Efficient Dynamic Voltage/Frequency Scaling Through Algorithmic Loop Transformation

Mohammad Ali Ghodrat
School of Information and Computer Sciences
University of California, Irvine
mghodrat@ics.uci.edu

Tony Givargis
School of Information and Computer Sciences
University of California, Irvine
givargis@ics.uci.edu

ABSTRACT

We present a novel loop transformation technique, particularly well suited for optimizing embedded compilers, where an increase in compilation time is acceptable in exchange for significant reduction in energy consumption. Our technique transforms loops containing nested conditional blocks. Specifically, the transformation takes advantage of the fact that the Boolean value of a conditional expression, determining the true/false paths, can be statically analyzed and this information, combined with loop dependency information, can be used to break up the original loop, containing conditional expressions, into a number of smaller loops without conditional expressions. Subsequently, each of the smaller loops can be executed at the lowest voltage/frequency setting yielding overall energy reduction. Our experiments with loop kernels from *mpeg4*, *mpeg-decoder*, *mpeg-encoder*, *mp3*, *qsdpcm* and *gimp* show an impressive energy reduction of 26.56% (average) and 66% (best case) when running on a StrongARM embedded processor. The energy reduction was obtained at no additional performance penalty.

Categories and Subject Descriptors

D.3.4 [**Processors**]: Compilers; I.1 [**Computing Methodologies**]: Symbolic and Algebraic Manipulation

General Terms

Algorithms

Keywords

Algorithmic Loop Transformation, Compiler Optimization for Low Power, Dynamic Voltage/Frequency Scaling

1. INTRODUCTION

Fueled by a growing demand for a rich set of functionality, the complexity of embedded systems and the underlying compute requirements continue to rise. Consequently, maintaining the energy consumption of a typical embedded system below an acceptable level poses a major design challenge. Moreover, for the large class of portable embedded systems, low-energy design has become a first class concern.

Aggressive compiler optimization, in particular those that address loops, can significantly improve the performance/energy requirement of the software, thus justifying the additional compilation time overhead. This is in particular true in the embedded system domain where software has become a key element of the design process and performance/energy requirement is of a critical concern. Furthermore, it is acceptable for a compiler intended for embedded computing to take longer to compile but perform aggressive optimizations, such as the ones presented in [28] and [8].

In what is know as intra-task Dynamic Voltage/Frequency Scaling (DVFS), the compiler statically annotates the application software with DVFS instructions at branch points (i.e., immediately following a branch instruction) such that all execution paths of the task execute in the same amount of time (i.e., the deadline) [9] [20] [22] [2] [12] [10]. Energy reduction is achieved whenever execution follows a path shorter than the critical path. Ideally, the DVFS instructions take effect instantaneously without incurring any power or time overhead.

In most implementations, however, there is a time-lag and an energy cost associated with voltage and/or frequency scaling. As a result, ideal intra-task DVFS is not practical. In particular, application of DVFS to loop bodies containing conditional branches usually is not feasible as the time/energy overhead of scaling at loop iteration level is greater than the potential saving in energy. As an example, using the formulas provided in [3], to switch from voltage/frequency pair (1.55 V, 624 MHz) to voltage/frequency pair (0.9 V, 104 MHz), we incur an overhead of 3.185 μS (time) and 26 μJ (energy). For the code sample shown in Figure 1, each iteration takes 0.26 μS and 0.33 μJ when executed at (1.55 V, 624 MHz). Thus, DVFS cannot feasibly be applied on a per iteration basis. A more practical solution may be to make a single adjustment for the entire execution of the loop rather than per iteration.

Our technique transforms loops containing nested conditional blocks. Specifically, the transformation takes advantage of the fact that the Boolean value of a conditional expression, determining the true/false paths, can be statically analyzed and this information, combined with loop dependency information, can be used to break up the original loop, containing conditional expressions, into a number of smaller loops without conditional expressions. Subsequently, each

CODES+ISSS'09, October 11–16, 2009, Grenoble, France.

