For the branch intense loops and straight codes, global scheduling techniques may yield better results [3]. On the other hand, for the loop intense computations, local scheduling techniques yield better parallelization [6]. Because local scheduling techniques use the cyclic regularity of loops, and do not unroll the iterations which causes large code expansion. In fact, most scientific programs have just a few number of conditional branches inside the loop [6].

Rotation Scheduling, a local scheduling method proposed by Chao and LaPaugh in [2], schedules cyclic DFGs (Data Flow Graphs) with resource constraints using loop pipelining. It transforms a given scheduling into a more compact scheduling by a retiming method. Although it results in a proper scheduling handling inter-iteration dependencies, it does not generate *pre-loop* and *post-loop* parts automatically, and does not have an anti-data dependency removal method.

Register allocation in scheduled loops is to allocate the registers in a way that minimizes register idle time while preventing lifetimes of the same loop variants corresponding to successive iterations from being assigned to the same physical register. Two current register allocation methods in local scheduling are *Modulo Variable Expansion*, and *Rotating Register Files* [5].

In modulo variable expansion method, successive iterations use different codes and different register sets, although successive iterations are overlapped. Iterations are unrolled to resolve resource contentions. This causes performance degradation.

In [5], Rau and et al present various heuristics for register allocation for software pipelined loops. They use one large rotating

DAC 96 - 06/96 Las Vegas, NV, USA ©1996 ACM, Inc. 0-89791-833-9/96/0006..$3.50

register file for all the variables, and offer various strategies for binding variables to registers. But their strategies are hard to implement, and have high computational complexity. In this method, they use single version of code. Each iteration has a different register set. Register sources and targets are dynamically renamed. Instruction's register specification points to *Iteration Control Pointer* (ICP) modulo the number of registers in rotating register file. ICP is decremented each time new stage starts so that new register file set is activated. Rotating register file is global pertaining to all variables.

We propose the use of local SRFs instead of global rotating register files to prevent loop variants being assigned to the same physical register. In SRF method, different register sets are assigned for different iterations, and each variable has a unique SRF, which can also be duplicated for concurrent accesses when necessary. Instead of modulo calculation hardware, SRFs use simple counters or PLAs to access proper registers. Also this method does not need variable increment or decrement for register control, which results in code expansion. To resolve data contentions, SRF method avoids iteration unrolling, another great performance degradation. The number of registers is determined after the code is generated for 'infinite' number of SRFs. We developed a local scheduling technique, *Expansion Scheduling*, to schedule loops for SRF-based VLIW-type of architectures.

After a brief discussion on design methodology in Section 2, Section 3 discusses code generation issues and presents SRF model and Expansion Scheduling method. Section 4 and Section 5 goes through an example to explain SRF allocation schema and discusses possible design choices. Section 6 shows results of our design tools for four applications.

2. Design Methodology

Major issues in ASP Design are as follows:

- *Concurrent Hardware and Software support for numerically intensive iterative applications*
- *Simple, pipelined execution units*
- *Multiple memory modules and register files*
- *Retargetable hardware and parallel code*

Figure 1 illustrates general design steps of our synthesis tool to realize a processor and generate parallel code for a given scientific computation such as image and DSP applications.

Parallel Code Generation

Our architecture has multiple execution units. Application code is mapped onto the execution units to achieve minimal execution time while preserving the semantics for correctness. There are certain possible hazards, namely data and control dependencies, and resource contentions.

Register Files and Busses

We employ two types of register files: *static register files* for loop-invariants[1], and *SRFs* for loop-variants[2]. SRFs offer automatic register renaming for removing anti data dependencies, accessing recursive values concurrently, and easy addressing.

Memory Organization Schema

In loops of scientific code, most indirect references are made to array elements. Array access patterns should be determined at compile time, and array elements should be placed into memory modules properly such that there will be no data contention between the pairs of elements that need to be accessed concurrently. We have developed a novel technique for distributing array elements, especially recursively and concurrently accessed elements, onto memory modules, which could be single- or multi-ported, single or multiple modules.

Clustering Execution Units

The objective of clustering is to lower the cost by minimizing the number of execution units. Depending on the initial constraints, we may start designing with unlimited or limited number of resources. We may design an architecture and generate parallel code, then try to cluster execution units such that no scheduling changes will be needed; or we may start designing for a limited number of resources and need not cluster execution units. We construct *compatibility graphs* for clustering.

