Exploiting VLIW Schedule Slacks for Dynamic and Leakage Energy Reduction *

W. Zhang, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, D. Duarte, and Y-F. Tsai
Microsystems Design Lab, Pennsylvania State University, University Park, PA 16802.
mdl@cse.psu.edu

Abstract

The mobile computing device market is projected to grow to 16.8 million units in 2004, representing an average annual growth rate of 28% over the five year forecast period [5]. This brings the technologies that optimize system energy to the forefront. As circuits continue to scale in future, it would be important to optimize both leakage and dynamic energy. Effective optimization of leakage and dynamic energy consumption requires a vertical integration of techniques spanning from circuit to software levels.

Schedule slacks in codes executing in VLIW architectures present an opportunity for such an integration. In this paper, we present compiler-directed techniques that take advantage of schedule slacks to optimize leakage and dynamic energy consumption. The proposed techniques have been incorporated into a cycle accurate simulator using parameters extracted from circuit level simulation. Our results show that a unified scheme that uses both dynamic and leakage energy reduction techniques is effective in reducing energy consumption.

1 Introduction

The recent trend has been to consider energy consumption at all phases of hardware and software design. However, many of the energy optimization techniques have focused on the circuit and architectural levels. At the software level, with a few exceptions, energy reduction has been primarily an outcome of performance-oriented optimizations. With the growing importance of limited energy devices, it is vital to design new energy-oriented optimizations [18]. These energy-oriented optimization techniques should focus on reducing energy consumption while keeping performance constant or on sacrificing some performance for much less energy consumption. A recent example of such energy-oriented optimizations is the exploitation of low-power operating modes found in some memory modules.

An important technique to reduce energy consumption is to exploit idleness of system components. The idleness of a system component can be exploited using different techniques. The first approach would be to exploit low-power sleep modes [1]. The trade-off to consider for such an approach would include the time to transition to and from the sleep state and the energy consumed during the transition process itself. Another commonly used approach to exploit idleness in CMOS-based devices is to prolong the operation by reducing the supply voltage. This approach reduces the dynamic energy consumption of the CMOS device, which is given by CV^2 [4], where C is the switched capacitance and V is the supply voltage. While the reduction in supply voltage brings a quadratic reduction in the dynamic energy consumption, it also increases the circuit delay of the CMOS device. Typically, the voltage scaling approach has been shown to provide more energy savings for CMOS devices as compared to completely shutting them down [15]. However, this trend could change as leakage energy becomes a comparable fraction of overall energy budget[1] and as the benefit of supply voltage scaling reduces with smaller supply voltages.

Many of the components in the CPU *datapath in* VLIW machines are not completely utilized during the execution. Slacks present due to lack of available instructions that exercise the specific components, data dependence relations between instructions, and schedule-specific design decisions are the major cause of idleness. These slacks can be exploited to reduce dynamic and leakage energy consumption of a given architecture in multiple ways. Three possible approaches can be considered to exploit the compile-time visible slacks for reducing dynamic energy consumption. In the first technique, the functional units of a given type are replicated and each unit is operated with a different supply voltage and a corresponding clock. This technique requires multiple supply voltage rails, multiple clock frequency domains and voltage converters for interfacing circuits operating at different voltage levels [16]. In [6], a limit study of exploitable slacks using a similar technique is presented for a dynamic instruction schedule processor. The second tech-

*This work was supported in part by Grants from GSRC and NSF CAREER Awards 0093082 and 0093085

[1]Leakage becomes the dominant part of energy consumption for 0.1 micron (and below) technologies [4].

1072-4451/01 $10.00 © 2001 IEEE

nique also replicates functional units of a given type but uses a different architectural implementation for each replica, all operating with the same supply voltage and clock domain. For example, different adders (e.g., carry look-ahead adder, carry select adder, ripple carry adder, etc.) possess different energy and performance tradeoffs. In the third technique, instead of replicating the functional units, the supply voltage and clock frequency to the unit are changed dynamically. In this technique, typically, Phase Locked Loops (PLLs) are employed to dynamically change clock frequency [10, 4]. As the change to a new frequency requires time on the order of a few microseconds, the first two techniques are easier to exploit using compiler-directed optimizations. From the compiler's viewpoint, the compilation strategies adopted for both the first and second technique are similar. Consequently, in this work, we focus only on the first alternative.

In addition to the dynamic energy reduction, two leakage control mechanisms are considered to exploit the compile-time visible slacks for reducing leakage energy. The first leakage control mechanism exploits the state dependence of the leakage current and sets the inputs that have the minimum leakage current when the units are idle [7]. The second mechanism eliminates leakage by cutting the power supply to the units [14]. These techniques can also be employed to reduce leakage in the unused replicated units when optimizing for dynamic energy.

In this paper, we make the following contributions:

- Evaluation of two leakage control mechanisms: input control and supply gating. The trade-off in energy consumption and performance impact of these mechanisms are evaluated through circuit-level simulation of three integer ALU (IALU) components designed in 0.25 micron technology. Further, the dynamic energy consumption and latency of these IALU components when operating at different supply voltages are obtained through simulation. These leakage and energy parameters have been integrated into a cycle-accurate VLIW energy simulation framework built upon the Trimaran infrastructure [17].
- Design and implementation of compiler techniques that utilize the evaluated leakage control and voltage scaling mechanisms to exploit the slacks available in a given schedule. We present algorithms that do not sacrifice any performance while reducing energy consumption as well as algorithms that sacrifice some performance.
- A unified mechanism that combines the benefits of both voltage scaling and leakage control in exploiting schedule slacks. This mechanism is experimentally compared to voltage scaling and leakage control techniques. Experimental results show that the unified scheme can reduce the energy consumption of IALU components by as much as 75% in a typical VLIW architecture.

The remainder of this paper is organized as follows. Architectural and circuit support for reducing dynamic and leakage energy is reviewed in Section 2. Compiler support for our approach to energy minimization is presented in Section 3. Experimental results showing the benefits of our algorithms are given in Section 4. Finally, conclusions are given in Section 5.

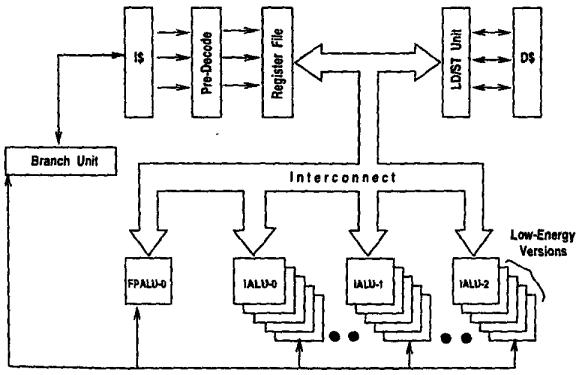

Figure 1. VLIW architecture with multiple IALU versions.

