Energy-Constrained Prefetching Optimization in Embedded Applications[*]

Juan Chen[1], Yong Dong[2], Hui-zhan Yi[3], Xue-jun Yang

School of Computer, National University of Defense Technology, Changsha 410073, China

[1]{juanchen@nudt.edu.cn}, [2]{luckpeople@163.com}, [3]{huizhanyi@nudt.edu.cn}

Abstract. In energy-constrained settings, most low-power compiler optimization techniques take the approach of minimizing the energy consumption while meeting no performance loss. However, it is possible that the available *energy budget* is not sufficient to meet the optimal performance objective. To limit energy consumption within a given energy budget, energy-constrained optimization approach is more significant. In this paper, we present an energy-constrained prefetching optimization approach through which memory or CPU stalls (caused by too early or too late prefetching) can be reduced so that energy budget is met. Optimal performance objective is achieved under a given energy budget. We evaluate the effectiveness of our energy-constrained prefetching optimization approach through simulations.

Keywords: software prefetching, energy-constrained, DVS, embedded applications.

1. Introduction

With the development of wireless, portable and embedded devices, power-aware systems have moved to the forefront of computer research in recent years, among which **Dynamic Voltage Scaling** (**DVS**) is major low-power techniques [1][2][3][4][5].

Many low-power techniques focus on minimizing the energy consumption with performance constraints. In this paper, we look at the problem from the other perspective: we consider optimized application with energy constraint (limitation). We assume that a limited energy budget (E_{budget}) is provided and it is usually less than the energy bound (E_{bound}) for the optimal performance objective. Such a situation is significant when the computing device depends on the battery power supply, such as military, space, and disaster recovery missions, etc. In this paper, we are concerned with energy-constrained prefetching optimization. With our approach, applications optimized with prefetching reach the optimal performance within the available energy budget. Clearly, the compiler can make use of well-known power-aware scheduling techniques (such as DVS) for energy savings. In scarce-energy settings where $E_{budget} < E_{bound}$, we have a serious problem: we have to sacrifice some prefetching optimization performance.

Software prefetching improves performance by effectively hiding memory access latency through overlapping memory access with computation [6]. However, software prefetching does not always implement perfect data prefetching because it requires prefetch instructions are inserted at the right places. Prefetching too early or too late may yield memory or CPU stalls. By reducing memory

[*] This work was supported by the National High Technology Development 863 Program of China under Grant No. 2002AA1Z2101 and No. 2004AA1Z2210.

frequency or CPU frequency or both, we not only can reduce these stalls but also can limit energy consumption within a given energy budget.

There usually exist two imperfect prefetch optimization cases: **CPU-bound case** and **memory-bound case**. In CPU-bound case, prefetching operation is too early or memory access latency is too short so that the prefetched data is provided before actually being used. Although memory access latency is completely hidden, prefetching too early makes memory access completes ahead of the actual data access. We refer to this time interval as *memory stalls*. The second case is memory-bound case, where memory access dominates the whole time and CPU stalls cannot be avoided due to prefetching too late or too long memory access latency. Figure 1 illustrates the behavior of prefetching.

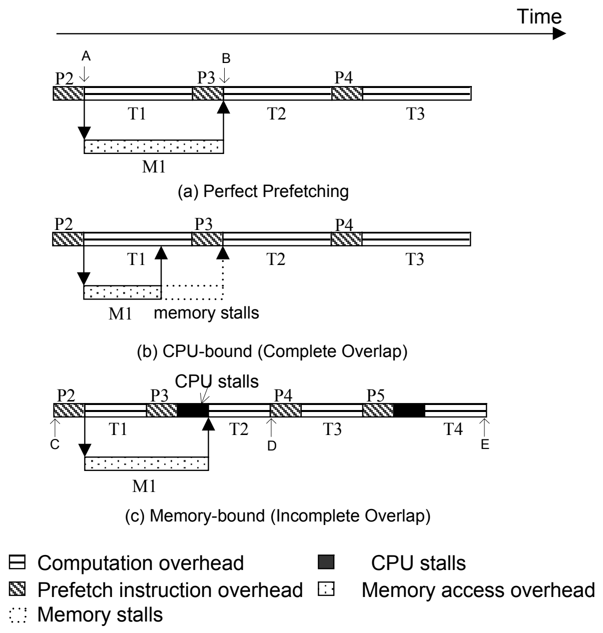


Figure 1: Prefetching Optimization. P2, P3, P4, P5 represent prefetch instrution overhead for the second, third, fourth and fifth iteration, respectively. M1 represents memory access overhead caused by prefetching. T1, T2, T3 and T4 represent computation overhead in the first, second, third and fourth iteration. Note: here iteration denotes the outmost iteration.

