Efficient and Scalable Compiler-Directed Energy Optimization for Realtime Applications

PO-KUAN HUANG and SOHEIL GHIASI
University of California, Davis

With continuing shrinkage of technology feature sizes, the share of leakage in total energy consumption of digital systems continues to grow. Coordinated supply voltage and body bias throttling enables the compiler to better optimize the total energy consumption of the system in future technology nodes. We present a compilation technique that targets realtime applications running on embedded processors with combined dynamic voltage scaling (DVS) and adaptive body biasing (ABB) capabilities. Considering the delay and energy penalty of switching between operating modes of the processor, our compiler judiciously inserts mode-switch instructions in selected locations of the code and generates executable binary that is guaranteed to meet the deadline constraint. More importantly, our algorithm runs very fast and comes reasonably close to the theoretical limit of energy optimization using DVS+ABB. At 65nm technology, we improve the energy dissipation of the generated code by an average of 33.20% under deadline constraints. While our technique's improvement in energy dissipation over conventional DVS is marginal (6.91%) at 130nm, the average improvement continues to grow to 13.19%, 22.97%, and 33.21% for 90nm, 65nm, and 45nm technology nodes, respectively. Compared to a recent ILP-based competitor, we improve the runtime by more than three orders of magnitude, while producing improved results.

Categories and Subject Descriptors: C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*; D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Leakage, energy-aware compiler, technology scaling

ACM Reference Format:

Huang, P.-K. and Ghiasi, S. 2007. Efficient and scalable compiler-directed energy optimization for realtime applications. ACM Trans. Des. Autom. Electron. Syst. 12, 3, Article 27 (August 2007), 16 pages. DOI = 10.1145/1255456.1255464 http://doi.acm.org/10.1145/1255456.1255464

Authors' address: P.-K. Huang, S. Ghiasi (contact author), Department of Electrical and Computer Engineering, University of California at Davis, One Shields Avenue, Davis, CA 95616; email: soheil@ece.ucdavis.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2007 ACM 1084-4309/2007/08-ART27 $5.00 DOI 10.1145/1255456.1255464 http://doi.acm.org/10.1145/1255456.1255464

1. INTRODUCTION

Microprocessors are one of the major contributors to energy consumption in embedded systems. Consequently, a number of circuit-level techniques such as DVS and ABB have been developed to reduce the energy consumption of the processor.

Quadratic dependence of active power on supply voltage, along with the lower-order impact of supply voltage on clock frequency, has motivated dynamic supply voltage scaling for processors. In this scheme the operating frequency and supply voltage of processors are throttled at runtime to save energy whenever full performance is not required. This technique was very effective in the old technology nodes, where the share of leakage energy in total energy consumption was negligible. The share of leakage energy, however, increases with the scaling of CMOS technology. Hence, conventional dynamic voltage scaling is less effective with advancement of technology [Duarte et al. 2002].

Adaptive body biasing (ABB) is another well-known CMOS design technique that allows runtime adjustment of transistor threshold voltage. Threshold voltage affects both the leakage and delay of transistors. Hence, its effect can be combined with supply voltage scaling to minimize the *total power consumption* for a given frequency [Martin et al. 2002].

We present a compilation methodology that targets embedded processors with joint DVS and ABB capabilities. We investigate hard realtime systems that have to meet the application deadline and have lightweight OS. Considering the delay and energy penalties associated with switching the processor's operating modes, our compiler judiciously inserts mode-switch instructions in the code, subsequently generating code that is optimized for overall energy consumption. The generated code is guaranteed to meet the execution deadline over the input data space.

We present a theoretical lower bound on energy reduction and subsequently show that our compiler approaches this limit reasonably well. Furthermore, our algorithm runs very fast and is readily scalable to large programs. Compared to baseline compilation, our compiler improves the energy consumption of the processor by 11.77%, 20.67%, 33.20%, and 48.72% in 130nm, 90nm, 65nm, and 45nm technologies, respectively. Compared to traditional DVS-only optimization, we improve the average energy consumption by 6.91%, 13.19%, 22.97%, and 33.21% in the four aforementioned technologies. Moreover, our algorithm improves the result of an ILP-based competitor by about 2%, while running more than three orders of magnitude faster.

2. RELATED WORK

Extensive research has been done to minimize the dynamic power consumption of a CMOS design. Dynamic voltage scaling is utilized in several fabricated academic and commercial processors [Burd et al. 2000]. DVS techniques adjust supply voltage and clock frequency to minimize dynamic power under timing constraints. Many task-level voltage scheduling algorithms have been proposed for such processors [Pillai and Shin 2001; Yao et al. 1995; Kim et al. 2003]. However, with the continuing shrinkage of device sizes, techniques that only target

dynamic power will not be effective [Duarte et al. 2002]. Adaptive body biasing has been utilized to reduce the leakage power consumption [Kesharvarzi et al. 1999; Kao et al. 2002]. Researchers have also studied the application of runtime ABB techniques [Duarte et al. 2002].