2 Architectural and Circuit Support

2.1 Support for Dynamic Energy Management

We assume a VLIW architecture (see Figure 1) composed of integer ALUs (IALUs), floating point ALUs (FPALUs), one load/store (LD/ST) unit, and one branch (BR) unit. Integer ALUs have different versions that possess different performance (latency) and energy consumption characteristics. In the instruction word, a few control bits are associated with each functional unit to select the appropriate low energy version of the IALU. These control bits are used to route the control signals/data to the appropriate versions. Our algorithms take an already scheduled code and reschedules it. In doing so, it sets the control bits to appropriate values and modifies start time (taking into account data dependencies) and execution length (latency) of operations. The compiler ensures that the rescheduling does not change the issue width. Multiple versions of functional units also involve some circuit overheads. First, when the supply voltages to the different versions are different, a level converter circuit is required to interface the circuits operating at different voltages [16]. Second, as the multiple versions of the functional units operate at different frequencies, there is also a need for multiple clock domains. We limit the overhead of the multiple clock distribution circuitry by using local clock dividers wherever possible. In addition to multiple clock domains, we also require multiple supply rails for supporting the different versions of the units.

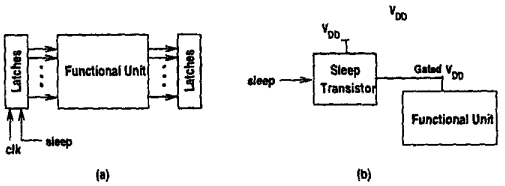

Figure 2. (a) Block level implementation of input vector control scheme. (b) Block diagram for gating supply voltage. When *sleep* signal is activated, the supply voltage to functional unit is gated.

Component	Input Vector Control			Power Supply Gating		
	% LR	IL	RL	% LR	IL	RL
Adder	66	1 cycle	0 cycle	100	480 cycles	480 cycles
Multiplier	18	1 cycle	0 cycle	100	800 cycles	800 cycles
Shifter	86	1 cycle	0 cycle	100	396 cycles	396 cycles

Figure 3. The effectiveness of the leakage control mechanisms. The leakage reduction column gives the average leakage current reduction due to the application of the control mechanism, and the initiation latency column indicates the time it takes for the control mechanism to take effect and recovery latency is the time required to return the functional unit to normal operational state.

2.2 Support for Leakage Energy Management

2.2.1 Input Vector Control

Many researchers have used models to estimate leakage and algorithms to find the minimum and maximum leakage of a given circuit [9]. Leakage, as in the case of dynamic power, depends on the input pattern. This is a consequence of the transistor stacking effect, where a simple two-transistor stack can reduce leakage by a factor of up to 10. Additional stacking can only provide incrementally more savings. The state of transistors in the stack, however, is determined by the inputs. The objective is to find the input pattern that maximizes the number of disabled transistors in a stack. Once this input pattern vector has been found, the input latches of the units can be designed such that a sleep signal sets the value of the unit's inputs to the desired state (See Figure 2(a)). The implementation of the input control technique requires minimal architectural support. The overhead of the input latches is quite small. For example, the area overhead for setting the inputs of the multipliers is less than 10%. A sleep signal is activated whenever the unit is idle. This signal is set in our approach by the compiler as described in Section 3.2. Note that if the switching incurred in setting the input to the desired sleep pattern causes a dynamic energy consumption larger than that produced by the reduction in leakage energy, this technique can increase overall energy consumption. Based on the relative values of the dynamic and leakage energy consumption and the duration of the idleness, the sleep signal needs to be activated intelligently.

We have quantified the leakage energy reduction for different integer ALU components using circuit-level simulation for 0.25 micron, 3.3V supply voltage and 0.48V threshold voltage. Random input patterns were generated for each unit to provide a 95% confidence of finding the input vector that provides the least leakage current [7] and simulations were done for each of them. The second, third and fourth columns of Figure 3 show the leakage power savings, initiation and recovery latencies due to the input control technique for three of the IALU components considered this work.

2.2.2 Power Supply Gating

There are many ways in which power supply gating approach can be implemented [4] but the basic idea is to disconnect the power supply from the unit so that idle units do not consume any leakage energy. Supply gating can be implemented using a sleep transistor that serves as a pass transistor. The supply line V_{DD} passes through this pass transistor to provide a gated supply voltage to the functional unit. The supply voltage to the functional unit is shut down when the sleep signal is activated to turn off the sleep transistor. The sleep transistor is built using a higher threshold voltage than the transistors in the functional unit. Thus, its leakage current in the off state is negligible. In our implementation, we use a sleep transistor per functional unit as shown in Figure 2(d). Implementation of the power supply gating needs careful consideration of sleep mode transistor sizing to consider performance and noise immunity issues. The initiation/recovery latency for this technique is influenced by the diffusion capacitances of the sized sleep transistors. It must also be noted that frequent switching of large sleep transistors has dynamic energy overheads. Thus, we utilize this technique only to shutdown units for longer durations. The fifth, sixth and seventh columns of Figure 3 show the leakage power savings, initiation and recovery latencies due to power supply gating, respectively.

3 Compiler Support

The objective of the compiler support explored in this paper is to exploit the idleness of functional units. In reducing dynamic and leakage energy, the compiler exploits the hardware support discussed in the previous section. The algorithms considered in this section start with an already available schedule. In our implementation, this schedule is obtained using basic block-level [13] or superblock-level [8] scheduling. We start with these *performance-oriented* schedules so as to have a minimal impact on performance while exploiting the slacks for energy reduction.