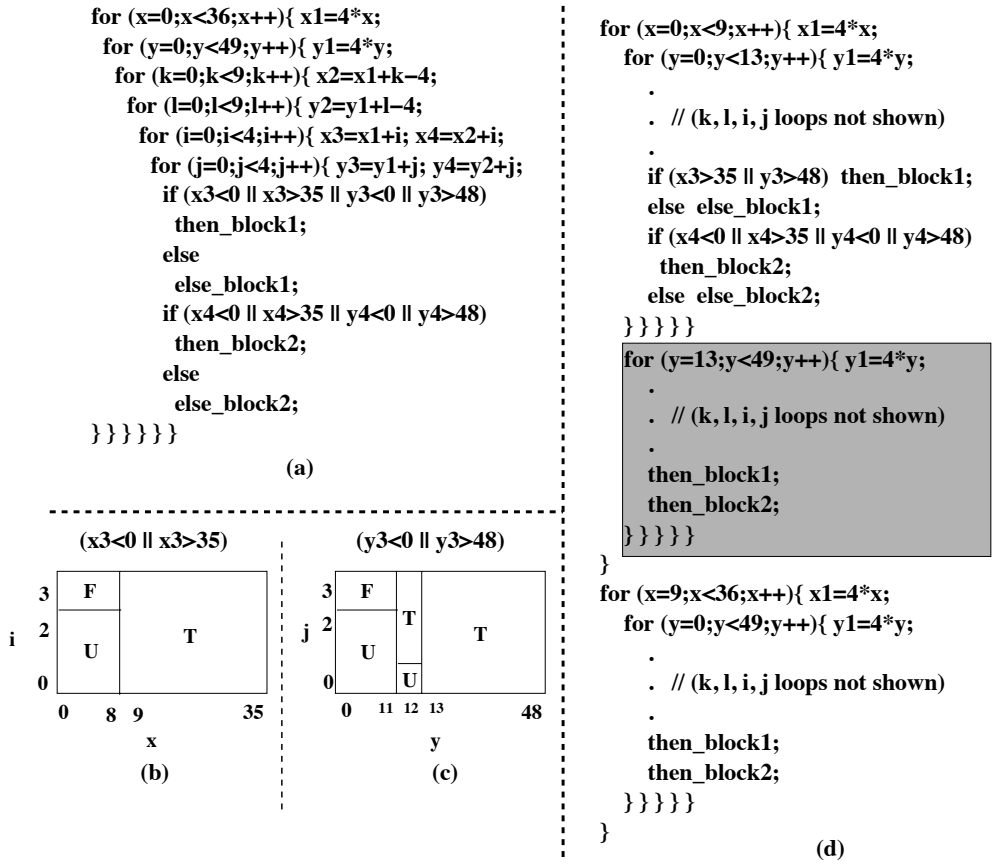

Figure 1: Motivational example

of the smaller loops can be executed at the lowest voltage/frequency setting yielding overall energy reduction.

The rest of this paper is organized as follows. In Section 2, we outline the related work. In Section 3, we formulate the problem and show a motivational example. In Section 4, we establish some preliminaries. In Section 5, we establish our low-energy code transformation technique. In Section 6, we show our experimental results. In Section 8, we conclude.

2. RELATED WORK

Dynamic voltage/frequency scaling (DVFS) techniques can be broadly categorized into interval-based techniques [9] [20], intra-task techniques [22] [2] and inter-task [12] [10] techniques. Interval-based technique is a history based technique, and it will assign a new voltage/frequency to the current interval based on the previous intervals' workload. In inter-task DVFS technique, the granularity of the voltage/frequency assignment is at the task-level. Each task will be assigned a different voltage/frequency based on its workload and its deadline. Intra-task DVFS techniques apply at finer granularity, modulating voltage/frequency of a single task as it executes. Our technique is considered an intra-task also, since it applies on the loop body.

Compiler optimization techniques for low power can be applied at different steps of compilation: common performance optimizations [1], instruction selection [25], instruction scheduling [26], register allocation [15], memory access time optimization [19] and source code transformation for low power [21] [6].

Our work can be categorized as a source code transformation technique. All the mentioned techniques are orthogonal to our technique and can be applied in parallel to ours. The work in [21], like our work, uses a compiler performance optimization that yields energy reduction as one of its byproducts.

Our proposed technique, to the best of our knowledge, is the first that addresses intra-task voltage scaling of loops with nested conditional branches, in particular addressing the voltage/frequency switching overhead.

3. PROBLEM FORMULATION

Loops contribute to a large amount of execution time, so any optimization (performance/power) on loops can have a significant improvement on the entire program. If a loop has a nested conditional block, each of its iteration can take different time, depending on which path of conditional block is taken. Given the cost of voltage/frequency scaling, a per iteration DVFS approach will yield poor energy (and possibly performance) savings, as the overhead of switching will add to a greater sum than the savings in power. In this paper, we provide a solution that allows for fine grain DVFS of loops that eliminates the before mentioned overhead problem.