Combined DVS and ABB optimization is well studied in circuit design. In Martin et al. [2002], a combined DVS+ABB technique is proposed for unrelated tasks. Also, an expression for obtaining the optimal tradeoff between bias voltage and supply voltage is derived. Yan et al. [2003] proposed an algorithm for a task graph with realtime constraints. None of the previous works on combined DVS+ABB has considered intratask-level optimization by program structure analysis. This is of particular interest to resource-constrained embedded applications that cannot admit complex task-level dynamic voltage scheduling policies, and demand a lightweight or no operating system to guarantee the realtime performance.

Issues of reducing leakage energy consumption of the processor are also discussed [You et al. 2006; Rele et al. 2002; Zhang et al. 2003]. In You et al. [2006], the authors proposed an architecture with a clock-gating mechanism and compiler framework for reducing the leakage power. Rele et al. [2002] provide a cooperative compiler and microarchitecture method to utilize the power gating mechanism of the processor. Zhang et al. [2003] built a compiler framework and energy optimization strategy to utilize the sleeping mode of functional units. These results demonstrate that a power-gating mechanism is effective for reducing the leakage energy consumption during the idle time of the function unit, while our works utilize the ABB technique to reduce the leakage energy consumption during the computation time of the processor.

Several research groups have proposed static intraprogram voltage scaling [Xie et al. 2004; AbouGhazaleh et al. 2003; Azevedo et al. 2002; Hsu and Kremer 2003]. An analytical study of the potential power savings using intraprogram DVS is reported in Xie et al. [2004]. The authors also propose an ILP-based approach whose savings come reasonably close to the analytical bounds. In Azevedo et al. [2002], check-points indicating the voltage scaling points are inserted into the program during compilation. Hsu and Kremer [2003] introduce an algorithm that identifies the program regions with time slack for the processor, and implement it as a source-to-source transformation. Compiler- and operating-system-level optimizations are coordinated in AbouGhazaleh et al. [2003]. None of these techniques considers leakage power, or the effect of technology scaling on the validity of the results. We utilize the generic power model derived in Martin et al. [2002], and perform intraprogram simultaneous DVS and ABB.

3. PROCESSOR MODEL AND OPERATING MODES

Our proposed compilation technique targets a processor with combined DVS and ABB capabilities that can operate at several discrete frequencies. According to Martin et al. [2002], each frequency has to be associated with a corresponding pair of supply and body bias voltages that allow operation of the processor at that frequency. The combination of the three parameters, namely,

Table I. Processor Operating Modes at 90nm

Operating frequency (MHz)	1000	800	600	400	200
Supply voltage (V)	1.63	1.47	1.29	1.11	0.95
Bias voltage (V)	−0.08	−0.17	−0.25	−0.35	−0.47

frequency, supply voltage, and body bias, constitutes an *operating mode* of the processor. The processor is assumed able to switch between operating modes by execution of a specialized instruction, referred to as a *mode-switch instruction*. Executing of a mode-switch instruction initiates the process of setting both the supply voltage and body bias of the processors to the target mode implied by the instruction. Note that frequency is a function of supply and body bias voltage, and need not be specified separately. Execution of the mode-switch instruction, or, equivalently, switching between modes, incurs delay and energy penalties. Both delay and energy penalties depend on the voltage difference of the two modes involved in switching.

We assume that our target processor can operate at five different clock frequencies, from 200 *MHz* up to 1 *GHz* at 200 *MHz* steps. We adopt the process technology and processor parameters from predictive technology models [Berkeley-Model; Andrei et al. 2004] and Intel XScale commercial processors, respectively. We obtain the energy-optimal supply and body bias voltages corresponding to each frequency by applying the conclusion in Martin et al. [2002]. Table I demonstrates the characteristics of the operating modes for our target processor in 90nm.

4. ILP-BASED INTRAPROGRAM SUPPLY AND BIAS VOLTAGE SCALING

The ILP-based approach aims to achieve this goal by the insertion of static mode-switch instructions on all control flow edges of the application. To formulate the problem as an ILP instance, profiling and simulation should be carried out to capture the frequency of executing each edge of the application control flow graph (CFG), and the average energy dissipation and delay of application basic blocks in each of the operating modes. To determine the appropriate mode for each edge of the CFG, a set of binary decision variables is assigned to each edge of the application CFG. Subsequently, integer linear constraints are formed to guarantee: (1) the assignment of each CFG edge to exactly one mode; (2) execution of the application, considering delay penalty when switching modes, within deadline. The objective function would be another integer linear expression that estimates the total energy consumption, including the energy penalty of mode-switches using integer variables [Xie et al. 2004; Huang and Ghiasi 2006].

The ILP-based technique has two major drawbacks. Firstly, the ILP formulation widely used for intraprogram frequency scaling is NP-hard. Therefore, its runtime is not scalable to large programs. Moreover, it inserts a mode-switch instruction before entering each basic block (1 mode switch per about 5 instructions, on average!). Some modes will be redundant, that is, they set the processor to the mode at which it is already operating, and can be removed using classic compiler optimization passes. Nevertheless, the performance and

energy overheads associated with mode switches partially diminishes the savings. In our previous work, we observed that the ILP solving time for typical applications of about 200 basic blocks exceeds 30 minutes on an ordinary desktop computer [Huang and Ghiasi 2006]. As expected, the runtime grows very fast with increase of program complexity. For example, it took the ILP solver more than 6 hours to solve problem instances associated with typical applications of about 500 basic blocks.