With DVS, we utilize CPU stalls and memory stalls to reduce memory frequency or CPU frequency or both so as to meet scarce energy budget. In CPU-bound case, the major method is to adjust memory frequency to eliminate memory stalls combined with small CPU frequency scaling. In memory-bound case, the major approach is to reduce CPU frequency to eliminate CPU stalls combined with small memory frequency scaling. Furthermore, it is necessary to adjust both of their frequencies instead of one of them for the optimal performance under energy constraint.

In this research effort, we introduce the notion *time* to characterize the performance benefits—*time* denotes the program execution time with software prefetching optimization. Optimal performance objective is minimizing *time*.

When energy is scarce, DVS capability (DVS for memory and CPU) guarantees that maximum performance objective will be met while staying within a given energy budget.

We simulate the impact of different parameters on our optimization. Experimental results show our approach is effective in implementing energy-constrained performance optimization.

The remainder of this paper is organized as follows. Section 2 reviews the related works. In section 3, we formulate an energy-constrained software prefetching problem to an optimization problem. Section 4 gives compilation strategy. Section 5 provides experimental results and analysis. Section 6 gives the conclusions.

2. Related Works

There are a lot of literatures on software prefetching techniques [6][7][8]. Todd [6] provided a comprehensive analysis on software prefetching. Ricardo Bianchini et al. [9] presented analytical models of the performance benefits of multithreading and prefetching. Our work built the energy model of prefetching.

DVS for CPU has been widely researched [1][2][3][4][10][11]. Furthermore, currently power-aware memory has been concerned by some researches besides DVS of CPU [5][12]. Pouwelse [12] alluded to the problem that the high cost of memory may dominate the overall power consumption of a system so that effective DVS of the CPU delivers marginal benefit. Xiaobo Fan et al. [5] exploited a positive synergistic effect between DVS and power-aware memories that can be transferred into lower power states. They believe that the best frequency/voltage for minimizing power consumption should be obtained by including memory power in the decision (The energy savings from DVS alone with standard memory are only 39% compared with energy savings of 89% by consistent method).

We exploited memory stalls and CPU stalls existing in imperfect prefetching and utilized them to reduce CPU or memory frequency/voltage for energy savings.

3. Energy-Constrained Prefetching Optimization

The prototypical loop that we optimize with prefetching looks like:

```
for ( i = 0 ; i < N ; i ++)  Compute ( i ) ;
```

After software prefetching optimization, the above loop is changed into:

```
// iteration 0, prefetch b blocks
prefetch ( 0, b );
for ( i = 0 ; i < N - step ; i += step ) {
    // prefetch b blocks for iterations i+step to i+2*step-1
    prefetch  ( i + step , b);
    // compute iterations i to i + step - 1
    for ( j = 0 ; j < step ; j ++ )  Compute ( i * step + j );
}
for ( j = 0 ; j < step ; j++ )  Compute ( i * step + j ) ;
```

In the above prefetching loop, several prefetch instructions are inserted. The major parameters involved in modeling the energy behavior of the above prefetching loop are the number of the cache blocks prefetched for each prefetching instruction, *b*; the number of prefetching instructions per iteration, N_b; the energy overhead of each prefetched cache block, E_b; the time overhead of each prefetched cache block, C_p; the total latency of cache misses per iteration, C_m; and the computation between two consecutive prefetch instructions, C_c. Table 1 summarizes these parameters.

Prefetching allows greater flexibility when trying to overlap memory access and computation, since the compiler or user can schedule the prefetches according to the memory access latency and the amount of work per iteration of each loop in the program. This intelligent scheduling of prefetches adjusts *b* and the iteration *step* size appropriately as the above code segment shows.

The optimal number of blocks to prefetch at a time depends on the available cache space; blocks prefetched into the cache may be replaced before being used. Another limitation is the number of prefetched blocks that can fit in a prefetch/transaction buffer. With these constraints, the amount of useful work that can be overlapped with memory access may be insufficient to hide memory latency completely. To analyze this problem, we simplify it as Figure 1 shows.

Figure 1 illustrates the behavior of the prefetching loop. The execution alternates between prefetch instruction and computation intervals. Each prefetch accessing cache blocks are to be used one computation block ahead. While it is possible to have loops that prefetch more than one computation block ahead, we can always transform such loops into an equivalent loop that prefetches a single computation block ahead.

In Figure 1(a), **P2** denotes prefetch instruction operation, where calculates the data address to be prefetched (b, N_b, E_b are involved). **M1** executes memory access operations caused by prefetch instruction (C_m, f_m are involved). **T2** accesses the prefetched data at point **B** (C_c, f_c are involved). Figure 1(a) shows perfect prefetching behavior. Memory accesses caused by prefetch are fully overlapped with computation, where no CPU stall or memory stall exists. If prefetech instructions cannot be inserted at the right places, two kinds of imperfect prefetchings occur as shown in Figure 1(b) and Figure 1(c). In terms of CPU-bound or memory-bound case, we adjust memory frequency f_m and CPU frequency f_c to meet energy budget.

