Compiler-Assisted Power Optimization for Clustered VLIW Architectures

Rahul Nagpal and Y. N. Srikant

Department of CSA
Indian Institute of Science
Bangalore, India
{*rahul,srikant*}*@csa.iisc.ernet.in*

Abstract

Clustered VLIW architectures solve the scalability problem associated with flat VLIW architectures by partitioning the register file and connecting only a subset of the functional units to a register file. However, inter-cluster communication in clustered architectures leads to increased leakage in functional components and a high number of register accesses. In this paper, we propose compiler scheduling algorithms targeting two previously ignored power-hungry components in clustered VLIW architectures, viz., instruction decoder and register file.

We consider a split decoder design and propose a new energy-aware instruction scheduling algorithm that provides 14.5% and 17.3% benefit in the decoder power consumption on an average over a purely hardware based scheme in the context of 2-clustered and 4-clustered VLIW machines. In the case of register files, we propose two new scheduling algorithms that exploit limited register snooping capability to reduce extra register file accesses. The proposed algorithms reduce register file power consumption on an average by 6.85% and 11.90% (10.39% and 17.78%) respectively, along with performance improvement of 4.81% and 5.34% (9.39% and 11.16%) over a traditional greedy algorithm for 2-Clustered (4-Clustered) VLIW machine.

1 Introduction

VLIW and clustered VLIW style of architectures are in widespread use in embedded devices[40][11]. VLIW architectures[12] leverage the large number of functional units connected to a large unified register file to exploit the explicit instruction level parallelism in embedded applications in order to successfully achieve the desired performance. However, operating a large number of functional units in parallel demands more ports. This worsen the power and delay problems in VLIW

architectures because for N arithmetic units area of register file grows as N^3, delay as $N^{3/2}$, and power dissipation as N^3 [36]. Clustered VLIW architectures[10] solve this scalability problem by partitioning the register file and connecting only a subset of the functional units to a register file. Inter-cluster communication is enabled by a point-to-point or a bus-based mechanism. Texas Instrument's VelociTI[40], HP/ST's Lx[11], Analog's TigerSHARC[14], and BOPS' ManArray[34] are examples of the recent commercial micro-architectures developed based on clustered ILP philosophy. IBM's eLite[7] is a research proposal for a novel clustered architecture.

The simplification of issue logic and trends towards smaller caches in embedded VLIW architectures leads to a significant fraction of the total power consumption taking place in functional units, instruction decoder, and register file. The contentions for the limited number of slow inter-cluster communication channels in the context of clustered VLIW architectures introduce many short idle cycles. This in turn leads to higher power consumption in clustered VLIW architectures because of the increase in the leakage component of power that dominates power consumption in the current (65nm) and future technologies. In the past, power optimization techniques have mostly focused on functional units. However, little work has been done on power optimization of another two major source of power consumption, namely, functional units and register file.

The instruction decoder is a well known source of power dissipation in superscalar architectures[28][23]. It is no surprise that its contribution to the overall processor power will be even higher in the context of contemporary wide issue VLIW and clustered VLIW architectures that demand facility to decode up to 8 (or more) instructions in parallel every cycle. Frequent access to the instruction decoder raises the temperature level and makes the leakage power even worse[29]. Thus, optimizing leakage power in instruction decoders is becoming more important by each process generation. A study in the context of Texas instruments' VelociTI architecture[38] attributes more than 50% of energy consumption to instruction fetch and decoding activity[6]. Though, the exact percentage may depend on the architecture and circuit details, earlier studies clearly indicate that 20% to 25% of the leakage power consumption in a VLIW architecture is attributed to instruction decoder. Instruction decoder is also a well known hotspot in VLIW chips and an aggressive power optimization for instruction decoder is important from that perspective as well[29].

The register file is another major source of power consumption in microprocessors[3][16][37]. For example, in Motorola M.CORE architecture[37], the register file consumes 16% of total processor power which is as high as 40% of the total datapath power[16]. In the context of embedded applications, register file power consumption is even higher and is shown to be up to 25% of the total processor power[3]. This huge power dissipation in a relatively small area occupied by the register file leads to significantly higher power densities. The register file has been clearly observed as a hot-spot in many studies with the temperature rising as high

as 115° C[29]. The impact of such a high power dissipation in a register file is visible in many ways. It severely impacts the mean time to failures, necessitates the need for sophisticated cooling and packaging techniques such as liquid cooling, fan mounted heat sinks etc., and impacts performance because of the need to cool-down periodically[29].

The traditional monolithic design of instruction decoder inhibits the leakage power management in instruction decoder. As a result, earlier work only focused on leakage power management for functional units mostly at a coarser granularity of loop level or block level[21]. However, the rising level of leakage power in current and future process technologies requires aggressive leakage power management even for short idle periods. In this paper, we consider a split instruction decoder design that enables the use of a hardware based scheme such as [9] for leakage power savings in instruction decoder. We also propose a scheduling algorithm in the context of VLIW and clustered VLIW architectures. Whereas, the purely hardware based scheme suffers from the problem of a limited program view, a compiler can analyze whole program regions and is capable of adjusting the number of operations decoded every cycle while maintaining the desired performance. The proposed scheme exploits the scheduling slacks of the instructions to maximize the simultaneous idle time and usage of decoders, thereby significantly reducing spurious transitions and hence improving the power savings over those obtained by a purely hardware based scheme. Moreover, since the proposed scheme keeps a limited number of decoders active and uses them as much as possible, it generates a more balanced schedule which helps to reduce the peak power and the step power[42] in instruction decoders.

In the context of a register file, though clustered architectures help to reduce the complexity of register files by reducing the number of ports, the number of register accesses in these architectures increases significantly because of the need for explicit inter-cluster communications. Thus, the power benefits obtained by clustering a VLIW architecture are annulled because of power penalty due to extra accesses. The other side effects of explicit inter-cluster move instructions are extra resource usage (such as execution slot, inter-cluster communication channel, and functional unit) as well as increased register pressure due to new live ranges. This increased contention for resources also affects performance by increasing the execution cycles. The increase in execution cycles in turn increases the energy consumption because of more leakage. Register snooping based clustered VLIW architectures provide very limited but very fast way of inter-cluster communication by allowing some of the functional units to directly read some of the operands from the register file of some of the other clusters. In this paper, we propose instruction scheduling algorithms that exploit such a limited register snooping capability to significantly reduce the register file energy consumption and to achieve better performance. It is important to note that the sort of limited register snooping capability has been proved useful in practice to gain performance close to the a flat architecture while gaining the clock speed benefit of completely clustered architectures. These archi-

tectures stand in between completely clustered architectures and flat architectures and offer a good architectural trade-off in terms of gaining clock speed while providing limited connectivity. It is important to mention that this is not same as having large number of register ports as the direct connectivity is provided in a very limited form as explained earlier [1] . The benefit of this style of limited connectivity has been realized in one of the well known commercial clustered architecture[38].

The rest of the paper is organized as follows. Section 2 presents the motivation and scheduling algorithms for power optimization in instruction decoder. Section 3 presents the motivation and scheduling algorithms for power optimization in register file. Section 4 presents our experimental setup, performance results, and a detailed analysis of results. Section 5 presents the related work. We conclude in section 6. Appendix A presents a formal description of cluster scheduling problem.

2 Power Optimization in Instruction Decoder

Earlier work in the context of leakage power management has mostly focused on functional units. The techniques proposed earlier are mostly at a coarser granularity of loop level or block level[21]. However, the rising level of leakage power in current and future process technologies requires aggressive leakage power management even for short idle periods. One such purely hardware based scheme for reducing leakage power in functional units in the context of a superscalar architecture is due to Albonesi et al.[9]. Their scheme utilizes the unique characteristics of dual-threshold domino logic with sleep mode that can transition between active mode and sleep mode without any performance penalty[24]. However, such a fast transition incurs moderate amount of energy penalty. Their scheme called 'MaxSleep' puts any integer ALU into low leakage mode after one cycle of idleness. Their results confirm the benefits of such an aggressive scheme. However, being a purely hardware based scheme, the benefits are severely (on average, by 30%) affected by frequent transitions from active mode to sleep mode and vice-versa because of many short idle periods.

The traditional monolithic design of instruction decoder inhibit the leakage power management. In this paper, we consider a split instruction decoder design that enables the use of a hardware based scheme such as [9] for leakage power savings in instruction decoder. Figure 6 presents the power savings obtained by 'MaxSleep', power savings obtained by a 'NoOverhead' scheme which is a hypothetical scheme (same as 'MaxSleep') but does not incur any transition power overheads and % power overhead of 'MaxSleep' due to transitions as compared to that of 'NoOverhead' scheme for a split instruction decoder design (that provides facility to de-

[1] In the rest of the paper, we refer to register snooping as the sort of limited free of cost cross-cluster register read capability provided by architecture as described here.

