Realization of Power Aware Software Prefetching as a Multi-Objective Optimization Problem

Sumanta Pyne, Krishanu Ray
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Kharagpur, India
{sumantapyne, roykrisanu}@gmail.com

Ajit Pal
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Kharagpur, India
apal@cse.iitkgp.ernet.in

***Abstract*—Software prefetching is a performance-oriented optimization technique, which is generally used to reduce the gap between processor speed and memory access speed. When software prefetching is applied to memory-intensive benchmark programs, the performance improves with higher power consumption. The present work provides a mechanism to transform a program with software prefetching to its power-aware equivalent. This is done by executing the software prefetching program at different voltage-frequency pairs. Besides reducing the power, the performance has been improved by adjusting the prefetch distance. XEEMU-Panalyzer simulator is used to evaluate the present work. Experimental results of the proposed scheme guarantees that performance improvement of software prefetching program is possible at the cost of less power consumption. The proposed work can enable a compiler to generate power aware software prefetching program.**

Keywords-software prefetching, prefetch distance, power-aware, multi-objective optimization problem, voltage-frequency pair, SPP to PASPP transformation algorithm.

I. INTRODUCTION

Power/Energy saving is vital issue not only for embedded and portable devices, but also for servers, personal, mainframe and super computers. This has lead to the design of compilers for power aware code generation [1]. It has been observed that some code optimization techniques can lead to performance gain as well as saving of energy. But it is not true for software prefetching. Software prefetching [6, 2] is a technique of inserting prefetch instructions into codes for memory reference that are likely to miss in the cache. This is either done by the programmer or a compiler. At runtime the inserted prefetch instructions bring the data into the processor's cache in advance of its use, thus overlapping the memory access with processor computation. Software prefetching eliminates cache misses causing improvement in performance. However, it increases power consumption. This leads to a problem of power-performance tradeoff for software prefetching programs. Fig.1 shows a C-program of 3D Jacobi's Kernel, while Fig.2 shows its software prefetching version. In [3] Agarwal et al have proposed the idea of low power software prefetching using *Dynamic Frequency and Voltage Scaling(DVFS)*[5], without degrading the performance. While Chen et al in [4] shows *DVFS* with adjustment of prefetch distance can provide power reduction as well as performance improvement.

```
#define n 100
double A[n][n][n],B[n][n][n];
int main()
  {
    int i,j,k;
    for(k=0;k<n-2;k++)
      {  for(j=0;j<n-2;j++)
           {
               for(i=0;i<n-2;i++)
                   {
                     B[k][j][i]=0.167*(A[k][j][i-1]
                     +A[k][j-1][i]+A[k][j][i+1]
                     +A[k][j+1][i]+A[k-1][j][i]
                     +A[k+1][j][i]);
                   }
           }
      }
    return 0;
  }
```

Figure 1. 3D Jacobi's Kernel

The present work formulates the problem of power reduction with performance gain as a multi-objective optimization problem. The solution of this optimization problem will guide the software prefetching program to achieve higher performance at the cost of minimum power consumption. Fig.3 shows the general structure of a *software prefetching program (SPP)*. The *SPP* consists of a *software prefetch code (SPC)*. An SPC nested in one or more loops form an *SPP*. *SPC* has three sections – *Prologue Loop Section (PLS), Unrolled Loop Section (ULS), Epilogue Loop Section (ELS)*. *PLS* has *p* prologue loops *(PL_1, PL_2 ,..., Pl_p)*, *ULS* has *p* unrolled loops *(UL_1, UL_2, ..., UL_p)*, *ELS* has *p* epilogue loops *(EL_1, EL_2, ..., EL_p)*, where $p \geq 1$. A prologue loop *(PL)* contains only data prefetching instructions. An unrolled loop *(UL)* contains both data prefetching and computation instructions to process the prefetched data. An epilogue loop *(EL)* contains only computation instructions to process prefetched data. Here, for each *j*, PL_j, UL_j and EL_j are related to a prefetch distance PD_j where $1 \leq j \leq p$. This implies *p* prefetch distances are associated with an *SPC*. *Prefetch distance (PD)* directs a *PL* to prefetch *PD* loop iterations before the data is referenced. The present work measures the *PD* using the formula *PD = ceiling (l/s)* as defined in [12, 13], where *l* is the memory access latency and *s* is the latency of one *UL* iteration.