Our energy-constrained prefetching optimization problem is stated as follows: given a loop L optimized with software prefetching and a DVS-enabled CPU and memory system, find the optimal CPU frequency f_c and memory frequency f_m, then the performance objective (time) is minimized while meeting a given energy budget.

3.1. Energy Constraint

The total energy consumption E_{total} consists of three parts: E_p, E_c, and E_m. That is,

$$E_{total} = E_p + E_c + E_m$$

where E_p represents the energy consumption of prefetching instructions. We assume the number of prefetched cache block per iteration is B, which is the product of b and N_b. Thus the energy consumed per loop iteration is $E_b \cdot b \cdot N_b$ and the energy consumption of the prefetching instructions E_p for N loop iterations is:

$$E_p = E_b \cdot b \cdot N_b \cdot N$$

E_c represents the energy consumption of CPU computation. Due to continuously variable CPU frequency, $P_c(f_c)$ is denoted the CPU power dissipation for CPU frequency of f_c. $\frac{C_c}{f_c}$ represents CPU computation time. Then CPU energy consumption during N loop iterations is:

$$E_c = P_c(f_c) \cdot \frac{C_c}{f_c} \cdot N$$

E_m represents memory energy consumption. Due to continuously variable memory frequency, $P_m(f_m)$ represents the memory power dissipation for memory frequency of f_m. $\frac{C_m}{f_m}$ represents memory access time. Then the memory access energy consumption for N iterations is:

$$E_m = P_m(f_m) \cdot \frac{C_m}{f_m} \cdot N$$

Thus E_{total} can be represented by:

$$E_{total} = E_b \cdot b \cdot N_b \cdot N + P_c(f_c) \cdot \frac{C_c}{f_c} \cdot N + P_m(f_m) \cdot \frac{C_m}{f_m} \cdot N$$

Given an energy budget E_{budget}, the following energy constraint must be satisfied:

$$E_b \cdot b \cdot N_b \cdot N + P_c(f_c) \cdot \frac{C_c}{f_c} \cdot N + P_m(f_m) \cdot \frac{C_m}{f_m} \cdot N \le E_{budget} \quad (1)$$

Power characteristics of CMOS is $P = \alpha C V^2 f$ and the relationship between supply voltage V and frequency f is $f \propto \frac{(V - V_{th})^{\beta}}{V}$, where V denotes supply voltage, f denotes clock frequency, P denotes power dissipation, α represents the activity factor, C represents the capacitance, V_{th} represents threshold voltage, and β is a proportional factor between 1 and 2. It is reasonable to assume frequency f is linearly proportion with voltage V so that we can get formula (2).

$$P = \alpha \cdot C \cdot f^3 \quad (2)$$

According to formula (2), we can achieve $P_c(f_c) = \alpha_1 \cdot C_1 \cdot f_c^3$ and $P_m(f_m) = \alpha_2 \cdot C_2 \cdot f_m^3$. So formula (1) is transformed to formula (3).

$$E_b \cdot b \cdot N_b \cdot N + k_1 \cdot f_c^2 \cdot C_c \cdot N + k_2 \cdot f_m^2 \cdot C_m \cdot N \le E_{budget} \quad (3)$$

where k_1 represents $\alpha_1 \cdot C_1$ and k_2 represents $\alpha_2 \cdot C_2$.

3.2. Optimal Performance Objective

In the previous section, we have defined performance objective—*time*. To calculate this value, two cases need to be considered. In CPU-bound case (Figure 1(b)), the total execution time of prefetching loop is calculated by prefetching instruction and CPU computation as follows.

$$T_{total} = N \cdot \left(\frac{b \cdot N_b \cdot C_p}{f_c} + \frac{C_c}{f_c}\right) \quad cond1$$

In memory-bound case, CPU stall occurs every other prefetching operation as Figure 1(c) shows. The time interval from **C** to **D** and the time interval from **D** to **E** are the same. We can conclude that the total execution cycles are *N/2* times as many as the execution cycles during **C** to **D**. Since the execution cycles during **C** to **D** are $\left(\frac{b \cdot N_b \cdot C_p}{f_c} + \frac{C_m}{f_m} + \frac{C_c}{f_c}\right)$, the total execution time is[1]

$$T_{total} = \frac{N}{2} \cdot \left(\frac{b \cdot N_b \cdot C_p}{f_c} + \frac{C_m}{f_m} + \frac{C_c}{f_c}\right) \quad cond2$$

where *cond1* and *cond2* represent CPU-bound case and memory-bound case, respectively. That is,

$$cond1 \quad represents \quad \frac{C_m}{f_m^0} \le \frac{b \cdot N_b \cdot C_p}{f_c^0} + \frac{C_c}{f_c^0}$$

$$cond2 \quad represents \quad \frac{C_m}{f_m^0} > \frac{b \cdot N_b \cdot C_p}{f_c^0} + \frac{C_c}{f_c^0}$$