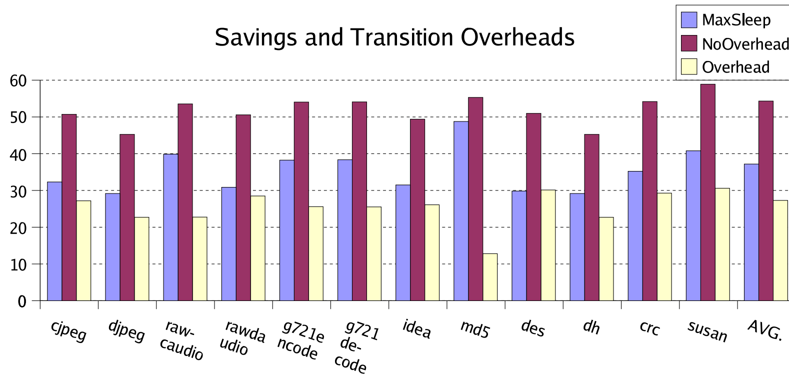

Fig. 1. % Savings for 'MaxSleep' and 'NoOverhead' Policies

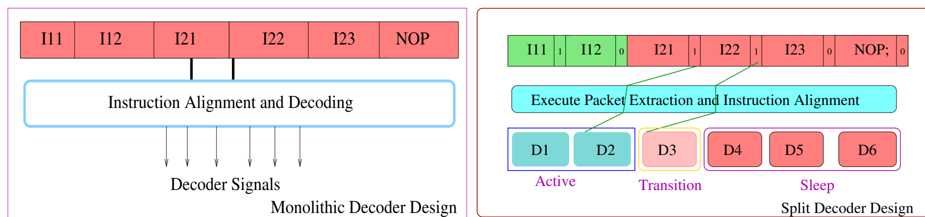

Fig. 2. (a)Traditional Monolithic Decoder Design (b) Split Decoder Design

code up to six instructions in parallel) for a 2-cluster configuration. These results clearly indicate that the 'NoOverhead' scheme is able to achieve an average savings of 56.86% in total power, whereas the average savings for 'MaxSleep' is only 38.92%. 'MaxSleep' has an average energy overhead of 29.37% (due to transitions) as compared to the 'NoOverhead' scheme. Thus, reducing the number of transitions will increase the idleness duration for decoders and improves the total power benefits of a hardware based scheme. Motivated by this, we have developed a scheduling algorithm in the context of VLIW and clustered VLIW architectures. Whereas the purely hardware based scheme suffers from the problem of a limited program view, a compiler can analyze whole program regions and is capable of adjusting the operations decoded every cycle while maintaining the desired performance. The proposed scheme exploits the scheduling slacks of the instructions to maximize the simultaneous idle time and usage of decoders, thereby reducing the number of transitions drastically. This reduction in the number of transitions leads to significant improvements in total power savings over those obtained by a purely hardware based scheme. Moreover, since the proposed scheme keeps a limited number of decoder active and use them as much as possible, it generates a more balanced schedule which helps to reduce the peak power (maximum power dissipated in a cycle) and the step power (cycle to cycle variation in power)[42] in instruction decoder.

2.1 Split Decoder Design

Decoding activity involves dividing a fetch packet into execute packets and then decoding individual micro-instructions in each execute-packet to issue signals. A parallel-bit is dedicated in a VLIW micro-instruction that specifies whether the next micro-instruction is in the same execute-packet (i.e., executes in the same cycle) or starts a new execute-packet. A traditional monolithic design of instruction decoder as shown in Figure 2 (a) inhibits any fine grained control for hibernating parts of the decoder circuit that are idle. A decoder circuit can be easily pipelined and split as shown in Figure 2 (b). This provides the benefits of ease of design and verification of circuit and performance benefits of pipelining[23] and also enables leakage power savings at the granularity of individual decoders. The performance benefits of pipelining the decode stage have already been identified and such a design is in use in many high performance commercial DSPs including the Texas Instruments' VelociTI[38]. However, the novelty of our approach lies in capitalizing on the power management capability of such a design as follows.

Due to variations in the ILP of the programs, the full issue width of the processor is rarely utilized continuously and hence several decoder will be idle most of the time. The split decoder design can leverage the capabilities of dual-threshold domino logic for fast transition from active mode to sleep mode and vice versa in less than a cycle (as used in [9] for functional units) to save tremendous amount of leakage power in mostly idle decoder circuit. However, in order to avoid the explicit penalty of activating a sleeping decoder, it is required to issue the activating signal one cycle in advance. Fortunately, the parallel-bit that specifies the parallel instructions in the current execute-packet can be used to drive the activation signal (see Figure 2 (b)). To avoid introducing any new hardware, in our machine model, we always keep first decoder active, and use the parallel-bits in execute packet to drive the active signal for the required number of decoders. It is important to note that these signals are activated during the first stage of decoding when the execute packet is being extracted and aligned from the fetch packet. Thus, by the time the micro-instructions reach stage two for actual decoding, the required number of decoders are in active state to perform the decoding.

2.2 The Scheduling Algorithm

The Elcor backend of the Trimaran infrastructure has a cycle scheduling algorithm designed and implemented for flat VLIW architectures. We have modified this algorithm to perform leakage power optimization for VLIW as well as clustered VLIW architectures. The scheduler controls the assignment of instructions to clusters so as to maximize the usage of active decode units and to keep the idle decode units idle as long as possible. The decode units in sleep mode are explicitly activated

Algorithm 1 The Main Scheduling Loop

```
if (Scheduling for a clustered configuration) then
  ClusterScheduling ← 1
end if
Initialize ReadyList with root operations of the dependence graph of the region to be scheduled
CurrentCycle ← 0
while (ReadyList is not empty) do
  Initialize EarlyCycle with CurrentCycle, and LateCycle with SchedulingCycle determined using performance driven
  scheduling
  slack = LateCycle − EarlyCycle
  while (Not all operations in ReadyList have been tried once) do
    (CurrentOperations ← UnSchedList.pop())
    AlternativeList ← DetermineSchedulingAlternatives(CurrrentOperation, ClusterScheduling)
    if (IsEmpty(AlternativeList) then
      CONTINUE
    end if
    TargetCluster ← 0
    SUCCESS ← FALSE
    if (ClusterScheduling) then
      TargetCluster ← DetermineBestCluster(CurrentOperation)
      AlternativeList ← DetermineSchedulingAlternatives(CurrentOperation, TargetCluster)
    end if
    CurrentAlternative ← AlternativeList.top()
    if DecoderActive(TargetCluster.Cluster) then
      Schedule CurrentOperation using CurrentAlternative in CurrentCycle on TargetCluster.Cluster using TargetClus-
      ter.CommOption
      SUCCESSS ← TRUE
    end if
    if (!SUCCESS and Slack ≤ SLACK_THRESHOLD) then
      FallBackAlternative ← AlternativeList.top()
      Schedule CurrentOperation using FallBackAlternative in CurrentCycle on TargetCluster.Cluster using Target-
      Cluster.CommOption
    else
      ReadyList.add(CurrentOperation)
    end if
  end while
  CurrentCycle ← CurrentCycle + 1
  ReadyList.update()
  updateFUStatus()
end while
```

only if not doing so impacts the performance. This ensures that decoder energy consumption because of spurious transitions from sleep mode to active mode and vice-versa is reduced. We follow an integrated approach[19][32] to cluster scheduling that makes the cluster assignment decision during temporal scheduling. This is in contrast to phase-decoupled approaches[8][25] which perform cluster assignment prior to temporal scheduling and are known to suffer from phase ordering problem[19][31]. Though, we follow an integrated approach, it is important to note that the proposed energy management scheme can be applied to gain benefits during temporal scheduling even if a phase decoupled scheme is used for spatial and temporal scheduling. Our integrated scheduling algorithm (Refer Algorithm 1) for leakage power optimization consists of the three main steps described as follows. Section 2.4 presents an example that illustrates the functioning of the algorithm in detail.

Procedure 2 DetermineBestCluster

```
FirstTarget.Cluster ← −1
FirstTarget.CommCost ← 1000000;
SecondTarget.Cluster ← −1
SecondTarget.CommCost ← 1000000
for (CurrentCluster ranging from FirstCluster through LastCluster) do
  Compute the Cross-path Requirements in CurrentCommOption
  Compute the Communication Cost in CurrentCommCost
  if (FU and Cross-paths required by CurrentOperation are available in CurrentCycle for CurrentCluster) then
    if (DecoderActive(CurrentCluster) and FirstTarget.cost > CurrentCommCost) then
      FirstTarget.CommCost ← CurrentCommCost
      FirstTarget.CommOption ← CurrentCommOption
      FirstTarget.Cluster ← CurrentCluster
    else
      if (SecondTarget.cost > currentCommCost) then
        SecondTarget.CommCost ← CurrentCommCost
        SecondTarget.CommOption ← CurrentCommOption
        SecondTarget.Cluster ← CurrentCluster
      end if
    end if
  end if
end for
if (FirstTarget.cluster! = −1) then
  RETURN FirstTarget
else
  RETURN SecondTarget
end if
```

2.2.1 Prioritizing the Ready Instructions

Instructions in the ReadyList are prioritized using a priority function that uses the instruction slack and the number of consumers of the instruction. Scheduling slack of an instruction is defined as the difference between the earliest start time and the latest finish time of the instruction. Instructions with less slack should be scheduled early and are given higher priority over instructions with more slack to avoid unnecessary stretching of the schedule. Instructions with the same slack values are further ordered in the decreasing order of the number of consumers. An instruction with a large number of successors is more constrained in the sense that its spatial and temporal placement affects scheduling of more number of instructions and hence should be given higher priority. Giving preference to an instruction with many dependent instructions also enables better future scheduling decisions by uncovering a larger portion of the graph.

Traditionally, slack is determined statically during dependence graph analysis before the scheduling begins, assuming a machine with infinite resources of each type. We quantify the slack of instructions while scheduling a region for the specific target machine by taking resource constraints into account. We first schedule the instruction using a simple cycle-by-cycle scheduler. The schedule time of the instructions is stored during this phase. In the second phase, this schedule time (Late cycle) is used to determine the slack of the instruction. In our implementation, slack is dynamically updated for all the operations in the ready list after every cycle. The earliest schedule time of an instruction is set to the current cycle, before scheduling for the current cycle begins (Early cycle). The slack is then determined as a difference of the Early cycle and the Late cycle. The dynamic update of slack after

each cycle ensures that any consumed slack is taken into account while scheduling instructions in the future cycles.