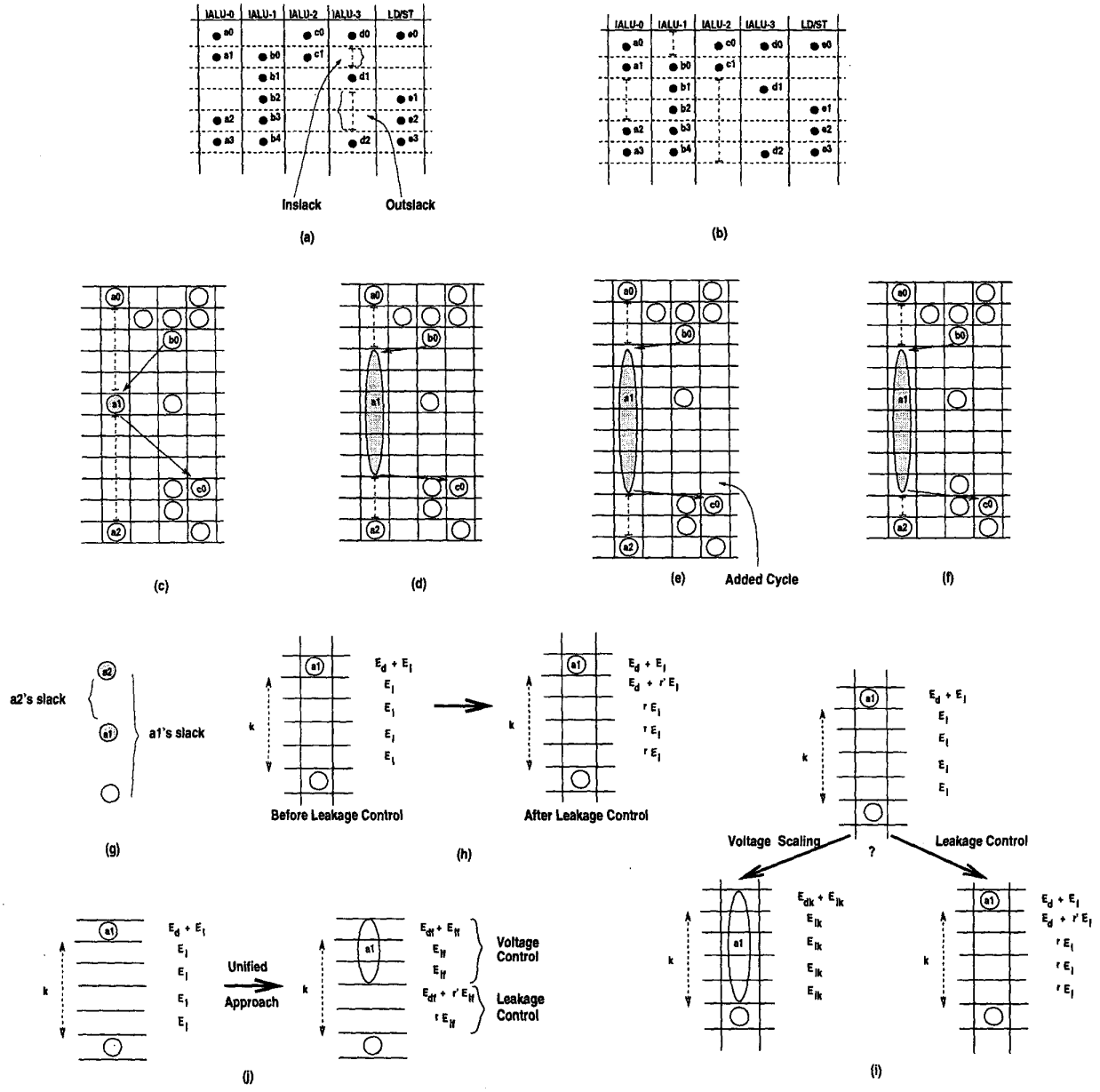

Figure 4. (a) An example schedule where inslack and outslack of operation $d1$ are explicitly marked. (b) Different slacks for the schedule given in (a). (c) An example schedule that is the output of a performance-oriented schedule. (d) Optimizing the slack of operation $a1$ in (c) using Algorithm I. (e) Optimizing the slack of operation $a1$ in (c) using Algorithm II. (f) Optimizing the slack of operation $a1$ in (c) using the modified form of Algorithm II. (g) Conflicting slacks. (h) Impact of leakage control mechanism on energy consumption. (i) Comparison of dynamic energy control and leakage control. (j) Unified approach which uses both leakage control and dynamic energy control on the same slack. Note that dynamic energy consumption is associated with the first cycle of operation for convenience. It actually spans over the entire duration of the operation.

Figure 4(a) presents an example input schedule for the energy optimization phase. All schedule figures in this paper are given as two-dimensional grids. In these schedule figures, each column denotes a functional unit and each row denotes a cycle. For example, in Figure 4(a), a schedule fragment for five functional units: four integer ALUs (IALU-0 thru IALU-3), and a load/store unit (LD/ST) is shown. Operation $a0$ is scheduled to be executed in the first cycle in IALU-0 whereas operation $c1$ is scheduled to be executed in IALU-2 in the second cycle.

For each operation, we define an *inslack* and an *outslack*. The outslack of an operation is the number of cycles between the end of the current operation and the start of the next (earliest) dependent operation. Similarly, the inslack of an operation is the number of cycles between the latest end time of all operations on which the current operation is dependent on and the start of the current operation. In the rest of the paper, the term *slack* refers to the sum of the number of cycles in both inslack and outslack. Note that this slack definition is with respect to data dependences between instructions and is independent of the underlying architecture. However, whether a given slack can be exploited or not depends not only on data dependences but also on the architecture in question. We define a slack as *exploitable* if the corresponding operation can be executed in a slower, low-energy version of a functional unit without violating any data dependencies.

As an example, the inslack and outslack of operation $d1$ are explicitly illustrated in Figure 4(a). Here, we assume that $d1$ is only dependent on $d0$ and that $d2$ is the only operation dependent on $d1$. To illustrate the difference between slack and exploitable slack, we consider the schedule fragment shown in Figure 4(c) which contains four IALUs. We focus on operation $a1$ which has four idle cycles before it and five idle cycles after it. Assuming that $a1$ is dependent on only $b0$ and that $c0$ is dependent on $a1$, we cannot exploit all of these nine slack cycles for reducing energy even if we have a low-energy unit with nine cycles of latency. This is because data dependences with $b0$ and $c0$ put a restriction on the number of cycles that can be used to execute $a1$. Consequently, the inslack and outslack for $a1$ are 2 and 3, respectively. Suppose now that we have two low-energy functional units that can execute $a1$ in 4 and 7 cycles, respectively. Since the operation $a1$ needs to complete in six cycles (latency of the fastest version + slack duration), we can only use the first functional unit. Therefore, the exploitable slack for this operation is 3 (note this is the difference between the latency on the low-energy version on which operation is scheduled and the latency of the original high-performance version).

Note that schedule-specific decisions impact the availability of low-energy functional units for exploiting slacks. Consider, for instance, operation $c1$ scheduled on IALU-2 in Figure 4(b). Assuming that this operation is not involved in any data dependence relation, it should normally be possible to prolong its execution and save energy. However, if we have only one low-energy IALU and decide to use it for $c0$, it will not be possible to exploit $c1$'s slack.

3.1 Algorithms for Dynamic Energy Reduction

Figure 4(c) shows how a given slack can be exploited to reduce dynamic energy consumption. Below we present two algorithms that take advantage of slacks by scheduling the operation in question in a slower, less energy-consuming version of the corresponding functional unit. Both algorithms assume that the VLIW architecture in question has multiple (low-energy) versions of each functional unit type. The first algorithm works without increasing the schedule length of the original performance-oriented scheduling. The second algorithm, on the other hand, allows a user-specified performance degradation if doing so leads to larger energy savings (as compared to the first algorithm). Both algorithms first order the operations to be exploited for energy optimization. Then, for each operation (in order), considering its exploitable inslack and taking into account number of IALU operations that can be issued in a cycle, the start time of the operation is set to the earliest possible time. After that, considering the outslack of the operation and the available low-energy versions of the corresponding unit, the most suitable low-energy version of the unit is selected and the operation is scheduled in that unit.