Synthesis and Implementation

At the level of scheduling, and register, memory and execution unit optimization, we end up having various design alternatives, for which we can figure out the performance and cost. Then we can choose the design that fits our needs best. Hardware implementation is the final step. We can simulate the design on Compass Design Automation Tools. For prototyping and production, we investigate using Field Programmable Logic Arrays (FPGAs) since, in reality, we have limited time for design and realization of ASP designs. FPGAs offer fast protoyping and less expensive production cost for a limited number of products[4].

3. Code Generation

3.1 Architecture Model

Figure 2 illustrates our VLIW-like processor-template having two execution units. In the template, register files having the same label are duplicates that hold the same values. Register files contain both static and SRFs to provide concurrent accesses to the same register value with the objective to remove anti-data dependencies automatically. We can start scheduling for a template with a certain number of execution units or determine optimum number of execution units afterwards according to initial design constraints.

3.2 Scheduling

Once a hardware template is selected, software-pipelined parallel code should be generated to utilize the hardware efficiently.

The basic functions of the Loop scheduler are:

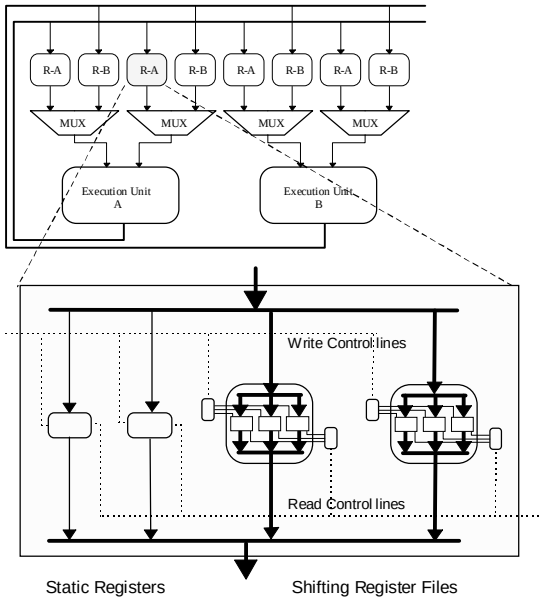
Step 1: Find lower bounds on T_{ii}. Choose maximum value.
Step 2: Initial scheduling
Step 3: Use initial scheduling and inter-iteration dependency cycles (from lower bounds).
Step 4: Check for resource conflicts.
Step 5: Local Rescheduling allow initiation of succeeding iterations at every T_{ii} without any resource or data contentions.

After one iteration is scheduled, *kernel*, *pre-loop*, and *post-loop* parts are constructed. *Pre-loop* is executed one time before *kernel*, generating a new compacted loop body. *Post-loop* part is executed also once after several repetitions of kernel.

One lower bound on **iteration initiation interval** T_{ii} is the maximum number of operations on any execution unit (resource limitation), another lower bound is the length of the largest cycle (inter-iteration dependency limitation). Minimal T_{ii} is the maximum of these two lower bounds. If the initial scheduling does not yield the minimal T_{ii} due to a cycle, operations belonging to that cycle are moved up or down to make them closer to each other so that theoretical minimal T_{ii} is achieved. After a minimal T_{ii} is found and justified on the initial scheduling, the remaining steps are completed, if possible. If not, T_{ii} is incremented until a proper scheduling is found.

[1] A new value is generated in each iteration for loop-variant variables, such as *recurrences*.
[2] Loop-invariant variables, such as constants or base addresses, are never modified during loop execution.

Figure 2: Architecture Model for 2 execution units



3.3 Data dependencies

Data dependencies are classified into two groups: *intra-iteration dependencies* when data-dependent operations belong to the same iteration, *inter-iteration dependencies* when data-dependent operations belong to successive iterations, such as *recurrences*. An inequality, which should hold for all data dependencies $\mathbf{Op_1 \rightarrow Op_2}$, where Operation $\mathbf{Op_1}$ ($\mathbf{Op_2}$) takes $\mathbf{E_1}$ ($\mathbf{E_2}$) cycles starting at cycle $\mathbf{C_1}$ ($\mathbf{C_2}$) is: $\mathbf{C_2 - C_1 - \mathit{min} + T_{ii} * \mathit{dif} f > 0}$.