2.2.2 Cluster Assignment

Once an instruction has been selected for scheduling, we make a cluster assignment decision. The primary constraints are :

- The chosen cluster should have at least one free resource of the type needed to perform this operation
- Given the bandwidth of the channels among clusters and their usage, it should be possible to satisfy the communication needs of the operands of this instruction on the cluster by scheduling these communications in the earlier cycles (so that operands are available at the right time).

Note that if we are scheduling for a plain VLIW architecture with no clustering, we assume that there is only one cluster (numbered 0) that is holding all the resources and the same algorithm is used. Selection of a cluster from the set of the feasible clusters is done as follows. A cluster with an active decoder to schedule the operation is given preference. If no such cluster is available or more than one such cluster is available, the one which reduces the communication cost gets preference. The communication cost is computed by determining the number and type of communications needed by a binding in the earlier cycles as well as the communication that will happen in the future. Future communications are determined by considering the successors of this instruction which have one of their parents bound on a cluster different from the cluster under consideration. This is due to the fact that if the instruction is bound to the cluster under consideration, it will surely lead to communication(s) in the future while scheduling the successors of the instructions. Although, we have experimented with many other heuristics for cluster assignment, the above mentioned heuristic seems to generate the best schedule in almost all cases[31].

2.2.3 Instruction Binding

A instruction binding scheme decides to bind or defer the chosen instruction to the selected cluster. The algorithm maintains a decoder map that explicitly keeps track of the status of each decode unit. A decode unit is marked to be in sleep mode after one cycle of idleness and is marked as activated on next use. If a decode unit is active in the target cluster, the instruction is bound to that cluster. Otherwise, the available slack of the instruction is considered. If the slack is below a threshold (we use the threshold value of 0 in our experiment), the instruction is bound anyway and the extra decoder unit required by the instruction is automatically woken up during execution. In case the instruction possesses enough slack, its scheduling is deferred to a future cycle and it is put back in the ReadyList. Note that the next time this

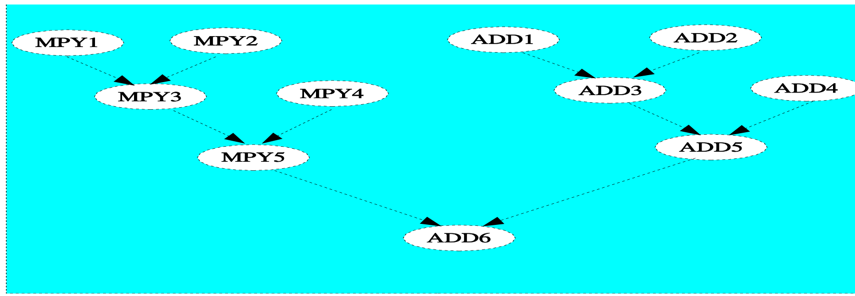

Fig. 3. An Example Data Dependency Graph

	Schedule 1
1	MPY1/D1, MPY2/D2, ADD1/D3, ADD2/D4
2	MPY4/D1, ADD4/D2, ADD3/D3
3	MPY3/D1, ADD5/D2
4	
5	MPY5/D1
6	
7	ADD6/D1
8	

	Schedule 2
1	MPY1/D1, MPY2/D2
2	MPY4/D1, ADD1/D2
3	MPY3/D1,ADD2/D2
4	ADD3/D1,ADD4/D2
5	MPY5/D1,ADD5/D2
6	
7	ADD6/D1
8	

	Schedule 3	
	Cluster 1	Cluster 2
1	MPY1/D1,ADD1/D2	MPY2/D3,ADD2/D4
2	MPY4/D1,ADD4/D2	
3	ADD3/D1	
4	MPY3/D1,ADD5/D2	
5		
6	MPY5/D1	
7		
8	ADD6/D1	
9		

	Schedule 4	
	Cluster 1	Cluster 2
1	MPY1/D1	MPY2/D2
2	MPY4/D1	ADD1/D2
3	ADD2/D1	ADD4/D2
4	MPY3/D1	
5	ADD3/D1	
6	MPY5/D1	
7	ADD5/D1	
8	ADD6/D1	
9		

Fig. 4. (a) Schedule 1 (b) Schedule 2 (c) Schedule 3 (d) Schedule 4

instruction is picked up for scheduling, its earliest scheduling time and hence the slack get updated. This guarantees that the slack of an instruction reduces monotonically and eventually goes below the threshold ensuring that it is scheduled. Hence the algorithm is guaranteed to terminate.

2.2.4 An Example

We now present an example to illustrate how the proposed scheduling algorithm gets power benefits without hurting performance. Figure 3 shows a data dependency graph and Figure 4 shows some schedules. Schedules 1 and 2 are for a plain VLIW architecture having two adders, two multipliers, and 4 decoders. We assume that the latency of an add operation is one cycle and the latency of a multiply operation is two cycles. Schedule 1 is generated by a traditional performance-oriented scheduler which schedules the instructions as early as possible and uses the slack value of instructions to break any contentions for resources and the total schedule length is 8 cycles. Total number of transition for Scheduler 1 are 3 as decoder D4, D3 and D2 each incur one transition in cycle 3, 4 and 5 respectively after idleness of 1 cycle.

Our energy efficient scheduler realizes the criticality of MPY operations and available slack for ADD operations and schedules the same data dependence graph as shown in schedule 2. Since deferring the execution of any MPY operation leads to stretching of schedules, they are scheduled in the same way as in the performance-oriented schedule 1. However, scheduling of ADD operations is delayed as well as serialized, exploiting the available slack of add operations. Notably, the scheduler determines the slack value available in scheduling an operation by first doing a performance-oriented scheduling pass on data-dependence graph and uses the estimate of schedule length from this pass to calculate the exact slack value available

in scheduling an instruction which is used to generate the schedule for energy efficiency. Schedule 2 uses only two decoders and incurs only one transition for D2 in cycle 7. Schedule 2 is also more balanced as compared to schedule 1 in terms of resource usage. The resource usage vector of the first schedule is (4,3,2,0,1,0,1,0) and that of second is (2,2,2,2,2,0,1,0). Thus cycle to cycle variation in resource usage is clearly reduced in schedule 2 as compared to schedule 1, which in turn helps in reducing step power and peak power dissipation[42]. Thus, it is clear that the proposed scheme is capable of reducing total leakage power consumption, transition energy overheads, as well as peak power and step power dissipation without affecting the performance.

Consider schedules 3 and 4 generated for a 2-clustered VLIW architecture (equivalent to above mentioned VLIW architecture) having 1 adder and 1 multiplier in each cluster and a bidirectional bus between the two clusters with 1 cycle transfer latency. Schedule 3 is generated by a performance-oriented scheduler. The extra delay of inter-cluster communication stretches the schedule from 8 cycles to 9 cycles as compared to the schedule 1. Again the total number of decoder transitions are 3.

Scheduling the same set of operations using our energy-efficient scheduler generates schedule 4. The major point to note is that the scheduler leverages the available slack due to inter-cluster communication to achieve the same 9-cycle schedule with only two decoders and only one transition for D2 in cycle 5. Finally schedule 4 is much more balanced : The resource usage vector of first schedule is (4,2,1,2,0,1,0,1,0) and that of the schedule 4 is (2,2,2,1,1,1,1,1,0).

3 Power Optimization in Register File

In context of register file, though clustered architectures help to reduce the complexity of register files by reducing the number of ports, the number of register accesses in these architectures increase significantly because of the need for explicit inter-cluster communications. Figure 5 demonstrates this fact clearly. The two add operations ADD1 and ADD2 when executed in separate clusters and one being dependent on the result of other necessitate an inter cluster move operation that leads to at least two extra register accesses (A3 and B1), one in each cluster. The other side effects of explicit inter-cluster move instruction are extra resource usage (such as execution slot, inter-cluster communication channel, and functional unit) as well as increased register pressure due to new live ranges (for example, live range associated with B1 in this case in cluster 2). Thus, the power benefits obtained by clustering a VLIW architecture are annulled because of energy penalty due to extra accesses.

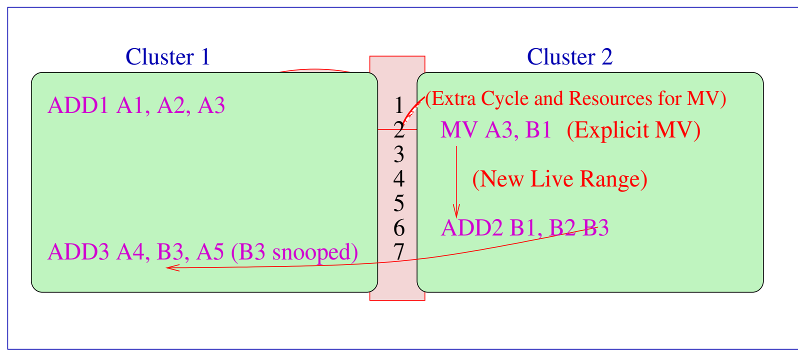

Fig. 5. Side Effects of Explicit Inter-Cluster Move operation

3.1 Motivation

Register snooping based clustered VLIW architecture such as [38] allows some of the functional units to read some of the operand from the register file of some of the other clusters directly without any extra delay by providing limited multiplexed cross-cluster register paths. Register snooping facility enables the immediate consumption of the value read from the source cluster and thus does not introduce new live ranges, extra register access, and resource usage (For example, register snooping facility as used in case of ADD3 in Figure 1 to read B3 from cluster 2). However, traditional cluster scheduling algorithms that are based on greedy heuristics such as scheduling an instruction as early as possible[19] lead to excessive inter-cluster communication on such an architecture. This in turn impacts performance because of the extra resource requirements of executing inter-cluster move instructions as well as the increase in the register pressure. The longer execution time in turn increases the overall system energy consumption. Some of the earlier heuristics[32] that are solely based on minimizing the communication suffer from high performance degradation on register snooping based clustered VLIW architectures. Figure 6 gives the quantitative results to support this argument. Figure 6 presents the percentage distribution of explicit inter-cluster move instructions w.r.t total number of instruction for a 2-clustered and a 4-clustered machine that supports two snoops or explicit move (one in each direction) between all pairs of neighboring clusters. The scheduling has been done using a greedy scheduling algorithm (called *Greedy* here onwards) similar to [19] but slightly modified to take benefits of register snooping capabilities wherever possible while retaining the core cluster assignment policy (Refer Algorithm 3 for an outline). The algorithm is a modified version of the list scheduling algorithm that considers instructions from the ReadyList based on priority and schedule them on a cluster that can execute the instructions at the earliest irrespective of inter-cluster communication that is incurred in the current and future cycles. This algorithm represents the state-of-the-art in cluster scheduling for clustered architectures without register snooping capabilities. However, it is clear that inter-cluster moves constitute a significant fraction of the overall instructions. This is clear from Figure 6 as the percentage of inter-cluster

