Micro-Preemption Synthesis: An Enabling Mechanism for Multi-Task VLSI Systems [*]

Kyosun Kim, Ramesh Karri
Department of ECE
University of Massachusetts
Amherst, MA 01002
{karri,kkim}@ecs.umass.edu

Miodrag Potkonjak
Department of Computer Science
University of California
Los Angeles, CA 90095
miodrag@cs.ucla.edu

Abstract - *Task preemption is a critical enabling mechanism in multi-task VLSI systems. On preemption, data in the register files must be preserved in order for the task to be resumed. This entails extra memory to save the context and additional clock cycles to restore the context. In this paper, we present techniques and algorithms to incorporate micro-preemption constraints during multi-task VLSI system synthesis. Specifically, we have developed: (i) algorithms to insert and refine preemption points in scheduled task graphs subject to preemption latency constraints, (ii) techniques to minimize the context switch overhead by considering the dedicated registers required to save the state of a task on preemption and the shared registers required to save the remaining values in the tasks, and (iii) a controller based scheme to preclude preemption related performance degradation.*

1 Introduction

1.1 Motivation

Task preemption is a critical enabling mechanism in a variety of application scenarios. Hard real-time computer systems have stringent timing requirements. In such systems, the deadlines for critical tasks are enforced by preempting less critical tasks. In soft real-time systems where infrequent deadline violations are tolerated, less important tasks are preempted to execute more important ones so as to meet some quality-of-service requirements. For example, in multimedia systems, video, voice, and data streams are scheduled and occasionally interrupted and resumed according to priority strategies to enforce soft end-to-end deadlines.
Along a different dimension, multi-task VLSI systems are becoming commonplace. For example, Motorola offers numerous DSP ASPPs [3]. An ASPP can be dynamically configured to one of the implemented tasks. Although the reconfiguration time of an ASPP may be very low (because reconfiguration entails moving from the final state of the current task to the start state of another task), it may not be acceptable for a critical task in a real-time system to wait until the current task is completed.

[*]This research was supported by an NSF CAREER grant MIP-9702676

On receiving a preemption request, the state of the active task must be saved, the context of the new task must be loaded and then executed. On completion of task execution, the state of the preempted task must be restored and the interrupted task resumed to completion. Important factors that should be considered while implementing task preemption include:

- **Preemption latency**: It is defined as the maximum time it takes from the instant a preemption request is received to the instant the task state is saved.
- **Context switch cost:** Hardware overhead incurred by the installation of a preemption handling scheme must be considered. A saved state should contain only enough information (and no more) so that the preempted task can be resumed at the precise point where it was interrupted. The task state should consist of the contents of the general purpose registers, the condition registers, and the relevant portion of background memory.
- **Performance degradation**: There are two main sources of performance degradation: (i) on a preemption request, some task states that have already been executed may be aborted. Retracing these aborted states adds to the finish time of the aborted task. (ii) Any scheme that saves the context of a preempted task in background memory may stall execution units.

In this paper we will present a systematic methodology for incorporating preemption constraints in multi-task VLSI systems. Specifically, we will show how **context switch cost** and **performance degradation** can be minimized while satisfying task specific throughput and **preemption latency** constraints.

1.2 A Motivating Example

Consider a system implementing two tasks A and B. Task A takes four clock cycles and task B takes six clock cycles for one iteration. Let A1, A2, A3, and A4 denote parts of task A that are executed in the first, second, third and fourth clock cycles respectively. Similarly, let B1, B2, B3, B4, B5 and B6 denote parts of task B that are executed in the first, second, third, fourth, fifth and sixth clock cycles respectively. The following assumptions were made while designing the micro-preemption controller.

1. A non-overlapping two-phase clock (CLK 1, CLK 2).
2. Activation of a new task (i.e. changing the SELECTED TASK signal) and transition of a task from one state to the next are synchronized with falling edge of CLK 2.
3. Setting and resetting of PREEMPT MASK is synchronized with falling edge of CLK 1.
4. A task preemption request is serviced when the PREEMPT MASK is low and SELECTED TASK is high and CLK 2 is falling.

