Compiler-Assisted Leakage Energy Optimization for Clustered VLIW Architectures

Rahul Nagpal
Department of Computer Science and Automation
Indian Institute of Science
Bangalore, India
rahul@csa.iisc.ernet.in

Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science
Bangalore, India
srikant@csa.iisc.ernet.in

ABSTRACT

Miniaturization of devices and the ensuing decrease in the threshold voltage has led to a substantial increase in the leakage component of the total processor energy consumption. Relatively simpler issue logic and the presence of a large number of function units in the VLIW and the clustered VLIW architectures attribute a large fraction of this leakage energy consumption in the functional units. However, functional units are not fully utilized in the VLIW architectures because of the inherent variations in the ILP of the programs. This underutilization is even more pronounced in the context of clustered VLIW architectures because of the contentions for the limited number of slow inter-cluster communication channels which lead to many short idle cycles.

In the past, some architectural schemes have been proposed to obtain leakage energy benefits by aggressively exploiting the idleness of functional units. However, presence of many short idle cycles cause frequent transitions from the active mode to the sleep mode and vice-versa and adversely affects the energy benefits of a purely hardware based scheme. In this paper, we propose and evaluate a compiler instruction scheduling algorithm that assist such a hardware based scheme in the context of VLIW and clustered VLIW architectures. The proposed scheme exploits the scheduling slacks of instructions to orchestrate the functional unit mapping with the objective of reducing the number of transitions in functional units thereby keeping them off for a longer duration. The proposed compiler-assisted scheme obtains a further 12% reduction of energy consumption of functional units with negligible performance degradation over a hardware-only scheme for a VLIW architecture. The benefits are 15% and 17% in the context of a 2-clustered and a 4-clustered VLIW architecture respectively. Our test bed uses the Trimaran compiler infrastructure.

EMSOFT'06, October 22–25, 2006, Seoul, Korea.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors-Code generation; Compilers; Optimization

General Terms

Algorithms

Keywords

Scheduling, Clustered VLIW Processors, Leakage Energy, Energy-Aware Scheduling

1. INTRODUCTION

The ongoing improvements in the semiconductor technology bring along various challenges[7]. One such challenge is the rising level of the leakage energy consumption in the logic. The transistor density doubles every eighteen months by packing more logic into the same area. However, this increase in the transistor density requires reducing the supply voltage in order to operate the circuit reliably. The reduction in supply voltage also requires reduction in the threshold voltage in order to maintain the speedup and this leads to an exponential rise in the leakage component of the energy consumption[25]. With the 70nm and smaller technologies currently in fabrication, the leakage energy is on par with the dynamic energy consumption. In future technologies the leakage energy will further dominate the overall energy consumption[32].

VLIW and clustered VLIW architectures rely on compile-time scheduling. This simplifies the issue logic by alleviating the need for a dedicated hardware for scheduling. Thus, a significant fraction of the total leakage energy consumption in VLIW architectures is attributed to functional units. The frequent access of functional units raises the temperature level and makes the leakage energy consumption even worse. Though, the exact percentage depends upon the architecture and circuit details, earlier studies report that 30% to 35% of the static energy conumption in a VLIW architecture is attributed to functional units[19]. An architecture level model developed in [8] also confirms that the leakage energy consumption in functional units constitutes a noticeable fraction of the overall processor leakage energy consumption despite having a smaller transistor count com-

pared to the caches. Thus, optimizing leakage energy in functional resources is becoming more important by each process generation.

However, these architectures are often designed targeting embedded domains where the real-time performance is of utmost importance. Thus, the design is often optimized for the peak performance and as a result, the functional units are underutilized due to the inherent variations in the ILP of the programs. Clustered VLIW architectures improve over the VLIW architectures by solving the scalability problem (in order to obtain a better clock rate) by distributing functional units among different clusters[13]. However, contentions for the limited number of slow inter-cluster communication channels introduce many short idle cycles and makes the utilization of functional units worse.

The underutilization of functional resources can be exploited to reduce leakage energy consumption. Some earlier work in this area reports leakage energy management at a coarser granularity of loop level[19] or block level[31]. However, the rising level of leakage energy in current and future process technologies requires aggressive leakage energy management even for short idle periods. One such purely hardware based scheme in the context of a superscalar architecture is due to Albonesi et al.[11]. 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[20]. However, such a fast transition incurs moderate amount of energy penalty. Their scheme 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.