When it is an intra-iteration dependency, iteration difference between $\mathbf{Op_1}$ and $\mathbf{Op_2}$, *diff*, becomes 0. Total computation time of the operations between $\mathbf{Op_1}$ and $\mathbf{Op_2}$ is *td*. If it is a true dependency *min*=$\mathbf{E_1}$–1, if anti dependency *min*=1–$\mathbf{E_1}$, if out dependency *min*=$\mathbf{E_1}$–$\mathbf{E_2}$. Note that $\mathbf{(Op_1 \rightarrow Op_2}$, true; *diff*=n) means ($\mathbf{Op_1}$[k] $\rightarrow$ $\mathbf{Op_2}$[k+n], true; *diff*=0) for every k. The two lower bounds for T_{ii} are: a) $T_{ii} \geq$ {number of operations executed in any processor} and b) $T_{ii} \geq$ {*td* / *diff* for all inter-iteration dependencies}.

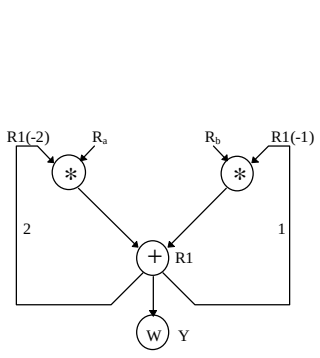
In scientific applications, recurrences (inter-iteration data dependencies) occur very often. This type of dependencies can cause iterations initiate less frequently, which, in turn, means less utilization and less efficiency, i.e., lower performance.

3.4 Dependency removal by Shifting Register Files

Data can be always directly accessed from and written to memory units, or initially accessed from memory units once, thereafter accessed from register files. Figure 3 illustrates a linear recurrence, R1[*i*] = Ra * R1[*i*-2] + Rb * R1[*i*-1], and its two types of scheduling. When data is always accessed from memory, overlapping successive iterations becomes minimal (T_{ii} = *6*). This means we can only start successive iterations at every six cycles, almost after one iteration is finished.

On the other hand, with SRFs, we attain a maximum overlapping (T_{iI} = 2) with a limited hardware cost for SRFs. We can start successive iterations at every second cycle because intermediate data is kept and can be accessed through SRFs instead of from memory modules. Since we allocate a new register set in the same SRF for each iteration, we can concurrently access different sets in the same SRF. For example, R1 belonging to the first iteration can be written while the previous value of R1 is read. If we didn't have SRFs, we would have to wait until the previous value is read to be able to write onto R1. Note that we do not consider the number of execution units in this example.

Figure 3: Example for overlapping



1. Data always accessed from memory modules:

Iteration 1 T_{ii} **= 6**
Read a, R_a
Read b, R_b
Read Y(2), R2
Read Y(1), R1
Mult R2, R_a, RA
Mult R1, R_b, RB
Add RA, RB, RC
Store RC, **Y(3)**

Iteration 2
Read a, R_a
Read b, R_b
Read **Y(3)**, R3
Read Y(1), R2
Mult R3, R_a, RA
Mult R2, R_b, RB
Add RA, RB, RC

2. Data accessed through register files (SRFs).

Overhead T_{ii} **= 2**
Read a, R_a
Read b, R_b
Store #y1, R1(-1)
Store #y0, R2(-2)

Iteration 1
Mult R_a, R1(-2), Rm
Mult R_b, R1(-1), Rn
Add Rm, Rn, **R1(0)**
Store R1(0), Y(3)

Iteration 2
Mult R_a, R1(-1), Rm
Mult R_b, **R1(0)**, Rn
Add Rm, Rn, **R1(1)**
Store R1(0), Y(4)

Iteration 3
Mult R_a, **R1(0)**, Rm
Mult R_b, **R1(1)**, Rn
Add Rm, Rn, R1(2)
Store R1(0), Y(5)

3.5 Expansion Scheduling

We have developed a scheduling technique, *Expansion Scheduling*, which alleviates the problems of Rotation Scheduling, thus finds a proper scheduling including all three parts, *pre-loop*, *kernel*, and *post-loop*, in a simpler way.