3.1.1 Algorithm I

The first algorithm is based on the idea that the low-energy units are used only if this does not increase the length of the performance-oriented schedule. This can be achieved by not prolonging the (compile-time estimated) execution of the operation beyond its outslack. The idea can be best explained using an example. Figure 4(d) shows how the operation $a1$ in Figure 4(c) is scheduled in a low-energy unit with a latency of 6. Note that the operation is scheduled to start at the fourth cycle and finish at the ninth cycle. However, if the latency of a candidate low-energy version was 7, it would not be possible to use this (version of the) unit for this operation.

A sketch of this algorithm is given in Figure 5. The algorithm takes a region of code to schedule (*region*) and a table (*table*) that gives energy consumption and latency for each IALU component. *compute_slack()* computes the inslack and outslack for each operation in the region and *build_slack_list()* builds a list of operations with slacks. In the for-loop, we employ a *selection heuristic* to determine the most beneficial operation candidate for slack exploitation. Note that it is very important to determine a suitable order of processing for operations as exploiting one slack might prevent another slack from being exploited. This is illustrated in Figure 4(g) where fully exploiting the slack for operation $a1$ prevents the outslack for operation $a2$ from being exploited

```
INPUT: A sequence of operations ("region") scheduled
       using a performance-oriented scheduler;
       A table that gives energy consumption and latency
       for each IALU component ("table")
OUTPUT: A scheduled set of operations where slacks have
        been exploited
Algorithm1(region,table,...)
  begin
    compute_slack(region);
    list = build_slack_list(region);
    curr_max = -1;
    next_op = NULL;
    for each operation op in list do
      if (heuristic_energy(op) > curr_max) then
        next_op = op;
        curr_max = heuristic_energy(op);
      endif
    endfor
    if (next_op != NULL) then
      stime = stime_old - inslack;
      latency = compute_latency(next_op);
      energy = compute_energy(next_op);
      update_region(region,next_op,stime,energy,latency);
      Algorithm1(region,table,...);
    endif
  end
```

Figure 5. Algorithm I.

if $a1$ depends on $a2$. Consequently, our selection heuristic evaluates each and every operation with a slack and calculates the potential energy gain if the associated slack is exploited. The potential gain is the difference in energy consumption between the fastest (and most energy-consuming) version of the unit and the most-energy saving version that does not distort any data dependence. Our approach, using the *heuristic_energy()* function, selects the operation with the largest potential gain (*curr_max* keeps the maximum potential energy gain found so far). After selecting an operation, the scheduler updates the code region, and calls itself with the updated region. *compute_latency()* return the largest latency value (from *table*) which is less than or equal to the sum of slack of the operation *next_op* and the minimum latency for executing the operation using the fastest version. In Figure 5, *compute_energy()* returns the corresponding energy value. Variables *latency* and *energy* keep the latency and energy consumption of the version on which *next_op* is scheduled. Finally, *stime* and *stime_old* are the updated and original start times of *next_op*.

3.1.2 Algorithm II

This algorithm attempts to increase dynamic energy savings further by allowing a user-specified increase in schedule length. Note that this might be a reasonable approach in many embedded/portable environments where energy consumption holds a first-class status. Informally, this algorithm checks whether, for a given operation with slack, using a more energy-saving version than the one that would normally be selected by Algorithm I is possible without exceeding a *performance degradation threshold* (PDT), a user-specified parameter. If so, it employs this more energy-saving option and considers the next operation (and its slack). An example application of this approach is illustrated in Figure 4(e), which shows the optimized version of Figure 4(c). As discussed in the previous subsection, the first algorithm would exploit the slack of operation $a1$ by scheduling its execution over six cycles. Note that considering the inslack and outslack, it is not possible to achieve a better result using the first algorithm even if we have a more energy-saving version (of the same unit) with a latency of 7 cycles. However, if we are allowed to increase the schedule length by one cycle, we can extend the execution of this operation to 7 cycles as shown in Figure 4(e). This can be achieved by inserting one empty cycle to the schedule.

We have also implemented a modified version of this algorithm. The modified version is more conservative in increasing schedule length. In fact, it does not increase schedule length unless it is strictly necessary; instead, it extends the available slack by moving around the other instructions in the code region of interest. Let us consider the example in Figure 4(c) under a scenario where we want to employ a low-energy unit with a latency of 7 cycles for operation $a1$. As shown in Figure 4(f), this can be achieved by scheduling $c0$ in the eleventh cycle instead of the tenth cycle. Note that this is possible only if $c0$ has an outslack and data dependences allow such a move. Since our experiments with this modified version generated very similar results to those obtained from Algorithm II, we drop this version from discussion in the remainder of the paper.

3.2 Algorithms for Leakage Energy Reduction

As leakage energy is becoming a significant portion of the overall energy consumption [3], it is also important to study software techniques to reduce leakage energy. The algorithms for leakage energy reduction exploit the slacks in the schedule by activating the appropriate leakage control mechanism. The compiler is provided with the information on the latency to invoke the leakage control mechanism, the latency to restore the unit to normal mode, the potential leakage energy reduction, and any additional overhead energy associated with the application of the leakage control mechanism.

It must be noted that the definition of the slack is different when considering leakage control. Since the application of leakage control techniques is not impacted by dependences, we define the slack for leakage control as the duration between two successive accesses to the unit in question. Note that this implies leakage control can exploit larger slacks than voltage scaling, which is restricted by the data dependences.

3.2.1 Input Vector Control

While abstracting the potential energy reduction of the input vector control mechanism, two important factors need to

be considered. First, it must be noted that the leakage current takes a few nanoseconds to settle to the minimum value corresponding to the activated input vector. Second, the activation of the input vector itself causes some dynamic energy consumption as the input to the unit is changed. Both these aspects are factored into the model presented to the compiler. Both the dynamic energy overhead and the leakage current settling time are state dependent. In our experiments, we assume only a single functional unit of each type, an average dynamic energy consumption when the input control mechanism is activated, and a single cycle settling time for leakage current. The average leakage energy reduction possible by activating the input vector found through circuit simulations explained earlier is provided to the compiler. The compiler analyses each slack and just before the first slack cycle, it inserts a command to activate the *sleep* signal shown in figure 2(a). Similarly, a command to *deactivate* the signal is inserted in the final cycle of the slack. Our current implementation works at the basic block level granularity and inserts commands to activate and deactivate the sleep signals when entering or leaving basic blocks. This reduces the ability to exploit larger slacks but makes the implementation easier.

Figure 4(h) (on page 4) shows the energy consumption with and without leakage control mechanism. Note that when the leakage control mechanism is employed, we incur extra *dynamic* energy consumption in the second cycle. The scheduling algorithm determines for each slack whether to activate the input control mechanism or not. It utilizes the following expression to determine whether the slack can be exploited:

$$E_d + (k+1)E_l \geq 2E_d + E_l + r'E_l + r(k-1)E_l \quad (1)$$

In this equation, E_l is the leakage energy per cycle and E_d is the dynamic energy per operation, r is leakage energy reduction factor (i.e., if the original leakage energy is E_l, the optimized energy is rE_l), k is the slack duration in cycles, r' is the leakage energy reduction factor during the current-settling time. Note that assuming $r = r'$ and a single-cycle current-settling time, we see that leakage energy is beneficial when $pk(1-r) - 1 > 0$, where $E_l = pE_d$.