In this paper, we propose and evaluate a compiler instruction scheduling algorithm that assists such a hardware based scheme in achieving better energy savings in the context of VLIW and clustered VLIW architectures. Whereas the hardware scheme suffers from a limited program view, a compiler can analyze whole program regions and is capable of orchestrating the mapping between operations and functional units. The proposed scheme exploits the scheduling slacks of the instructions to maximize the simultaneous idle time and usage of functional units, thereby reducing the number of transitions drastically. This reduction in the number of transitions leads to significant improvements in energy savings over those obtained by a purely hardware based scheme. Moreover, since the proposed scheme keeps a limited number of functional units active and use them as much as possible, it generates a more balanced schedule which helps to reduce the peak power and the step power[35].

The rest of the paper is organized as follows. Section 2 provides a motivation for this work along with some quantitative results. Section 3 describes our new instruction scheduling algorithm and presents an example to show the benefits of the proposed scheme. Section 4 provides detailed experimental results and analysis. Section 5 describes the earlier work done in the area of instruction scheduling for clustered architectures, architectural approaches for leakage energy management, and energy-aware scheduling for VLIW architectures. Section 6 concludes this paper with future directions for this work.

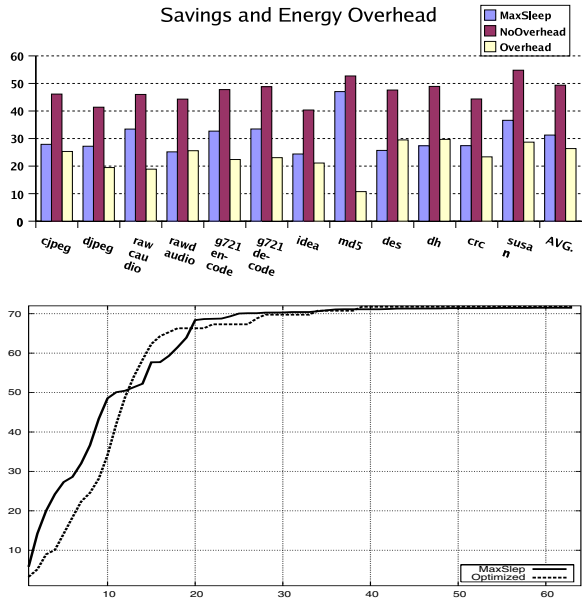

Figure 1: (a) % Savings for 'MaxSleep' and 'NoOverhead' Policies (b) % Cumulative Distribution of Idle Cycles

2. MOTIVATION

The VLIW and clustered VLIW class of architectures are in widespread use in the embedded domain. The primary reason for their success in this domain is high level of ILP in the embedded workload and the suitability of a compiler scheduling algorithm to map this explicit ILP to available hardware. In order to satisfy the demand for high performance in embedded applications (most of which are real-time applications), these architectures use more and more number of functional units. However, the inherent variations in the ILP of the programs lead to underutilization of functional units. We observe that integer ALUs are idle for 60% of the time on an average for a collection of media benchmarks. This idleness figure is for a VLIW configuration having only a moderate number of ALUs so as to achieve 95% of the peak performance (details of our experimental setup and energy model appear in a later section). The idleness is even more pronounced for a clustered VLIW configuration because of the contention for a limited number of slow interconnects which manifests itself in the form of many short idle cycles.

A hardware based scheme such as 'MaxSleep' proposed in [11] puts a functional unit into low leakage mode after an idleness of one cycle and thus saves leakage energy. However, if there are many short idle cycles then there are many transitions and the transition overheads adversely affect the benefit gained by such a scheme. Figure 1 (a) presents the energy savings obtained by a 'MaxSleep', energy savings obtained by a 'NoOverhead' scheme which is a hypothetical scheme (same as 'MaxSleep') but does not incur any transition energy overheads and % energy overhead of 'MaxSleep' due to transitions as compared to that of 'NoOverhead' scheme for integer ALUs in a 2-cluster configuration. These results clearly indicate that the 'NoOverhead' scheme is able to achieve an average savings of 50%

in total energy, where as the average savings for 'MaxSleep' is only 31%. 'MaxSleep' has an average energy overhead of 26% (due to transitions) as compared to the 'NoOverhead' scheme. These results are also in agreement with the results presented in [11]. Thus, reducing the number of transitions will increase the idleness duration for functional units and improves the energy benefits of a hardware based scheme.