Imagine that we can partition the loop iteration space into a series of disjoint subspaces with this property that in each subspace either of *then* part or *else* part of the conditional

block is executed but not both. In this case, if the worst case execution time for the *then* or *else* part is less than the worst case execution time of the original conditional block, then each subspace can finish its job sooner than the worst case execution time for the conditional block, and we can use this time slack to enable DVFS technique. Below we present a motivational example:

3.1 Motivational example

Consider the code shown in Figure 1-a, which is a loop kernel from *MPEG4*. If the conditional expression $(x3 < 0||x3 > 35)||(y3 < 0||y3 > 48)$ is analyzed statically (shown in Figure 1-b and c), we realize that $(x3 < 0||x3 > 35)||(y3 < 0||y3 > 48)$ is true when $9 \leq x \leq 35$ or $13 \leq y \leq 48$ as explained in [7]. The loop in Figure 1-a can be transformed as shown in Figure 1-d as explained in [8]. Now if we measure the time taken for the highlighted nested loop (loop on $y \in [13, 48]$) to execute on StrongARM SA1100, we obtain 130454 cycles. The original loop nest executing on StrongARM SA1100 takes 6380215 cycles. Considering a constant execution time (before and after transformation), we can use the difference in execution time (6249761 cycles) to enable DVFS for the given loop nest. In this case, by lowering the voltage and frequency of the processor from ($V_{dd} = 1.55V$ *and* $f = 624MHz$) to ($V_{dd} = 1.1V$ *and* $f = 312MHz$), the code will take 420 μS time and 1.08 mJ energy instead of the original 10224 μS time and 50.6 mJ energy. This is 97.86% reduction in energy for the mentioned loop nest.

4. PRELIMINARIES

In this section we review two previous contributions which are the basis of this work. In Section 4.1, we review the analysis technique behind conditional expression evaluation [7] and in Section 4.2 we review the transformation technique for loops with nested conditional blocks [8] which is the basis of our first step in our code transformation technique presented in this paper(see Section 5).

4.1 Domain Space Partitioning

In this subsection we summarize the analysis technique developed in [7] and used for our transformation. Without loss of generality, the remaining discussions in the paper will use C/C++ notation. Every program can be represented as a *Control Data Flow Graph (CDFG)* intermediate form. A CDFG is a graph that shows both data and control flow in a program. The nodes in a CDFG are *basic blocks*. Each basic block contains straight lines of statements with no branch except for the last statement and no branch destination except for the first statement. The edges in a CDFG represent the control flow in the program.

As defined in [7], a conditional expression *cond_expr* is either a *simple condition* or a *complex condition*. A simple condition is in the form of ($expr_1$ *ROP* $expr_2$). Here, $expr_1$ and $expr_2$ are *arithmetic expressions* and *ROP* is a relational operator ($=, \neq, <, \leq, >, \geq$). An arithmetic expression is formed over the language ($+$, $-$, $\times$, constant, variable). A complex condition is either a simple condition or two complex conditions merged using *logical operators* (&&, ||, !).

An *integer interval* of the form $[a, b]$ represents all possible integer values in the range a to b, inclusively. The operations ($+$, $-$, $\times$, $/$) can be defined on two intervals $[a, b]$ and $[c, d]$. We refer the interested reader to [16] for a full coverage of interval arithmetic.

We define an n-dimensional space to be a box-shaped region defined by the Cartesian product $[l_0, u_0] \times [l_1, u_1] \times ... \times [l_{n-1}, u_{n-1}]$. Hence, for a given program with n input integer-variables x_0, x_1, ..., x_{n-1}, the *program domain space* is an n-dimensional space defined by the Cartesian product $[min_0, max_0] \times [min_1, max_1] \times ... \times [min_n, max_n]$, where min_i and max_i are defined based on the type of the variable x_i (e.g. if x_i is of type *signed character* then $min_i = -128$ and $max_i = 127$).

Given the conditional expression *cond_expr* with variables $x_1, x_2, ..., x_k$, the domain space partitioning problem [7] is to partition the domain space of *cond_expr* into a minimal set of k-dimensional spaces $s_1, s_2, ..., s_n$ with each space s_i having one of `true`(T), `false`(F), or `unknown`(U) Boolean value. If space s_i has a Boolean value of `true`, then *cond_expr* evaluates to `true` for every point in space s_i. If space s_i has a Boolean value of `false`, then *cond_expr* evaluates to `false` for every point in space s_i. If space s_i has a Boolean value of `unknown`, then *cond_expr* may evaluate to `true` for some points in space s_i and `false` for others.