Figure 6(a) shows (for different values of p and k) when leakage energy reduction is beneficial for an energy reduction factor of 0.5. It can be observed that the leakage control is beneficial in all cases except when both p and k are small. In other words, in order for the leakage control mechanism to be beneficial, the per cycle leakage energy should be comparable to the dynamic energy per operation and there should be a sufficient number of cycles in the slack to compensate for the extra dynamic energy consumption due to input vector control. Figure 6(b) shows when leakage energy reduction is beneficial (when $p = 1$ and $p = 0.5$) for different reduction factor values. We see that the leakage control mechanism is useful when k is larger than 10 or r is small enough (i.e, overall leakage energy reduction is high).

3.2.2 Power Supply Gating

The use of power supply gating demands different optimization strategies since it requires a significant amount of time for the circuit to transition back to active (fully-operational) state. Consequently, to take advantage of this mechanism, the compiler should focus on larger program scopes such as nested loops and procedures. In our current implementation, we exploit only power supply gating for totally unused units throughout program execution.

3.3 Combining Dynamic and Leakage Energy Reduction

In the previous sections, dynamic and leakage energy reduction have been explored individually. As the relative magnitudes of dynamic and leakage energy become comparable, it will become important to reduce both in an integrated fashion. In this section, two alternate approaches for combining voltage scaling and leakage reduction by the compiler are presented. Since our current voltage scaling scheme for dynamic energy reduction works on instruction granularity, we consider here only input vector control for the leakage reduction.

In the first approach, the compiler determines whether it is better to exploit dynamic or leakage energy reduction for each slack duration independently. To achieve this, it uses the following expression to determine which technique to employ (see Figure 4(i)):

$$E_{dk} + (k+1)E_{lk} < 2E_d + (1 + r' + (k-1)r)E_l \quad (2)$$

where E_{dk} is the dynamic energy consumed when voltage scaling is employed to increase the latency of the operation to exploit the slack of k cycles, and E_{lk} is the corresponding leakage energy per cycle when voltage scaling is employed. Note that leakage energy also scales down with supply voltage scaling. If the condition specified by the above expression is satisfied, voltage scaling is employed to exploit the slack; otherwise, the leakage control mechanism is activated.

Figure 7 shows for each slack duration, the energy consumed when no optimization is performed, when only voltage scaling is applied, and when only the input vector control is applied. It is interesting to note that, based on slack duration, either voltage scaling or input vector control mechanism generates the best result, motivating an integrated approach that employs different energy reduction mechanisms depending on slack size. In Figure 7(a), voltage scaling is the preferred technique for slacks of duration less than 10 cycles. Beyond this crossover point, the leakage reduction mechanism becomes favorable. This crossover point is influenced by the number of available functional units with different supply voltages (which is three in Figure 7(a)). When we

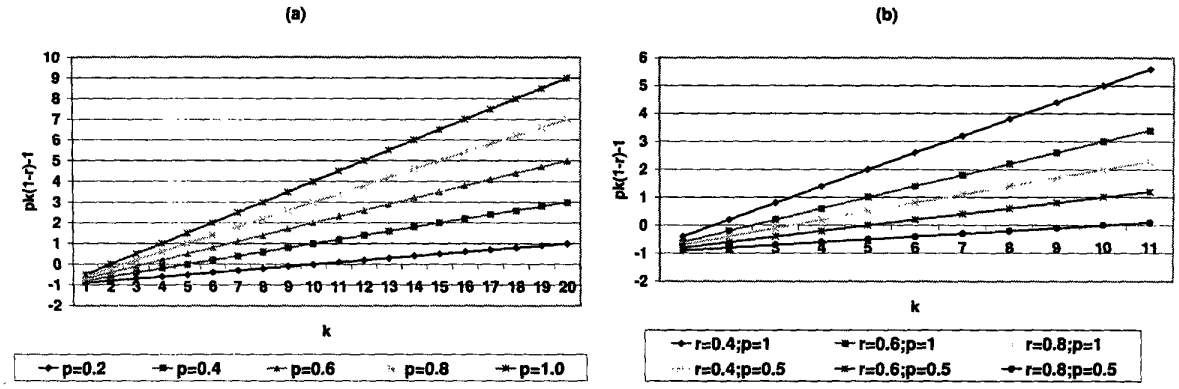

Figure 6. Sensitivity of leakage energy benefits to parameters k and p for a fixed value of $r = r' = 0.5$. In obtaining these results, energy values for a 0.1micron, 1V adder are used.