Motivated by this, we have developed a scheduling algorithm in the context of VLIW and clustered VLIW architectures that leverage the available slack in scheduling instructions in order to keep the idle functional units idle for a longer duration while maximizing the utilization of active functional units. Figure 1 (b) shows the average cumulative distribution of idle cycles in integer ALUs for a 2-clustered machine on our collection of benchmarks. The graph for 'MaxSleep' clearly shows many small idle cycles constitute a large percentage of overall cycles. 50% of total 71% idle cycles have a duration less than or equal to 10 cycles. The graph after applying our scheduling scheme is shown with title optimized. This shows that the many small idle cycles have been converted to large idle cycles by reducing transitions and only 34% of overall idle cycles are now less than 10 cycle. Idle cycle of length between 10 to 20 cycles constitute 32% of total idleness for 'Optimized' scheme while for the 'MaxSleep' scheme this is only 18%. This clearly shows that our scheme is able to exploit the slack to reduce the number of transitions thereby increasing the duration of idle periods.

3. THE SCHEDULING ALGORITHM

The Elcor backend of the Trimaran infrastructure has a cycle scheduling algorithm designed and implemented for flat VLIW architectures[4][5]. We have modified this algorithm to perform leakage energy optimization for VLIW as well as clustered VLIW architectures. Another loop has been added inside the main scheduling loop of the cycle scheduler to perform cluster scheduling in an integrated fashion. The integrated approach[30][24][17] to cluster scheduling makes the cluster assignment decision during temporal scheduling. This is in contrast to phase-decoupled approaches[6][10][21] which perform cluster assignment prior to temporal scheduling. Essentially, our integrated scheduling algorithm for leakage energy optimization consists of the following main steps.

1. Prioritizing the ready instructions
2. Assignment of a cluster to the selected instruction
3. Assignment of functional unit to selected instruction in target cluster

In what follows, we describe how each of these step is performed in our algorithm. An outline is shown in Algorithm 1.

3.1 Prioritizing the Ready Instructions

Instructions in the ReadyList are prioritized using a priority function that uses instruction slack and number of consumers of the instruction. Instructions with less slack should be scheduled early and are given higher priority over instruction with more slack to avoid unnecessary stretching of the schedule. Instructions with the same slack values are

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 schduling
  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, TragetCluster)
    end if
    while (CurrentAlternative = AlternativeList.pop()) do
      if (FU in CurrentAlternative are active) then
        Schedule CurrentOperation using CurrentAlternative in
        CurrentCycle on TargetCluster.Cluster using TragetClus-
        ter.CommOption
        SUCCESSS ← TRUE
      end if
    end while
    if (!SUCCESS and Slack ≤ SLACK_THRESHOLD)
    then
      FallBackAlternative ← DetermineBestAlternative(
                          AlternativeList, TargetCluster)
      Wakeup FU in FallBackAlternative
      Schedule CurrentOperation using FallBackAlternative in
      CurrentCycle on TragetCluster.Cluster using TragetClus-
      ter.CommOption
    else
      ReadyList.add(CurrentOperation)
    end if
  end while
  CurrentCycle ← CurrentCycle + 1
  ReadyList.update()
  updateFUStatus()
end while
```

Procedure 2 DetermineBestCluster