For example, consider $cond\_expr : 2 \times x_0 + x_1 + 4 > 0$ (domain space $[-5, 5] \times [-5, 5]$). Figure 2 shows the partitioned domain space and the corresponding Boolean values [7].

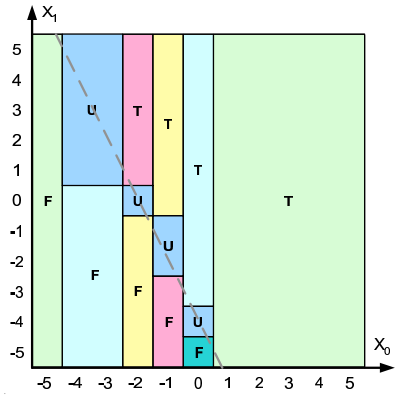

Figure 2: Partitioned Domain of $2x_0 + x_1 + 4 > 0$

4.2 Iteration space partitioning

In this subsection we summarize the loop transformation technique developed in [8] and used for our transformation. The proposed transformation technique presented in [8] targets loops that follow the normalized template shown in Figure 3-a. Here, there are n loop nests, with n indices $x_1, ..., x_n$. The body of the inner most loop contains at least one conditional block, called the *target conditional block*. Here, st_{cond_expr} computes the value of the branch condition v.

The proposed transformation in [8] decomposes the original nested loops of Figure 3-a into three parts, as shown in Figure 3-b. The first part sets up one or more nested loop structures with iteration spaces for which the st_{cond_expr} is known to be `true` at compile time. Likewise, the second part sets up one or more nested loop structures with iteration spaces for which the st_{cond_expr} is known to be `false` at compile time. The third part sets up one or more nested loop structures with an iteration space for which the st_{cond_expr}

can not be statically evaluated. The three parts combined cover the entire iteration space of the original nested loops. Since the evaluation of st_{cond_expr} is eliminated in parts one and two, the decomposed code executes substantially fewer instructions than the original code.

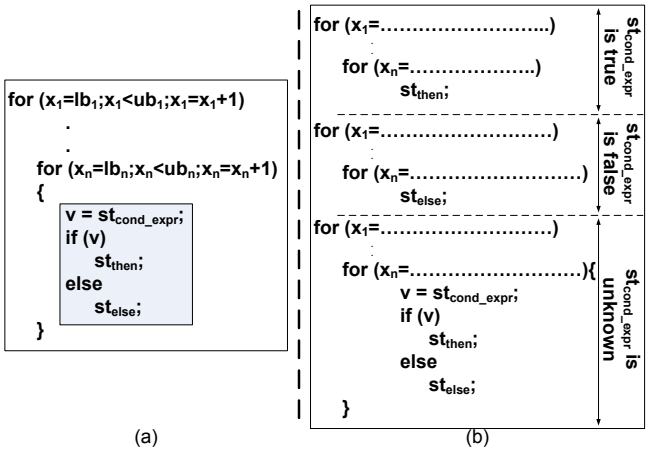

Figure 3: Loop transformation

5. TECHNICAL APPROACH

Figure 4 shows the proposed transformation technique whose inputs come from the output of the technique presented in [8]. For selected loop nests, we add two function calls at the beginning and the end of the loop nest. The first function call assigns a lower energy voltage/frequency pair to the processor. The second function call changes the voltage/frequency pair to another operating point.

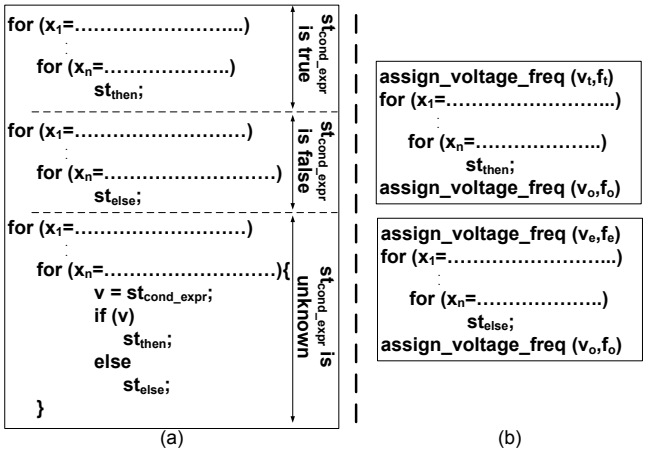

Figure 4: Voltage/frequency scaling transformation

Figure 5 shows our proposed methodology for low energy code transformation in seven steps. In step (1), we extract all the loops which have the template shown in Figure 3-a. In step (2), we use the work in [7] to evaluate the conditional expression nested in each loop. In step (3), we use the work in [8] to transform the loop, which gives us a series of nested loops for disjoint subspaces of the iteration space in the form shown in Figure 3-b.