The present work considers a processor that run with *m* voltage-frequency *(v, f)* pairs. A voltage-frequency pair is considered as (v_i, f_i) where $1 \leq i \leq m$. (v_1, f_1) is the peak *(v, f)* pair, and (v_m, f_m) is the lowest *(v, f)* pair. Fig.4 shows the

```
#include"dvfs.h"
#define n 100
#define PD 16
double A[n][n][n],B[n][n][n];
int main()
{
  int i,j,k;
  for(k=0;k≤ n-1;k++)
  {
    for(j=0;j≤ n-1;j++)
    {       //SPC begins
     for(i=0;i<PD;i+=4)  //Prologue Loop(PL1)
      {
       PREFETCH(&A[k][j][i]);
       PREFETCH(&A[k][j+1][i]);
       PREFETCH(&A[k][j-1][i]);
       PREFETCH(&A[k-1][j][i]);
       PREFETCH(&A[k+1][j][i]);
       PREFETCH(&B[k][j][i]);
      }
      for(i=0;i<n-PD-2;i+=4)   //Unrolled Loop(UL1)
      {
       PREFETCH(&A[k][j][i+4+PD]);
       PREFETCH(&A[k][j+1][i+4+PD]);
       PREFETCH(&A[k][j-1][i+4+PD]);
       PREFETCH(&A[k-1][j][i+4+PD]);
       PREFETCH(&A[k+1][j][i+4+PD]);
       PREFETCH(&B[k][j][i+4+PD]);
       B[k][j][i]=0.167*(A[k][j][i-1]+A[k][j-1][i]+A[k][j][i+1]+
      A[k][j+1][i]+A[k-1][j][i]+A[k+1][j][i]);
      B[k][j][i+1]=0.167*(A[k][j][i]+A[k][j-1][i+1]+A[k][j][i+2]+
       A[k][j+1][i+1]+A[k-1][j][i+1]+A[k+1][j][i+1]);
       B[k][j][i+2]=0.167*(A[k][j][i+1]+A[k][j-1][i+2]+A[k][j][i+3]+
      A[k][j+1][i+2]+A[k-1][j][i+2]+A[k+1][j][i+2]);
      B[k][j][i+3]=0.167*(A[k][j][i+2]+A[k][j-1][i+3]+A[k][j][i+4]+
       A[k][j+1][i+3]+A[k-1][j][i+3]+A[k+1][j][i+3]);
      }
      for(i=n-PD-1;i<n-2;i++) //Epilogue Loop(EL1)
       {
        B[k][j][i]=0.167*(A[k][j][i-1]+A[k][j-1][i]+A[k][j][i+1]+
        A[k][j+1][i]+A[k-1][j][i]+A[k+1][j][i]);
       }
          //SPC ends
    }
  }
  return 0;
}
```

Figure 2. 3D Jacobi's Kernel with software prefetching

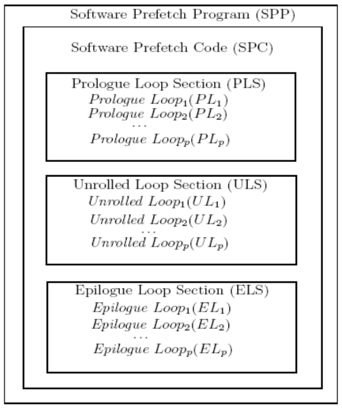

Figure 3. General structure of Software Prefetching Program(SPP)

general structure of Power *Aware Software Prefetching Program (PASPP)* which contains an extra block in its *SPC*

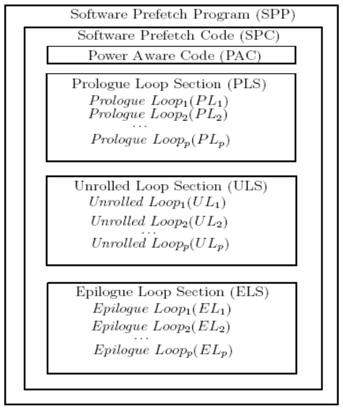

Figure 4. General structure of Power Aware Software Prefetching Program(PASPP)

called *Power Aware Code (PAC)*. The goal of the present work is to transform a given *SPP* (as in Fig.2) to a *PASPP* (as in Fig.5), so that it can take full advantage of the processor under consideration in terms of performance improvement and power awareness.

Software prefetching increases average power consumption of *SPP*. Increase in average power consumption also increases heat dissipation. Heat dissipation reduces system reliability and increases leakage power consumption. The average power dissipation is important in context of battery operated portable devices as it decides the battery life time. The reduction of average power will result longer battery life time. The power-delay product (energy consumption) of *SPP* is lesser than that of non-prefetched version (original program). The present work targets in reduction of average power consumption of *PASPP*. The performance of *PASPP* is much better than that of the original program and is at most as good as that of *SPP*. The power-delay product of the *PASPP* is lesser than those of the original program and *SPP*.