In order to accelerate the solution time, it is reasonable to employ heuristics to reduce the number of constraints in the ILP instance. This would result in a tradeoff between the quality of the solution (energy savings) and solution time that might lead to an acceptable balance of the two. In our study, we filtered out the constraints associated with basic blocks that do not significantly contribute to total energy consumption of the application. For example, eliminating some constraints in the case of the *susan* testbench allowed us to solve the ILP instance on the order of tens of seconds while degrading the energy consumption by 12%. However, the gap in energy consumption between the original-ILP and simplified-ILP increases with the growth of application size. Consequently, heuristics applied on top of ILP-based approaches should be viewed as temporary solutions that somewhat push the limitations, rather than deliver truly scalable strategies.

5. EFFICIENT AND SCALABLE MODE SWITCHING

In addition to compilation time, the ILP-based approach has yet another problem. The ILP formulation mainly depends on the control flow behavior of the train input. The solution of the train input will be optimal. However, the optimal setting for train input is suboptimal for different inputs. We cannot have consistent energy savings as we apply different inputs. Once the actual input significantly differs from the train input, we not only have ignorant energy savings, but also violate the deadline of the application. Therefore, we develop an efficient and scalable mode switching algorithm to overcome the compilation-time problem and acquire consistent energy savings while we apply various inputs.

The ILP-based approach cannot have consistent energy savings for test inputs because its operating mode setting is fixed based on the control flow behavior of the train input. Once we apply some inputs having different control flow behavior from the train input, the fixed operating mode setting will kill the energy savings. Therefore, we proposed an algorithm called CPIM (check-point insertion method) which inserts the check-points to the program statically. However, these check-points can determine the next scaling frequency by checking the existence of the slack. Thus, although we insert these check-points statically, our operating mode setting is still adjusted dynamically as we apply different inputs. According to our experimental result, our proposed algorithm is slightly worse than the ILP-based approach as we apply train input. However, it outperforms the ILP-based approach as we apply various different inputs. Another important contribution is the shrinkage of compilation time for large applications. We will describe CPIM in detail in this section.

5.1 Optimal Scaling Frequency

Our compiler optimization goal is to minimize the application's total energy consumption by judiciously inserting mode-switch instructions in the code while guaranteeing the execution deadline of the application. We also consider the energy and latency penalties of switching between modes. We develop a very intuitive algorithm that guarantees meeting the execution deadline of the application.

The basic idea of our method is the following: At each point of execution, by knowing the maximum (i.e., worst-case) number of cycles required to finish the execution of the application and the time that is left before violating the deadline, the next operating frequency F_{next} would be the slowest possible frequency that guarantees executing the application without violating the deadline.

$$F_{next} = \frac{WCRC + S_d}{T_L}, \quad (1)$$

where *WCRC* denotes the *worst-case required cycles* from that specific point to finish the execution of the application, T_L stands for "time left" in seconds, and S_d refers to the delay penalty, in cycles, for switching between two modes. The motivation for this scaling equation is to spread the required cycles so as to execute the application over the remaining time. In other words, we try to run the processor as slowly as possible to meet the deadline for the workload at hand. This equation is proved to be theoretically optimal if continuous frequencies are available [Qu 2001]. In practice, however, processors can only run at a number of discrete frequencies and the available frequency immediately larger than F_{next} would be the right choice.

5.2 WCRC Calculation Algorithm

In order to utilize this equation and obtain the next scaling frequency for each node of the control flow graph, we need to estimate both WCRCs for each node of the application CFG.

In general, the software timing analysis used to calculate the WCET of an embedded application can also be used to estimate the WCRC of the program. However, we should avoid performing exhaustive path enumeration approach in order to reduce the compilation time of the large embedded application. Therefore, we next present a WCRC calculation algorithm which is simple and time efficient. Our algorithm is similar to the nonenumeration approach proposed in Suhendra et al. [2006]. We calculate the WCRC for each loop in the program by tracking its heaviest path and calculate the WCRC for entire program by traversing the control flow structure of the program in a bottom-up manner. We assume that the worst-case execution time (WCET) over the input data space, as well as the input associated with it, is known. Note that this assumption is not unreasonable because guaranteeing the execution time without knowledge of the WCET and associated input is not feasible [Li et al. 1999].

The concept of our proposed algorithm is described as follows. We use two bottom-up traversals to calculate the WCRC for each control flow edge of the program. The first traversal calculates the WCRC for loops inside the program.

```
WCRC-Main(M)
For each procedure P in the program by reverse topological order in the call graph Do
  For each loop L in the procedure in decreasing order of nesting depth Do
      G = L without the back edge;
      Loop-WCRC(G) = G.LoopBound(G) × WCRC-Estimation(G);
  G' = P without back edge;
  Procedure-WCRC(G') = WCRC-Estimation(G');
Let M be the main procedure in the program; return M
End

Function WCRC-Estimation
  Replace each Loop in the procedure by a dummy node with its WCRC as latency
  Replace each call site of procedure P by a dummy node with its WCRC as latency
  For each node u in reverse topologically sorted order Do
    pre-edge-wcrc(u) = Max(suc-edge-weight(u))+weight(u)
End
```

Fig. 1. WCRC calculation algorithm.