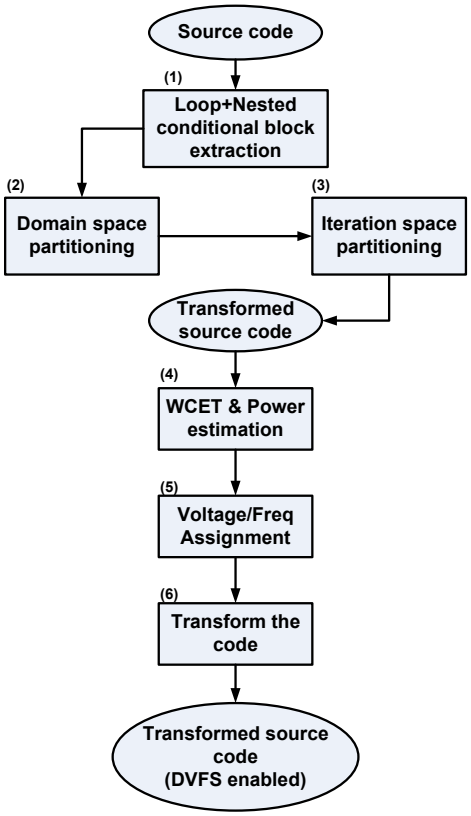

Figure 5: Block diagram of our proposed methodology

In step (4), we use measurement-based prediction or static analysis technique [17] [14] [27] to estimate the worst case execution time and power for original and transformed loop partitions. One can use a functional/power simulator to gather these numbers also, which is the method that we have used.

In step (5), we use a 0-1 Integer Programming technique to assign each loop partition a pair of (voltage, frequency) which minimizes the total energy consumption of the system. Assume the following:

- Variable $vf_{i,j}$ is a binary integer that is 1 when loop i has the voltage/frequency pair (v_j, f_j),
- $E_{i,j}$ and $T_{i,j}$ are the energy and the execution time of the loop i when it runs with voltage/frequency pair (v_j, f_j)
- $i \in [1, \#loop\,partitions]$
- $j \in [1, \#voltage/frequency\,pairs]$
- Minimize: $\sum_i \sum_j vf_{i,j} \times E_{i,j} + E_{overhead}$
- With the constraint: $\sum_i \sum_j vf_{i,j} \times T_{i,j} + T_{overhead} \leq DEADLINE$
- $E_{overhead} = \sum_i \sum_j vf_{i,j} \times E_{ov}(i,j)$
- $T_{overhead} = \sum_i \sum_j vf_{i,j} \times T_{ov}(i,j)$

Table 1: Selected Application List

Benchmark #	Application	Function description	Conditional expressions
B1	mpeg4	Motion estimation	$(x3 < 0 \| \| x3 > 35 \| \| y3 < 0 \| \| y3 > 48)$ $(x4 < 0 \| \| x4 > 35 \| \| y4 < 0 \| \| y4 > 48)$
B2	qsdpcm	Video compression	$((4 * x + vx - 4 + x4 < 0) \| \|$ $(4 * x + vx - 4 + x4 > (N/4 - 1)) \| \|$ $(4 * y + vy - 4 + y4 < 0) \| \|$ $(4 * y + vy - 4 + y4 > (M/4 - 1)))$
B3	gimp	Create Kernel	$(32 * x - 2 * i + 1)^2 + (32 * y - 2 * j + 1)^2 < 4096$
B4	mpgdec	Initialize Decoder	$(i < 0), (i > 255)$
B5	mpgenc	Ver./Hor. Filter,2:1 Subsample	$(i < 5), (i < 4), (i < 3), (i < 2), (i < 1)$
B6	mp3	Layer 3 Psych. Analysis	$j < sync\_flush, j < BLKSIZE$

where $E_{i,j}$ and $T_{i,j}$ are calculated in step (4). $DEADLINE$ is the time taken by the original code (Figure 3-a) running at the maximum frequency of the CPU. $E_{overhead}$ and $T_{overhead}$ are the total energy and time overhead incur in voltage/frequency switchings. $E_{ov}(i,j)$ and $T_{ov}(i,j)$ are the energy and time overhead when there is a switching from voltage/frequency (v_i, f_i) to voltage/frequency (v_j, f_j), and are pre-computed using the following formulas [3] [30]:

$$E_{ov}(i,j) = (1-\mu) \times C \times |v_i^2 - v_j^2|$$
$$T_{ov}(i,j) = (2 \times C/I_{MAX}) \times |v_i - v_j|$$

Based on [3] and [30], μ represents the energy efficiency of the power regulator which is considered to be 90%. Also, C is the voltage regulator's capacitance which is considered to be $10\mu F$ and I_{MAX} is the maximum allowed current which is assumed to be $1A$.