```
FirstTarget.Cluster ← −1
FirstTarget.CommCost ← 1000000;
SecondTarget.Cluster ← −1
SecondTarget.CommCost ← 1000000
for (CurrentCluster ranging from FirstCluster through LastClus-
ter) do
  Compute the Cross-path Requirements in CurrentCommOption
  Compute the Communication Cost in CurrentCommCost
  if (FU and Cross-paths required by CurrentOperation are avail-
  able in CurrentCycle for CurrentCluster) then
    if (FU under consideration is in active mode and FirstTar-
    get.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
```

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.

Scheduling slack of an instruction is defined as the difference between the earliest start time and the latest finish time of the instruction. Traditionally, slack is determined statically during dependence graph analysis before the scheduling begins, assuming a machine with infinite resources of each type. This calculation is inherently pessimistic as any real machine will have contentions for resources which prolongs the execution time. Since our algorithm exploits slack of instructions to delay their execution in order to save energy without affecting performance, a better quantification of available slack is of utmost importance. 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 begin (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.

3.2 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 functional unit of the type needed 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

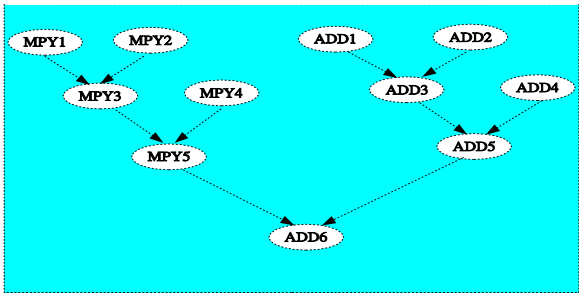

Figure 2: An Example Data Dependency Graph

Schedule 1

1	MPY1/M1, MPY2/M2, ADD1/A1, ADD2/A2
2	MPY4/M1, ADD4/A1, ADD3/A2
3	MPY3/M1, ADD5/A1
4	
5	MPY5/M1
6	
7	ADD6/A1
8	

Schedule 2

1	MPY1/M1, MPY2/M2
2	MPY4/M1, ADD1/A1
3	MPY3/M1,ADD2/A1
4	ADD3/A1
5	MPY5/M1,ADD4/A1
6	ADD5/A1
7	ADD6/A1
8	

Schedule 3

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

Schedule 4

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

Figure 3: (a) Schedule 1 (b) Schedule 2 (c) Schedule 3 (d) Schedule 4

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[27].

3.3 Functional Unit Binding

A functional unit binding scheme decides the binding of a chosen instruction to a functional unit. The algorithm maintains a FU map that explicitly keeps track of the status of each functional unit. A functional unit is marked to be in sleep mode after one cycle of idleness and activated on next use.

If the functional unit required for the instruction under consideration is active in the target cluster, it is bound as usual. 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 functional unit required by the instruction is woken up. In case there is more than one alternative available (for activating), the functional unit which is in sleep mode for a longer duration is woken up in order to amortize the cost of waking up. 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 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 comes below the threshold ensuring that it is scheduled. Hence the algorithm is guaranteed to terminate.

3.4 An Example

In this subsection, we present an example to illustrate how the available slack of instructions is exploited by the proposed scheduling algorithm to get energy benefits without hurting performance. Figure 2 shows an example data dependency graph and Figure 3 shows some schedules. Let us first discuss schedule 1 and schedule 2 for a plain VLIW architecture having two adders (namely A1 and A2) and two multipliers (namely M1 and M2) both of which are pipelined. 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.

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, the scheduling of ADD operations is delayed as well as serialized, capitalizing on 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 takes same execution cycles as schedule 1 but is better than schedule 1 in many ways. First of all, schedule 2 make use of only one adder compared to schedule 1 which clearly reduces the leakage energy consumption because the second adder is always in low-leakage mode. Secondly, there are fewer transitions from active to low leakage mode and vice versa in schedule 2 as compared to schedule 1. Assuming availability of hardware mechanisms that put a functional unit into low leakage mode after 1 cycle of idleness, the number of transitions from active mode to low leakage mode and vice-versa for M1, M2, A1, and A2 are 2, 2, 4, and 2 for schedule 1 and 2,2,2, and 0 for schedule 2. Schedule 2 is also more balanced as compare 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 (2,2,2,1,2,1,1,0). 