Journal of
Low Power Electronics
Vol. 8, 1–20, 2012

Branch Target Buffer Energy Reduction Through Efficient Multiway Branch Translation Techniques

Sumanta Pyne* and Ajit Pal

Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, West Bengal 721302, India

(Received: 15 June 2012; Accepted: 5 October 2012)

Branch Target Buffer (BTB) plays an important role for pipelined processors in branch prediction during the execution of loops, if-then-else, call-return, and multiway branch statements. It has been observed that 20% of instructions in a program are related to branch. Access to BTB consumes 10% of total energy consumption of a program in execution. The present work introduces the use of *K–d* tree and pattern matcher to generate efficient code, i.e., lesser execution time, for multiway branch. However, instead of enhancing performance, Voltage Frequency Scaling (VFS) can be applied to achieve energy efficiency without degradation in performance. The present work is evaluated on a wide range benchmark programs. The BTB energy saving in the present work lies in the range 20% to 80% with small improvement performance as well. The total energy reduction is in the range 3–12%.

Keywords: Multiway Branch, *K–d* Tree, Pattern Matcher, Voltage Frequency Scaling, Branch Target Buffer, Energy, Performance.

1. INTRODUCTION

The present work introduces some techniques to reduce Branch Target Buffer (BTB) energy consumption through efficient translation of multiway branch. Low energy code generation is an important aspect of modern compilers.[1] It has been observed that 20% of instructions in a program are branch instructions.[2] BTB consumes 10% of total energy consumption of a program in execution.[10, 11]

In most of the high-level languages, the construct 'Multiway Branch' (*MB*) is widely used for the selection of one out of several possible blocks of code to be executed. For example, it is the *case* statement in Pascal, it is the *switch* statement in C and it is the *SELECT* statement in Fortran 90. Figures 1(a) and (b) shows the multiway branch as switch and *if-then-else* ladder, respectively, containing n branch destinations. Where, BC_j is the block of code at jth branch destination (BD_j), $1 \leq j \leq n$. One or more index variables form an index expression. The index expression should match with the jth matching value ($value_j$) to jump to BD_j and execute BC_j.

In modern processors Dynamic Branch Prediction is done and Branch Target Buffer (BTB) is commonly used to improve the performance of execution of branch instructions. Dynamic Branch Prediction uses the information about taken or not taken branches gathered at run-time to predict the outcome of a branch. BTB is a small cache memory used to hold the branch history and the target addresses corresponding to different branch instructions.

There are three possible alternatives for the implementation of multiway branch. The three implementations are based on the way the index expression with $value_j$ is searched to find out BD_j. These are linear search, binary search or hashing.[3, 4] For a given *MB* the compiler implements either B_{linear}, B_{binary}, or B_{hash} on the basis of value(s) of index expression(s). B_{linear}, B_{binary}, and B_{hash} requires $O(n)$, $O(\log_2 n)$ and $O(1)$ BTB accesses, respectively, to find out the target address of the BD_j. The first choice of the compiler is to implement a B_{hash}. The generation of B_{hash} depends on the possibility to find a hash function by analyzing the values matched by the index expression(s). This may not be possible for every *MB*. But it is always possible to generate a B_{binary}. However, the simplest implementation is the B_{linear}. In case of *if-then-else ladders*, most of the modern compilers generate B_{linear}, when multiway branch decision depends on more than one index expressions. The present work shows that it is possible to implement B_{hash} or B_{binary} for such *if-then-else ladders*. It introduces the utility of *k–d* tree[6] to generate B_{binary}. Many modern programming languages like C# and Ruby supports *MB* where the index expression values are strings.

*Author to whom correspondence should be addressed.
Email: sumantapyne@gmail.com

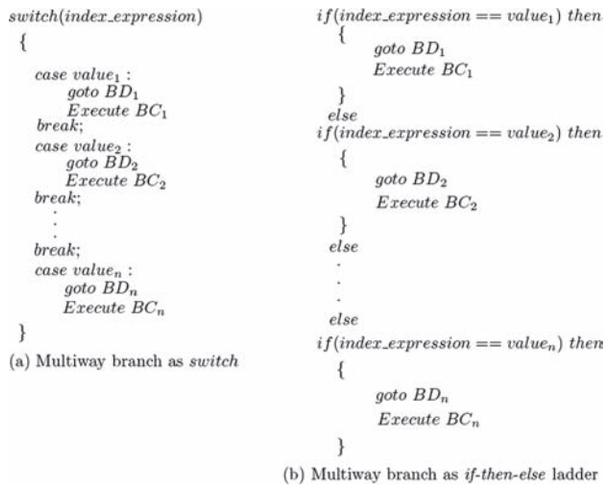

Fig. 1. Equivalent forms of multiway branch.