A simulation snapshot of micro-preemption in the two-task VLSI system is shown in figure 1[1]. To minimize the controller and context switch overhead, we mandate that task A can be preempted in STATEs A1 and A3 alone. These are called **task preemption points**. Initially the system is in STATE A4. When the system goes to STATE A1, execution of a new task is requested by setting the SELECTED TASK to B and the data inputs to appropriate data values (these have not been shown here for simplicity). However, it is not a valid preemption request (since PREEMPT MASK is high). Even when a valid preemption request arrives in STATE A2 (i.e. all conditions in item 4 are satisfied) task A is not aborted immediately. Rather, the computation **rolls forward** to end of state A3 (the next preemption point) before the preemption request is serviced. Notice that it has taken two clock cycles from the time a valid preemption request arrived (beginning of STATE A2) to the time the new task (task B) became active (end of STATE A3). From the point of view of task B, this is its **preemption latency**. From the point of view of the multi-task VLSI system as a whole (and task A in particular) rolling forward of the computation has eliminated the performance degradation due to an immediate abort. In a nutshell, preemption points A1 and A3 have partitioned the execution of task A into two **critical sections** {A2, A3} and {A4, A1}. Similarly, **task preemption points** B2, B4 and B6 for task B partition it into three critical sections {B1, B2}, {B3, B4}, and {B5, B6}.

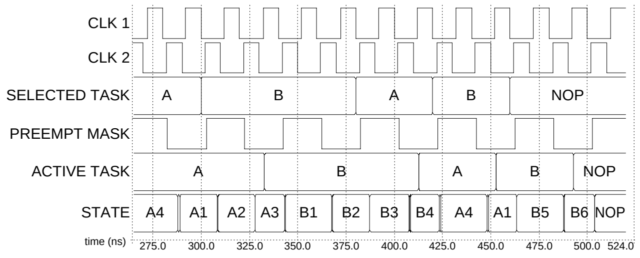


Figure 1: Simulation snapshot showing preemption request and servicing mechanism in a two-task VLSI system

[1]the controller has been implemented using 1μ SCMOS standard cell library and simulated using IRSIM.

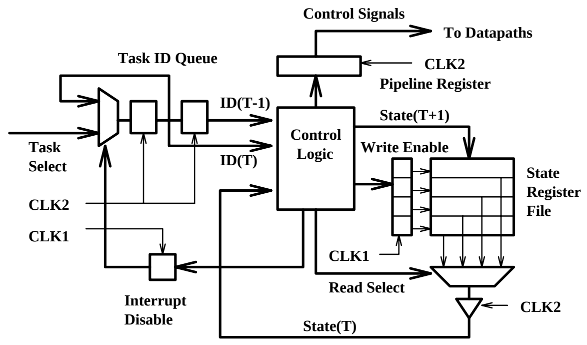


Figure 2: Controller for a multi-task VLSI system supporting micro-preemption

The controller is shown in figure 2. It is a collection of finite state machines (one for each implemented task) and has a state register file that holds the identification of the currently active task. At every clock cycle, a different task can be initiated or a preempted task resumed by the task select signal. The controller signals are pipelined so that the controller delay does not affect the critical path.
The rest of the paper is organized in the following way. We first briefly survey the related work along several dimensions. Next, we will discuss the computational and hardware models. In sections 3 and 4, we introduce our approach, formulate the micro-preemption synthesis problems, and describe the proposed algorithms for micro-preemption synthesis. Experimental results are presented in section 5. In section 6, we conclude by summarizing the results.

1.3 Related Research

Reconfigurable computing platforms are attracting a lot of attention recently. A fast growing billion dollar Field Programmable Gate Array (FPGA) industry is supported by a number of commercial and research tools [12]. A number of special purpose reconfigurable computers have been built. Early work in this direction includes the systems realized at University of Texas, Austin (TRAC) [5]. The Splash system enables reconfigurability to more than 100 different configurations which are well suited for several computational tasks in molecular biology [7]. Several generations of data path reconfigurable video-processors with accompanying compilation support have been developed at University of California, Berkeley [13]. Recently, Application Specific Programmable Processors (ASPPs) [16] have been introduced as an excellent candidate for multifunctional datapaths with frequent context switching. Though their functionalities must be determined in the design phase, a single ASPP implementing multiple functions obtains significant area savings when compared with the dedicated ASIC implementations of the functions.
Research in implementing interrupts is outlined next. The IBM 360/91 supports precise and imprecise interrupt handling [1]. Hwu and Patt [2] proposed a checkpointing approach to handling interrupts. The checkpoints (which incur some penalty in processor