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 [35]. Thus, it is clear that the proposed scheme is capable of reducing leakage energy 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 corresponding VLIW schedule 1. ADD3 (MPY3 resp.) is scheduled in cycle 3 (cycle 4 resp.) because it takes a cycle to transfer the result of ADD2 (MPY2 resp.) from cluster 2. The Total schedule length is 9 cycles.

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 map all the operation to just cluster 1, keeping cluster 2 completely idle, thereby saving even more leakage energy. The number of transitions from active mode to low leakage mode and vice-versa for M1, M2, A1, and A2 are 2, 2, 4, 2 for schedule 1 and 2, 0, 2, 0 for schedule 2 respectively. Finally schedule 2 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 second is (1,1,2,2,1,2,1,0).

4. EXPERIMENTAL EVALUATION

4.1 Setup

We have used the Trimaran suite[4] 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 back-end. 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[22][1] *(viz. cjpeg, djpeg, rawcaudio, rawdaudio, g721encode, g721decode, md5, des, and idea)*, two from netbench[15][3] *(viz. crc, and dh)*, and one *(susan)* is from MiBench[16][2]. We have tried other benchmarks from these suits 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 an unclustered, a two-cluster machine and a four-cluster 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 number of functional units selected for the VLIW configurations are such that the performance achieved using this configuration is within 95% of the peak performance achieved by using many more functional units. This moderate number of functional resources guarantees

that the benefits reported have not been obtained by trivially putting the numerous idle functional units into the low leakage mode. Also, we report results only for Integer ALUs which are heavily used and pose a challenge for any leakage energy management scheme. Thus the benefits reported here have not been magnified by the leakage energy benefits of the load-store, branch, and FP units which are mostly idle.

4.2 Energy Model

We have used the same analytical energy model as in [11] to directly compare the energy benefits of the proposed scheme over the pure hardware based scheme proposed in[11]. We briefly describe this model here. The reader is referred to [11] for details. The total energy in a functional unit in this model is determined as follows:

$$\mathbf{E'_{total}} = \mathbf{DynamicEnergy} + \mathbf{LeakageEnergy} + \mathbf{TransitionEnergy} + \mathbf{SleepModeEnergy}$$

$$\mathbf{E'_{total}} = \mathbf{n_A}(\alpha\mathbf{E_A} + (\mathbf{1}-\mathbf{D})\mathbf{E_{S_1}}) + (\mathbf{n_A D} + \mathbf{n_{UI}}) * (\alpha\mathbf{E_{s_0}} + (\mathbf{1}-\alpha)\mathbf{E_{s_1}}) + \mathbf{M_z}((\mathbf{1}-\alpha)\mathbf{E_A} + \mathbf{E_{Sleep}}) + \mathbf{n_Z E_{s0}}$$

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.

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

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 :

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

The technology parameters that we have used (s=0.01 and $E_{Sleep}/E_A = 0.01$) are also the same as in [11] in order to compare the benefits of our scheduling algorithm to the hardware-only scheme. Considering the current 70nm fabrication technology where leakage energy is on par with dynamic energy, we set p to 0.5 .α is activity factor and $\mathbf{D}$ is the duty cycle of the clock. We use a typical value of 0.5 for both of these parameters in our simulation as in [11]. Sensitivity results with different values of α and $\mathbf{p}$ can be found in associated technical report[28].

4.3 Results