We note that in addition to all the transformed loop partitions, in this step, we assign a voltage/frequency pair to the original (unoptimized/untransformed) loop partitions also (the *unknown* section in Figure 3-b).

In step (6) the code is transformed by adding the function calls needed to change the voltage and frequency at the boundaries of the loop partitions.

6. EXPERIMENTAL RESULTS

To evaluate the proposed code transformation technique, several *loop kernels* from *MediaBench* [4] application suite were chosen. We also experimented with an *mp3* encoder implementation obtained from [29], an *mpeg4* full motion estimation obtained from [5], GNU Image Manipulation Program (*gimp*) [24] and also *qsdpcm* [23] video compression algorithm obtained from [13].

By *loop kernel*, we mean the region of code that was impacted by the transformation. For example, if the transformed code was a for-loop with a nested conditional block within, then the energy taken to execute that entire for-loop before and after the optimization was used to determine the energy reduction percentage. The characteristics of the loop kernels selected for our experiments are listed in Table 1. In Table 1, *conditional expressions* column shows the particular conditional expression(s). If there are more than one conditional expression in a loop kernel, then we run our algorithm for each instance of the conditional expression separately (i.e., the algorithm is run iteratively as long as improvements are obtained). Also, in Table 1, *Application* column shows the origin of the loop kernel and *Function description* column shows the functionality of the code where the kernel is taken from. We applied our transformation technique at the source level to each of the chosen benchmarks, compiled the original and the transformed code, and measured the improvement. We did these experiments using the StrongARM SA100 for which we had a power simulator available [18]. Moreover, the StrongARM SA100 is very similar to Intel PXA270 [11] which is a good candidate for applying DVFS. Table 2 shows several voltage/frequency pairs obtained from the Intel PXA270 manual [11].

Table 2: Voltage/Frequency points for experiments

Frequency(MHz)	624	520	416	312	312	208	104
Voltage(V)	1.55	1.45	1.35	1.25	1.1	1.15	0.9

6.1 Results

For the experiments we followed the steps shown in the block diagram of Figure 5. After applying the first three steps, we obtain a series of loop partitions, some of which are transformed and some are not. We compare the energy for three scenarios:

- Original: The original energy and power is measured for the loop without any transformation and for the highest voltage/frequency point (V=1.55 V and f=624 MHz). The measured time from this step is used as the deadline for the next two methods ($DEADLINE$ in the 0-1 Integer Programming formulation of Section 5). The energy and power results for the 6 selected benchmarks are shown in the 2nd and the 3rd columns of Table 3.

- Coarse-grain DVFS: The coarse-grain DVFS applies the DVFS technique once and for the entire transformed loop (Figure 3-b). For frequency assignment, we find the lowest frequency (f_{CG}) that the transformed code can run and meet the deadline (i.e., DEADLINE in the 0-1 Integer Programming formulation). Assuming that the transformed code takes C_{trans} cycles to execute, and the measured power for transformed code for the (V_{CG}, f_{CG}) is $P_{f_{CG}}$, then:

$$T_{trans} = C_{trans} \times (1/f_{CG})$$
$$E_{trans} = T_{trans} \times P_{f_{CG}}$$

Where T_{trans} is the time to run the transformed loop with the selected frequency. This time might be lower than $DEADLINE$ and since we desire a unique point of reference for all the 3 scenarios, we place the CPU into idle mode for the duration of time up-to the deadline:

$$T_{idle-cg} = DEADLINE - T_{trans}$$
$$E_{idle-cg} = T_{idle-cg} \times P_{idle}$$

Here, we use the power consumption of the idle mode for different frequency settings from the Intel PXA270 manual [11] to compute $E_{idle-cg}$. We compute the total energy as follows:

$$E_{coarse-grain} = E_{trans} + E_{idle-cg}$$

The energy and power results for the 6 selected benchmarks are shown in the 4th and the 5th columns of Table 3.