performance) are used to divide the sequential instruction stream into smaller units to reduce the cost of resumption. Sohi [9] integrated the functions of reservation stations and reorder buffers into the register update unit to realize precise interrupts. In addition, Smith and Pleszkun [6] presented architectural solutions such as saving the intermediate state of vector instructions and saving a sequence of instructions that must be executed before saving the program counter. Mosberger et. al. [15] presented a software-only solution to the synchronization problem in uniprocessors. Their idea was to execute atomic sequences without any hardware protection, and to roll the sequence forward to the end, thereby preserving atomicity.

Behavioral synthesis has been active research area for more than two decades [8], and numerous outstanding systems have been built targeting both data path oriented and control oriented applications [8]. Synthesis systems that optimize power, testability and fault-tolerance [14] have been developed.

2 Computational/Hardware Models

Our computational model for a single task is homogeneous synchronous data flow [4]. Within this model, a task is represented as a hierarchical Control Data Flow Graph $G(N, E, T)$ (or CDFG), with nodes N representing the flow graph operations, and the edges E and T respectively the data and timing dependences between the operations.

In modern designs a variety of register file models have been used [10]. From among them we have selected the dedicated register file hardware model. This model clusters all registers in register files and each file is then connected only to the inputs of the corresponding execution units. An important benefit of the chosen hardware model is that it reduces the interconnect at the expense of additional registers.

3 Issues and Our Approach

On preemption, the data in the register files must be preserved somewhere in order for a task to resume. In general purpose microprocessors, these values are transferred to background memory before an interrupt is serviced. This technique is not acceptable in multi-task VLSI systems due to the attendant performance penalty. Alternately, a register windowing technique is used in the Sparc architecture [11]. In this scheme, data is saved in registers within the processor even when a new computation environment is required. However, it entails non-negligible area overheads for duplicated registers. In contrast, we propose an intuitively simple technique by classifying the edges in the CDFG and the registers hold them into two groups:

- **Dedicated registers**[2] (R_d^t) store the values of edges of a task that straddle preemption points. These edges that straddle a preemption point are called the **red** edges, and represent intermediate values essential to resume the task if preempted.
- **Shared registers** (R_s^t) are shared by the values associated with the remaining edges (of all tasks) in the system. These edges that do not straddle a preemption point are called the **green** edges. The dedicated registers of a task can also be used to store the values associated with the green edges in the task. However, the shared registers cannot be assigned to red edges.

[2]**coefficient registers** (R_c^t) that hold the constants used by the task are not targeted during the context switch optimization. This is because, generally these constants differ from one task to another and cannot share registers.

Since dedicated registers cannot be shared between tasks, the associated context switch overhead is the sum of the dedicated registers over all tasks. On the other hand, the context switch overhead due to shared registers is the maximum value across all tasks. Overall, the context switch cost of a multi-task VLSI system with task set T is:

$$| R | = \sum_{t \in T} | R_d^t | + \max_{t \in T} | R_s^t | \qquad (1)$$

Performance degradation resulting from aborting a task is eliminated by (i) partitioning the task states into **critical sections**, (ii) executing critical sections and (iii) preserving atomicity of a critical section by **rolling forward** to the end of a critical section on preemption. This is analogous to the classical approach to **precise interrupts**.

Next, we present algorithms for (i) preemption point insertion and (ii) preemption context synthesis that minimizes the context switch overhead during multi-task VLSI system synthesis. The optimization problem can be defined as follows:

Given an underlying hardware model and N scheduled tasks, each with its own time bound (λ) and maximum preemption latency (τ), insert preemption points, and bind edges to registers, so that the context switch overhead is minimized.