We use the initial scheduling from our scheduler to find inter-iteration dependency hazards and their types, which could be cyclic or not. Then we can determine a minimal T_{ii}. As explained above, we schedule operations in the order of their execution times. Operations remain at their time slots and execution units as long as there is no resource or data contention. If resource contention is the only problem for operation **n**, we try to find another execution unit for **n** without incrementing its starting cycle. If there are also data contentions, we increment starting time cycle of **n**, which we call as *shifting down* operation **n**. All the succeeding (children) operations causing intra-iteration data dependency hazards also need to be shifted down recursively to cover all successive operations. In case of a possible conflict due to an inter-iteration dependency, resource conflicting operations are shifted down. If none of these shifts works, then we try to shift **n** back to clear data and resource contentions. If we cannot find a solution, then we increment T_{ii} and try expansion again. However, our experiments show that we can obtain a scheduling for minimal T_{ii} most of the time.

An example is given in Table 1 to illustrate the expansion concepts. This example is excerpted from [2]. Operations with a '-' sign are inter-iteration dependent operations, e.g. operation **1** uses the output of operation **7** generated in the previous iteration.

Table 1: An example

	Operations										
	1	2	3	4	5	6	7	8	9	10	11
Operands	**11**	**11**	**11**	**1**	**3**	**4**	**5**	**11**	**11**	**8**	**-9**
	-7	**-9**	**-10**	**2**	–	**-7**	**6**	**-7**	**-9**	**-10**	–

Table 2: Initial Scheduling and Final Kernel

Initial Scheduling

time	Execution Units +	*
0	--	**11**
1	**2**	**9**
2	**1**	--
3	**4**	--
4	**3**	**6**
5	**5**	--
6	**8**	**7**
7	--	**10**

New loop body

time	Execution Units +	*
0	**5(i-1)**	**11(i)**
1	**8(i-1)**	**7(i-1)**
2	**2(i)**	**9(i)**
3	**1(i)**	**10(i-1)**
4	**4(i)**	--
5	**3(i)**	**6(i)**

Table 3: Instances

		Instance 1				**Instance 2**				**Instance 3**				**Instance 4**			
		1.iter.		2.iter.		1.iter.		2.iter.		1.iter.		2.iter.		1.iter.		2.iter.	
time	mod	P1	P2			P1	P2			P1	P2			P1	P2		
0	0	-	11			-	-			-	11			-	11		
1	1	2	9			-	11			-	-			-	-		
2	2	1	-			2	9			2	9			2	9		
3	3	4	-			1	-			1	-			1	-		
4	4	3	6			4	-			4	-			4	-		
5	5	5	-	P1	P2	3	6	P1	P2	3	6	P1	P2	3	6	P1	P2
6	0	8	7	-	11	5	-	-	-	5	-	-	11	5	-	-	11
7	1	-	10	2	9	8	7	-	11	8	7	-	-	8	7	-	-
8	2	-	-	1	-	-	10	2	9	-	10	2	9	-	-	2	9
9	3	-	-	4	-	-	-	1	-			1	-	-	10	1	-
10	4	-	-	3	6	-	-	4	-			4	-	-	-	4	-
11	5	-	-	5	-	-	-	3	6			3	6	-	-	3	6
12	0			8	7			5	-			5	-			5	-
13	1			-	10			8	7			8	7			8	7
14	2			-	-			-	10			-	10			-	-
15	3			-	-			-	-							-	10
16	4			-	-			-	-							-	-
17	5			-	-			-	-							-	-

One lower bound for T_{ii} based on resources is 6 (ceiling of 11/2). Another lower bound based on inter-iteration dependencies caused by cyclic dependencies is 4 (length of the largest cycle that is between 1 and 7, containing operations 1,4,6,7) in terms of time steps. Thus minimal T_{ii} becomes 6. Table 2 shows the initial scheduling for two execution units.

Four instances of expansion are shown in Table 3. Blocks represent modular structures. Two attached blocks illustrate overlapping and possible conflicts. Region from time cycle 6 to time cycle 11 is the kernel. For example, the final 4th instance shows that while operation 5 of the first iteration is executed by unit 1, operation 11 of the second iteration is executed by unit 2 concurrently. From cycle 0 to cycle 5 is pre-loop, from cycle 12 to cycle 17 is post-loop. The first instance is the initial scheduling, in which resource conflicting pairs of operations are shaded, such are 7 and 11, 10 and 9. Operation 11 is *shifted down*, therefore all the related operations, which are all the operations in this case, are also *shifted down*, this is shown in the second instance. Since it did not resolve any conflicts, operation 11 is *shifted up*, which resolves one conflict, this is shown in the third instance. In the last instance, operation 10 is *shifted down* to resolve the last conflict. After four instances, we obtained a final scheduling for the minimal T_{ii} (= 6). Table 2 shows the new loop body.