The proposed scheme is evaluated with XEEMU [10, 11]. XEEMU is a power simulator that simulates Xscale processor. It is an extension of Simple Scalar computer architecture simulator and uses Panalyzer as its power model. Xscale processor supports data prefetching. Xscale processor can operate at nine *(v, f)* pairs as shown in TABLE - I. So in the present work value of *m* is 9. For a given program XEEMU measures the time taken by the program in seconds as well as in cycles. It measures total energy consumption both in Watt × Cycle and in Joule. It measures average power dissipation in Watts.

TABLE I.

i	v_i(Volt)	f_i(MHz)
1	1.5	733
2	1.4	666
3	1.3	600
4	1.2	533
5	1.1	466
6	1.1	400
7	1.0	333
8	1.0	266
9	1.0	200

II. Present Work

The present work formulates the problem of transforming *SPP* to *PASPP*. Given a processor with *m (v, f)* pairs and an *SPP*, generate a *PASPP* using the solution of the following optimization problem represented by Formula 1.

Formula 1: Multi-Objective Optimization Problem (MOOP)
Goal 1: Minimize energy (E) consumed by the PASPP.
Goal 2: Minimize time (T) taken by the PASPP.
subject to

$$E = \sum_{i=1}^{m} e_i x_i \le \min(E_{prefetch}, E_{no\_prefetch}),$$

$$\sum_{i=1}^{m} t_i x_i \le T, \sum_{i=1}^{m} x_i = N, and, x_i \ge 0.$$

where *E*, is the total energy consumed by *PASPP* in Watts*Cycles. e_i and t_i are energy (in Watts*Cycles) consumed and time (in µsec) taken, respectively, per execution of *SPC*, when executed at (v_i, f_i). x_i is the number of times the *SPC* is executed at (v_i, f_i). *T* is the total time taken by *PASPP* in µsec. $T_{prefetch} \le T < T_{no\_prefetch}$. $T_{prefetch}$ and $T_{no\_prefetch}$ are time taken to execute the *SPP* and its non-prefetched version, respectively, when executed at (v_1, f_1). Their unit is µsec. *N* is the total number of times the *SPC* is executed. *N* is a function of input size *n*. For example, the *SPP* of 3D Jacobi's Kernel has $N=n^2$. $E_{prefetch}$ and $E_{no\_prefetch}$ are energy (in Watts*Cycles) consumed by *SPP* and its non-prefetched version, when executed at (v_1, f_1). The solution of this problem will find the value of $x_1, ..., x_m$. The value of x_i indicates that the *PASPP* will spend $(x_i/N)*100\%$ of its execution time at (v_i, f_i). The solution of this formula enables the *PASPP* to adjust the *(v, f)* pair while in execution to achieve performance gain at the cost of minimum energy consumption. The following steps can help a compiler to achieve this.

1. Consider a program without software prefetching as shown in Fig.1 and transform it to an *SPP* as shown in Fig.2.

2. Find *N* from the *SPP* obtained in the previous step.

3. Find e_i, t_i, PD_{ji} for each (v_i, f_i) and store them in *TEPD_TABLE*. PD_{ji} is the prefetch distance associated with PL_j, UL_j, EL_j in the *SPC* of *SPP*, when executed at (v_i, f_i). *TEPD_TABLE* is a table having *m* records, each record having the following attributes — *t, e,* and an array of *p* elements named *pd*, they store e_i, t_i, and PD_{ji}, respectively.

4. Find $T_{prefetch}$, $E_{prefetch}$, $T_{no\_prefetch}$ and $E_{no\_prefetch}$ by executing the *SPP* and its non-prefetched version, respectively, at (v_1, f_1).