Initially, all tasks are scheduled in an integrated fashion by considering their word length, precision, hardware and topological similarities. Using the number of edges straddling a clock cycle as an estimate of the context switch overhead, preemption points are inserted. The resulting preemption point set for each task will have more than the minimum number of preemption points. In the next step these preemption point sets are refined. Finally, the preemption context is synthesized by binding edges to registers subject to preemption constraints. The output is then passed through hardware mapping and layout generation tools to synthesize a multi-task VLSI system.

3.1 Preemption Point Insertion

Towards investigating preemption point insertion, consider a task with five edges ($e_1, ..., e_5$), an application latency of eight clock cycles and an edge-to-register binding shown in figure 3. The register file has one input port and one output port which are accessed at the first half cycle and the last half cycle, respectively.

The register overhead of a preemption point can be estimated as the number of edges straddling it. For instance, assuming a preemption latency of three yields

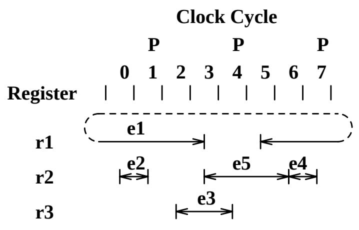


Figure 3: Preemption point insertion and register binding. The dotted line shows a cyclic dependency.

clock cycles 1, 4, and 7 as preemption points as shown in figure 3. The preemption points are marked by 'P'. On preemption point insertion, e_1 and e_5 become red edges and are assigned to two dedicated registers r_1 and r_2. e_3 becomes a green edge and is assigned to shared register r_3. e_2 and e_4 become green edges but are assigned to dedicated register r_2. Initially, preemption points are inserted one task at a time using a polynomial heuristic time algorithm **InsertPreemptionPoints()**. It incrementally inserts preemption points (into each task) such that the number of edges straddling the preemption points is minimized until the preemption latency constraint is satisfied.

3.2 Preemption Point Refinement

Minimizing dedicated registers alone does not reduce the context switch overhead. Instead, it may increase the number of shared registers and hence the total context switch overhead. For example, assume preemption points are inserted at clock cycles 0 and 5. Consider the following scenarios:

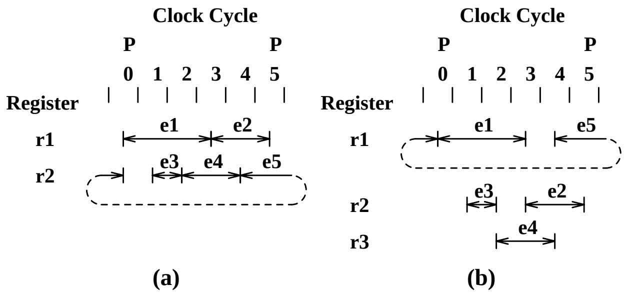


Figure 4: Shared register overhead

- **scenario 1 (figure 4 (a)) :** Red edges e_1 and e_5 are assigned to dedicated registers r_1 and r_2, respectively. This results in two dedicated registers and zero shared registers for the task with a context switch overhead of two registers.
- **scenario 2 (figure 4 (b)):** Red edges e_1 and e_5 are bound to dedicated register r_1. The green edges e_2, e_3 and e_4 are bound to shared registers r_2 and r_3. This results in one dedicated register and two shared registers with a context switch overhead of three registers.

Scenario 1 is superior to **scenario 2** if all other tasks in the system do not require shared registers. **Scenario 2** is superior to **scenario 1** if at least one of the remaining tasks uses more than two shared registers. Based on these observation it is clear that both the shared and dedicated registers must be considered in an integrated manner to optimize the context switch overhead.

```
    RefinePreemptionPoints $(T, P)$ {
1:  for each $t_i \in T$ {
2:    for $(j \leftarrow 0; j <| P |; j{+}{+})$ {
3:      $R[i][j] \leftarrow$ PreemptionContextSynthesis$(E_i, P)$;
4:      $p \leftarrow p_k$ s.t. NumberOfEdges$(p_k)$ is
          $\max_{p_l \in P}$ NumberOfEdges$(p_l)$ and
          MaxPreemptionLatency$(P - \{p_k\}) \le \tau_i$;
5:      if $(p = \phi)$ break;
6:      $P \leftarrow P - \{p\}$; /* Prune a preemption point */
      }
    }
7:  $PCCost_{best} \leftarrow infinite$;
8:  while $((C \leftarrow$ GenerateConfiguration$()) \ne \phi)$ {
9:    $PCCost \leftarrow$ PreemptionContextCost$(R, C)$;
10:   if $(PCCost < PCCost_{best})$
        $PCCost_{best} \leftarrow PCCost$;
    }
  }
```

Figure 5: Algorithm for preemption point refinement - For each task t_i in task set T, E_i is the set of edges, λ_i is the input latency, and τ_i is the preemption latency.