4. Register File Organization

Register allocation in scheduled loops is to allocate the registers in a way that minimizes register idle time. In [5], Rau and et al present various heuristics for register allocation for software pipelined loops. They use one large register file for all the variables, and offer various strategies for binding variables to registers. But their strategies are hard to implement, and have high computational complexity.

On the other hand, we allocate one dedicated SRF for each variable. By having separate SRFs, we can decode the register references at run time by dedicated hardware decoders instead of selecting proper registers at compile time, which is a time consuming process.

At the software development stage we assumed that we had infinite number of *shifting* and *static* register files, however, in reality, just a limited number of register files is enough to achieve the same performance. We have considered various register selection alternatives ranging from having a full degree of register duplication wherever necessary and no duplication at all (just single register files). Also we may have dedicated buses to each register file, or have just a single bus to serve all register files. After being selected and assigned onto buses, register files should be bound to variables; and patterns of register read and write should be extracted to construct control path. Register assignment onto buses and degree of register duplication are two factors determining the ratio of *performance/cost*. Higher degree of register duplication results in less *data communication* time (i.e., higher performance), and higher cost.

We have developed a *variable-register binding* schema to allocate variables onto a set of optimum number of SRFs; and also a SRF allocation technique, which generates *register access patterns*, and *control path*. As explained above, we allocate *loop-variants* onto dedicated SRFs to preserve the subsequent values. A loop-variant, *V*, having a life range from StartTime to EndTime must be allocated onto a SRF, having $\left\lceil (EndTime - StartTime + 1) / T_{ii} \right\rceil$ number of registers, called *register_count*.

For consistency, only $\mathbf{Mod}_{\mathbf{register\_count}}$ **N**th register can be updated during the **N**th iteration. Thus once a value of *V* is updated and written onto a register of its SRF, it will be available during its life range.

Table 4: DO-loop example

```
do i = 1,35
        s = s + a[i]
        a[i] = s * s * a[i]
end do
```

An example of a DO-loop code (partly adopted from [5]) and its scheduling including starting times of operations and life ranges of variables are shown in Table 4 and Table 5.

Table 5: Scheduling for the DO-loop

time	Operations		Life Ranges
0	**A** V1 = Mem[V2 (–1)]	Unit 1	V1: 0 - 19
	B V2 = V2 (–1) + V6	Unit 2	V2: 0 - 22
13	**C** V3 = V1 + V3 (–1)		V3: 13 - 16
15	**D** V4 = V3 * V3		V4: 15 - 19
18	**E** V5 = V4 * V1		V5: 18 - 20
20	**F** Mem[V2 (–1)] = V5		

Each operation takes one cycle to finish. Note that T_{ii} is found as 2. The code has five *loop-variants* V1, V2, V3, V4, and V5; and one *loop-invariant* V6, which holds the address increase value (=4) for V2. SRFs dedicated to variables V1, V2, V3, V4, and V5 are R1, R2, R3, R4, and R5 respectively; and these SRFs must have 10, 12, 2, 3, and 2 registers respectively, according to the formula for *register_count*. For example, variable V1 is allocated onto SRF R1, which must have (19-0+1)/2=10 registers so that V1's values will be kept for 20 cycles, throughout its life range. V6 is allocated onto a single static register, R6.

Table 6: Kernel

	P1	**P2**	**P3**	**P4**	**P5**	**P6**
0	**A** (n+10)	**B** (n+10)			**E** (n+1)	**F** (n)
1			**C** (n+4)	**D** (n+3)		

After expansion scheduling, kernel of the new compacted code is formed as shown in Table 6. All the operations are distributed over separate execution units, i.e. over six execution units. As explained above, kernel contains operations from different iterations. For example, operation A of the tenth iteration and operation E of the first iteration are concurrently executed. Register references are found as explained above. For example, R1 refers to 0th register of its SRF during the 10th iteration. For brevity, the pre-loop and the post-loop parts are omitted.