We have performed a detailed experimental evaluation of the proposed scheme in terms of the reduction in the number of transitions and the associated energy savings. We present results for the hardware-only scheme from [11] called 'MaxSleep' as well as for our scheduling scheme that assists the hardware based scheme. We call this scheme 'Optimized'. 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 overheads of transitions. This scheme represents a theoretical ideal against which a leakage energy management scheme can be compared for its effectiveness.

Figure 4 (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 48.34%, 53.97%, and 58.29% 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 4 (b) shows the energy overhead of 'MaxSleep' and 'Optimized' schemes as compared to the 'NoOverhead' scheme. 'MaxSleep' and 'Optimized' schemes show average energy overheads of 23.59% and 13.32% respectively as compared to the 'NoOverhead' scheme. The proposed 'Optimized' scheme reduces the total energy overhead by 11.85% 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. These results are in agreement with the results presented in [11], where the author mention that the 'MaxSleep' scheme incurs 30% more energy overheads than the 'NoOverhead' scheme (some difference compared to our evaluation is due to change in workload). In [11] the evaluation is based on the spec benchmark in the context of superscalar architecture, whereas our evaluation uses the media benchmark in the context of VLIW and clustered VLIW architectures.

The benefit of our scheme is even more pronounced in the context of clustered architectures. In the context of 2-clustered architecture 'MaxSleep' and 'Optimized' have average energy overheads of 26.36% and 13.26% respectively as compared to the 'NoOverhead' scheme (Refer Figure 5 (a)). The energy benefits of 'Optimized' over Maxsleep is 15.11% in context of 2 clustered architecture. For a 4-clustered configuration, 'MaxSleep' and 'Optimized' incur 27.02% and 12.15% overhead as compared to 'NoOverhead' scheme (Refer Figure 5 (b)). The 'Optimized' scheme improves over the 'MaxSleep' scheme on the average by 16.92% 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 during which functional units are waiting for operands to arrive from other clusters. 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.

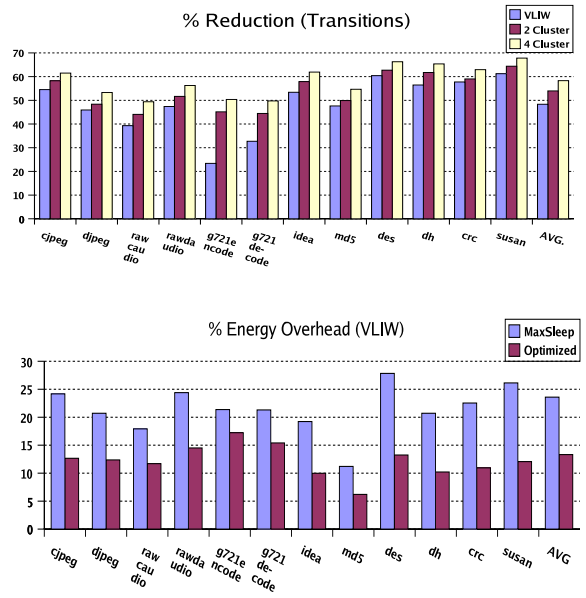

Figure 4: (a) % Reduction in Transitions with scheduling w.r.t. Hardware only Scheme (b) % Increase in energy w.r.t Hypothetical No-overhead Scheme (VLIW)

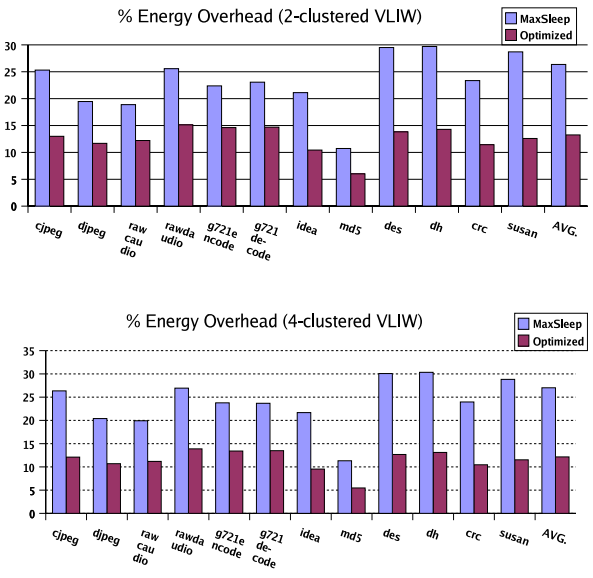

Figure 5: % Increase in energy w.r.t Hypothetical No-overhead Scheme (a) 2 Cluster (b) 4 Cluster

5. RELATED WORK

In this section, we briefly describe the earlier work done in the area of instruction scheduling for clustered architectures, architectural approaches for leakage energy management, and energy aware scheduling for VLIW architectures.

5.1 Instruction Scheduling for Clustered Architectures

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. A major argument in favor of this approach is that a partitioner having a global view of a DFG can perform a better job of reducing inter-cluster communication and load-balancing. The proposals in this direction are due to Ellis[12], Desoli[10], Gonzalez[6], Lapinskii[21], Mahlke[9], Lee[23], and Nystrom[29]. However, the phase-decoupled approach is known to suffer from the phase ordering problem. Since the spatial scheduler has only an approximate knowledge of load on clusters, usage of functional units, and cross-paths, approximate load-balancing often leads to cluster assignments which unnecessarily constrain the temporal scheduler in the later phase. Moreover, some of these schemes are designed for reducing inter-cluster communication and end up reducing the ILP in the program in this pursuit[30][17].

An integrated approach to scheduling combats the phase-ordering problem by combining spatial and temporal scheduling decisions in a single phase. The integrated approach considers instructions ready to be scheduled in a cycle and the available clusters in some priority order. The priority order for considering instructions is decided based on mobility, scheduling alternatives, the number of successors of an instruction etc. Similarly, the priority order for considering clusters is decided based on communication cost of assignment, earliest possible schedule time etc. An instruction is assigned a cluster to reduce communication or to schedule it at the earliest. The proposals in this direction are due to Ozer[30], Leupers[24], Kailas[17], Zalamea[36], and Nagpal[27][26].