For each task, we start from the preemption point sets generated by the insertion step. We then generate a list of candidate preemption point sets by pruning preemption points with large context switch overhead (steps 2-6 in **RefinePreemptionPoints()** in figure 5). Both dedicated and shared registers are used to compute the context switch overhead. Since the peak usage of shared registers cannot be known a priori, edges are bound to registers (using **PreemptionContextSynthesis()**) to evaluate the context switch overhead, exactly. This pruning technique is possible because for each task, preemption point insertion usually inserts more preemption points than are necessary. Finally, the best preemption point set one for each of the tasks is obtained by using the context switch cost function given by equation 1. This is summarized in steps 7-10.

Consider a multi-task VLSI system implementing three tasks, t_1, t_2, t_3 shown in figure 6. Following the preceding steps, task t_1 has two candidate preemption point sets (PPS) with context switch overheads (CSO) (3, 4) and (2, 5). Similarly, tasks t_2 and t_3 have four and three preemption point sets, respectively. The context switch overhead for each preemption point set is given as the two-tuple, (# of dedicated registers, # of shared registers). Selecting preemption point set 2 for t_1, preemption point set 4 for t_2 and preemption point set 3 for t_3 will result in a context switch cost of (2 + 2 + 1) + max (5, 7, 5) = 12. The context switch cost of selecting preemption point set 2 for t_1, preemption point set 3 for t_2 and preemption point set 2 for t_3 is (2 + 3 + 1) + max (5, 5, 5) = 11. From among the $2 \times 4 \times 3 = 24$ configurations, this has the lowest context switch overhead.

t_1	
PPS	CSO
1	(3, 4)
2	(2, 5)

t_2	
PPS	CSO
1	(5, 4)
2	(4, 5)
3	(3, 5)
4	(2, 7)

t_3	
PPS	CSO
1	(2, 4)
2	(1, 5)
3	(1, 5)

Figure 6: Candidate preemption point sets for tasks t_1, t_2 and t_3

3.3 Preemption Context Synthesis

The optimization problem associated with preemption context binding can be defined as:
Given a scheduled task and a set of preemption points, bind the edges to the registers so that (i) the red edges are bound to dedicated registers, and (ii) the total number of registers is minimized.

```
PreemptionContextSynthesis $(E, P)$ {
  1. **Classify** edges into red and green
  2. **Bind** red edges to dedicated registers
  3. **Bind** green edges to dedicated or shared registers
}
                              (a)
Classify $(E, P)$ {
  $Red \leftarrow \phi$; $Green \leftarrow E$;
  foreach $e \in E$
    foreach $p \in P$
      if (lifetime of $e$ overlaps $p$) {
        $Green \leftarrow Green - \{e\}$; $Red \leftarrow Red \cup \{e\}$; }
}
                              (b)
Bind $(E)$ {
  repeat {
    $e \leftarrow e_k$ s.t. $| e_k.nbr |$ is $\max_{e_i \in E} | e_i.nbr |$;
    $e.reg \leftarrow \min r$ s.t. $r \neq n.reg \; \forall n \in e.nbr$;
  } until $((E \leftarrow E - \{e\}) \neq \phi)$;
}
                              (c)
```

Figure 7: Algorithms for preemption context synthesis

The algorithm, outlined in figure 7, minimizes the number of dedicated registers first and then minimizes the number of shared registers. Initially, the algorithm groups the edges into red and green edges using **Classify ()**. Then the red edges are bound to dedicated registers. Finally, the green edges are bound. The ordering is important since while green edges can be bound to either the dedicated or the shared registers, red edges can only be bound to dedicated registers. A graph coloring heuristic **Bind** () (outlined in figure 7 (c)) is used for binding. The edge with the largest number of bound neighbors (*nbr*) is selected and bound to a register (*reg*) which is not bound to any of its neighbors.