5. Run *SPP to PASPP Transformation Algorithm.*

The rest of this section will discuss each of these steps in details.

```
#include"dvfs.h"
#define n 100

double A[n][n][n], B[n][n][n];
int main()
{
  int i, j, k, count=0, PD=16;
  for(k=0;k≤ n-1;k++)
   {
     for(j=0;j≤ n-1;j++)
      {      //SPC begins
        if(count==1690) //PowerAwareCode(PAC)
        {  setfrequency(2);
           setvoltage(2);
           PD=12;
        }
        count++;
       for(i=0;i<PD;i+=4)  //Prologue Loop(PL1)
        {
         PREFETCH(&A[k][j][i]);
         PREFETCH(&A[k][j+1][i]);
         PREFETCH(&A[k][j-1][i]);
         PREFETCH(&A[k-1][j][i]);
         PREFETCH(&A[k+1][j][i]);
         PREFETCH(&B[k][j][i]);
        }

        for(i=0;i<n-PD-2;i+=4)       //Unrolled Loop(UL1)
        {
          PREFETCH(&A[k][j][i+4+PD]);
          PREFETCH(&A[k][j+1][i+4+PD]);
          PREFETCH(&A[k][j-1][i+4+PD]);
          PREFETCH(&A[k-1][j][i+4+PD]);
          PREFETCH(&A[k+1][j][i+4+PD]);
          PREFETCH(&B[k][j][i+4+PD]);
          B[k][j][i]=0.167*(A[k][j][i-1]+A[k][j-1][i]+A[k][j][i+1]+
         A[k][j+1][i]+A[k-1][j][i]+A[k+1][j][i]);
         B[k][j][i+1]=0.167*(A[k][j][i]+A[k][j-1][i+1]+A[k][j][i+2]+
          A[k][j+1][i+1]+A[k-1][j][i+1]+A[k+1][j][i+1]);
          B[k][j][i+2]=0.167*(A[k][j][i+1]+A[k][j-1][i+2]+A[k][j][i+3]+
         A[k][j+1][i+2]+A[k-1][j][i+2]+A[k+1][j][i+2]);
         B[k][j][i+3]=0.167*(A[k][j][i+2]+A[k][j-1][i+3]+A[k][j][i+4]+
           A[k][j+1][i+3]+A[k-1][j][i+3]+A[k+1][j][i+3]);
        }
        for(i=n-PD-1;i<n-2;i++) //Epilogue Loop(EL1)
         {

          B[k][j][i]=0.167*(A[k][j][i-1]+A[k][j-1][i]+A[k][j][i+1]+
           A[k][j+1][i]+A[k-1][j][i]+A[k+1][j][i]);
         }

            //SPC ends
      }
   }
  setvoltage(1);
  setfrequency(1);
 return 0;
}
```

Figure 5. 3D Jacobi's Kernel with power aware software prefetching

A. A Program and its SPP version

Consider a source program and find out the opportunities of having an *SPP* version of it. This can be done either by the programmer or by a compiler. To do this, the present work considers the algorithm by Mowry et al in [12], which takes *O(p)* time, where *p* is the number of *PLs*, *ULs* and *ELs* in the *SPC* of the *SPP*, or in other words it is the number of prefetch distances associated with the *SPC*.

B. Finding N from SPP

The *SPC* in the *SPP* obtained from the source program. An *SPC* nested in one or more loops form an *SPP*. The nested loop helps to find *N*. When an *SPC* is not nested in any loop, *N* is considered as 1. This takes *O(k)* time where *k* (≥ 1) is the nesting level of the nested loop which contains the *SPC*. In Fig.2, *N* is n^2 and *k* is 2.

C. Formation of TEPD_TABLE

The following code fragment enables the formation of *TEPD_TABLE* in *O(p)* time, because *m* is constant for a given processor.

```
for (i = 1; i ≤ m; i++)
 {
  for (j = 1; j ≤ p; j++)
   {
    Execute SPC at (v_i, f_i) for one iteration and store its execution time in s;
    PD_ji=ceiling(l/s);
    TEPD_TABLE[i].pd[j]=PD_ji;
   }
  Execute SPC at (v_i, f_i) for once and store the execution time and energy
  consumed in t_i and e_i respectively;
      TEPD_TABLE[i].t= t_i;
      TEPD_TABLE[i].e= e_i;
 }
```

Where, *m* is the number of *(v, f)* pairs and *p* is the number of prefetch distances associated with the *SPC*. *TEPD_TABLE[i].t* and *TEPD_TABLE[i].e* represents the *t (time)* and *e (energy)* attributes of the i^{th} record of *TEPD_TABLE*, respectively. *TEPD_TABLE[i].pd[j]* is the j^{th} *PD* of the i^{th} record of *TEPD_TABLE*, i.e. *TEPD_TABLE[i].pd[j]* stores PD_{ji}.