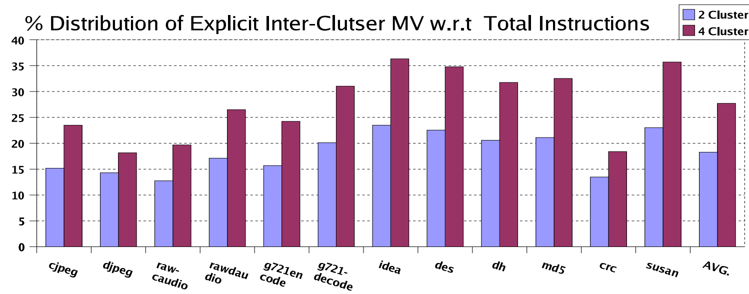

Fig. 6. % Distribution of Explicit Inter-Cluster move w.r.t Total Instructions

move is as high as 24% for a 2-clustered machine and 37% for a 4-clustered machine. On average the percentage of inter-cluster move instructions with respect to other instructions is 18.3% and 27.7% for a 2-clustered and a 4-clustered machine respectively. As mentioned earlier, each of these extra inter-cluster move engages resources (such as execution slot, inter-cluster communication channel, functional unit) and it creates extra register accesses and new live ranges. This leads to performance degradation. We observe that the cycle count increases by 15% and 23% with respect to a hypothetical BASE architecture having the same number of resources in a single cluster. Extra inter-cluster move instructions also explicitly increase the code size.

The large number of inter-cluster move instructions can be reduced if the register snooping capability of clustered VLIW architectures is harnessed properly. A scheduler can leverage the benefits of snooping by scheduling operations among clusters in such a way that the free ICC facility can be utilized to the maximum possible extent. This demands spreading the computation among clusters in such a way that most of the communication can be accommodated using free-of-cost communication facility without compromising the available ILP in the application and requiring explicit move operations only rarely. This demands not only reducing the requisite communication in the cycle under consideration but also taking care of the communication requirements that may arise in the future. This will help not only to reduce the performance overhead due to excessive inter-cluster communication but also the significant power savings in register files by eliminating extraneous access. In what follows, we propose two such scheduling algorithms. The first algorithm called *Simple* tries to schedule instruction in a priority manner but instead of trying to reduce the inter-cluster communication, it rather uses the register snooping capability as much as possible and minimizes the need for explicit inter-cluster communication. This algorithm gives reasonably good saving in register file power as compared to algorithm *Greedy* with significant performance improvements. The second algorithm, *Aggressive* utilizes the scheduling slack of instructions to selectively defer their execution in order to use register snooping capability over explicit inter-cluster move. *Aggressive* improves the performance over *Simple* marginally but further reduces the register file power significantly.

We have modified the cycle scheduling algorithm in Elcor backend of the Trimaran infrastructure[1] to do cluster scheduling in an integrated fashion using different heuristics (*Greedy*, *Simple*, and *Aggressive*). The selection of a feasible cluster for binding an instruction in a cycle is done based on the same feasibility constraints as described earlier in section 2.2.2. The algorithms differ in selection of an instruction from the ReadyList, selection of the target cluster for binding instruction from set of feasible clusters, and the decision mechanism to defer an instruction for consideration in future cycles. In what follows, we describe the specific functioning of each of these three algorithms in detail.

Algorithm 3 Out Line of the Scheduling Algorithm *Greedy*

Determined EarlyCycle, LateCycle and Slack for each instruction in the Dependence Graph
Initialize ReadyList with root operations of the dependence graph of the region to be scheduled
$CurrentCycle \leftarrow 0$
while (ReadyList is not empty) **do**
  while (Not all operations in ReadyList have been tried once) **do**
    $(CurrentOperations \leftarrow UnSchedList.pop())$
    $TargetCluster \leftarrow DetermineEarliestCluster(CurrentOperation)$
    Schedule CurrentOperation in CurrentCycle on TargetCluster.Cluster using TargetCluster.CommOption
  end while
  $CurrentCycle \leftarrow CurrentCycle + 1$
  $ReadyList.update()$
end while

3.2.1 Algorithm Greedy

Algorithm *Greedy* presented here (Refer Algorithm 3 for an outline) is a slightly modified version of the one presented in [19] that extends the traditional list scheduling algorithm for cluster scheduling. This algorithm selects instructions in the ReadyList in decreasing order of priority. Priority of an instruction is determined based on instruction slack. Slack in this case is calculated as a difference between earliest possible schedule time and latest possible schedule time for an infinite resource machine. The selected instruction is greedily assigned to the cluster where it can be executed at the earliest irrespective of any communication that happened in earlier cycles as well as those that will happen in future cycles (as a side effect of this binding). However, in our implementation of *Greedy*, in order to ensure a fair comparison, we use the register snooping facility to read the operands (wherever possible) from other clusters subject to the core cluster assignment policy (as described above) of *Greedy*. Explicit inter-cluster communication instructions are scheduled

in earlier cycles for all inter-cluster communications that can not be accommodate using the available register snooping facility in the current cycle.

3.2.2 Algorithm Simple

Algorithm *Simple* (Refer Algorithm 4 for an outline) tries to schedule the instructions in a manner such that the register snooping capability is used to the maximum extent and explicit inter-cluster communication is minimized. An instruction is selected from ReadyList based on a two component ordering function. The two components of the ordering function are instruction slack and number of consumers. Slack is calculated in the same as in Algorithm *Greedy*. The instructions with less slack are scheduled early and are given priority over instructions with more slack in order to avoid unnecessary stretching of the schedule. Among instructions with the same slack values, the one with more number of successors is scheduled early because it has more constraints and its scheduling impacts spatial and temporal placement of a larger number of instructions. Once an instruction has been selected from the ReadyList, the selection of a cluster from the set of feasible clusters is done according to the following heuristic. Unlike *Greedy* that simply assigns an instruction to a cluster that can execute the instruction at the earliest, *Simple* determines the average communication cost per transfer for each cluster by taking into account the communication required in earlier cycles, current cycle as well as communication required in future cycles as a side effect of a binding. The instruction is finally assigned to that cluster which can execute the instruction at the earliest but with the minimal average communication cost. Specifically, the average communication cost of a binding is calculated according to communication cost metric given in Equation 1.

$$\begin{aligned} avg_comm_cost = (A * current_comm + B * future \\ _comm + C * explicit_mv)/total_num_comm \end{aligned} \tag{1}$$

In Equation 1 *explicit_mv* takes care of communications that can be due to non-availability of *snooping* facility in a particular cycle (because of cross-path saturation). Only those clusters which have enough communication slots and required resources for scheduling a move instruction in the earlier cycles are considered. *current_comm* is determined by the number of operands that reside on a cluster other than the cluster under consideration and can be *snooped* using the *register snooping* facility available in the current cycle. *future_comm* is determined by considering the successors of this instruction which have one of their parents bound to a cluster different from the current one. In case some of the parents of one of the successors are not yet bound, the calculation is done assuming that they can be bound to any of the clusters with equal probability. The selection of A, B and C is architecture specific and depends on available communication options in a clustered architecture and their relative cost. We found that A=0.25, B=0.5 and C=1.0

work well in practice for an architecture that has limited *register snooping* facility available (such as the one we consider) and allows at most two snoops per cycle (one in each direction) between all pairs of neighboring clusters. In general, the values of A, B and C are tuned based on the target architecture. Since *explicit_mv* is a pure overhead in terms of resource requirements and register pressure, it is assigned double the cost of the *current_comm* to discourage these explicit moves and this favors the *register snooping* facility in the cost model proposed. *future_comm* is also assigned a smaller cost optimistically assuming that most of them will be accommodated in the free-of-cost communication slot. The values of different constants can be changed to reflect the ICC model under consideration.