4 Experimental Results

Micro-preemption synthesis techniques proposed in this paper were validated on a set of DSP, video, control and communication applications. The selected applications span a wide range of complexities in computational structures and include Arai's fast DCT algorithm (ARAI), decimate-by-four wave digital filter (DECBY4), four-state linear controller (FSLC), Winograd's DFT for N = 8 (FFT8), digital wavelet transform (WAVELET) and ninth degree bireciprocal WDF with Butterworth response (WDF9). Synthesis modules for hardware mapping and layout generation from HYPER high level synthesis system were used to complete the synthesis trajectory.

4.1 Register Overhead Evaluation

multi-task VLSI system	allocation			# of registers		
	+	-	*	0-p	1-p	all-p
{ARAI, FFT8, WAVELET}	2	2	1	64	65 2%	86 34%
{FIR20, VETT, VOLTERRA}	2	2	2	81	88 9%	105 30%
{DECBY4, FSLC, NC, WANG}	2	3	2	119	134 13%	170 43%
{WDF7, WDF9, WDFB}	2	4	2	62	80 29%	91 47%
{DIF, LDI_LP, WDF5}	2	2	1	40	54 35%	68 70%
{ADAPT, LEE, CASCADE}	2	2	2	57	79 39%	92 61%

Table 1: Register overhead associated with micro-preemption

The results of six multi-task VLSI systems are summarized in table 1. The first column shows the applications implemented in each system. The next three columns summarize the hardware allocation. The last three columns give the number of registers for the case when no preemption points are inserted (0-p), when one preemption point is inserted (1-p), and when preemption points are inserted at all clock cycles (all-p). Using the 0-p case as the base line, the register overhead for the 1-p case varies from 2% to 39%. At the other extreme, the register overhead for the all-p case varies from 30% to 70%.

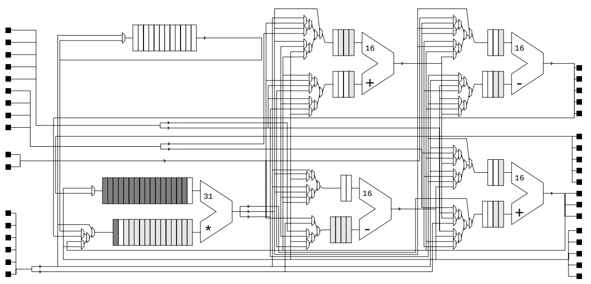
multi-task VLSI system	area (mm^2)		over head
	0-p	all-p	
{ARAI, FFT8, WAVELET}	40.8	43.2	6%
{FIR20, VETT, VOLTERRA}	94.2	98.0	4%
{DECBY4, FSLC, NC, WANG}	84.6	90.6	7%
{WDF7, WDF9, WDFB}	90.9	96.1	6%
{DIF, LDI_LP, WDF5}	28.4	31.4	11%
{ADAPT, CASCADE, LEE}	50.3	54.4	8%

Table 2: Area overhead associated with micro-preemption

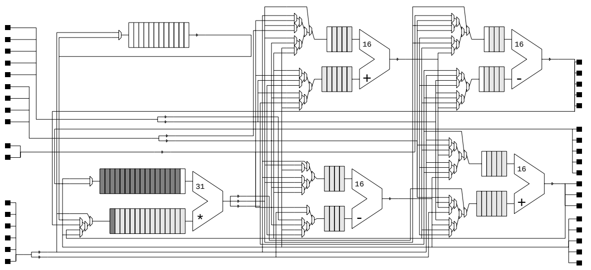
We completed the synthesis trajectory by passing these designs through the hardware mapping and layout synthesis phase. The area overhead for the six designs using actual layouts are summarized in table 2. The areas are reported for the 0-p case and the all-p case. Again using the 0-p case as the basis, the area overhead for the all-p case varies from 4% to 11% as shown in the last column.