[1] Here we assume $N \bmod 2 = 0$. If $N \bmod 2 = 1$, $T_{total} = \frac{N+1}{2} \cdot \left(\frac{b \cdot N_b \cdot C_p}{f_c} + \frac{C_m}{f_m} + \frac{C_c}{f_c}\right) - \frac{C_m}{f_m}$. When N is large enough, this little difference caused by N can be ignored.

where f_c^0 and f_m^0 represent initial CPU and memory frequencies, respectively.

The optimal performance objective is:

$$\min T_{total} = \begin{cases} \min(N \cdot (\frac{b \cdot N_b \cdot C_p}{f_c} + \frac{C_c}{f_c})) & cond1 \\ \min(\frac{N}{2} \cdot (\frac{b \cdot N_b \cdot C_p}{f_c} + \frac{C_m}{f_m} + \frac{C_c}{f_c})) & cond2 \end{cases} \tag{4}$$

3.3. Energy-Constrained Optimization Problem

Based on previous analysis, when CPU-bound case, energy-constrained optimization problem is represented as follows.

CPU-bound case

$$\min(N \cdot (\frac{b \cdot N_b \cdot C_p}{f_c} + \frac{C_c}{f_c})) \tag{5}$$

$$E_b \cdot b \cdot N_b \cdot N + k_1 \cdot f_c^2 \cdot C_c \cdot N + k_2 \cdot f_m^2 \cdot C_m \cdot N \le E_{budget} \tag{6}$$

$$C_m / f_m - (b \cdot N_b \cdot C_p + C_c) / f_c \le 0 \tag{7}$$

$$f_c^{'} \le f_c \le f_c^{''} \tag{8}$$

$$f_m^{'} \le f_m \le f_m^{''} \tag{9}$$

memory-bound case

$$\min(\frac{N}{2} \cdot (\frac{b \cdot N_b \cdot C_p}{f_c} + \frac{C_m}{f_m} + \frac{C_c}{f_c})) \tag{10}$$

$$E_b \cdot b \cdot N_b \cdot N + k_1 \cdot f_c^2 \cdot C_c \cdot N + k_2 \cdot f_m^2 \cdot C_m \cdot N \le E_{budget} \tag{11}$$

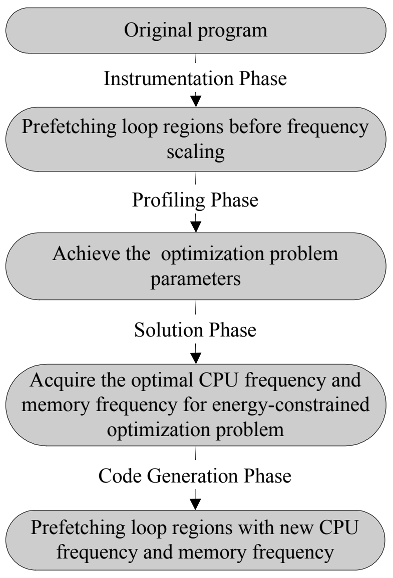
$$C_m / f_m - (b \cdot N_b \cdot C_p + C_c) / f_c > 0 \tag{12}$$

$$f_c^{'} \le f_c \le f_c^{''} \tag{13}$$

$$f_m^{'} \le f_m \le f_m^{''} \tag{14}$$

Here formula (7) and (12) guarantee CPU-bound case is still CPU-bound during frequency scaling and memory-bound case is still memory-bound. Inequality (8)-(9) and inequality (13)-(14) define the ranges of CPU and memory frequency scaling. The detailed meaning of each parameter and variable can be found in Table 1.