Table 7: SRF usage for 11 iterations

	R3		R5		R4			R1										R2										
t	0	1	0	1	0	1	2	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	10	11
0								**0**										**0**										
1								**0**										**0**										
2								**0**	1									**0**	1									
3								**0**	1									**0**	1									
4								**0**	1	2								**0**	1	2								
5								**0**	1	2								**0**	1	2								
6								**0**	1	2	3							**0**	1	2	3							
7								**0**	1	2	3							**0**	1	2	3							
8								**0**	1	2	3	4						**0**	1	2	3	4						
9								**0**	1	2	3	4						**0**	1	2	3	4						
10								**0**	1	2	3	4	5					**0**	1	2	3	4	5					
11								**0**	1	2	3	4	5					**0**	1	2	3	4	5					
12								**0**	1	2	3	4	5	6				**0**	1	2	3	4	5	6				
13	**0**							**0**	1	2	3	4	5	6				**0**	1	2	3	4	5	6				
14	**0**							**0**	1	2	3	4	5	6	7			**0**	1	2	3	4	5	6	7			
15	**0**	1			**0**			**0**	1	2	3	4	5	6	7			**0**	1	2	3	4	5	6	7			
16	**0**	1			**0**			**0**	1	2	3	4	5	6	7	8		**0**	1	2	3	4	5	6	7	8		
17	2	1			**0**	1		**0**	1	2	3	4	5	6	7	8		**0**	1	2	3	4	5	6	7	8		
18	2	1	**0**		**0**	1		**0**	1	2	3	4	5	6	7	8	9	**0**	1	2	3	4	5	6	7	8	9	
19	2	3	**0**		**0**	1	2	**0**	1	2	3	4	5	6	7	8	9	**0**	1	2	3	4	5	6	7	8	9	
20	2	3	**0**	1		1	2	10	1	2	3	4	5	6	7	8	9	**0**	1	2	3	4	5	6	7	8	9	10
21	4	3		1	3	1	2	10	1	2	3	4	5	6	7	8	9	**0**	1	2	3	4	5	6	7	8	9	10
22	4	3	2	1	3		2	10	11	2	3	4	5	6	7	8	9	**0**	1	2	3	4	5	6	7	8	9	10	11
23	4	5	2		3	4	2	10	11	2	3	4	5	6	7	8	9		1	2	3	4	5	6	7	8	9	10	11

Table 7 illustrates the overall register allocations and life ranges for 23 time cycles. First line shows the SRFs R3, R5, R4, R1, and R2. Second line shows internal registers of these SRFs. Leftmost vertical line represents time cycles. Note that registers used for **1**st iteration are represented as **1'**s, and so on. Therefore even during the 11th iteration, SRF R2 keeps its value from the first iteration. It is easy to see, from this table, that registers are read and written periodically, which form patterns. As it will be explained below, simple control units can handle these periodic register activation.

5. Implementation and Datapath

Execution Unit Clustering. As seen in Table *6*, utilization rate of execution units is just %50. To achieve the same performance (T_{ii} = 2) with less number of execution units, operations must be clustered in a way that neither register conflicts nor execution unit conflicts will occur. In our example, we clustered E and C, and A and B, as seen in Table 8. Since operations E and C have V1 as a common input, they can share the same SRF for V1.

Table 8: New compacted loop body

	P1	P2	P3	P4
0	**A** (n+10)	**B** (n+10)	**E** (n+1)	**F** (n)
1	**D** (n+3)		**C** (n+1)	

Note that four operations in the first line, and two operations in the second line are two VLIW-type of instructions. In other words, each instruction contains several operations, each of which controls separate hardware units concurrently.

Memory allocation. In our example, A and F have concurrent accesses to/from memory (A reads while F writes). Since we cannot read from and write to the same unit concurrently, we need to write the results into a separate memory unit to prevent data conflicts.

Data Communication. Data is transferred internally from producing operation to consuming operation if both operations are executed in the same execution unit; otherwise it is transferred externally. **Internal data transfers** can be realized by *bypassing* and *internal forwarding* methods without any explicit scheduling. **External data transfers** can be realized either via *duplicated register files* or by *explicitly scheduled data transfer operations* in order to prevent possible bus contentions caused by bus limitations. In case of communication via register files, we need no explicit bus cycles.