D. Finding $T_{prefetch}$, $E_{prefetch}$, $T_{no\_prefetch}$ and $E_{no\_prefetch}$

Execute the *SPP* at (v_1, f_1) to obtain $T_{prefetch}$ and $E_{prefetch}$. Execute the non prefetch version of *SPP* at (v_1, f_1) to obtain $T_{no\_prefetch}$ and $E_{no\_prefetch}$.

E. SPP to PASPP Transformation Algorithm

This algorithm converts an *SPP* to a *PASPP*. The algorithm starts with $T_{prefetch}$ as initial value of *T* and increase the value of *T* by 1% of $T_{prefetch}$ until an optimal solution is found. On finding an optimal solution, the values of $x_1, ..., x_m$ are stored in *X[1], ..., X[m]* respectively. Then the *PAC* is generated.

1) SPP to PASPP Transformation Algorithm(SPP-PASPP)

```
SPP to PASPP Transformation Algorithm
Input: T_no_prefetch, T_prefetch, TEPD_TABLE, N and SPP.
Output: PASPP.
Initialization:
 1. for (i = 1; i ≤ m; i++)
        X[i] = 0;
 2. T = T_prefetch ;
Algorithm
Step 1. If(T≥ T_no_prefetch) then
          {
            Report "Failure" and goto Step 5;
          }
Step 2. Solve the MOOP defined in Formula 1 by Goal Programming
          using the information in TEPD_TABLE;
Step 3. If the MOOP has an optimal solution then
          {
            for (i = 1; i ≤ m; i++)
                     X[i] = x_i;
          }
          Otherwise
          {
            T=T + 0.01*T_prefetch ;
            goto Step 1;
          }
Step 4. Call Procedure Power_Aware_Code_Generator(X);
Step 5. Stop;
```

SPP-PASPP finds the least possible value of *T* such that, $T_{prefetch} \leq T < T_{no\_prefetch}$, as defined in *Formula 1*.

SPP-PASPP solves the *MOOP* in *Formula 1* using *Goal Programming* [15], where *Goal 1* has higher priority than *Goal 2*. A *Goal Programming Problem* can be reduced to a *Linear Programming Problem* and solved by using *Simplex Method* [14]. In the worst case, time taken by *Simplex Method* is an exponential function of *m*. For a given processor *m* is always constant. In the present work the value of *m* is 9, and it remains same for any input. So, the time taken to solve the optimization problems is *O(1)*. Steps 1-3 takes $O(floor(((T_{no\_prefetch}-T_{prefetch})/T_{prefetch})*10^2))$ time and in Step 4 *Power Aware Code Generator(PACG)* takes *O(p)* time. So, total effort required by the algorithm is $O(floor(((T_{no\_prefetch}-T_{prefetch})/T_{prefetch})*10^2)+p)$.

2) Power Aware Code Generator (PACG)

```
Power_Aware_Code_Generator(X)
{
 char S[100]; int xcounter=X[1]; boolean first = true;
  for (i = 1; i ≤ m; i++)
  {
   if(X[i]>0)
    {
       if(first==false)
        {
          sprintf(S, "else");
          Insert_Code(S);
        }
       else
        {
          first = false;
        }
      sprintf(S,"if(count==%d){setvoltage(%d);setfrequency(%d);",
      xcounter, i, i);
     Insert_Code(S);
     if(p==1)
      {
       sprintf(S, "PD=%d;", TEPD_TABLE[i].pd[1]);
       Insert_Code(S);
      }
     else
      {
       for (j = 1; j ≤ p; j++)
        {
         sprintf(S, "PD%d=%d;", j, TEPD_TABLE[i].pd[j]);
         Insert_Code(S);
        }
      }
     sprintf(S, "}");
     Insert_Code(S);
    }        // end of if(X[i]>0)
   xcounter+=X[i];
  }          //end of for (i = 1; i ≤ m; i++)
 sprintf(S, "count++");
 Insert_Code(S);
} // end of PACG
```

Power Aware Code Generator (PACG) is a procedure that inserts *Power Aware Code (PAC)* in the *SPP* to form the *PASPP*. After the least possible value of *T* is obtained, *SPP-PASPP* calls this procedure. *PACG* has a parameter *X* which contains the solution of the optimization problem solved before the algorithm reaches Step 4. *PACG* uses the C library function *sprintf*. *Insert_Code* is another procedure that enables *PACG* to insert the desired code in the *PAC* block of the *PASPP*. *PACG* takes *O(p)* time because *m* is constant for a given processor. Fig.5 shows the *PASPP* of JACOBI which contains an integer variable *count* initialized to *zero* and a *PAC* containing *if* statement. The *PACG* inserts an *if-else* ladder followed by a statement *count++*. The *if-else* ladder and *count++* statement collectively forms the *PAC*. The *count* variable counts the number of time the *SPC* is executed. The *if-else* ladder helps the *PASPP* to compare the *count* with the number of times the *PASPP* should be executed at a *(v, f)* pair and switch to the desired *(v, f)* pair with change in prefetch distance.