The present work helps to generate efficient code called $B_{pattern}$ for such *MB* using pattern matching. It considers a source code containing m *MB*s and translates them to m *TMB*s (Translated Multiway Branch) as shown in Figure 2. The MB_l is a based on B_{linear}, on the other hand TMB_l utilizes either a B_{binary} or a B_{hash}.

As TMB_l utilizes either a B_{binary} or a B_{hash}, its execution time is smaller than that of B_{linear}. However, instead of enhancing performance, it is possible to reduce energy consumption by scaling down the voltage along with frequency, commonly known as Voltage and Frequency Scaling (VFS). However, the processor on which the code is executed should be a special type of processor that can operate at different voltages and frequencies, such as Strong ARM 1100. Here, we have used Intel's XScale processor, which works on nine different voltage–frequency (v, f) pairs and supports VFS. Table I shows the (v, f) pairs supported by XScale. The (v_1, f_1) is the peak (v, f) pair and (v_9, f_9) is the least. The BTB Energy Reduction Algorithm with VFS Algorithm takes MB_l as input and generates TMB_l as output, for $1 \leq l \leq m$. The VFS Algorithm scales down the (v, f) to minimize energy consumed by *MB* and the execution of *TMB* finishes within the *deadline*, i.e., $T_{translated} \leq deadline$. Where, $deadline = T_{linear}$ is execution time of *MB* which is a B_{linear}. $T_{translated}$ is the execution time of *TMB* which is based on either B_{binary} or B_{hash}. It may be noted that the voltage–frequency

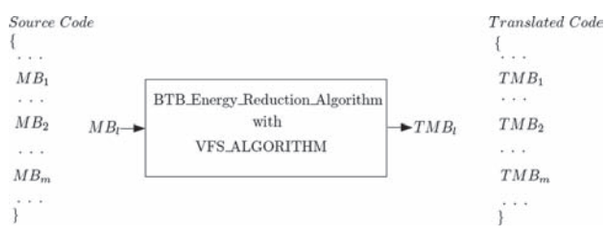

Fig. 2. Agenda of the present work.

Table I. Voltage–frequency pairs supported by XScale.

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

pairs supported XScale processor have limited number of discrete values. As a consequence, the chosen voltage–frequency pair for a particular *TMB* may not fully utilize the slack, i.e., the difference between $(T_{linear} - T_{translated})$. This can be used to achieve small enhancement in performance of the *TMB* along with energy efficiency provided by the chosen voltage–frequency pair. This work is applicable to *MB* where the estimation of time taken and energy consumed to jump to BD_j and execute BC_j can be done at compile time.

The proposed scheme is simulated on XEEMU,[7,8] which simulates Intel's XScale processor. The related works are discussed in Section 2. Section 3 illustrates the proposed scheme with illustrative examples and explains the application of VFS. Section 4 describes the experimental setup and evaluates the proposed scheme with benchmark programs. Section 5 concludes the present work with its future scopes.

2. RELATED WORKS

The past works on BTB energy/energy reduction were implemented either by hardware or by software. Both techniques concentrated on the reduction of BTB access.

2.1. Hardware Techniques

In Ref. [9] Deris et al. introduced Speculative BTB Access (SABA), to identify cycles where there is no control flow instruction among those fetched, at least one cycle in advance. By identifying such cycles and eliminating unnecessary BTB accesses BTB energy reduction varies between 6–15% with an average performance loss of 1.5%.

In Ref. [10] the non-necessary accesses to BTB are reduced by taking into account this fact that there exists distances between different consecutive branch instructions. This method decides the access to BTB by a constant value and a counter. After an instruction entrance, the BTB is accessed if the counter is zero, and if the instruction is a branch instruction and exists in the BTB the counter is reset. The approach achieves BTB energy saving by 25%.

In Ref. [11] the authors introduced the use of a static BTB that achieves the similar performance to the traditional branch target buffer but which eliminates most of

the state updates thus reducing the energy consumption of the BTB significantly. They also introduce a correlation based static prediction scheme into a dynamic branch predictor so that those branches that can be predicted statically or can be correlated to the previous ones will not go through normal prediction algorithm. This reduces the activities and conflicts in the branch history table. It saves 43.9% energy of the branch prediction unit without degradation of performance.

Hu et al. in Ref. [12] proposed two approaches to reduce BTB accesses. The first approach expects the distance of every two dynamic branch instructions to be a constant N, where N can be statically profiled, and forces BTB to response for N instructions after a BTB hit. The second approach dynamically predicts the address of the next branch instruction, and accesses BTB only on the predicted address. This reduces 22.033% of useless BTB access.