Algorithm 4 Out Line of the Scheduling Algorithm *Simple*

```
Determine EarlyCycle, LateCycle, Slack and NumDependent (Number of Suc-
cessor instructions) for each instruction in Dependence Graph
Initialize ReadyList with root operations of the dependence graph of the region
to be scheduled
CurrentCycle ← 0
while (ReadyList is not empty) do
   while (Not all operations in the ReadyList have been tried once) do
      (CurrentOperations ← UnSchedList.pop())
      TargetClusterList ← DetermineMinCommClusters(CurrentOperation)
      TargetCluster ← DetermineEarliestClusters(
                  CurrentOperation, TargetClusterList)
      Schedule CurrentOperation in CurrentCycle on TargetCluster.Cluster using
      TargetCluster.CommOption
   end while
   CurrentCycle ← CurrentCycle + 1
   ReadyList.update()
end while
```

3.2.3 Algorithm Aggressive

Algorithm *Simple* described in the last subsection maximizes the usage of the register snooping facility available in architecture while minimizing the need for explicit inter-cluster move operations. This not only helps to reduce register file power but also improves performance significantly. However, it is still greedy to some extent because it does not defer scheduling low priority instructions (considered for scheduling later in a cycle) that incur high communication cost. This is because, low priority instructions use explicit inter-cluster communication as a result of saturation of available register snooping facility by high priority instructions considered early in the cycle. Algorithm *Aggressive* (Refer Algorithm 5 for an outline) improves over *Simple* by deferring these instructions for consideration in later cycles. *Aggressive* algorithm makes the decision based on communication cost and on available scheduling slack of instructions so that deferring instructions does not impact performance by stretching the schedule. In contrast to Algorithms *Greedy* and

Algorithm 5 Out Line of the Scheduling Algorithm *Aggressive*

Initialize ReadyList with root operations of the dependence graph of the region to be scheduled
Determined NumDependent (Number of Successor instructions) for each instruction in Dependence Graph
$CurrentCycle \leftarrow 0$
while (ReadyList is not empty) **do**
  Initialize EarlyCycle with CurrentCycle, and LateCycle with SchedulingCycle determined using performance driven scheduling
  $slack = LateCycle - EarlyCycle$
  while (Not all operations in ReadyList have been tried once) **do**
    $(CurrentOperations \leftarrow UnSchedList.pop())$
    $TargetClusterList \leftarrow DetermineMinCommClusters(CurrentOperation)$
    $TargetCluster \leftarrow DetermineEarliestClusters($
        $CurrentOperation, TargetClusterList)$
    if ($Slack \geq SLACK_THRESHOLD)$ and
      $(TargetCluster.CommCost \geq Comm_Threshold)$ **then**
      ReadyList.add(CurrentOperation)
    else
      Schedule CurrentOperation in CurrentCycle on TargetCluster.Cluster using TargetCluster.CommOption
    end if
  end while
  $CurrentCycle \leftarrow CurrentCycle + 1$
  $ReadyList.update()$
end while

Simple, Algorithm *Aggressive* depends upon realistic calculation of slack to be able to be able avoid the stretch of schedule as well as exploiting the opportunities to avoid explicit inter-cluster communication. Thus, in this case slack is determined taking into account the resource constraint on the real machine and updated dynamically in the same way as explained earlier (Refer section 3.2.1 for the realistic calculation and update of slack).

Algorithm *Aggressive* selects an instruction from the ReadyList based on the above determined dynamic slack and the number of consumers. After the instruction is selected from ReadyList, a set of feasible clusters for scheduling the instruction are determined based on the above mentioned constraints. The average communication cost of scheduling the instruction on each of the clusters is then determined and the cluster that can schedule the instruction earlier with minimum average communication cost is considered for scheduling. However, unlike algorithm *simple*, instead of binding the instructions blindly to a cluster, algorithm *Aggressive* takes into account the average communication cost for each communication as determined by the cost metric explained above and the available instruction slack as calculated and updated dynamically. If the average communication slack of the instruction is

above than a communication threshold (COMM_THRESHOLD) and the slack of the instruction in above a slack threshold (SLACK_THRESHOLD), the scheduling of the instruction is deferred by decrementing its slack and putting it back into ReadyList. The values of SLACK_THRESHOLD and COMM_THRESHOLD decide the degree of aggressiveness of the scheduler. Higher slack thresholds ensure that possible performance degradation because of deferring instructions is less but at the same time this also offer only limited opportunities to defer instructions. On the other hand, low values of slack provide more opportunities for deferring instructions there by scheduling them using register snooping facility as much as possible. But too low a value of slack can also lead to instruction serialization because of resource contentions and thus affect the performance. The value of COMM_THRESHOLD decides the aggressiveness of the scheduler in terms of avoiding the inter-cluster move over the register snooping facility. Lower the value, more will be the bias towards using register snooping facility over explicit inter-cluster communication. We have experimented with a class of schedulers by changing values of COMM_THRESHOLD and SLACK_THRESHOLD. We present the results (in experimental section) based on a moderate value of $SLACK_THRESHOLD = 2$ and a low value of $COMM_THRESHOLD = .35$ which provides a scheduler a heavy bias towards using register snooping facility but with only limited possibility of impact on performance.

4 Experimental Evaluation

4.1 Setup

We have used the Trimaran suite for our experimentation. Trimaran was developed to conduct state-of-the-art research in compilation techniques for ILP architectures with a specific focus on VLIW class of architectures. We have modified the Trimaran suite to generate and simulate code for a variety of clustered VLIW configurations. The machine description module has been upgraded to describe various clustering related parameters such as the number of clusters, number and types of functional units in each cluster, interconnection network parameters such as number and types of buses between different clusters, and their latency parameters. These parameters are fed to the parameterized machine-dependent optimization modules in the backend. Major modifications have been performed in the Trimaran scheduler and register allocator module (which was originally written for a class of flat VLIW architectures) to faithfully account for the conflicts due to limitations on the number of available functional units and registers in a cluster as well as the limitations on the number of available cross-paths between clusters. The scheduler has been modified to implement the scheduling algorithm described in the last section. We have used twelve benchmarks out of which nine are from mediabench[26] *(viz. cjpeg, djpeg, rawcaudio, rawdaudio, g721encode, g721decode, md5, des, and idea)*, two

from netbench[15] *(viz. crc, and dh)*, and one *(susan)* is from MiBench[17]. We have tried other benchmarks from these suites as well but these are the only ones which compiled successfully and executed correctly in the Trimaran framework and hence we report results for them.

We present results for a 2-clustered machine and a 4-clustered VLIW machine as compared to an equivalent hypothetical unclustered BASE VLIW machine. The unclustered VLIW configuration has 4 ALUs, 2 load-store units, 1 branch unit, and 64 registers. The 2-clustered configuration has 2 ALUs, 1-load store units, 1 branch unit and 32 registers in each cluster, whereas the 4-clustered configuration has 1 ALU, 1-load store unit, 1 branch unit and 16 registers in each cluster. The communication among data values is possible by explicit move instructions as well as register snooping facility with a limitation of two snoops (or explicit moves) par cycle between all pairs of neighboring clusters.

4.1.1 Energy Model

We have used the same general analytical energy model proposed in [9] for combinational circuits to directly compare the decoder energy benefits of our compiler-assisted scheme over the pure hardware based scheme proposed in[9]. However, unlike [9] that target leakage energy in functional units, we target leakage energy savings in instruction decoder. We briefly describe this model here. The reader is referred to [9] for details. The total energy in a decode unit is determined as follows:

$$\begin{aligned}\mathbf{E'_{total}} &= \mathbf{DynamicEnergy} + \mathbf{LeakageEnergy} + \\ &\quad \mathbf{TransitionEnergy} + \mathbf{SleepEnergy}\\ \mathbf{E'_{total}} &= \mathbf{n_A}(\alpha\mathbf{E_A} + (\mathbf{1}-\mathbf{D})\mathbf{E_{S_1}}) + (\mathbf{n_A D} + \mathbf{n_{UI}}) * (\alpha\mathbf{E_{s_0}} + (\mathbf{1}-\alpha)\mathbf{E_{s_1}}) + \\ &\quad \mathbf{M_z}((\mathbf{1}-\alpha)\mathbf{E_A} + \mathbf{E_{Sleep}}) + \mathbf{n_Z E_{s0}}\end{aligned}$$

Here $\mathbf{n_A}$ is the number of active cycles, $\mathbf{n_{UI}}$ is the number of uncontrolled idle cycles, $\mathbf{n_Z}$ is the number of sleep cycles and $\mathbf{M_z}$ is the number of transitions. We have determined these values differently for each configuration by using the trimaran simulator. $\mathbf{E_{s_0}}$ and $\mathbf{E_{s_1}}$ are low leakage and high leakage energy and are related by the following equations.

$\mathbf{E_{s_0}} = \mathbf{s} * \mathbf{E_{S_1}}, \mathbf{0.0001} \le \mathbf{s} \le \mathbf{0.01}$ and $\mathbf{E_{s_1}} = \mathbf{p} * \mathbf{E_A}, \mathbf{0} \le \mathbf{p}$

Where $\mathbf{p}$ is the ratio of the maximum leakage energy expended to the maximum energy for evaluation per unit of time (1 cycle). After simplifying and normalizing the equations with respect to active energy, the following model for total energy consumption is obtained :

$$\begin{aligned}\mathbf{E_{total}} &= \mathbf{n_A}(\alpha + (\mathbf{1}-\mathbf{D})\mathbf{p}) + (\mathbf{n_A D} + \mathbf{n_{UI}}) * (\alpha\mathbf{sp} + (\mathbf{1}-\alpha)\mathbf{p}) \\ &\quad + \mathbf{M_z}((\mathbf{1}-\alpha) + \mathbf{E_{Sleep}}/\mathbf{E_A}) + \mathbf{n_Z sp}\end{aligned}$$

The technology parameters that we have used (s=0.01 and $E_{Sleep}/E_A = 0.01$) are also the same as in [9]. Considering the current 65nm fabrication technology where leakage energy is on par with dynamic energy, we set p to 0.5. α is the activity factor and **D** is the duty cycle of the clock. We use a typical value of 0.5 for both of these parameters in our simulation.