III. Experimental Methodology And Results

The present work is simulated on an architectural simulator named XEEMU. XEEMU extends SimpleScalar with functionalities of Xscale processor and Panalyzer power model. Xscale supports software prefetching and work on multiple *(v, f)* pairs. The performance and power of the non-prefetched version, *SPP*, *PASPP* are measured with the help of this simulator. The information in the *TEPD_TABLE* is also obtained using XEEMU.

A. Experimental Methodology

As discussed in section II, the parameters $T_{no\_prefetch}$, $T_{prefetch}$, $E_{no\_prefetch}$, $E_{prefetch}$, PD_{ji}, e_i, t_i are measured on XEEMU. The present work implements three high level functions ─ *setvoltage, setfrequency,* and *PREFETCH*. Each of these functions are implemented using inline assembly facility available in C programming. *setvoltage(i)* will set the supply voltage to v_i Volts, *setfrequency(i)* will set the clock frequency to f_i MHz, and *PREFETCH(data_address)* will fetch a data block to the L1 data cache. The present work assumes a split 4 Kbyte 8-way set-associative L1 cache with 32 byte cache blocks, and a unified 128 Kbyte 4-way set-associative L2 cache with 64 byte cache blocks. The memory access overhead *l* at peak *(v,f)* pair (v_1,f_1) (as in TABLE-I) is 170ns. As the *(v, f)* pair is scaled down this delay increases [7]. To fix this memory access time overhead at lower *(v, f)* pairs, the prefetch distance is adjusted. During the switching from a *(v, f)* pair to another, there are time and energy overhead. To measure this, the present work uses the mathematical model proposed by Burd et al in [9].

The simulation is based on experimental evaluation that employs six benchmarks, representing two classes of data intensive applications. TABLE-II lists the benchmarks along with their problem sizes and memory access patterns. JACOBI performs a 3D Jacobi relaxation. MM represents matrix multiplication. DET finds the determinant of a square matrix. MOLDYN [8] performs non-bonded force calculation for key molecular dynamic applications. IP represents inner product of two vectors. Very Long Integer Addition (VLIA) adds two integers of very long size. Each integer is stored in a dynamic array, where each array element represents a place-value. The sum is also stored in a dynamic array.

TABLE II.

Benchmark	Input	n	N
JACOBI	one n×n×n 3D matrix(affined array) of 8 byte real numbers	10^2	n^2
MM	two n×n 2D matrices(affined array) of 8 byte real numbers	2×10^2	n^2
DET	one n×n 2D matrices(affined array) of 8 byte real numbers	2×10^2	n^2
MOLDYN	three indexed arrays of size n, and time = 10000	128K	Time
IP	two arrays(affined arrays) of n 8 byte real numbers	10^5	1
VLIA	two dynamic arrays (affined arrays) of n integers	10^3	1

B. Experimental Results

The approach described in section II is followed. The performance and power of the optimized codes are measured with the help of the simulator. All programs are built with Gcc-O2 option. TABLE-III shows *TEPD_TABLE* for JACOBI. TABLE-IV shows performance and power (average power) comparison of different approaches. The third column from left shows the power and time taken by

TABLE III.

i	t_i(µsec)	e_i(Watt×Cycle)	Pd
1	396.27	301414	16
2	433.00	250785	12
3	477.55	213549	11
4	532.45	176665	9
5	60.674	145748	7
6	699.55	136370	5
7	834.89	116155	3
8	1037.73	108874	2
9	1372.25	101509	1

TABLE IV.