In Ref. [13] the authors studied two mechanisms that reduce dynamic energy dissipation. The first one is a serial-BTB configuration. The second mechanism is the filter-BTB, a combination of a low energy counting Bloom filter placed in front of a conventional BTB. They also studied the effect of placing a small 32 entry direct-mapped BTB, functioning as a bypass, in parallel with the first two mechanisms. The filter-BTB reduces the number of lookups relative to a conventional BTB and the dynamic energy dissipated. The serial-BTB variant only accesses the data array of the BTB upon a hit, therefore for most of the accesses the actual energy dissipated is only what is dissipated by accessing the tag array. The bypass is used in parallel to either the filter-BTB or the serial-BTB and reduces the performance cost by providing a low latency response in case of a hit. By integrating these mechanisms into a BTB design the scheme achieved an average reduction of 51% in the dynamic energy dissipation of the BTB. These benefits come at a small performance cost that is on average slightly less than 1.2%.

In Ref. [14] Kahn et al. investigated three architectural methods to reduce the leakage energy dissipated by the BTB data array. The first method (called here window) periodically places the entire BTB data array into drowsy mode. A drowsy entry is woken up by the first access in the time interval and remains active for the remainder of the interval (window). There is an associated performance loss which is related to the size of the window, since there is a delay when a specific line must be woken up. The second method, awake line buffer (ALB), limits the number of active BTB entries to a predetermined maximum. While this reduces energy dissipation it comes with a performance penalty that is relative to the size of the buffer. ALB, however, reduces the energy dissipation of the data array more than the window method. The third method, 2-level ALB (2L-ALB), uses a two level buffer with the identical number of combined entries as the previous method. This method exploits the fact that many branches operate numerous times in a fixed sequence. By predicting the next BTB access, 2L-ALB achieves further reduction in leakage energy without incurring any further performance loss, compared to the ALB method.

Levison et al. in Ref. [15] proposed two BTB designs that fit the tight energy budgets of embedded processors. In the first design, the energy consumption of a single BTB access is reduced by reading only the lower part of the predicted target address bits. This design has energy savings of up to 25% dynamic energy, with effectively no performance degradation. In the second design, they avoid redundant BTB accesses to the same set by using a small buffer that holds the most recently accessed set. This design results in 75% dynamic energy savings at the cost of up to 0.64% system slowdown in a 2-way BTB, and 80% dynamic energy savings at the cost of up to 0.58% system slowdown in a 4-way BTB.

In Ref. [16] Baniasadi et al. introduced branch predictor prediction (BPP) which reduces branch prediction energy dissipation by selectively turning on and off two of the three tables used in the combined branch predictor BPP which relies on a small buffer that stores the addresses and the sub-predictors used by the most recent branches executed. They refer to this buffer to decide if any of the sub-predictors and the selector could be gated without harming performance. They show that on the average and for an 8-way processor, BPP can reduce branch prediction energy dissipation by 28% and 14% compared to non-banked and banked 32 k predictors respectively. This comes with a negligible impact on performance (1% max).

The authors in paper Ref. [17] proposed to use the loop cache to reduce static energy consumption as well as dynamic one. They combined it with CMOS circuits having sleep mode, and thus instruction cache can go to sleep mode when the loop cache is active. They also apply the technique to branch target buffer, and its static and dynamic energy consumption is reduced by up to 40.4% and 40.7%, respectively.

In Ref. [18] Tomas et al. analyzes at what extent tag and target address lengths could be reduced to benefit both dynamic and static energy consumption, silicon area, and access time, while sustaining performance. The tag length and the target address could be reduced by about a half and one byte, respectively with no performance losses. BTB energy savings can reach about 35%.

Levison et al. in Ref. [19] propose a novel micro-architectural method referred to as Shifted-Index BTB with a Set-Buffer, which reduces both dynamic and static energy. It achieves up to 80% reduction in dynamic energy is achieved at the cost of up to 0.64% system slowdown. 58% reduction is static energy is also achieved by applying low-leakage energy techniques that mesh well with the Set-Buffer design.

In Ref. [20] Deris et al. introduce Branchless Cycle Prediction (BLCP) which predicts cycles where there is no branch instruction among those fetched, at least one cycle in advance. They avoid accessing BTB during such cycles.

By using BLCP, it is possible to reduce BTB energy dissipation by 32% while paying a negligible performance cost (average: 0.2%).

The paper Ref. [21] proposes an energy-aware branch predictor by accessing the BTB selectively. To enable the selective access to the BTB, the PHT (Pattern History Table) in the proposed branch predictor is accessed one cycle earlier than the traditional PHT if the program is executed sequentially without branch instructions. As a side effect, two predictions from the PHT are obtained through one access to the PHT, resulting in more energy savings. In the proposed branch predictor, if the previous instruction was not a branch and the prediction from the PHT is untaken, the BTB is not accessed to reduce energy consumption. If the previous instruction was a branch, the BTB is always accessed, regardless of the prediction from the PHT, to prevent the additional delay/accuracy decrease. The proposed branch predictor reduces the energy consumption by 29–47% with little hardware overhead, not incurring additional delay and never harming prediction accuracy.