4. Compilation Strategy



Original program
Instrumentation Phase
Prefetching loop regions before frequency scaling
Profiling Phase
Achieve the optimization problem parameters
Solution Phase
Acquire the optimal CPU frequency and memory frequency for energy-constrained optimization problem
Code Generation Phase
Prefetching loop regions with new CPU frequency and memory frequency

Figure 2: Compilation Framework

System Parameter	Meaning	Value
E_b	Energy consumption by a cache block prefetched	10 pJ
b	The number of cache block prefetched by a prefetching instruction	4
C_p	A prefetched cache block computation time	1 cycles
k_1	The coefficient for $P_c = k_1 \cdot f_c^3$	7.72e-27
k_2	The coefficient for $P_m = k_2 \cdot f_m^3$	1.09e-24
f_c^0	Initial CPU frequency	1000 MHz
f_m^0	Initial memory frequency	133 MHz
$f_c^{'}$	The lower bound of continous CPU frequency	433 MHz
$f_c^{''}$	The upper bound of continuous CPU frequency	1000 MHz
$f_m^{'}$	The lower bound of continuous memory frequency	83 MHz
$f_m^{''}$	The upper bound of continuous memory frequency	133 MHz
Program Parmeter	**Meaning**	**Value**
N	Prefetching loop iteration counts	Program specified
C_c	Computation time in once iteration (cycles)	Profiled
C_m	Memory access time in once iteration (cycles)	Profiled
N_b	The number of prefetching instructions in one iteration	Optimization specified
E_{budget}	Energy budget	specified

Table 1: System parameters and program parameters for our optimal problem solution

Our energy optimization approach is profile-driven. The prototype implementation consists of four phases. It starts by instrumenting the original program at selected program locations (instrumentation phase). The instrumented code is then simulated, drawing out some profiled data, such as C_c, C_m (profiling phase). Once the profiling is done, all the parameters of this optimization problem are determined. The next work is to obtain the optimal CPU and memory frequencies through solving the above optimization problem (solution phase). Finally, the frequency-setting calls are inserted at the appropriate situations so that the selected region is executed at the appropriate frequency and the rest of the program is executed at the highest frequency (code generation phase). In the future work, we will extend our approach to deal with other program segments besides regular loop segments. The whole compilation framework is shown in Figure 2.

In the next experiments, we will analyze the impact of parameters on our optimization approach and then choose a set of array-dominated applications to validate our optimization approach.

5. Experiments

5.1. Simulation Platform

In our simulation, CPU and memory both are DVS-enabled. We build a simple energy model for this DVS-enabled CPU and memory in terms of formula (2). That is,

$$P_c = (7.72e-27)\cdot f_c^3 \quad (15) \qquad P_m = (1.09e-24)\cdot f_m^3 \quad (16)$$

The above formula (15) and (16) also refer to Transmeta's Crusoe TM5900 processor parameters [13]. The summary of power specifications for TM5900 CPU and DDR can be found in [15]. All the parameters and profiled data for this optimal problem are shown in Table 1.

We use SimpleScalar tool set [14] to profile some necessary parameters such as C_c and C_m. Modified SimpleScalar tool set models a 1 GHz 4-way issue dynamically-scheduled processor. This simulator models all aspects of the processor including the instruction fetch unit, the branch predictor, register renaming, the functional unit pipelines, and the reorder buffer. This modified SimpleScalar tool set enables software prefetching through adding a prefetch instruction to the ISA of the processor model. In addition, our simulator also models the memory system in detail. A split 8-Kbyte direct-mapped L1 cache with 32-byte cache blocks, and a unified 256-Kbyte 4-way set-associative L2 cache with 64-byte cache blocks are assumed. Such cache configuration can meet our input data sets.

5.2. Parameters Analysis

5.2.1. the Impact of E_{budget} on Optimization

We only consider scarce-energy settings, where $E_{budget} < E_{bound}$. To describe the degree of energy scarcity, we define α as the ratio of energy budget to energy bound.

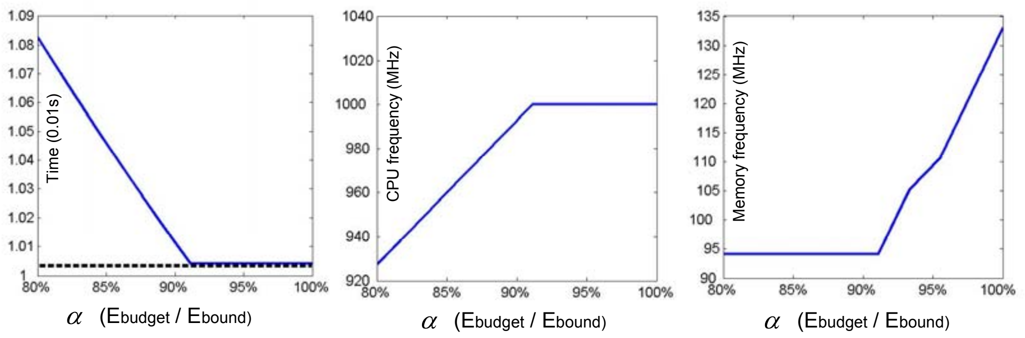
$$E_{budget} = \alpha \cdot E_{bound}$$

As α increases, energy budget is more approach to energy bound so that performance is more approach to the optimal value as Figure 3(a) and Figure 4(a) show, where blue solid lines represent execution time under different α values and black dot lines represent the optimal performance when energy bound is reached.