To determine the energy consumption in register files, we have modified eCACTI[27] to model a register file as a small tag free storage structure in eCACTI similar to [33]. The register file access statistics are collected for each program and each scheduler using Trimaran. The access statistics are used to determine the total register file energy by multiplying it with per access energy as determined using eCACTI[27].

4.2 Results

We have performed a detailed experimental evaluation of the proposed algorithms in terms of energy savings and performance benefits for 2-clustered and 4-clustered machine. These results are presented in separate subsection below. Since the contribution of power consumption of any particular component to the overall processor power consumption depends on lot of parameters such as exact configuration as well as fabrication process and technology, we take an approach to evaluate the relative power savings in the target component (such as decoder or register file) by the proposed scheme. The exact benefit can be determined by scaling these benefit depending on the details of the exact design. It is important to note that the proposed techniques are opportunistic in nature and seek to exploit power benefits without degrading performance. So application of these techniques does not necessarily affect the power consumption in other components. Another important point to note is that the development of these techniques is also motivated by the fact that decoder and register file are well known hotspot in the processor. So the proposed no-overhead software-only techniques are still attractive due to thermal benefit even if the power savings achieved do not translate to huge amount of overall power savings at the processor level for certain configurations.

4.2.1 Decoder Power Optimization

For decoder energy savings, we present results for the 'AlwaysActive' scheme that doesn't not apply any leakage energy management, the hardware-only scheme from [9] called 'MaxSleep' used in the context of decode units that puts a decode unit into low leakage mode after one cycle of idleness, and our scheduling scheme called 'Optimized' that assists the hardware based scheme by reducing undesirable transitions. The results are presented in comparison with a hypothetical scheme called 'NoOverhead' that is the same as 'MaxSleep' but does not incur any of the energy

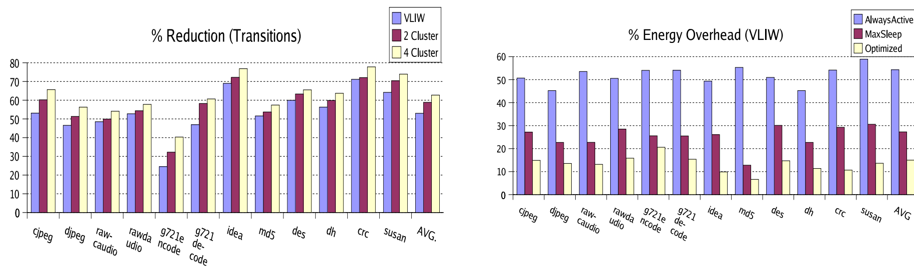

Fig. 7. (a) % Reduction in Transitions with scheduling w.r.t. Hardware only Scheme (b) % Increase in total energy w.r.t Hypothetical No-overhead Scheme (VLIW)

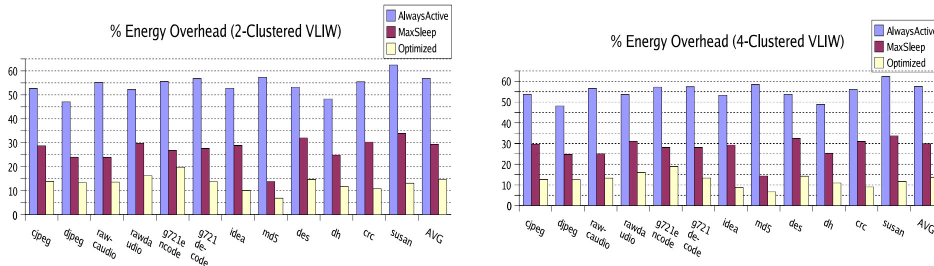

Fig. 8. % Increase in total energy w.r.t No-overhead Scheme (a) 2 Cluster (b) 4 Cluster

overheads of transitions. This scheme represents a theoretical ideal against which a leakage energy management scheme can be compared for its effectiveness. Though our main focus is on clustered architectures, we also present results in the context of VLIW architectures for the sake of comparison and completeness.

Figure 7 (a) shows the percentage reduction in the number of transitions due to our algorithm as compared to the hardware-only scheme. We observe that the number of transitions reduce by 53%, 58.88%, and 62.74% for VLIW, 2-Clustered VLIW, and 4-Clustered VLIW respectively. The reduction in the number of transitions depends on the total available slack in scheduling instructions as well as the distribution of idle cycles in the benchmark. Benchmarks like des, dh, crc, and susan have many short idle cycles and our algorithm is able to exploit the available slack in these applications to avoid many transitions. In the case of g721encode and g721decode, the available slack is relatively less and consequently the reduction is also less.

Figure 7 (b) shows the total energy overhead of 'AlwaysActive', 'MaxSleep' and 'Optimized' schemes as compared to the 'NoOverhead' scheme. 'AlwaysActive', 'MaxSleep' and 'Optimized' schemes show average energy overheads of 54.33%, 27.29%, and 14.99% respectively as compared to the 'NoOverhead' scheme. The proposed 'Optimized' scheme reduces the total energy overhead by 14.46% over the 'MaxSleep' scheme which is significant taking into account that it is a purely software based scheme and does not incur any hardware overhead.

The benefits of our scheme are even more pronounced in the context of clustered

architectures. In the context of 2-clustered architecture 'AlwaysActive','MaxSleep' and 'Optimized' have average energy overheads of 56.86%, 29.37% and 14.6% respectively as compared to the 'NoOverhead' scheme (Refer Figure 9 (a)). The energy benefits of 'Optimized' over Maxsleep is 17.3% in the context of 2 clustered architecture. For a 4-clustered configuration, 'AlwaysActive','MaxSleep', and 'Optimized' incur 57.51%, 29.88%, and 13.7% overhead as compared to 'NoOverhead' scheme (Refer Figure 9 (b)). The 'Optimized' scheme improves over the 'MaxSleep' scheme on the average by 18.74% in the context of 4-clustered architectures. The reasons for more savings in the context of clustered architectures are as follows. Clustering brings along extra contentions for a limited number of slow cross-paths (for inter-cluster communication). This leads to many short idle cycle for instruction decoders. A purely hardware based scheme with traditional scheduling algorithm undergoes transitions for such many short idle cycles and suffers the associated energy penalty. In contrast to the performance-oriented scheduling algorithm which is designed for utilizing the resources spread over different clusters to achieve a better performance, our energy-aware scheduling algorithm sometime limits the spreading of operations, if it can fetch some energy benefits without hurting performance. Thus, some of the extra slack which is available while scheduling for clustered architectures due to contention for inter-cluster communication is utilized to gain energy benefits in our algorithms. Finally, our algorithm suffers a very marginal performance loss of 0.18% in the context of VLIW architceture as compared to performance oriented scheduler. The average performance loss in the context of 2-clustered and 4-clustered architecture is 0.32% and 0.45% respectively. The reason for this performance loss is inherent inaccuracies in determining the available slack. Due to this, slack is sometime over-estimated which in certain cases lead to performance penalty due to serialization of operations. However, our results clearly show that it is rare and its overall effect on performance is negligible. This is because our algorithm is conservative in exploiting slack to save energy.

4.2.2 Register Power Optimization

Register file energy savings and performance improvement of *Simple* and *Aggressive* are presented with respect to *Greedy* for a 2-clustered and a 4-clustered machine.

Figure 9 (a) and Figure 9 (b) show the percentage energy savings of *Simple* and *Aggressive* as compared to *Greedy* for a 2-Clustered and a 4-Clustered machine respectively. We observe that *Simple* and *Aggressive* obtain energy savings of up to 9% and 16% over *Greedy* for 2-clustered machine. The average energy savings of *Simple* and *Aggressive* over *Greedy* are 6.85% and 11.90% for 2-Clustered VLIW machine. The benefits of *Simple* and *Aggressive* over *Greedy* are up to 14% and 20% with the average being 10.39% and 17.78% for 4-Clustered VLIW machine. The benefits are clearly high for 4-clustered machine as compared to 2-clustered machine because of higher number of explicit moves in 4-clustered machine (as can

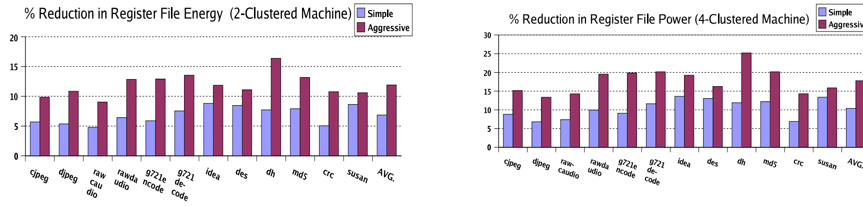

Fig. 9. % Register File Energy Savings w.r.t *Greedy* Scheme (a) 2 Cluster (b) 4 Cluster

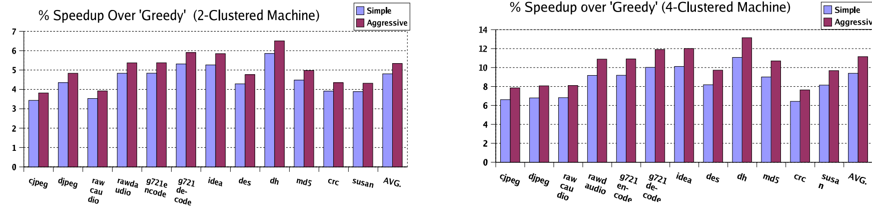

Fig. 10. % Speedup w.r.t. 'Greedy' Algorithm (a) 2 Cluster (b) 4 Cluster

be seen in Figure 6) that could be effectively converted to snoops by our algorithm using greater degree of register snooping available in 4-clustered architecture. The benefit among benchmarks depends on the percentage of explicit inter-cluster move instruction and available scheduling slack of instruction in a benchmark. Benchmarks such as dh, md5 and g721-decode obtain more savings in register file energy which can be attributed to a higher fraction of inter-cluster move instructions in these benchmarks as well as a good amount of available scheduling slack in instructions. However, it is clear that all the benchmarks benefit significantly in terms of register file power. An important point to note is that *Aggressive* gives more benefit in the context of a 4-clustered machine as compared to *Simple* which implies the need for aggressively exploiting the slack for highly clustered architectures in order to reap power as well as performance benefits (as is described in next paragraph).