- Fine-grain DVFS: For each benchmark, for each transformed loop partition and for each voltage frequency points (V_j, f_j) mentioned in Table 2, we ran the power simulator [18] and measured the number of cycles (C_i) and total power consumption $(P_{i,j})$ reported by the simulator for each transformed loop partition. For each voltage/frequency point (V_j, f_j), we compute the energy and the time for each transformed loop partition by:

$$T_{i,j} = C_i * (1/f_j)$$
$$E_{i,j} = P_{i,j} * T_{i,j}$$

The $E_{i,j}$'s and $T_{i,j}$'s are the inputs to step (5) in Figure 5, which will return the set of voltage/frequency assignments for the loop partitions. For loop partition i, we define the index of assigned voltage aV_i and the index of assigned frequency aF_i. Similar to coarse-grain DVFS, there may be some idle time, during which we place the CPU into idle mode to obtain the total energy as follows:

$$T_{idle-fg} = DEADLINE - \sum T_{i,aV_i}$$
$$E_{idle-fg} = T_{idle-fg} \times P_{idle}$$

The energy for the fine grain DVFS $(E_{fine-grain})$ can be computed as below:

$$E_{fine-grain} = \sum E_{i,aV_i} + E_{idle-fg}$$

The energy and power results for the 6 selected benchmarks are shown in the 6th and the 7th columns of Table 3.

Figure 6 shows the energy reduction percentage for both coarse-grain and fine-grain methods compared to the original case. As can be seen in Figure 6, on average fine-grain scenario does better in energy reduction compared to coarse-grain, when compared to original version (26.56% reduction for fine-grain and 22.7% reduction for coarse-grain). Also in best case, fine-grain can gain 66% energy reduction compared to original, but coarse-grained can gain 56% energy reduction compared to original. This is expected, since fine-grain matches itself better to the speed need of the software, but coarse grain mostly runs faster and then sits idle. As can be seen in Table 3, for *mp3* and *mpegdec*, the fine-grain approach essentially achieves the same performance as the coarse-grain, yielding similar energy savings.

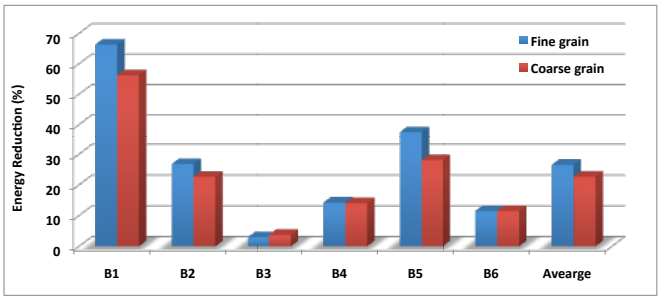

Figure 6: Energy reduction percentage for the selected benchmarks

7. ACKNOWLEDGEMENT

This work was in part supported by grant #0749508 from the National Science Foundation.

8. CONCLUSION

We presented a novel loop transformation technique, particularly well suited for optimizing embedded compilers, where an increase in compilation time is acceptable in exchange for significant energy consumption decrease. Our contribution was specifically, enabling DVFS technique for the loop nests containing conditional blocks without adversely affecting their execution time. Specifically, the transformation takes advantage of the fact that the Boolean value of a conditional expression, determining the true/false paths, can be statically analyzed and this information, combined with loop dependency information, can be used to break up the original loop, containing conditional expressions, into a number of smaller loops without conditional expressions. Subsequently, each of the smaller loops can be executed at the lowest voltage/frequency setting yielding overall energy reduction. Applying the proposed transformation technique on loop kernels taken from *Mediabench*, *mpeg4*, *qsdpcm* and *gimp*, we measured an impressive energy reduction of 26.56% (average) and 66% (best case) when running on a StrongARM embedded processor. The energy reduction was obtained at no additional performance penalty.

9. REFERENCES

[1] A. Aho. *Compilers principles, techniques and tools.* Addison Wesley, Reading, Massachusetts, 1988.

[2] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Profile-based dynamic voltage scheduling using program checkpoints. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, pages 168–175, 2002.

[3] T. D. Burd and R. W. Brodersen. Design issues for dynamic voltage scaling. In *ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design*, pages 9–14, 2000.