Benchmark	Items	Original	SPP	PASPP
JACOBI	Power(W)	4.3	7.2	4.18
	Time(sec)	7.54	3.5	4.28
MM	Power(W)	4.7	10.2	4.57
	Time(sec)	8.25	3.23	3.72
DET	Power(W)	5.1	11.3	4.92
	Time(sec)	6.23	2.84	3.32
MOLDYN	Power(W)	4.73	7.54	4.69
	Time(sec)	10.28	2.76	3.57
IP	Power(W)	3.57	8.12	3.53
	Time(sec)	2.57	0.95	1.47
VLIA	Power(W)	4.2	7.82	4.17
	Time(sec)	1.53	0.657	0.884

the original non-prefetched version of the benchmarks. The next column shows the outcome of *SPP* where performance is enhanced at the cost of higher power dissipation. The power dissipation of *SPP* increases due to increase in the number of instructions, and overlapped memory access and CPU computation. The rightmost column shows the power and performance of the *PASPP*. The *PASPP* is based on MOOP. Fig.5 shows the *PASPP* version of JACOBI. *SPP-PASPP* gives higher priority to energy minimization. For this reason *PASPP* programs perform well at the cost of lesser power consumption. Fig.6 shows the power consumed by different units of the processor all three versions of JACOBI. The power consuming units of the system shown here are register renaming (rename), branch prediction unit (bpred), instruction window (window), load-store queue (lsq), register file (regfile), instruction cache (icache), L1 data cache (dcache), L2 data cache (dcache2), ALU (alu), output bus (resultbus), write buffer (write_buffer). TABLE-V shows an average of 60.34% performance and 21.56% energy gained by SPP with respect to that of the original program. TABLE-VI shows an average of 49.18% performance and 50.19% energy gained by *PASPP* with respect to that of the original program. TABLE-VII shows an average of 28.82% performance lost and 34.50% energy gained by *PASPP* with respect to that of *SPP*. TABLE-VIII shows power and time overhead due to the *PAC* and switching of *(v, f)* pairs. TABLE-IX shows time spent by the *PASPPs* at different *(v, f)* pairs with different prefetch distances.

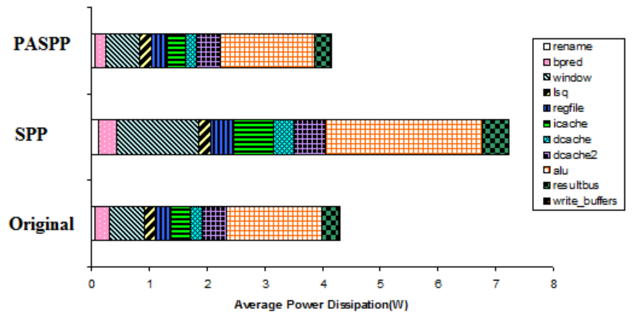

Figure 6. Detailed power dissipation at different units for three versions of 3D Jacobi's Kernel

TABLE V.

Benchmark	Performance gain by SPP with respect to the original (in %)	Energy gain by SPP with respect to the original (in %)
JACOBI	53.58	22.27
MM	60.84	15.02
DET	54.41	-1.00
MOLDYN	73.15	57.2
IP	63.03	15.9
VLIA	57.05	20.0
Average	**60.34**	**21.56**

TABLE VI.

Benchmark	Performance gain by PASPP with respect to the original (in %)	Energy gain by PASPP with respect to the original (in %)
JACOBI	43.23	44.82
MM	54.90	56.15
DET	46.70	48.59
MOLDYN	65.27	65.56
IP	42.80	43.44
VLIA	42.22	42.63
Average	**49.18**	**50.19**

TABLE VII.

Benchmark	Performance loss by PASPP with respect to the SPP (in %)	Energy gain by PASPP with respect to the SPP (in %)
JACOBI	22.28	29.00
MM	15.17	48.40
DET	16.90	49.10
MOLDYN	29.34	19.54
IP	54.73	32.73
VLIA	34.55	28.24
Average	**28.82**	**34.50**

TABLE VIII.

Benchmark	Items	PASPP
JACOBI	Power(W)	0.119
	Time(sec)	0.192
MM	Power(W)	0.267
	Time(sec)	0.202
DET	Power(W)	0.228
	Time(sec)	0.183
MOLDYN	Power(W)	0.198
	Time(sec)	0.194
IP	Power(W)	0.079
	Time(sec)	0.037
VLIA	Power(W)	0.067
	Time(sec)	0.041

TABLE IX.