In order to find the WCRC for each loop, we calculate the latency of the heaviest acyclic path, which is that acyclic path inside the loop with the largest latency. The first traversal keeps track of the longest path inside the loop in bottom-up order. As we acquire the latency of the heaviest acyclic path of the loop which is equal to h, and the loop execution bound, which is equal to e, the WCRC for the loop is equal to $h * e$. As for nested loops, our algorithm will start from the innermost. The first traversal reports the WCRC for each loop. Then, we replace each loop structure of the program by a dummy basic block. The latency of the dummy basic block is equal to the WCRC of its corresponding loop structure acquired in the first traversal. Finally, another bottom-up traversal of the program structure is made to estimate the WCRC for each control flow edge of the program.

Figure 1 illustrates our algorithm in detail. Function WCRC—Estimation is the bottom-up tracer which calculates and reports the WCRC for each point inside the DAG structure. A *pre—edge—wcrc(u)* inside the function denotes the WCRC of precedent edges of the basic block u. Its value is equal to the maximum value of successive edges of the basic block u plus the latency of the latter. Function WCRC—Main(M) separates the estimation step for different procedures. For each procedure, the WCRCs of individual loops are first estimated. As we estimate the WCRC of the procedure, the loops inside the procedure are treated as black boxes with weights equal to their WCRCs. Likewise, while we estimate the WCRC of the entire program, the procedures inside the program are treated as black boxes with weights equal to their WCRCs. Our proposed algorithm is time efficient and simple to implement. Its complexity is $O(m)$, where the m is the number of edges in the control flow graph.

We develop a fast and effective algorithm to calculate the WCRC by traversing the graph in reverse order, assuming that the worst-case execution time (WCET) over the input data space and the input associated with it are known. Note that this assumption is not unreasonable because guaranteeing the execution time without knowledge of the WCET and associated input is not feasible [Li et al. 1999].

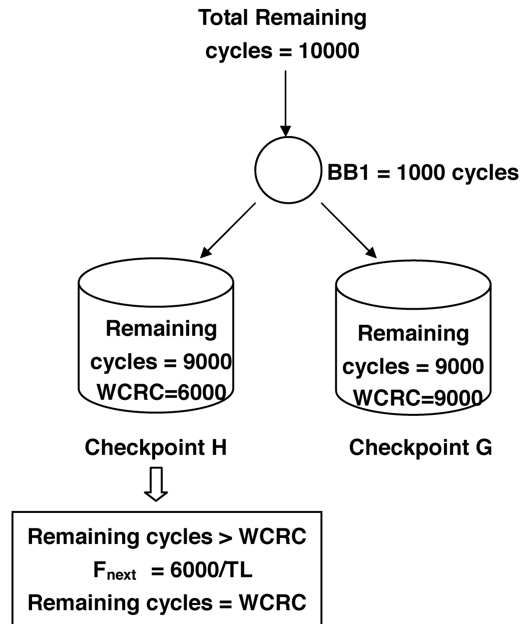

Fig. 2. Example function of a check-point.

5.3 Check-Point Insertion

After determining WCRC values for all CFG edges, we instrument the code to access the time elapsed and number of cycles executed so far from the operating system. We also need to consider the penalty required for accessing the operating system. We assume that we consume 100 cycles for accessing the time elapsed and number of cycles executed.

Capturing the number of cycles executed so far enables the compiler to determine the *remaining cycles* at each point. The number of remaining cycles is simply the maximum number of cycles required (or the WCRC at the entry) minus the number of cycles executed so far. Note that the number of remaining cycles is a function of the input data, and not generally equal to WCRC.

However, the two become equal for some input data only for nodes on the critical execution path. Comparing WCRC and remaining cycles provides a sense of *criticality*: If the remaining cycles are larger than WCRC, there is some slack available. Depending on the amount of slack, Eq. (1) could be used to determine and switch to the next frequency.

The aforementioned steps are implemented at particular points of execution in regions called *check-points*. Figure 2 illustrates an example check-point and its function. Let us assume that the worst case required cycles determined by WCRC analysis for edges G and H are 9,000 and 6,000, respectively. Furthermore, we assume that basic block 1 requires 1,000 cycles to run. This means that the critical path of the application and the WCRC at the entry of program snippet shown in Figure 2 is 10,000 cycles. As shown in Figure 2, we insert check-points for edges, G and H to update the counters used to track

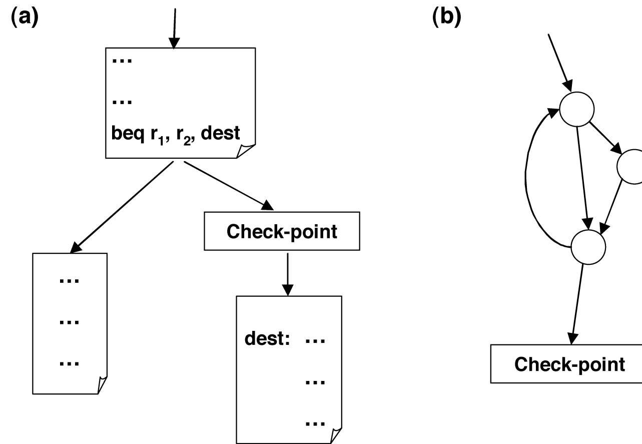

Fig. 3. Check-points are inserted onto selected edges.

the number of remaining cycles. When the execution goes to check-point G, the number of remaining cycles is 10,000–1,000 = 9,000, because 1,000 cycles are spent for executing basic block 1. Check-point G finds that the number of remaining cycles is no greater than WCRC at that check-point. This means it is not safe to lower the frequency at this moment because it would violate the deadline.