5.2 Architectural Approaches for Leakage Energy Management

Study of leakage energy management at the architectural level has mostly focused on storage structure such as cache. Yang et al., propose power supply gating of L1 cache cells[34]. Kaxiras et al., dynamically adjust the interval after which a cache line is put into low leakage mode[18]. Flaunter et al., propose a state-preserving drowsy cache design and a simple control scheme which is able to deliver most of the leakage energy benefits[14].

In contrast to storage structures, little work has been done on architecture level leakage energy management in the context of functional units. Our work directly improves over the work due to Albonesi et al. [11]. This work proposes and evaluates an architectural policy for aggressively controlling leakage energy in integer ALUs. The 'MaxSleep' policy puts a functional unit into low leakage mode after

one cycle of idleness. This scheme depends on dual threshold domino logic circuit with sleep mode proposed in [20] which has no delay penalty of transition between active mode and sleep mode. Their performance evaluation using an analytical energy models in the context of spec benchmarks for superscalar architectures shows that for technology such as 70nm, the leakage energy benefit gained by such an aggressive scheme is significant. However the overhead of transitions from active mode into low-leakage mode and vice-versa are significant (on an average 30% when compared to a 'NoOverhead' scheme).

5.3 Energy-Efficient Scheduling

Zhang et al.,[37] have proposed a rescheduling scheme to reduce dynamic and leakage energy in the functional units of a VLIW processor by exploiting the remnant slack of a performance-oriented schedule. In contrast, our approach works on raw unscheduled code with all the available slack for scheduling and complements a hardware based mechanism for leakage energy management. Kim et al.,[19] 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). The calculation is done at the loop level granularity assuming that there is little variation in the ILP within the loop. Their scheme keep only canonical subset of functional units that is sufficient to exploit this approximated ILP active. In contrast, our approach adaptively applies leakage energy management at a finer granularity based on available ILP. Gupta et al.,[31] 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. ISA and architectural support is needed to switch on and off the functional unit at the boundaries of power blocks and nullify spurious on-off. Kim et al.,[35] 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. Though our algorithm is not directly designed towards improving the peak power and step power dissipation, it generates a more balanced schedule. The closest to our work is the work by Vardhan et al.[33]. They extend the standard list scheduling algorithm but prioritize instruction selection (from the ready queue) based on a closeness metric. This algorithm is evaluated for an in-order superscalar processor.

6. CONCLUSIONS AND FUTURE DIRECTIONS

In this work, we have proposed a new energy-aware instruction scheduling algorithm for VLIW and clustered VLIW architectures that is capable of reducing the number of transitions by exploiting the scheduling slack of instructions. The experimental evaluation reveals that the proposed scheme is able to reduce the number of transitions by approximately 48% (more for clustered architectures). This results in 11.85% energy savings in the context of VLIW architecture while 15.11% and 16.92% energy savings in the context of 2-clustered and 4-clustered VLIW architecture respectively, as compared to a purely hardware based scheme. In addition, the proposed scheme is able to generate a more balanced schedule that help in reducing the peak power and step power dissipation of the processor. In future, we would like to integrate the proposed scheme for leakage energy management with the slack based approach to dynamic energy management.

7. REFERENCES

[1] MediaBench . *http://cares.icsl.ucla.edu/MediaBench/.*

[2] MiBench. *http://www.eecs.umich.edu/mibench/.*