For CPU-bound case (Figure 3(a)), when $\alpha = 91.1\%$, performance has reached the optimal value instead of $\alpha = 100\%$. In contrast, for memory-bound case (Figure 4(a)), performance doesn't achieve the optimal level until $\alpha = 100\%$. This shows that in CPU-bound case, the optimal performance can be reached at less energy budget (less than energy bound). While in memory-bound case, the optimal performance must be reached with energy bound. In other words, in CPU-bound case, some energy can be saved without performance loss. While in memory-bound case, energy saving must cause performance loss.

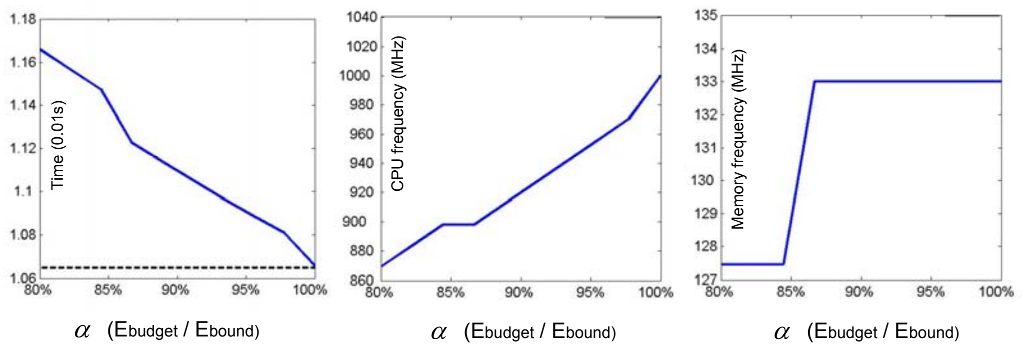
Figure 3(b)-(c) and Figure 4(b)-(c) show CPU and memory frequency variances under the same conditions. In CPU-bound case (Figure 3(b)-(c)), when $\alpha > 91.1\%$ CPU frequency keeps at 1GHz while memory frequency is climbing and it reaches the highest point until $\alpha = 100\%$. Due to objective (5), the performance value in CPU-bound is determined by CPU frequency. Therefore, when CPU frequency reaches the highest value, performance reaches the optimal value. After that, memory frequency increase consumes unnecessary energy, which explains why some energy savings can be obtained without performance loss in CPU-bound case.

In contrast, in memory-bound case, performance is determined by both CPU frequency and memory frequency in terms of objective (10). Only when both of them reach the highest values, the performance reaches the optimal.



(a) time (b) CPU frequency (c) memory frequency

Figure 3. The impact of E_{budget} on the optimization results when CPU-bound case. Cc=1000, Cm=90



(a) time (b) CPU frequency (c) memory frequency

Figure 4. The impact of E_{budget} on the optimization results when memory-bound case. Cc=1000, Cm=150

5.2.2. the Impact of C_c and C_m on Optimization

We also simulated the changes of time as C_c and C_m vary shown in Figure 5, where $\alpha = 90\%$. From point **Ai** to **Bi** (i=1, 2, 3, 4), the varying trends occur an exception. That is because the cases before **Ai** belong to memory-bound while the cases after **Bi** belong to CPU-bound and the calculation formulas for these two cases are different.