On the other hand, if the execution goes to check-point H, the number of remaining cycles is still 10,000–1,000 = 9,000, however, the WCRC for the execution of the rest of the program is 6,000 cycles. This means that instead of operating under the original frequency, which must be greater than or equal to $9{,}000/T_L$, we can scale down the frequency to $6{,}000/T_L$ without violating the deadline. After scaling down the frequency, the execution continues and we set the number of remaining cycles from 9,000 to 6,000. Essentially, the mode switching check-point can be thought of as a virtual entry point for the rest of the application.

In the preceding example, we intentionally ignored the delay penalty associated with switching between modes. Let us assume S_d is the switching delay in cycles. As we take the latency penalty into account, the functionality of the check-point only needs a small modification. In this case, instead of comparing the number of remaining cycles to WCRC, the check-point would compare remaining cycles to WCRC + S_d, where S_d is the switching delay in cycles. If the number of remaining cycles are reasonably larger than WCRC + S_d, it means that we can scale down the frequency. If the worst-case remaining cycles are larger than WCRC but not reasonably larger than WCRC + S_d, the mode-switch will not be executed. However, we will not waste the existing slack because the execution slack is captured in the number of remaining cycles, and can be utilized in upcoming check-points.

We can insert check-points on three type of edges. The first type is forward branches (see Figure 3(a)). The reason is that branching might create changes in the number of remaining cycles. In addition, check-points can be inserted

on the edges that immediately follow a loop body (see Figure 3(b)). The reason we insert check-points here is that it is likely to have slack right after the loop because the actual number of iterations in the loop can be less than the iterations in the worst case. The third option is to insert check-points in the first basic block of the loop. By adding check-points in the first basic block, we can exploit the slack for each iteration of the loop. We set the minimum distance between two check-point during check-point insertion, thereby avoiding the redundant switch penalty brought by check-points. The minimum distance between two check-points needs to be larger than the average switching penalty. We usually set the minimum distance between check-points to be about 10–20 times the average switching penalty. If the next desired frequency is between two processor frequencies, we set the process to run at the faster frequency to ensure meeting the deadline constraint. However, calculating the elapsed time take this into account, and ultimately incorporate it into T_L at upcoming check-points.

We will have a huge switch penalty when we have too many check-points in the application or insert them into the wrong place. The easiest way of solving this issue is to set minimum distance between two check-points. Loops are another source of repeated the switch penalties. We intend to avoid inserting check-points in the first basic block of the loop unless the size of loop is large.

A challenge for our heuristic is the discreteness in operating voltages and frequencies, that is, we cannot select our next frequency arbitrarily. Our scheme for calculating the number of remaining cycles and elapsed time enables us to efficiently address this issue.

In order to determine WCRC values and insert check-points, our algorithm visits each edge of the application CFG only three times. Therefore, its runtime complexity is $O(m)$, where m is the number of edges in the application control flow graph. For real applications, control flow graphs are sparse graphs in which the number of control flow edges grows linearly with the number of nodes (i.e., basic blocks). Consequently, our algorithm runs very efficiently and is readily scalable to large applications.

6. QUANTITATIVE ANALYSIS AND VALIDATION

6.1 Experimental Setup

In order to explore the effectiveness of our technique, we have developed two compilation flows based on the aforementioned algorithms, namely, the ILP-based strategy and check-point insertion method (CPIM). Both of the compilers generate executable code for our target processor. Figure 4 illustrates our experimental setup for the CPIM.

We use the MachineSUIF compiler framework [Smith and Holloway 2002] to extract the control flow graphs of the applications. We simulate program performance and energy to estimate the power and latency of application basic blocks by using our XTREM-based cycle-accurate DVS+ABB simulator [Huang

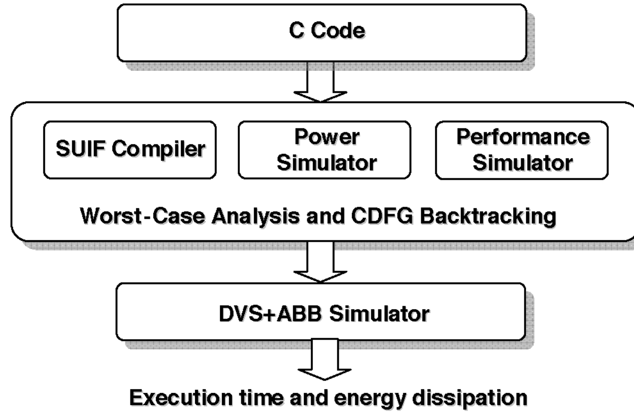

Fig. 4. The setup of experiments for the check-point insertion method (CPIM).