Benchmark	Percentage of execution time spent by PASPP at different *(v, f)* and PD
JACOBI	16.9% at (v_1, f_1), PD = 16 83.1% at (v_2, f_2), PD = 12
MM	2.0% at (v_1, f_1), PD = 32 84.5% at (v_2, f_2), PD = 26 13.5% at (v_3, f_3), PD = 18
DET	5.15% at (v_1, f_1), PD = 40 67.23% at (v_2, f_2), PD = 34 27.62% at (v_3, f_3), PD = 29
MOLDYN	20.17% at (v_1, f_1), $PD_1 = 2, PD_2 = 3, PD_3 = 3$ 79.83% at (v_3, f_3), $PD_1 = 2, PD_2 = 3, PD_3 = 3$
IP	100.0% at (v_3, f_3), PD = 24
VLIA	100.0% at (v_3, f_3), PD = 9

IV. Related Works

In [3] Deepak et al showed energy saving of *SPP* without performance loss using DVFS. Chen et al introduced the idea of energy saving of *SPP* with performance improvement in [4] by means of DVFS with prefetch distance adjustment. The present work represents the problem of minimizing power dissipation of *SPP* with performance gain as a MOOP, as discussed in section II. This helps to automate the concept of transforming a non-prefetch source code to an *SPP* and then finally to a *PASPP*. A compiler can use the methods described in the present work to optimize *SPPs* that will execute on architectures having multiple *(v, f)* pairs.

V. Conclusions

The present work provides an idea of transforming *SPP* to *PASPP*. The experimental results show the *PASPPs* can perform well at the cost of lesser power dissipation. The proposed methods can enable a compiler to generate *PASPP*. In future the authors would like to extend the present work in the following directions — *(i)* power/performance tuning at programmer's will, *(ii)* reduction of prefetch instructions, and *(iii) SPP* to *PASPP* transformation that will be independent of input size.

Acknowledgment

The authors of the present work are grateful to Professor Zoltàn Herczeg, Department of Software Engineering, University of Szeged, Hungary for his cooperation during the installation of XEEMU.

References

[1] Vivek Tewari, Sharad Malik and Andrew Wolfe, "Compilation Techniques for Low Energy", in the proceedings of 1994 Symposium on Low-Power Electronics, San Diego, CA, October 1994.

[2] Todd C. Mowry, "Tolerating Latency through Software-Controlled Data Prefetching", Doctor dissertation, Standford University, March 1994.

[3] Deepak N. Agarwal, Sumitkumar N. Pamnani, Gang Qu, and Donald Yeung, "Transferring Performance Gain from Software Prefetching to Energy Reduction", in proceedings of the 2004 International Symposium on Circuits and Systems (ISCAS2004), Vancouver, Canada.

[4] Juan Chen, Yong Dong, Huizhan Yi, and Xuejun Yang, "Power-Aware Software Prefetching", ICESS 2007, LNCS 4523, pp. 207–218.

[5] Fen Xie, Margaret Martonosi and Sharad Malik, "Intraprogram Dynamic Voltage Scaling: Bounding Opportunities with Analytic Modeling", ACM Transactions on Architecture and Code Optimization,Vol.1, No.3, September 2004. pp 323-367.

[6] D. Callahan, K. Kennedy, and A. Porterfield, "Software Prefetching" , in the proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, CA, April 1991.

[7] Sakurai T. and Newton A, "Alpha-power model, and its application to CMOS inverter delay and other formulas", IEEE Journal Solid-State Circ. Vol. 25, 1990. pp. 584-594.

[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. Vol 6, 2004.

[9] T. Burd and R. Brodersen, "Design issues for dynamic voltage scaling", in the proceedings of International Symposium on Low Power Electronics and Design (ISLPED-00), June 2000.

[10] Zoltàn Herczeg, Akos Kiss, Daniel Schmidt, Norbert Wehn and Tibor Gyimothy, "XEEMU: An Improved XScale Power Simulator", PATMOS 2007, LNCS 4644, pp. 300–309.

[11] Zoltan Herczeg, Akos Kiss, Daniel Schmidt, Norbert Wehn and Tibor Gyimothy, "Energy simulation of embedded XScale systems with XEEMU", Journal of Embedded Computing - PATMOS 2007 selected papers on low power electronics archive, Volume 3 Issue 3, August 2009.

[12] Mowry, T.C., Lam, S. and Gupta, A, "Design and Evaluation of a Compiler Algorithm for Prefetching" , Proc. Fifth International Conf. on Architectural Support for Programming Languages and Operating Systems, Boston, MA, Sept. 1992, pp. 62-73.

[13] Klaiber, A.C. and Levy, H.M, "An Architecture for Software-Controlled Data Prefetching", Proc. 18th International Symposium on Computer Architecture, Toronto, Ont., Canada, May 1991, pp. 43-53.

[14] Hamdy A. Taha, "Operations Research: An Introduction",8th Edition, Chapter 3, pp 90, PHI Learning Private Limited.

[15] Hamdy A. Taha, "Operations Research: An Introduction",8th Edition, Chapter 8, pp 338, PHI Learning Private Limited.