Register Files. There is a range from having no register file duplications to having optimally maximum register file duplications. Implementations with no duplications at all yields a *lower bound* on both cost (due to less register files) and performance (due to explicitly scheduled bus cycles). On the other hand, implementations with optimally maximum duplications, which emulates a full-crossbar among execution units and register files, yields an *upper bound* on performance with a reasonably high cost. In this case, each register file has a bus attached to it, therefore there is no need to schedule busses explicitly.

Implementation. Figure 4 illustrates a crossbar connection emulating a full crossbar. This architecture can be implemented in a number of ways depending on the design constraints, which are *system response time* and *implementation cost.*

A data path implementation of this crossbar is shown in Figure 5. There are four execution units, and two operand buffers for each of them. Two-to-one multiplexers let execution units receive correct data. SRF R2 is duplicated three times, R3 twice. Other SRFs R1, R4, and R5 are not duplicated. Total number of registers dedicated for loop-variants is 55. An additional single register is dedicated for loop-invariant V6.

Figure 4: Architecture

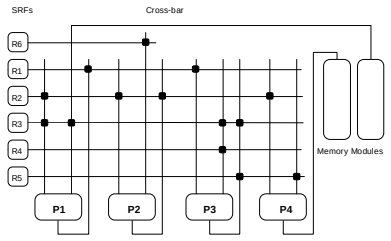
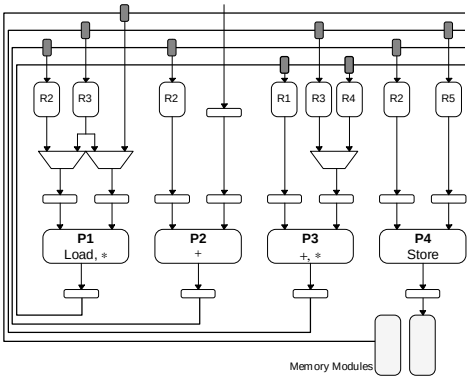


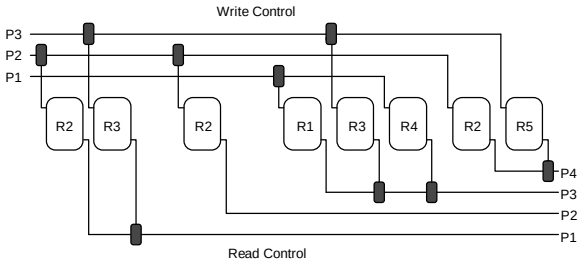
Figure 5: Optimum implementation



Controlpath. Figure 6 shows the control path for the design example. There are eight register files to control, some of which are duplicates. Duplicate SRFs emulate multi-ported memories.

Functional units can only read from SRFs that are dedicated to their own input. For example, P3 reads from SRFs R1, R3, R4 dedicated to itself only but writes into all duplicates of SRF R3 as well as SRF R5, which are read by only P4. In other words, the same value can be concurrently read by different execution units by means of duplicate SRFs. However all duplicates of a SRF are updated concurrently by a single execution unit. For example, duplicates of SRF R2 is read concurrently by P1 and P2. There are two separate control lines for *write control* and *read control* to allow concurrent reading(writing) from(to) separate registers in the same register file. Thus, within the same SRF, one register can be read while another one is being written.

Figure 6: SRF control bus



Implementation of register read/write patterns can be realized by a simple counter or by a PLA. For example, for P2, a simple counter counting up to 12 is enough to activate proper registers of SRF R2. But P3 needs a PLA since it reads from different registers of SRF R1. In fact, P3 needs two dedicated control units to determine proper registers of SRFs R5 and R3. As a result, every functional unit must have certain dedicated units to activate proper registers.

6. Results

We have implemented the algorithms on a Sparc 2 workstation in C programming language. We have run the program for many benchmarks, four of which are shown in Table 9. Experiments are finished only within seconds.

Table 9: Applications

		number of operations	length of largest cycle
1	5th order elliptic filter	43	19
2	Bandpass	42	13
3	AR Filter	54	0
4	Differential Equation	16	9

In the following graphs in Figure 7, lower lines show the resulting T_{ii} values for different number of execution units when expansion scheduling is applied. Succeeding iterations can start at every T_{ii} cycle without causing any data hazards or resource contentions. Upper lines show execution completion time of one iteration when we do not overlap succeeding iterations. Thus we can compare the effect of overlapping.