Table II. Application Execution Times and MILP Solution Times (sec)

Benchmark	Application Domain	# Basic Block	Exec. Time @600MHz	Deadline	Average MILP Solution Time	Ave. Backtracking Time	Speed Up
dijkstra	network	36	32.54	37.42	5.22	0.62	8.42
patricia	network	138	52.14	63.38	183	1.38	132.75
susan	automotive	203	43.14	52.8	1588	4.64	342.28
jpeg-dec	consumer	212	45.41	54.87	1613	4.63	348.49
gsm-dec	telecom	556	53.51	65.98	22451	14.69	1528.34

and Ghiasi 2006]. To perform worst-case analysis, we select the most complex input of each application as its *train input*.

For each application, there are also a number of simpler input data which would result in faster program execution. We refer them as *test input*. By using the train input to profile the application, we can capture the WCET and maximum number of loop iterations. After worst-case analysis, we apply CPIM, including WCRC analysis and check-point insertion. Finally, we generate the code and simulate it using our simulation framework to measure the energy and performance of the generated code.

Table II summarizes the complexity, execution time, and compilation time for both ILP and CPIM for selected applications from Guthaus et al. [2001]. The selected application domains justify the need for the execution deadline constraint and realtime operation of the generated code.

Table II reports the baseline execution-time of the applications (using train input) with no frequency scaling when our proposed processor runs at 600*MHz*. In order to investigate the effect of deadline relaxation on the quality of different frequency scaling methods, we have carried out extensive experiments using five different deadlines for each application. The first four are determined by averaging adjacent execution times (e.g., execution times @800*MHz* and @600*MHz*). For example, the first deadline is equal to the average execution time of 1*GHz* and 800*MHz* frequencies, with no frequency scaling mechanism. The last (fifth) deadline is set to 95% of the execution time at the slowest mode,

that is, running the processor at 200*MHz*. We would like to point out that the results are very consistent over different deadlines, and the improvements generally grow with more relaxed ones.

6.2 Experimental Result

We implemented the experimental flows depicted in Figure 4 and generated code for the five applications listed in Table II. In the first set of experiments, we tried to quantify the effectiveness of our approach by simulating the performance and energy consumption with the same input (train input). The compilation, corresponding simulations, and analysis are performed using the train input in the first set of experiments. In the second part of our experiment, we simulate a number of test inputs under the setting determined by the train input. We have one train data inputs and five test inputs for each application. The train input is the one associated with worst-case execution time (WCET).

The energy optimization techniques used in the experiment are ILP-based DVS-only (without body bias), ILP-based DVS+ABB, and CPIM-based DVS+ABB. To better measure the optimality gap of these techniques, we also adopted the analytical modeling and optimality analysis existing in the literature [Qu 2001; Xie et al. 2004], and applied it to our testbenches and processor model. When we calculate this analytical energy lower bound, we make some assumptions on the theoretical energy model of the processor. First of all, the ideal energy processor model has no switch latency or energy. Secondly, the ideal processor has exactly the same frequency-voltage pairs as our realistic model and cannot switch to other frequencies or voltages arbitrarily. As for memory, we keep it asynchronous with the processor in our ideal model. Then we use the simulator to determine the total execution cycles and time, total idle cycles and time for cache misses, and total cycles and time for processor operation.

According to Qu [2001], we need only two frequencies in the optimal discrete voltage schedule. We then apply the equation in Qu [2001] to calculate two consecutive frequencies for the analytical model. By using two consecutive frequencies and the total time for processor operation, we can get the energy consumed by the processor operation. As for the energy consumed during cache misses, we estimate the static energy consumption by applying the total cache-miss-time to the average static power consumption under zero bias voltage.

Note that the optimal energy dissipation predicted by such analytical modelings is only a lower bound on the amount of energy dissipation using any dynamic voltage scaling (either intratask or intertask) technique. The bounds not tight, and in practice not feasible.

Figure 5 and Table II show that CPIM reduces the compilation time by more than three orders of magnitude for large programs, while achieving energy savings that are very close to the ILP-based results. More importantly, the results are only about 10–20% away from the theoretical loose bound of energy savings, which means that our method comes reasonably close to the theoretical limit of DVS+ABB technology.

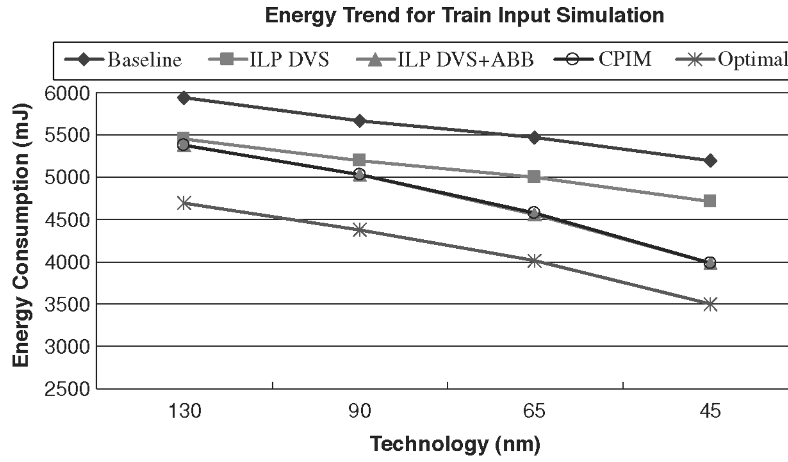

Fig. 5. Energy trend over the technology size for train input.