Figure 10 (a) and Figure 10 (b) show the percentage reduction in execution time of different algorithm *Simple* and *Aggressive* over *Greedy* Algorithm. *Simple* and *Aggressive* obtains performance benefits of 4.81% and 5.34% as compared to *Greedy* for a 2-clustered machine. The performance improvement of *Simple* and *Aggressive* is 9.39% and 11.16% as compared to *Greedy* in the context of 4-clustered architectures. The performance benefit due to reduction of inter-cluster move is due to many reasons. The resource contentions (for functional units and inter-cluster path) is reduced as well as the extra slot used by an inter-cluster move can be utilized to schedule regular instructions. Again the benefit is clearly much more in the context of a 4-clustered architecture because of more number of moves that get converted into snoops by the proposed scheduling algorithms. *A salient feature of the proposed scheme is that it simultaneously provides energy and performance benefit rather than gaining energy benefit at the cost of performance degradation.*

Our early experimental observations demonstrate that the algorithms for decoder power saving and register file power savings can be combined into an integrated algorithm in a profitable manner. This is because both the algorithms are opportunistic in nature with regard to exploiting slack to save power without affecting performance. Though, it is clear that the combination of two algorithms do not interfere negatively with each other, the overall benefit obtained is limited by the total available slack. We have seen instances where combined algorithm provides roughly the same performs as the individual algorithm. However, there are also instances where using one takes away available opportunities for other thereby the benefit are not additive when applied together (as explained above).

Combining decoder power saving algorithm with that of functional unit power saving algorithm is relatively simpler. This is because goal of both the algorithms is to reduce unnecessary transitions and increase the average idleness duration. In fact uniformity of decoders makes it easier and less conflicting to integrate decoder power saving heuristic with that of functional unit power saving heuristic. However, we do not venture into integration of multiple heuristics because focus of this paper is more on saving power in two of the important hots-pots of the chip namely decoder and register file.

5 Related Work

Earlier proposals for scheduling on clustered VLIW architectures can be classified into two main categories, viz., phase-decoupled approaches and phase-coupled approaches. A phase-decoupled approach to scheduling works on a data flow graph (DFG) and performs partitioning of instructions into clusters to reduce inter-cluster communication while approximately balancing the load among clusters. The annotated DFG is then scheduled using a traditional list scheduler while adhering to earlier spatial decisions[8][5]. An integrated approach to scheduling combats the phase-ordering problem by combining spatial and temporal scheduling decisions in a single phase [19] [31][30].

Study of leakage energy management at the architectural level has mostly focused on storage structure such as cache[13]. Some of the earlier work has targeted energy efficiency in functional units. [9] proposes an architectural policy for aggressively controlling leakage energy in integer ALUs. However the overhead of transitions from active mode into low-leakage mode and vice-versa are significant. Zhang et al.[43] have proposed a rescheduling scheme to reduce dynamic and leakage energy in the functional units of a VLIW processor. The dynamic energy saving scheme proposed by [43] depends upon multiple implementation of same function-

ality such as simple adder and carry look ahead adder for addition. This scheme tries to achieve the leakage power savings in functions units by applying energy aware scheduling to code that is already scheduled and resource allocated by a performance oriented scheduler. It is important to note that this leaves only a remnant slack from a performance oriented heuristic for an energy oriented heuristic to gain only limited energy savings. In contrast, our scheme is to perform performance and energy efficient scheduling in an integrated phase on an unscheduled code thereby use the knowledge about total available slack to meet performance requirement while saving energy. Our early experiments showed that such an approach generate better energy savings than an approach used in [43].

Kim et al.[21] have proposed a leakage energy management scheme for VLIW processors that approximates the ILP available in the program using heuristics (as the exact estimation problem is itself NP complete). Gupta et al.,[35] propose a novel data structure called power-aware flow graph. Their leakage energy management scheme in the context of superscalar processors works over this graph to determine larger program regions called power blocks which offer opportunities to save leakage energy. Yun et al.,[42] have proposed a modulo scheduling algorithm that produces a more balanced schedule for software pipelined loops with an objective to reduce the peak power and step power dissipation. Kannan et al.,[20] have proposed temperature and process variation aware power reduction techniques for functional units.

To the best of our knowledge, the only work for energy optimization in the context of instruction decoder is due to Kuo et al.[23]. Kuo et al.[23] consider instruction decoding as in superscalar architectures and propose to split (horizontally partition) instruction decoder circuitry into two or more sub-decoders based on execution frequencies of different instructions. They also propose to do pipelining (vertical partitioning) of the instruction decoder to achieve energy and area benefits. The experimental results of Kuo et al., based on physical synthesis clearly demonstrates that the horizontal and vertical partitioning of the instruction decoder is in general useful in reducing the design complexity, power consumption, area overhead and delay because of simplification of circuitry. In contrast to the work of Kuo et al.[23], partitioning of instruction decoder in our work is geared more toward VLIW and clustered VLIW architectures that demands decoding of large number of instructions in parallel. Thus, compared to functionally asymmetric partitioning of Kuo et al, we consider partitioning of instruction decoder circuitry into functionally identical individual sub-decoders each of which can be controlled independently. The pipelining of decoder as considered by us is more natural in VLIW context where a fetch packet needs to be broken into execute packets and the current execute packet needs to be aligned before actual decoding can begin. Apart from general benefits of a partitioned design as demonstrated by Kuo et al., partitioned decoder design in our proposal also provides an opportunity for fine grained leakage energy management in the instruction decoder.

Most of the earlier work for register file power optimization has been done at architecture level in the context of changing register file organizations to improve register file energy consumption. For example, [4] proposed a hierarchical register file, and [22][18] use auxiliary storage structures to optimize register file power by reducing the ports. Tseng et al.[41] propose five different modifications to the register file and the pipeline that in combination give significant benefits in total register file power consumption.

Another recent work in the context of a superscalar processor proposes a scheduling algorithm for partially bypassed superscalar processors that maximizes the number of operand reads from datapath bypasses thereby reducing the register file access and associated power consumption[33]. However, the proposed algorithm suffers some performance degradation as it tries to schedule the dependent instructions close to each other. In contrast, our work is in the context of explicitly scheduled clustered processors and gains both energy and performance benefits. [2] proposes a mechanism to put unused registers into low-leakage mode thereby saving leakage power of register file. In contrast, our work is geared towards reducing dynamic power of register files by reducing the extra accesses.

6 Conclusions and Future Directions

In this work, we propose compiler scheduling algorithms in the context of widespread embedded clustered VLIW for power optimization in two major source of power consumption namely, instruction decoder and register file. We consider a split instruction decoder design that enables energy optimizations in the instruction decoder. We evaluate a purely hardware based scheme that gains energy benefits for short idle cycles delimited by frequent transitions. We also propose a new energy-aware instruction scheduling algorithm that provides 14.5% and 17.3% benefit in overall power consumption on an average over the purely hardware based scheme in the context of 2-clustered and 4-clustered VLIW machines.

Higher number of register access due to a large number of inter-cluster move instructions make register files another major source of power consumption apart from degrading performance in clustered VLIW architectures. In the context of register files, we have proposed two new scheduling algorithms for register snooping based clustered VLIW architectures. Our experiments show that the proposed algorithms, *Simple* and *Aggressive* reduce register file power consumption on an average by 6.85% and 11.90% (10.39% and 17.78%) respectively along with performance improvement of 4.81% and 5.34% (9.39% and 11.16%) over traditional algorithm *Greedy* for a 2-Clustered (4-Clustered) VLIW machine.

In the future, we are planning to evaluate the temperature benefits of the proposed power reduction techniques using temperature models such as HotSpot[39]. We are

also planning to integrate and experimentally evaluate the proposed schemes with energy management techniques for functional units.

References

[1] S. G. Abraham, W. M. Meleis, and I. D. Baev. Efficient Backtracking Instruction Schedulers. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pages 301–308, 2000.

[2] J. L. Ayala, A. Veidenbaum, and M. Lpez-Vallejo. Power-aware compilation for register file energy reduction. *International Journal of Parallel Program.*, 31(6):451–467, 2003.

[3] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Profile-based dynamic voltage scheduling using program checkpoints in the copper framework. In *Proceedings of Design, Automation and Test in Europe Conference (DATE)*, March 2002.

[4] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *Proceedings of the international symposium on Microarchitecture*, pages 237–248, Washington, DC, USA, 2001.

[5] M. Chu, K. Fan, and S. Mahlke. Region-based Hierarchical Operation Partitioning for Multicluster Processors. *SIGPLAN Notices*, pages 300–311, 2003.

[6] K. D. Cooper and T. Waterman. Understanding energy consumption on the c62x. In *Proceedings of the Work. on Compilers and Operating Systems for Low Power*, 2002.

[7] J. Derby and J. Moreno. A High-performance Embedded DSP Core with Novel SIMD Features. In *Proceedings of 2003 International Conference on Acoustics, Speech, and Signal Processing*, 2003.