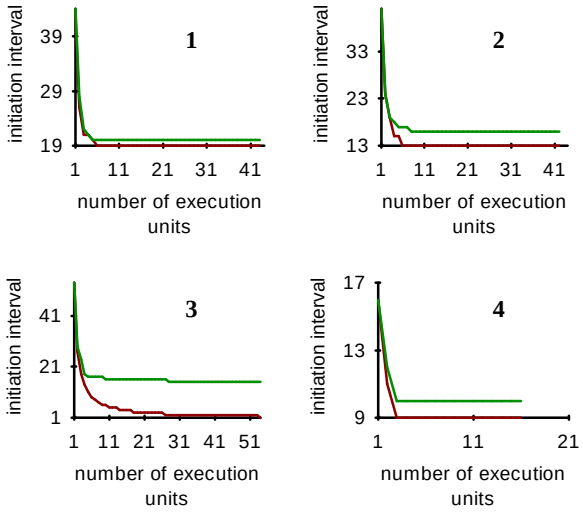
Note that we have obtained maximum throughput for the third application. The reason is that our scheduler finds optimum scheduling for DFGs with no cyclic inter-iteration dependencies. Yet even for those with cyclic inter-iteration dependencies, T_{ii} values eventually converge to the best possible values as well.

To see the effect of overlapping in terms of cycles, lets consider the third application running on a processor with 6 execution units. When overlapping is applied, minimal T_{ii} (=9) is achieved. When overlapping is not applied, it can only be 17. Assume that we run this kernel 1,000 times. In the former case, it finishes roughly in 9,000 cycles plus time for pre-loop and post-loop parts to finish. In the latter case, it finishes in 17,000 cycles. For 18 execution units, T_{ii} becomes 3, when overlapped, 16 when not overlapped. This means 3,000 cycles instead of 16,000 cycles for a thousand iterations when Expansion Scheduling is applied with a cost of 18 execution units. We can also see the optimum number of execution units from the graphs. For example, 6 execution units for the first application is enough to attain maximum performance/cost ratio.

Size of Register Files. The number of registers required for the first application is 55, for the second 50, and for the forth 18. For the third application, it varies from 56 to 226 due to high degree of overlapping while T_{ii} converges to 1.

Figure 7: Initiation intervals

Upper lines: scheduling without *Expansion Scheduling*
Lower lines: scheduling with *Expansion Scheduling*
Minimal T_{ii} values are placed on the center.



7. Conclusion and future work

In this paper, we presented a methodology for application specific processor synthesis, addressed the problem of scheduling recurrences and suggested a novel scheduling method for scheduling iterative computations for resource-constraint architectures. Our scheduling method is based on extensive use of register files of two types: static and Shifting Register Files. We also presented a comprehensive register file organization schema to achieve minimal number of register files. We have shown that patterns of register accesses can be easily obtained, and generated at run time by simple controllers such as counters.

We have also completed a schema to distribute array variables onto memory modules; and designed an execution unit template using Compass Design Automation Tool. We plan to work on automatic generation of, VHDL description of, resulting processor for a given architecture.

References

1. Breternitz M, Shen J. *Architecture synthesis of high-performance application-specific processors.* DAC 1990.
2. Chao L, LaPaugh A. *Rotation Scheduling: A Loop Pipelining Algorithm.* 30th Design Automation Conference. 1993.
3. Johnson M. *Superscalar microprocessor design.* Prentice Hall. 1991.
4. Lala P. *Digital System Design Using Programmable Logic Devices.* Prentice Hall, 1991.
5. Rau B, Lee M, Tirumalai P, Schlansker M. *Register Allocation for Software Pipelined Loops.* Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, SIGPLAN Notices, July 1992.
6. Rau B, Schlandker M, Tirumalai P. *Code Generation Schema for Modulo Scheduled Loops.* MICRO 1992.
7. Valle, M. et al. *A VHDL-based Design Methodology: the Design Experience of an High Performance ASIC Chip.* EURO-DAC'94, 1994.
8. Woundsma R, Meerbergen J. *Consumer Applications: A Driving Force for High-Level Synthesis of Signal-Processing Architectures.* IEEE Micro. Aug 1992.