When compared to the background memory based schemes, the proposed scheme does not need (i) additional ports of the register files which are used to save/restore data to/from the background memories without stalling currently running task, (ii) additional buses to interconnect the register files to the background memories, and (iii) additional control logic to compute memory addresses. Since the preemption requests can be serviced in parallel without stalling the execution of the tasks, there is no performance penalty.
As the preemption latency increases, the number of dedicated registers decreases, the number of shared registers increases, and the total number of registers decreases monotonously. Finally, the number of coefficient registers is invariant to the preemption latency.

4.2 Register Transfer Level Analysis



(a) preemption latency is 75% of the input latency.



(b) preemption latency is 25% of the input latency.

■ Constant ▨ Dedicated □ Shared

Figure 8: Microarchitecture for the multi-task VLSI system implementing {ARAI, FFT8, WAVELET} for two preemption latencies.

The microarchitecture with preemption points for the multi-task VLSI system implementing {ARAI, FFT8, WAVELET} are shown in figure 8. Registers in the register files are classified as one of three types by using different gray levels as shown in the legend in figure 8. The number of registers increases by 15% (from 73 to 84) as the preemption latency decreases by 50%. However, since there is no interconnect overhead, the increase in the total chip area is very small.

5 Conclusions

We presented techniques and algorithms to incorporate micro-preemption constraints during multi-task VLSI system synthesis. The area overhead of the proposed scheme is under 12%. We have also implemented a controller based scheme to eliminate the performance degradation by (i) partitioning the task states into **critical sections**, (ii) executing critical sections and (iii) preserving atomicity by **rolling forward** to the end of the critical sections on preemption.

References

[1] D.W. Anderson, F.J. Sparacio, and F.M. Tomasulo, "The IBM System/360 Model 91: Machine philosophy and instruction-handling," IBM Journal, 11(1), pp. 8-24, 1967.

[2] W.W. Hwu and Y.N. Patt, "Checkpoint repair for high performance out-of-order execution machines," IEEE Trans Comp (36)12, pp. 1496-1514, 1987.

[3] G.D. Hillman, "DSP56200: An Algorithm-Specific Digital Signal Processor Peripheral", Proc IEEE, 75(9), pp. 1185-1191.

[4] E. A. Lee and D. C. Messerschmitt, "Static Scheduling of Synchronous Data flow Programs for Digital Signal Processing", IEEE Trans on Comp, 36(1), pp. 24-36, 1987

[5] G.J. Lipovsky, M. Malek, "Parallel Computing: Theory and Comparison", John Wiley.

[6] J.E. Smith and A.R. Pleszkun, "Implementing precise interrupts in pipelined processors," IEEE Trans ComP, 37(5), pp. 562-273, May 1988.

[7] M. Gokhale, et al., "SPLASH: A Reconfigurable Linear Logic Array", ICPP, 1990.

[8] M. C. McFarland and A. C. Parker and R. Camposano, "The high-level synthesis of digital systems", Proc IEEE, 78(2), pp. 301-318, 1990.

[9] G.S. Sohi, "Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers," IEEE Trans Comp, 39(3), pp. 349-359, 1990.

[10] J. Rabaey, et al., "Fast Prototyping of Data Path Intensive Architectures," IEEE Design & Test, 8(1), pp. 40-51, 1991

[11] R.J. Baron and L. Higbie, Computer Architecture Case Studies, Addison-Wesley, pp. 234-237, 1992.

[12] A. El Gamal, J. Rose, A. Sangiovanni-Vincentelli, "Synthesis Methods for Field Programmable Gate Arrays", Proc. of IEEE, 81(7), pp. 1013-1029, 1993.

[13] A.K. Yeung, J.M. Rabaey, "A 2.4 GOPS data-driven reconfigurable multiprocessor IC for DSP", ISSCC, pp. 108-109, 1995.

[14] K. Kim, R. Karri and M. Potkonjak, "Heterogeneous Built-In Resiliency of Application Specific Programmable Processors," Proc. of Intl Conf on CAD, pp. 406-411, Nov 1996.

[15] D. Mosberger, P. Druschel and L. Peterson, "Implementing Atomic Sequences on Uniprocessors Using Rollforward," Software-Practice and Experience, 26(1), pp. 1-23, Jan 1996.

[16] K. Kim, R. Karri and M. Potkonjak, "Synthesis of Application Specific Programmable Processors," Proc. of 34th DAC, pp. 353-358, Jun 1997.