Table 3: Measured Energy (Joule) and Power (Watt) numbers for the selected loop kernels (same deadline)

	Original)		DVFS coarse-grain		DVFS fine-grain	
Application	Energy	Power	Energy	Power	Energy	Power
mpeg4	2.856	4.821	1.250	2.111	0.962	1.624
qsdpcm	0.221	5.451	0.171	4.218	0.162	3.991
gimp	1.269	5.634	1.223	5.426	1.231	5.466
mpgdec	0.00084	5.053	0.00072	4.345	0.00072	4.332
mpgenc	0.0217	5.268	0.0156	3.788	0.0136	3.292
mp3	0.00121	4.843	0.00107	4.285	0.00106	4.285

[4] C. L. et. al. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[5] H. Falk and P. Marwedel. Control flow driven splitting of loop nests at the source code level. In *Proceedings of DATE*, pages 410– 415, 2003.

[6] Y. Fei, S. Ravi, A. Raghunathan, and N. K. Jha. Energy-optimizing source code transformations for operating system-driven embedded software. *Trans. on Embedded Computing Sys.*, 7(1):1–26, 2007.

[7] M. Ghodrat, T. Givargis, and A. Nicolau. Equivalence checking of arithmetic expressions using fast evaluation. In *Proceedings of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 147–156, 2005.

[8] M. Ghodrat, T. Givargis, and A. Nicolau. Control flow optimization in loops using interval analysis. In *Proceedings of CASES*, pages 157 – 166, 2008.

[9] K. Govil, E. Chan, and H. Wasserman. Comparing algorithm for dynamic speed-setting of a low-power cpu. In *MobiCom '95: Proceedings of the 1st annual international conference on Mobile computing and networking*, pages 13–25, 1995.

[10] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. B. Srivastava. Power optimization of variable voltage core-based systems. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 176–181, 1998.

[11] Intel. Intel xscale technology-basd embedded processor. Available as http://www.intel.com/design/embeddedpca/applicationsprocessors/302302.htm.

[12] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *ISLPED '98: Proceedings of the 1998 international symposium on Low power electronics and design*, pages 197–202, 1998.

[13] I. Issenin and N. Dutt. Data reuse driven energy-aware mpsoc co-synthesis of memory and communication architecture for streaming applications. In *CODES-ISSS 2006*, pages 294–299, 2006.

[14] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C. S. Kim. An accurate worst case timing analysis for risc processors. *IEEE Trans. Softw. Eng.*, 21(7):593–604, 1995.

[15] H. Mehta, R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh. Techniques for low energy software. In *ISLPED '97: Proceedings of the 1997 international symposium on Low power electronics and design*, pages 72–75, 1997.

[16] R. Moore. *Interval analysis*. Prentice-Hall, Englewood Cliffs, N. J., 1966.

[17] K. D. Nilsen and B. Rygg. Worst-case execution time analysis on modern processors. In *LCTES '95: Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pages 20–30, 1995.

[18] U. of Michigan. The simplescalar-arm power modeling project. Available as http://www.eecs.umich.edu/ panalyzer/.

[19] P. R. Panda and N. D. Dutt. Low-power memory mapping through reducing address bus activity. *IEEE Trans. Very Large Scale Integr. Syst.*, 7(3):309–320, 1999.

[20] T. Pering, T. Burd, and R. Brodersen. Voltage scheduling in the iparm microprocessor system. In *ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design*, pages 96–101, 2000.

[21] A. Peymandoust, T. Simunic, and G. de Micheli. Low power embedded software optimization using symbolic algebra. In *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, pages 1052–1058, 2002.

[22] D. Shin, J. Kim, and S. Lee. Intra-task voltage scheduling for low-energy, hard real-time applications. volume 18, pages 20–30, 2001.

[23] P. Stobach. A new technique in scene adaptive coding. In *Proceedings of EUSIPCO*, 1988.

[24] T. G. Team. Gnu image manipulation program. Available as http://www.gimp.org/.

[25] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *IEEE Trans. Very Large Scale Integr. Syst.*, 2(4):437–445, 1994.

[26] V. Tiwari, S. Malik, A. Wolfe, and M. T.-C. Lee. Instruction level power analysis and optimization of software. *J. VLSI Signal Process. Syst.*, 13(2-3):223–238, 1996.

[27] V. Venkatachalam and M. Franz. Power reduction techniques for microprocessor systems. *ACM Comput. Surv.*, 37(3):195–237, 2005.

[28] M. Wolfe. How compilers and tools differ for embedded systems. In *Proceedings of the CASES*, page 1, 2005.

[29] www.mp3tech.org. Iso mp3 sources. Available as http://www.mp3-tech.org/programmer/sources/dist10.tgz.

[30] F. Xie, M. Martonosi, and S. Malik. Efficient behavior-driven runtime dynamic voltage scaling policies. In *Proceedings of CODES+ISSS*, pages 105–110, 2005.