[3] NetBench. *http://cares.icsl.ucla.edu/NetBench/.*

[4] Trimaran System. *http://www.trimaran.org/.*

[5] S. G. Abraham, W. M. Meleis, and I. D. Baev. Efficient Backtracking Instruction Schedulers. In *Proc. of Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 301–308, 2000.

[6] A. Aleta, J. M. Codina, J. Sanchez, and A. Gonzalez. Graph-partitioning based Instruction Scheduling for Clustered Processors. In *Proc. of Intl. Symp. on Microarchitecture*, pages 150–159, 2001.

[7] S. Borkar. Design Challenges of Technology Scaling. *IEEE Micro*, 19(4):23–29, 1999.

[8] J. A. Butts and G. S. Sohi. A Static Power Model for Architects. In *Proc. of the Intl. Symp. on Microarchitecture*, pages 191–201, New York, NY, USA, 2000.

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

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

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

[12] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, 1986.

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

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

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

[16] M. Guthaus, J. Ringenberg, and D. Ernst. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. *IEEE 4th Annual Workshop on Workload Characterization*, 2001.

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

[18] S. Kaxiras, Z. Hu, and M. Martonosi. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. In *Proc. of the Intl. Symp. on*

Computer Architecture, pages 240–251, New York, NY, USA, 2001.

[19] H. S. Kim, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. Adapting Instruction Level Parallelism for Optimizing Leakage in VLIW Architectures. In *Proc. of Conf. on Language, Compiler, and Tool for Embedded Systems*, pages 275–283, 2003.

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

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

[22] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proc. of Intl. Symp. on Microarchitecture*, 1997.

[23] W. Lee, D. Puppin, S. Swenson, and S. Amarasinghe. Convergent Scheduling. In *Proc. of Intl. Symp. on Microarchitecture*, pages 111–122, 2002.

[24] R. Leupers. Instruction Scheduling for Clustered VLIW DSPs. In *Proc. of Intl. Conf. on Parallel Architectures and Compilation Techniques*, page 291, Washington, DC, USA, 2000.

[25] T. N. Mudge. Power: A First Class Design Constraint for Future Architecture and Automation. In *Proc. of the Intl. Conf. on High Performance Computing*, pages 215–224, London, UK, 2000. Springer-Verlag.

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

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

[28] R. Nagpal and Y. N. Srikant. Compiler-Assisted Leakage Energy Optimization for Clustered VLIW Architectures. Technical Report, Dept. of CSA, Indian Institute of Science (http://www.archive.csa.iisc.ernet.in/TR), 2005.

[29] E. Nystrom and A. E. Eichenberger. Effective Cluster Assignment for Modulo Scheduling. In *Proc. of 31st annual ACM/IEEE Intl. Symp. on Microarchitecture*, pages 103–114, 1998.

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

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

[32] D. Sylvester and H. Kaul. Power-Driven Challenges in Nanometer Design. *IEEE Design and Test of Computers*, 18(6):12–22, 2001.

[33] K. A. Vardhan and Y. N. Srikant. Transition Aware Scheduling: Increasing Continuous Idle-Periods in Resource Units. In *Proc. of the Conf. on Computing frontiers*, pages 189–198, New York, NY, USA, 2005.

[34] S.-H. Yang, B. Falsafi, M. D. Powell, K. Roy, and T. N. Vijaykumar. An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-Submicron High-Performance I-Caches. In *Proc. of the Intl. Symp. on High-Performance Computer Architecture*, page 147, Washington, DC, USA, 2001.

[35] H. Yun and J. Kim. Power-aware Modulo Scheduling for High-Performance VLIW Processors. In *Proc. of Intl. Symp. on Low Power Electronics and Design*, pages 40–45, 2001.

[36] J. Zalamea, J. Llosa, E. Ayguade, and M. Valero. Modulo Scheduling with Integrated Register Spilling for Clustered VLIW Architectures. In *Proc. of Intl. Symp. on Microarchitecture*, pages 160–169, 2001.

[37] 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 *Proc. of Intl. Symp. on Microarchitecture*, pages 102–113, 2001.