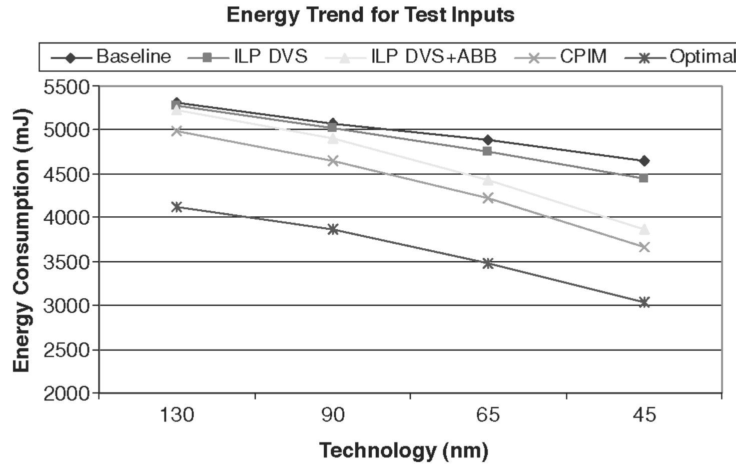

Fig. 6. Energy trend over the technology size for test inputs.

According to our experiment result, CPIM acquires 14.3% energy savings for the baseline energy and outperforms ILP-based DVS by about 6.16% on average under 65nm technology because CPIM also optimizes leakage energy. Compared to the theoretical energy lower bound, CPIM is about 14.53% worse than the theoretical value. CPIM outperforms ILP-based DVS and DVS+ABB techniques by about 11.40% and 8.8%, respectively, on average, under 65nm technology. Compared to the theoretical energy lower bound, the check-point method is about 19.65% away from the theoretical value. CPIM has an obvious advantage over the ILP-DVS technique for advanced technology. The reason is that CPIM is designed for DVS+ABB optimization.

Figures 5 and 6 illustrate the energy trends over device sizes for our optimized techniques using train input and test input simulation, respectively. The

chart shows that the difference between baseline and optimized techniques becomes larger in the advanced technology. Therefore, energy savings will increase greatly as device size shrinks.

6.3 Discussion

Based on the experiment, CPIM can achieve the same energy savings as the other methods, but reduces the compilation time greatly. This is because the ILP technique is based on integer linear programming, which grows exponentially with problem instance complexity. On the other hand, CPIM runtime depends on the number of basic blocks (i.e., control flow edges) in the application. Based on this observation, the complexity of CPIM will be in $O(m)$, where m denotes the total number of edges in the application CFG. The CPIM visits all edges of the CFG three times, and runs in linear time of input size.

CPIM acquires even more energy savings when we test our optimal setting by executing different inputs. Different from the ILP technique, our heuristic will assign the operating mode of the processor based on the execution progress. If the execution progress goes to the noncritical path, the check-point can determine whether it is worth executing a mode-switch based on the existing slack. If the slack is not sufficient to outperform the switch latency, the check-point will not execute the mode-switch instruction. The check-point executes mode-switch instructions only when it is worthwhile to do so. Therefore, CPIM will not waste the slack in the meaningless mode switch instruction. If the check-point does not execute the mode-switch instruction, the executing overhead of the check-point is relatively slight. However, the ILP-based technique inserts mode-switch instructions whose target operating mode are fixed to the edge of CDFG. Every fixed mode-switch instruction will be executed, regardless of existence of slack. Therefore, a difference in the control flow behavior between the train input and test input might downgrade the energy savings.

Since CPIM exploits the energy savings of the application by adaptively utilizing the existing slack, it is closer to the theoretical limit of the energy savings when executing test inputs. However, ILP comes slightly closer to the theoretical limit of the energy savings when executing the train input. The reason is that ILP is the optimal solution for train input, while its static operating mode setting cannot acquire optimal energy savings for different test inputs.

When we have some small slack which cannot be used to scale down the processor frequency, the check-point will leave these small slacks in the worst-case remaining cycles and these can usually be used in the future. This mechanism increases the opportunities to make use of slack and reduce the waste of the slack when the number of scaling frequencies is few.

7. CONCLUSIONS

We present a methodology to combine dynamic voltage scaling and adaptive body biasing during compilation of an application targeting a DVS+ABB-enabled embedded processor. Compiler-level analysis is particularly useful for embedded and realtime systems that demand lightweight operating systems. Additionally, compilers can exploit program execution trace information

that is not visible to the operating system, hence can assist dynamic voltage schedulers.

We develop a compiler framework that quickly and efficiently generates code for a DVS+ABB-enabled processor. Experimental results advocating the effectiveness of our approach show that the energy dissipation gap between leakage-aware and conventional DVS grows with technology scaling. Moreover, they show that our compiler's result come reasonably close to the theoretical limits of energy savings using this method.