[8] G. Desoli. Instruction Assignment for Clustered VLIW DSP Compilers: A New Approach. Technical Report, Hewlett-Packard, 1998.

[9] S. Dropsho, V. Kursun, D. H. Albonesi, S. Dwarkadas, and E. G. Friedman. Managing Static Leakage Energy in Microprocessor Functional Units. In *Proceedings of the International Symposium on Microarchitecture*, pages 321–332, Los Alamitos, CA, USA, 2002.

[10] P. Faraboschi, G. Brown, J. A. Fisher, and G. Desoli. Clustered Instruction-level Parallel Processors. Technical report, Hewlett-Packard, 1998.

[11] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: A Technology Platform for Customizable VLIW Embedded Processing. In *Proceedings of 27th annual International Symposium on Computer architecture*, pages 203–213, 2000.

[12] J. A. Fisher. Very Long Instruction Word Architectures and the ELI-512. In *25 years of the International symposia on Computer architecture (selected papers)*, pages 263–273, 1998.

[13] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy Caches: Simple Techniques for Reducing Leakage Power. In *Proceedings of the International Symposium on Computer Architecture*, pages 148–157, Washington, DC, USA, 2002.

[14] J. Fridman and Z. Greefield. The TigerSHARC DSP architecture. *IEEE Micro*, pages 66–76, 2000.

[15] B. M.-S. Gokhan Memic and W. Hu. NetBench: A Benchmarking Suit for Network Processor. *CARES Technical Report*, 2002.

[16] D. R. Gonzales. Micro-risc architecture for the wireless market. *IEEE Micro*, 19(4):30–37, 1999.

[17] M. Guthaus, J. Ringenberg, and D. Ernst. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. *IEEE workshop on Workload Characterization*, 2001.

[18] Z. Hu and M. Martonosi. Reducing register file power consumption by exploiting value lifetime. In *Workshop on Complexity Effectice Design at ISCA-27)*, June 2000.

[19] K. Kailas, A. Agrawala, and K. Ebcioglu. CARS: A New Code Generation Framework for Clustered ILP Processors. In *Proceedings of International Symposium on High-Performance Computer Architecture*, page 133, 2001.

[20] D. Kannan, A. Shrivastava, S. Bhardwaj, and S. Vrudhul. Power reduction of functional units considering temperature and process variations. *VLSI-DESIGN*, pages 533–539, 2008.

[21] H. S. Kim, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Adapting Instruction Level Parallelism for Optimizing Leakage in VLIW Architectures. In *Proceedings of Conference on Language, Compiler, and Tool for Embedded Systems*, pages 275–, 2003.

[22] N. S. Kim and T. Mudge. Reducing register ports using delayed write-back queues and operand pre-fetch. In *Proceedings of the international conference on Supercomputing*, pages 172–182, New York, NY, USA, 2003. ACM Press.

[23] W.-A. Kuo, T. Hwang, and A. C.-H. Wu. Decomposition of Instruction Decoders for Low-power Designs. *ACM Transaction on Design and Automation of Electronic Systems*, 11(4), 2006.

[24] V. Kursun and E. G. Friedman. Low swing Dual Threshold Voltage Domino Logic. In *Proceedings of the ACM Great Lakes Symposium on VLSI*, pages 47–52, New York, NY, USA, 2002.

[25] V. S. Lapinskii, M. F. Jacome, and G. A. De Veciana. Cluster Assignment for High-Performance Embedded VLIW processors. *ACM Transaction on Design and Automation of Electronic Systems*, pages 430–454, 2002.

[26] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. *International Symposium on Microarchitecture*, 1997.

[27] M. N. Mamidipaka. *Power estimation of low-power high-performance memory structures*. PhD thesis, Long Beach, CA, USA, 2004. Chair-Dutt, Nikil.

[28] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: speculation control for energy reduction. In *Proceedings of the international symposium on Computer architecture*, pages 132–141, Washington, DC, USA, 1998. IEEE Computer Society.

[29] M. Mutyam, F. Li, V. Narayanan, M. Kandemir, and M. J. Irwin. Compiler-directed thermal management for vliw functional units. *SIGPLAN Not.*, 41(7):163–172, 2006.

[30] R. Nagpal and Y. N. Srikant. A Graph Matching Based Integrated Scheduling Framework for Clustered VLIW Processors. In *Proceedings of ICPP Workshop on Compile and Runtime Techniques Parallel Computing*, pages 530–537, 2004.

[31] R. Nagpal and Y. N. Srikant. Integrated Temporal and Spatial Scheduling for Extended Operand Clustered VLIW Processors. In *Proceedings of Conference on computing frontiers*, pages 457–470, 2004.

[32] E. Ozer, S. Banerjia, and T. M. Conte. Unified Assign and Schedule: A New Approach to Scheduling for Clustered Register File Microarchitectures. In *Proceedings of International Symposium on Microarchitecture*, pages 308–315, 1998.

[33] S. Park, A. Shrivastava, N. Dutt, A. Nicolau, Y. Paek, and E. Earlie. Bypass aware instruction scheduling for register file power reduction. In *Proceedings of the conference on Language, compilers and tool support for embedded systems*, pages 173–181, 2006.

[34] G. G. Pechanek and S. Vassiliadis. The ManArray Embedded Processor Architecture. In *Proceedings of Euromicro Conference*, pages 348–355, 2000.

[35] S. Rele, S. Pande, S. Onder, and R. Gupta. Optimizing Static Power Dissipation by Functional Units in Superscalar Processors. In *Proceedings of 11th International Conference on Compiler Construction*, pages 261–275, 2002.

[36] S. Rixner, W. J. Dally, B. Khailany, P. Mattson, U. J. Kapasi, and J. D. Owens. Register organization for media processing. *Proceedings of International Symposium on High Performance Computer Architecture*, pages 375–386, 2000.

[37] J. Scott. Designing the low-power m.core architecture. In *Power Driven Microarchitecture Workshop at ISCA98*, June 1998.

[38] N. Seshan. High VelociTI Processing. *IEEE Signal Processing Magazine*, March 1998.

[39] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 2–13, New York, NY, USA, 2003. ACM.

[40] Texas Instruments Inc. TMS320C6000 CPU and Instruction Set reference Guide. http://www.ti.com/sc/docs/products/dsp/c6000/index.htm, 1998.

[41] J. H. Tseng and K. Asanovic. Energy-efficient register access. In *SBCCI '00: Proceedings of the 13th symposium on Integrated circuits and systems design*, page 377, Washington, DC, USA, 2000. IEEE Computer Society.

[42] H. Yun and J. Kim. Power-aware Modulo Scheduling for High-Performance VLIW Processors. In *Proceedings of 2001 International Symposium on Low Power Electronics and Design*, pages 40–45, 2001.

[43] W. Zhang, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, D. Duarte, and Y.-F. Tsai. Exploiting VLIW Schedule Slacks for Dynamic and Leakage Energy Reduction. In *Proceedings of International Symposium on Microarchitecture*, pages 102–113, 2001.

A Cluster Scheduling Problem

We are given a set of operation types, a set of resource types, and a relation between these two sets. This relation may not be strictly a mapping in general. This is because a resource can perform more than one kind of operation and an operation can be performed on more than one kind of resource. There can be more than one instance of each type of resource. Resource instances can be partitioned into sets each one representing a cluster of resources. We use the following notation:

O	Set of operations
o	An individual operations
R	Set of resources
r	An individual resource
N(RT)	Number of instance of resource type *RT*
N(c,RT)	Number of instance of resource type *RT* in cluster *c*
BW(ci,cj)	Communication bandwidth between cluster *i* and cluster *j*
NT(t,ci,cj)	Number of transfer between cluster *i* and cluster *j* in a time step t
RT(r)	Type of resource *r*
OT(o)	Type of operation *o*
C(r)	Cluster of resource *r*
l	An individual triple of *L*
ES(o_i)	The subset of edges having o_i as successor

Given a data flow graph, which is a partially ordered set (poset) of operations the problem is to assign each operation a time slot, a cluster, and a functional unit in a chosen cluster such that the total number of time slots needed to perform all the operations in poset are minimized while the partial order of operations is honored

and neither any resource nor the ICC facility is over committed. Formally we are given:

(1) A set of operation types *OT*, a set of resource types *RT*, and a set of clusters *C*
(2) A relation between two sets given by *OR*: *OT*$\rightarrow$ *RT* such that $OR(o_i)$ for an operation o_i is a set of resource types $RT_i \subset$ *RT* that can perform these operations, and
(3) A set *L* of triples *(t,c,r)* where *t* is a time slot, $c \in C$ and $r \in R$

A poset of operations can be represented as a directed acyclic graph G(V,E) where V is a set of operations in the poset and E represent the set of edges. Edges are labeled with a number representing the time delay needed between the head operation and the tail operation. Scheduling in this context is the problem of computing a mapping S from a poset P to a set of triple L such that the number of time steps needed to carry out all the operations in the poset is a minimum subject to these constraints (apart from other architectural constraints).

(1) $forall$ *i such that* $S(o_i) = l \in L$, $l.t \geq max(w(e_j))$ *for all* $j \in ES(o_i)$.
(2) $RT(l.r) \in OR(o_i)$
(3) *C(l.r)=l.c*
(4) $\forall\, t\, \forall\, c\, \forall\, rt \sum (l_i.t = t) \wedge (l_i.c = c) \wedge (RT(l_i.r) = rt) \leq N(c,rt)$
(5) $\forall\, t\, \forall i\, \forall j\; NT(t,ci,cj) \leq BW(ci,cj)$