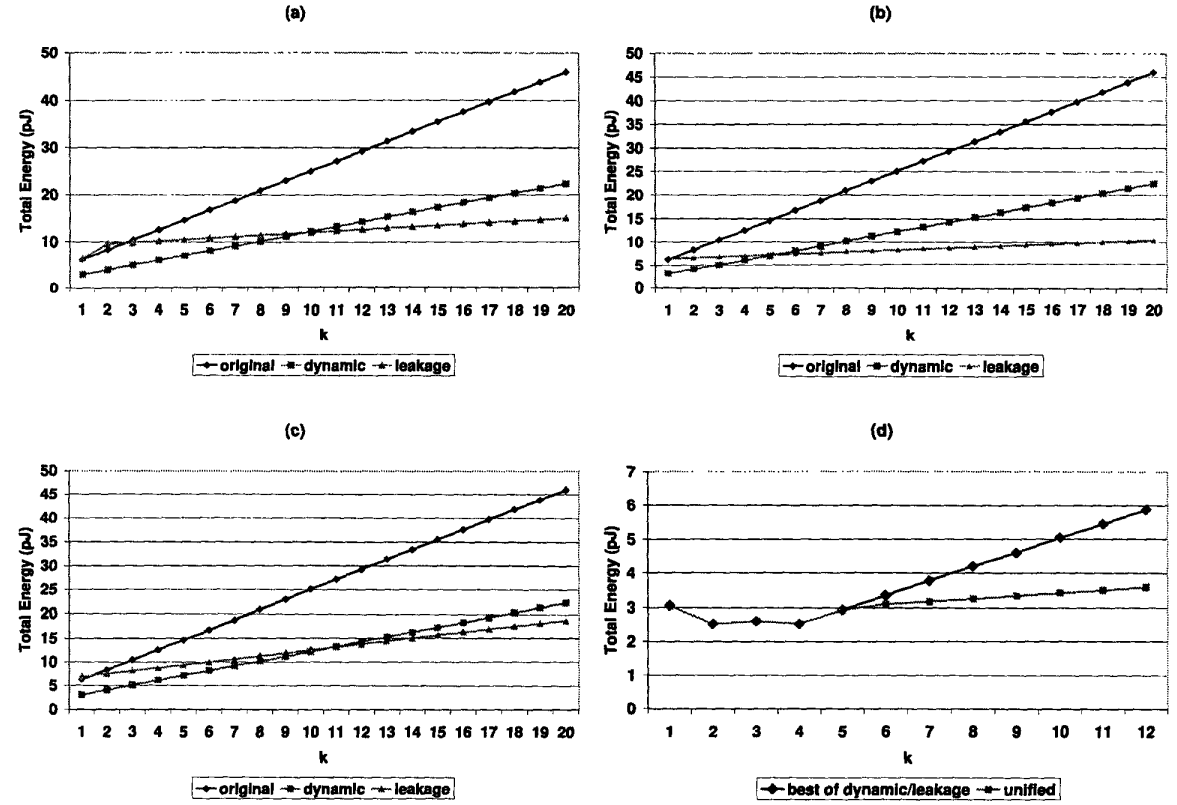

Figure 7. (a-c) Comparison of voltage scaling and input vector control. In (a), r is 0.1 and the number of supply voltages is three (1V, 0.7V,0.55V). In (b), r is 0.1 and the number of supply voltages is two (1V and 0.7V). In (c), r is 0.3 and the number of supply voltages is two (1V and 0.7V). (d) Unified approach with three supply voltages (1V, 0.7V, 0.55V) and r being 0.3. In obtaining these results, energy values for 0.1 micron adder are used.

reduce the number of available supply voltages to two (e.g., in an attempt to reduce unit replication and multiple supply line overheads), the crossover point shifts to a slack duration of 5 cycles as can be observed from Figure 7(b). This shift is caused by the inability to exploit larger slacks due to the non-availability of functional units using the third supply voltage. In contrast, the input vector control is able to amortize the initial dynamic energy consumption overhead better with larger slacks. The leakage energy reduction factor, r, (when the leakage control mechanism is used) also influences the location of the crossover point. Figure 7(c) shows that the crossover point increases to 10 when the leakage reduction is less, even in the presence of just two supply voltages. These results emphasize the need for the compiler designers to be aware of parameters that can influence their technique of choice for slack exploitation.

In the second approach, which we call *unified*, the goal is to investigate whether both the leakage and dynamic energy management schemes can be exploited at different intervals of *the same slack*. The idea is illustrated in Figure 4(j). Basically, the compiler selects an optimum value f ($< k$) up to which dynamic energy is optimized and beyond which leakage energy is optimized. This scheme is in contrast to the previous approach that determines a single technique to employ for each slack. Figure 7(d) shows how the unified approach compares to using the best of the dynamic and leakage control mechanisms individually for each slack duration explained in the previous paragraph. We see that the unified approach performs best as it combines the best of both the techniques. Since the individual slack exploitation for leakage or dynamic energy control (discussed in the previous paragraph) is subsumed by this technique, we only consider the unified scheme in the rest of the paper.

4 Experimental Evaluation

In this section, we discuss our implementation and simulation environment (Section 4.1), introduce our benchmark codes (Section 4.2), and present our results (Section 4.3).

4.1 Simulation Platform and Implementation

Trimaran is a compiler infrastructure to provide a vehicle for implementation and experimentation in state-of-the-art research in compiler techniques for Instruction Level Parallelism (ILP) [17]. A program flows through `IMPACT`, `Elcor`, and the cycle-level simulator. `IMPACT` applies machine-independent classical optimizations and transformations to the source program, whereas `Elcor` is responsible for machine-dependent optimizations and scheduling. The proposed algorithms are implemented in `Elcor`. The increase in compilation time due to our algorithms was around 20% on average. The cycle-level simulator was modified to record the activity of the IALU units. Further, the simulator was augmented to support a cache hierarchy. This recorded information was used along with energy parameters to evaluate the energy consumption. The energy estimation is activity-based in which energy consumption is based on number of accesses to the components. The dynamic energy parameters and leakage reduction factors used are based on actual circuit-level simulation of the components. Figure 8 shows the energy parameters of three IALU components (adder/subtracter, shifter and multiplier) for three supply voltages. These numbers are based on actual layouts performed in 0.25 micron technology. Scaling factors [2] are applied to these values to obtain corresponding values for 0.10 micron with a 1V supply voltage and 0.2V threshold voltage. The leakage reduction numbers are extracted from the Figure 3. The default configuration for our experiments uses four IALU, two FPALU, one LD/ST unit and one branch unit.

IALUt Component	Supply Voltage	Dynamic Energy (pJ)	Latency
`Adder`	3.3V	66.6	1 cycle
	2.1V	26.9	2 cycles
	1.7V	17.6	3 cycles
`Multiplier`	3.3V	258.0	8 cycles
	2.1V	104.5	14 cycles
	1.7V	68.4	27 cycles
`Shifter`	3.3V	66.1	4 cycles
	2.1V	26.8	8 cycles
	1.7V	17.5	16 cycles

Figure 8. Energy characteristics for the three 32-bit components using 0.25micron technology. The threshold voltage for these designs is 0.48V.

We present results for the different optimizations both using compile time metrics and runtime metrics. The energy savings estimated at the compile time are provided by analyzing the slacks using `Elcor` (without taking into account conditionals and loop bounds) and are called *static results*. The energy savings at run time are estimated using the cycle-accurate simulator and are called *dynamic results*. It must be observed that the dynamic results depend on the number of times each portion of the schedule is executed. All energy saving numbers are reported with respect to an architecture that uses a performance oriented schedule with no support for voltage scaling or leakage control. As we apply our techniques to the IALU, we consider the energy consumed only by the IALU operations.

4.2 Benchmark Codes

To evaluate the effectiveness of our algorithms, we used a suite of fifteen programs from different benchmark sets. The important characteristics of these codes are given in Figure 9. The third column in this figure gives the number of slacks

Program	Source	Number of Slacks	Avg. Slack Length	Exploitable Slacks (%)
099.go	SpecInt95	2,741	3.99	34%
124.m88ksim	SpecInt95	1,809	3.78	32%
129.compress	SpecInt95	632	3.47	44%
130.li	SpecInt95	1,077	4.22	49%
132.ijpeg	SpecInt95	1,360	4.27	49%
convolution	DSPstone	33	1.50	50%
dot_product	DSPstone	29	1.75	25%
fir	DSPstone	59	2.00	75%
n_complex_updates	DSPstone	61	1.53	90%
n_real_updates	DSPstone	64	1.56	67%
cordic	Mediabench	456	5.73	14%
idea	Mediabench	668	3.76	75%
nbradar	Mediabench	441	6.60	61%
paraffins	Trimaran	383	2.34	52%
rawcaudio	Mediabench	83	1.74	54%

Figure 9. Benchmark characteristics. The average length of slacks is the static length obtained for all IALU operations (including those without slack) in the basic block schedule.