REFERENCES

AbouGhazaleh, N., Mossé, D., Childers, B., Melhem, R. G., and Craven, M. 2003. Collaborative operating system and compiler power management for real-time applications. In *Proceedings of the IEEE Real Time Technology and Applications Symposium*. 133–143.

Andrei, A., Schmitz, M., Eles, P., Peng, Z., and Al-Hashimi, B. M. 2004. Overhead-Conscious voltage selection for dynamic and leakage energy reduction of time-constrained systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 518–523.

Azevedo, A., Issenin, I., Cornea, R., Gupta, R., Dutt, N., and Veidenbaum, A. 2002. Profile-Based dynamic voltage scheduling using program checkpoints. In *Design Automation and Test in Europe*. 168–175.

Berkeley-model. http://www-device.eecs.berkeley.edu/~ptm/introduction.html.

Burd, T., Pering, T., Stratakos, A., and Brodersen, R. 2000. A dynamic voltage scaled microprocessor system. *IEEE J. Solid-State Circ. 35*, 11 (Nov.), 1571–1580.

Duarte, D., Tsai, Y., Vijaykrishnan, N., and Irwin, M. J. 2002. Evaluating run-time techniques for leakage power reduction. In *Proceedings of the International Conference on VLSI Design*. 31–38.

Duarte, D., Vijaykrishnan, N., Irwin, M. J., Kim, H.-S., and McFarland, G. 2002. Impact of scaling on the effectiveness of dynamic power reduction schemes. In *Proceedings of the International Conference on Computer Design*. 382–387.

Guthaus, M., Ringenberg, J., Ernst, D., Austin, T., and Mudge, T. 2001. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*. 3–14.

Hsu, C.-H. and Kremer, U. 2003. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In *Proceedings of the Conference on Programming Language Design and Implementation*. 38–48.

Huang, P. and Ghiasi, S. 2006. Leakage-Aware intraprogram voltage scaling for embedded processors. In *Proceedings of the Design Automation Conference*.

Kao, J., Miyazaki, M., and Chandrakasan, A. 2002. A 175-mv multiply-accumulate unit using an adaptive supply voltage and body bias architecture. *J. Solid-State Circu. 37*, 11 (Nov.), 1545–1554.

Kesharvarzi, A., Narenda, S., Borkar, S., Hawkins, C., Roy, K., and De, V. 1999. Technology scaling behavior of optimum reverse body bias for leakage power reduction in ICS. In *Proceedings of the International Symposium Low Power Electronics and Design*. 252–254.

Kim, W., Kim, J., and Min, S. 2003. Dynamic voltage scaling algorithm for fixed-priority realtime systems using work-demand analysis. In *Proceedings of the International Symposium on Low Power Electronics and Design*. 396–401.

Li, Y., Malik, S., and Wolfe, A. 1999. Performance estimation of embedded software with instruction cache modeling. *ACM Trans. Des. Autom. Electron. Syst. 4*, 3, 257–279.

Longrun. 2007. http://www.transmeta.com/technology/architecture/longrun.html.

Martin, S., Flautner, K., Mudge, T., and Blaauw, D. 2002. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. 721–725.

Pillai, P. and Shin, K. 2001. Real-Time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the ACM Symposium on Operating System Principles*. 89–102.

Qu, G. 2001. What is the limit of energy saving by dynamic voltage scaling? In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*. 560–563.

RELE, S., PANDE, S., ONDER, S., AND GUPTA, R. 2002. Optimizing static power dissipation by functional units in superscalar processors. In *Proceedings of the 11th International Conference on Compiler Construction*. 261–275.

SMITH, M. AND HOLLOWAY, G. 2002. An introduction to machine suif and its portable libraries for analysis and optimization. Tech. Rep., Division of Engineering and Applied Sciences, Harvard University.

SUHENDRA, V., MITRA, T., ROYCHOUDHURY, A., AND CHEN, T. 2006. Efficient detection and exploitation of infeasible paths for software timing analysis. In *Proceedings of the Design Automation Conference*.

XIE, F., MARTONOSI, M., AND MALIK, S. 2004. Intraprogram dynamic voltage scaling: Bounding opportunities with analytic modeling. *ACM Trans. Architecture Code Optimization 1*, 3 (Sept.), 1–45.

XSCALE. 2007. http://www.intel.com/design/intelxscale.

YAN, L., LUO, J., AND JHA, N. 2003. Combined dynamic voltage scaling and adaptive body biasing for heterogeneous distributed real-time embedded systems. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. 30–37.

YAO, F., DEMERS, A., AND SHENKER, S. 1995. A scheduling model for reduced CPU energy. In *Proceedings of the Symposium on Foundations of Computer Science*. 374–382.

YOU, Y., LEE, C., AND LEE, J. 2006. Compilers for leakage power reduction. *ACM Trans. Des. Autom. Electron. Syst. 11*, 1, 147–164.

ZHANG, W., KANDEMIR, M., VIJAYKRISHNAN, N., IRWIN, M., AND DE, V. 2003. Compiler support for reducing leakage energy consumption. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. 1146–1147.

Received September 2006; revised January 2007; accepted March 2007