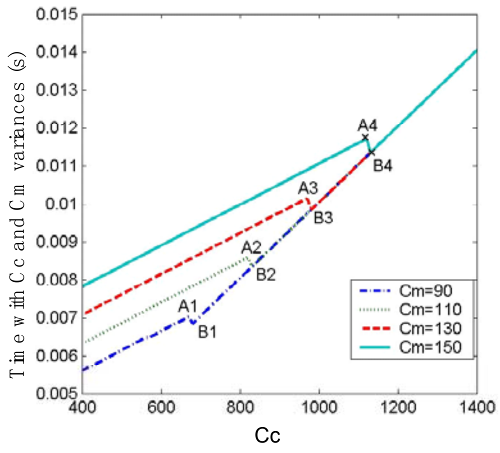


Figure 5. Time variances with Cc and Cm variances

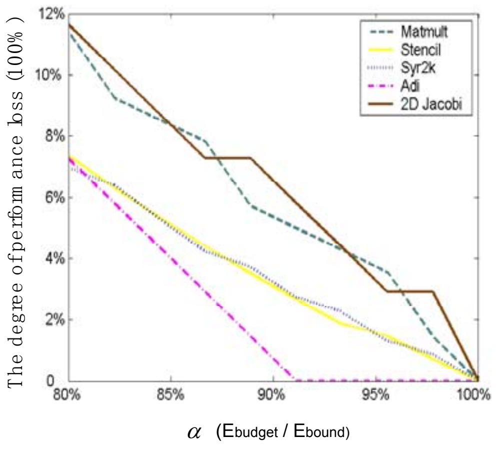


Figure 8. The degree of performance loss for each benchmark

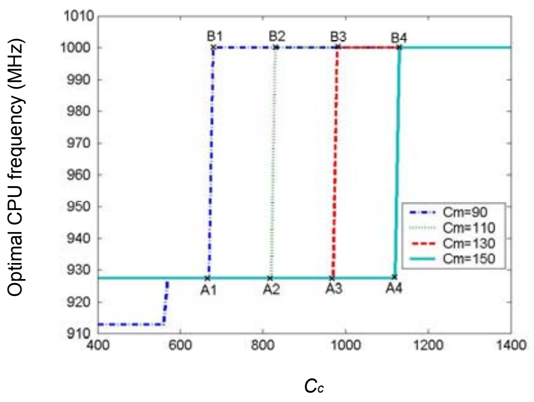


Figure 6. Optimal CPU frequency variances with C_c and C_m variances

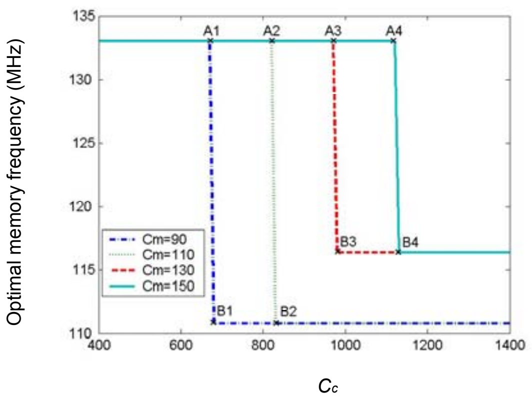


Figure 7. Optimal memory frequency variances with C_c and C_m variances

Figure 6 and Figure 7 give the optimal CPU and memory frequency settings as C_c and C_m vary. From these two graphs, the optimal CPU frequency value is non-decreasing as C_c increases while the optimal memory frequency value is non-increasing as C_c increases under a fixed C_m value. The C_c value and C_m value at points **Ai** and **Bi** (i=1, 2, 3, 4) are the same with those in Figure 5. Therefore, we can contrast Figure 6 and Figure 7 with Figure 5. Here $\alpha = 90\%$.

5.3. Experimental Results

After the detailed parameters analysis, we choose a set of array-dominated applications to test the effectiveness of our energy optimization approach. They include Matmult, Stencil, Adi, 2D Jacobi and Syr2k. These benchmarks descriptions are given in Table 2.

Benchmarks	description	The number of Array	Size
Matmult	matrices product	3	1024*1024
Adi	the core base benchmark of Livermore	6	1024*1024*3
2D Jacobi	2D Jacobi relaxation	2	1024*1024
Stencil	a stencil computing program for five dots	2	1024*1024
Syr2k	Rank-2K update computation program for solving zonal symmetry matrix from BLAS	3	1024*1024

Table 2. The description of benchmarks

Assume the time under energy constraint is *Time* and the time without energy constraint is *InitialTime*, the degree of performance loss can be calculated by formula (17).

$$Performance\ Loss\ (100\%) = \frac{\frac{1}{InitialTime} - \frac{1}{Time}}{\frac{1}{InitialTime}} \times 100\% \qquad (17)$$

memory-bound benchmarks	α	CPU Fre. (MHz)	Memory Fre.(MHz)	Perf. Loss
Matmult	82.2%	898	122	9.3%
	86.7%	927	122	7.8%
	91.1%	942	127	5.0%
	95.6%	971	127	3.6%
	100%	1000	133	0
Stencil	82.2%	782	133	6.3%
	86.7%	840	133	4.4%
	91.1%	898	133	2.7%
	95.6%	942	133	1.5%
	100%	1000	133	0
Syr2k	82.2%	811	133	6.4%
	86.7%	869	133	4.2%
	91.1%	913	133	2.7%
	95.6%	956	133	1.3%
	100%	1000	133	0

memory-bound benchmarks	α	CPU Fre. (MHz)	Memory Fre.(MHz)	Perf. Loss
Adi	82.2%	942	88.6	5.8%
	86.7%	971	88.6	2.9%
	91.1%	1000	88.6	0
	95.6%	1000	111	0
	100%	1000	133	0
2D Jacobi	82.2%	898	122	10.2%
	86.7%	927	122	7.3%
	91.1%	942	127	5.8%
	95.6%	971	127	2.9%
	100%	1000	133	0