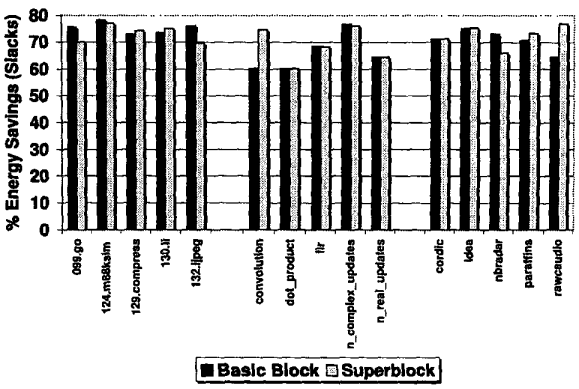

Figure 10. Percentage energy savings considering only IALU operations with slacks when using basic block and superblock scheduling (static results).

in each code and the fourth column gives the average slack length (in cycles). The fifth column shows the percentage of exploitable slacks. On average, 51.4% of the slacks are exploitable and the average slack length is 3.21. We measured the distribution of slacks across different types of operations and observed that, in these codes, more than 88% of the slacks, on the average, occur with integer ALU operations using our default configuration. This provides a strong motivation for us to focus on these operations for exploiting slacks.

4.3 Results and Discussion

4.3.1 Dynamic Energy Reduction

In this subsection, we assume that there are three versions for each IALU (with each version operating with a different supply voltage) in the default configuration that can be used simultaneously (that is a total of 12 IALU versions) The energy value corresponding to each version is shown in Figure 8. It should be emphasized that the leakage energy contribution for this technology (0.25 micron) is not significant as compared to dynamic energy consumption (around 3% of overall energy for a junction temperature of 110C).

Figure 10 shows that a 70.8% energy saving is possible when only considering energy consumption of operations with exploitable slacks (i.e., the operations that can be optimized by Algorithm I). This is obtained starting with a basic-block oriented performance schedule. The corresponding number when superblock scheduling is used is 71.6%. These static results show that the proposed approach can cover the slacks successfully across applications from different benchmark suites.

In order to evaluate how this translates to actual energy savings, Figure 11(a) provides the overall energy savings obtained by running the benchmarks through the simulator (dynamic results). The average energy savings by using Algorithm I which employs voltage scaling is 32.3%. Further energy savings can be obtained at the cost of performance when Algorithm II with three different PDTs is used as observed from the figure. The average energy savings across the different benchmarks for these three PDTs are 50.1%, 63.7% and 71.3%, respectively. It can be observed that the additional energy savings are small when moving from 20% to 50% performance degradation. This is because of the limited number of supply voltages that imposes an inherent limit of exploitable slack duration. Further, the incremental energy savings corresponding to a fixed slack duration starts to reduce as slack duration increases. On the average, the actual performance degradation at runtime across the different benchmarks for Algorithm II is 11.5%, 17.6% and 42%, respectively, for 10%, 20% and 50% PDTs. The energy behavior of Algorithm I and Algorithm II when superblock scheduling is employed is shown in Figure 11(b). It is observed that slacks can be exploited successfully providing comparable energy savings even with superblock scheduling.

4.3.2 Comparing Voltage Scaling, Leakage Control and Unified Schemes

In this subsection, we use the 0.1 micron, 1V supply voltage technology energy parameters as leakage and dynamic energy become comparable in this technology. The leakage energy reduction factor (r) for the adder, shifter and multiplier used were 0.34, 0.14, and 0.72, respectively. Further, we use a value of $p = 1$, that is leakage energy per cycle is equal to the dynamic energy per operation executed on that unit. The voltage scaling technique is the same as that

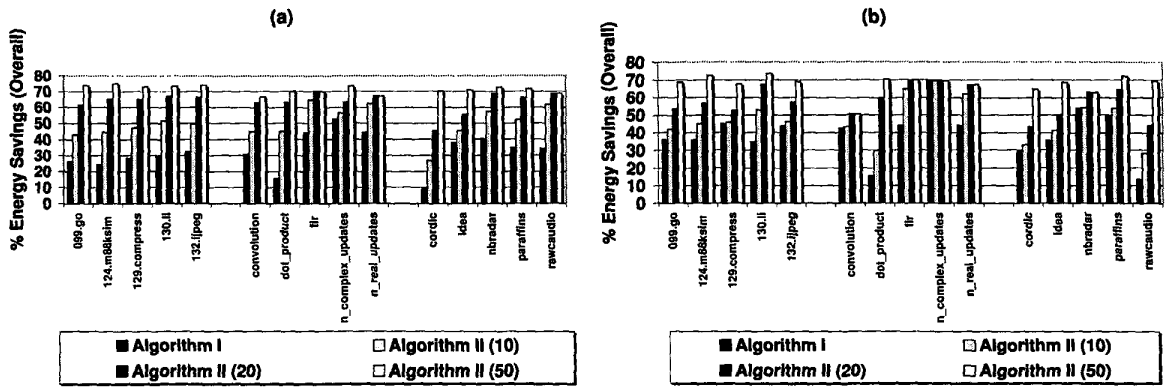

Figure 11. Overall runtime energy savings percentage for all IALU operations when voltage scaling is applied in conjunction with (a) basic-block scheduling (b) super-block scheduling. Algorithm II numbers are for three different performance degradation thresholds of 10%, 20% and 50%.

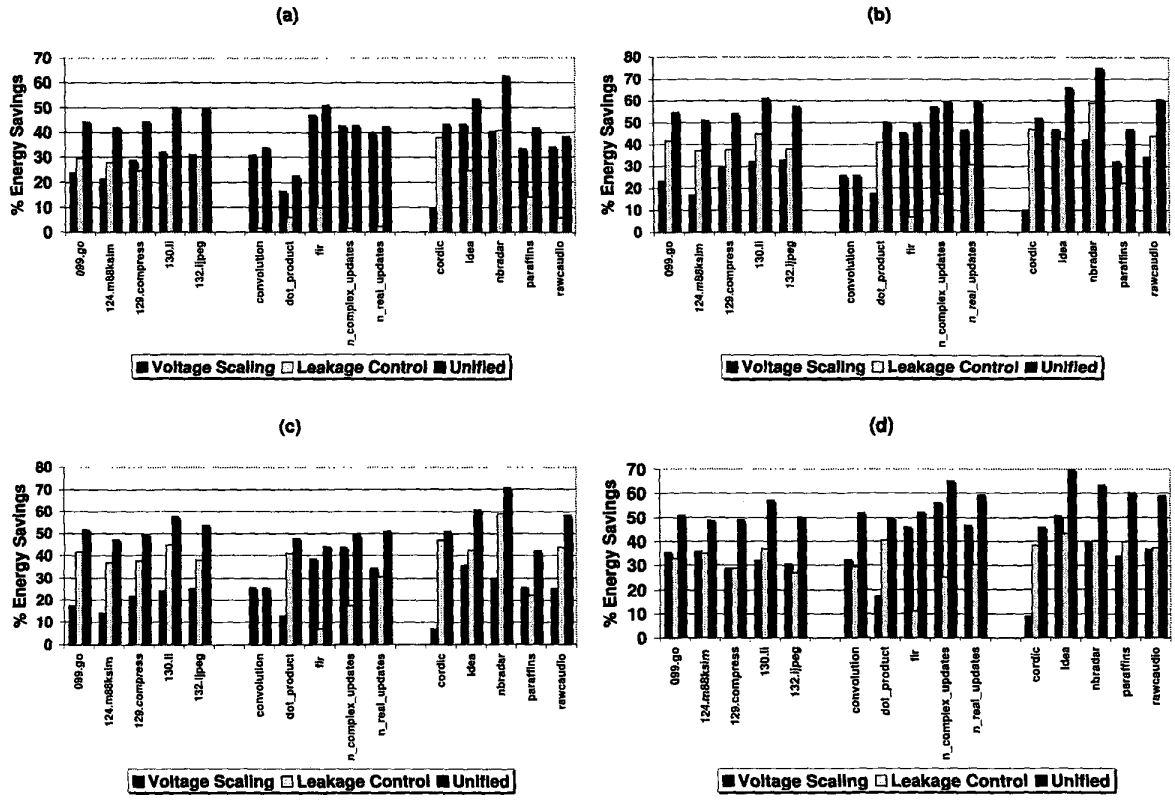

Figure 12. Energy savings for all IALU operations as compared to original case with no voltage scaling or leakage control (a) Static results with three supply voltages and basic block scheduling (b) Dynamic results with three supply voltages and basic block scheduling (c) Dynamic results with two supply voltages and basic block scheduling (d) Dynamic results with three supply voltages and superblock scheduling.

used for Algorithm I in the previous section. When optimizing for only leakage energy (using input vector control), it is assumed that only the highest performance version of each IALU is available. All low-energy versions are supply-gated to completely eliminate leakage energy. Finally, the unified scheme employs a combination of leakage control and voltage scaling as explained earlier. In the unified scheme, input vector control is applied whenever a functional unit becomes idle.

Figure 12(a) gives the energy savings of the three schemes obtained from compiler (static results) when using basic block scheduling. Note that these are cumulative energy savings across all basic blocks. In eleven out of fifteen cases voltage scaling is estimated to perform better than leakage reduction at compile time. The unified approach is the best in all cases. The corresponding dynamic results are shown in Figure 12(b). The effectiveness of each scheme depends on the exploitable slack durations in the schedule. It can be observed that for nine of the benchmarks, the leakage control mechanism outperforms voltage scaling. The reason that there are larger energy gains in favor of leakage control when we move from static to dynamic results is two-fold. First, in some benchmarks, basic blocks with slacks larger than the average are executed more frequently. Second, the compiler visible slacks are prolonged during execution due to unexpected delays such as cache miss stalls. We also note that the unified scheme brings an average of 22.2% and 20.9% improvement over the voltage scaling and input vector control mechanisms, respectively. Figure 12(c) gives the dynamic results when only two supply voltages (1V and 0.7V) are employed. Due to the limited number of supply voltages, the average benefits from voltage scaling reduce from 32.7% to 25.4% on the average. Consequently, the gains due to unified scheme also reduce from 54.9% to 50.7% on the average. Figure 12(d) shows the dynamic results when superblock scheduling is employed with three supply voltages. It can be observed that as compared to basic block scheduling, the voltage scaling performs better than leakage control. This is due to the reduction in the slack length duration. Specifically, when moving from basic block to superblock, average slack length reduces by 12%.

5 Conclusions

This work presents a novel approach to optimizing energy consumption of processor IALUs. The basic idea is to have the compiler analyze the schedule slacks in a VLIW architecture and exploit them using dynamic and leakage energy reduction mechanisms. Based on the duration of exploitable slacks and available functional units with different supply voltages, dynamic or leakage energy reduction become more favorable. We also present and evaluate a unified energy optimization approach which exploits voltage scaling and input vector control for reducing dynamic and leakage energy control within a given slack. The proposed techniques have been implanted within a compiler and the resulting schedules are simulated using parameters extracted from circuit-level simulation. Our results show that combining voltage scaling and input control performs better than using either of these strategies independently.

References

[1] L. Benini and G. De Micheli. System-level power optimization: techniques and tools. ACM Transactions on Design Automation of Electronic Systems, 5(2), pp.115-192, April 2000.

[2] S. Borkar. Design challenges of technology scaling. IEEE Micro. pp.23-27, July-August 1999.

[3] J. A. Butts and G. Sohi. A Static Power Model for Architects. In Proc. *International Symposium on Microarchitecture*. December 2000.

[4] A. Chandrakasan, W. J. Bowhill, and F. Fox. *Design of High-Performance Microprocessor Circuits*. IEEE Press, 2001.

[5] Computing Market Dynamics. Mobile Computing Devices: A New Era in Personal Computing. Report No. CMC00-005MC, Aug. 2000.

[6] J. Casmira and D. Grunwald. Dynamic Instruction Scheduling Slack. In Proc. *2000 Kool Chips Workshop*, December 2000.

[7] J. P. Halter and F. Najm. A gate-level leakage power reduction method for ultra-low-power CMOS circuits. In Proc. *IEEE Custom Integrated Circuits Conference*, pp. 475–478, 1997.

[8] W. W. Hwu et. al. The superblock: an effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, Kluwer Academic Publishers, 1993, pp. 229–248.

[9] M. Johnson, D. Somasekhar and K. Roy. Models and algorithms for bounds in CMOS circuits. *IEEE Transactions on CAD of Integrated Circuits and Systems*, Vol. 18, No. 6, pp. 714–725, June 1999.

[10] A. Klaiber. The technology behind Crusoe processors. *Whitepaper*, Transmeta Corporation, January 2000.

[11] T. Kuroda and T. Sakurai. Threshold-voltage control schemes through substrate-bias for low-power high-speed CMOS LSI design. *Journal of VLSI Signal Processing Systems*, 13(2/3):191–201, Aug. 1996.

[12] S. A. Mahlke et. al. Effective compiler support for predicated execution using hyperblock. In Proc. *the 25th International Symposium on Microarchitecture*,, pp.45-54, Dec. 1992.

[13] S. S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, San Francisco, California, 1997.

[14] S. Mutoh et. al. 1-V power supply high-speed digital circuit technology with multithreshold-voltage CMOS. *IEEE Journal of Solid State Circuits*, vol. 30, no. 8, pp. 847–854, Aug. 1995.

[15] T. Okuma, T, Ishihara, and H. Yasuura. Real-time task scheduling for a variable voltage processor. In Proc. *International Symposium on System Synthesis*, November 1999.

[16] K. Roy and S. C. Prasad. *Low-Power CMOS VLSI Circuit Design*. Wiley Interscience, 2000.

[17] Trimaran home page, http://www.trimaran.org

[18] M. C. Toburen, T. M. Conte and M. Reilly. Instruction scheduling for low power dissipation in high performance processors. Power Driven Microarchitecture Workshop. June 1998.