Table 3. Optimal CPU and memory frequency settings and performance loss for each benchmark with variable

As α value varies, the performance loss for each benchmark is shown in Figure 8. Table 3 gives the detailed experimental data for the optimal CPU and memory frequency setting and performance loss percentage. In these five benchmarks, Matmult, Stencil and Syr2k belong to memory-bound cases and Adi and 2D Jacobi belong to CPU-bound cases. Experimental results validate our conclusions from the following aspects:

- Benchmark Adi reaches the optimal performance with 91.1% of energy bound, which validates that in CPU-bound case, the optimal performance can be reached under less energy budget (less than energy bound).
- For each benchmark belonging to memory-bound case, the optimal performance is only achieved with energy bound.
- The optimal CPU and memory frequency variances conform to Figure 3(b)-(c) and Figure 4(b)-(c).

6. Conclusions

Energy consumption is more and more important for battery-powered embedded systems due to the need for longer battery life and portability. Compiler-directed optimization techniques are more and more concerned. In our article, we provide an energy-constrained prefetching optimization approach, which obtains the optimal performance with energy constraint through adjusting CPU and memory frequencies. For CPU-bound case and memory-bound case, frequency scaling shows the different characteristics. It is necessary to adjust both of CPU and memory frequencies for the optimization problem. We analyze the impact of several parameters on our energy-constrained optimization problem and draw some conclusions. Finally, experimental results validate the effectiveness of our approach.

Reference

1. Gang Qu. What is the Limit of Energy Saving by Dynamic Voltage Scaling? In ICCAD. page 560-563. 2001.
2. H. Saputra, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. S. Hu, C-H. Hsu, U. Kremer. Energy-Conscious Compilation Based on Voltage Scaling. In LCTES'02-SCOPES'02, June 19-21, 2002, Berlin, Germany.
3. Fen Xie, Margaret Martonosi and Sharad Malik. Compile-Time Dynamic Voltage Scaling Settings: Opportunities and Limits. In PLDI'03, June 9-11, 2003, San Diego, California, USA.
4. Woo-Cheol Kwon and Taewhan Kim. Optimal Voltage Allocation Techniques for Dynamically Variable Voltage Processors. In DAC'03, June 2-6, 2003, Anaheim, California, USA.
5. Xiaobo Fan, Carla S. Ellis and Alvin R. Lebeck. The Synergy between Power-aware Memory Systems and Processor Voltage Scaling. In *Proceedings of the Workshop on Power-Aware Computer Systems (PACS'03)*, Dec 2003.
6. Todd C. Mowry. Tolerating Latency Through Software-Controlled Data Prefetching. Ph.D. thesis, Stanford University, Computer System Laboratory, March 1994.
7. Shimin Chen, Phillip B. Gibbons and Todd C. Mowry. Improving Index Performance through Prefetching. In *Proceedings of the 2001 SIGMOD International Conference on Management of Data*. Page 235-246. May 2001.
8. Abdel-Hameed Badawy, Aneesh Aggarwal, Donald Yeung, and Chau-Wen Tseng. The Efficacy of Software Prefetching and Locality Optimizations on Future Memory Systems. Journal of Instruction-Level Parallelism. June 2004.
9. Ricardo Bianchini and Beng-Hong Lim. Evaluating the Performance of Multithreading and Prefetching in Multiprocessors. Journal of Parallel and Distributed Computing (JPDC), special issue on Multithreading for Multiprocessors, August 1996.
10. C. Hsu, U. Kremer, and M. Hsiao. Compiler-Directed Dynamic Voltage/Frequency Scheduling for Energy Reduction in Microprocessors. In *International Symp. On Low Power Electronics and Design (ISLPED)*, pages 275-278, August 2001.
11. Chung-Hsing Hsu. Compiler-Directed Dynamic Voltage and Frequency Scaling for CPU Power and Energy Reduction. Ph. D. Dissertation. New Brunswick, New Jersey. October 2003.
12. J. Pouwelse, K. Langendoen, and H. Sips. Dynamic Voltage Scaling on a Low-Power Microprocessor. In *the Seventh Annual International Conference on Mobile Computing and Networking 2001*, pages 251-259, 2001.
13. Crusoe Processor Model TM5700/TM5900 Data Book. http://www.transmeta.com/crusoe_docs/tm5900_databook_040204.pdf
14. D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. CS TR 1342, University of Wisconsin-Madison, June 1997.
15. Juan Chen, Yong Dong, Xue-jun Yang. Energy Optimization for Software Prefetching in Embedded Applications. *In the proceedings of Asia and South Pacific International Conference on Embedded SoCs (ASPICES 2005)*. July 5-8, 2005, Bangalore, India.