Briejer et al. in Ref. [22] proposed energy-efficient dynamic branch predictors for the Cell SPE, which normally depends on compiler-inserted hint instructions to predict branches. The prediction scheme predecodes instructions when they are fetched from the local store and accesses the BTB only for branch instructions, thereby saving energy compared to conventional dynamic predictors that access the BTB for every instruction. The authors also introduce branch warning instructions which initiate branch prediction before the actual branch instruction is fetched. This allows fetching the instructions starting at the branch target and thus completely removes the branch penalty for correctly predicted branches. For a 256-entry BTB, a speedup of up to 18.8% is achieved. The energy consumption of the branch prediction schemes is estimated at 1% or less of the total energy dissipation of the SPE and the average energy-delay product is reduced by up to 6.2%.

2.2. Software Techniques

Software techniques like loop unrolling and loop fusion reduce BTB access as well as BTB energy consumption. In Ref. [23] Yang et al. study the impact of loop optimizations such as loop unrolling and software pipelining in terms of performance and energy tradeoffs. Zhu et al. in Ref. [24] consider the effect of loop fusion on energy. Loop fusion combines corresponding iterations of different loops. It decreases program run time increasing instruction per cycle (IPC), by reducing loop overhead. The fusion-induced improvements in program energy are slightly smaller than improvements in program run time. If IPC is held constant, however, by reducing frequency and voltage-particularly on a processor with multiple clock domains then energy improvements may significantly exceed run time improvements. They demonstrate energy savings ranging from 7–40%, with run times ranging from 1% slowdown to 17% speedup.

3. PRESENT WORK

The present work proposes BTB Energy Reduction Algorithm which takes MB_l as input and produces TMB_l as output. Figure 3(a) shows the format of an MB_l. Here MB_l is a B_{linear} enclosed in a loop, which executes p times, where $p \geq 1$. B_{linear} contains a multiway branch construct having n branch destinations. In other words, B_{linear} can be considered as an *if-then-else ladder* having n branch destinations. The proposed scheme applies VFS. The VFS_Algorithm finds the opportunity to scale down (v, f) of TMB_l. Table II shows the two cases of VFS algorithm. These cases are based on the input dependency of p, where, p is the number of times the MB_l will execute. The value of p is input dependent means p's value is obtained at runtime as an input. If p is input independent, then its value is always a constant. The proposed scheme considers two different forms of VFS algorithm. Figures 3(b) and (c) show the format of the *TMB* produced by different forms of VFS algorithm. The variable min_vf_pair $(1 < min\_vf\_pair \leq 9)$ in Figure 3(c) implies that execution of *TMB* at $(v_{min\_vf\_pair}, f_{min\_vf\_pair})$ will minimize the energy consumed by it. In Figure 3(c) $P[min\_vf\_pair]$, $P[min\_vf\_pair\text{-}1]$, $P[min\_vf\_pair\text{-}2], \ldots, P[2]$ are the minimum values

Fig. 3. Format of the multiway branch (MB) and translated multiway branch (TMB) codes.

Table II. Input dependency.

Case	p	VFS_Algorithm
A	Input independent	VFS_Algorithm_A
B	Input dependent	VFS_Algorithm_B

of p required to execute TMB_l at voltage–frequency pairs $(v_{min\_vf\_pair}, f_{min\_vf\_pair})$, $(v_{min\_vf\_pair\text{-}1}, f_{min\_vf\_pair\text{-}1})$, $(v_{min\_vf\_pair\text{-}2}, f_{min\_vf\_pair\text{-}2}), \ldots, (v_2, f_2)$, respectively. The subroutines *setVoltage* and *setFrequency* helps to scale up and scale down the (v, f) pair at runtime.

3.1. Illustrative Examples

To demonstrate the efficacy of the approach, three illustrative examples are provided in this section. In the illustrative example *EX*1 the *MB* can be implemented as B_{linear}, B_{hash}, or B_{binary}. For the illustrative example *EX*2 compilers generate B_{linear}. It is hard to implement B_{hash} for *EX*2. In this case, *k–d* tree[6] is used to implement B_{binary} for *EX*2. The illustrative example 3 *EX*3 deals with an *MB* where the index variables and values are strings. The branching takes place on string matching. The compilers generate B_{linear} for such *MB*s. Here it introduces the use of pattern matcher to generate time and energy efficient $B_{pattern}$ code for *EX*3. The assembly language used in this paper is based on the instruction set of XScale processor. The assembly language codes for B_{linear} and B_{hash} are generated by xscale-gcc-elf compiler. The B_{binary} and $B_{pattern}$ codes are generated by traversal of *k–d* tree and pattern matcher graph, respectively. The experimental values in Tables III–V are obtained by executing the possible B_{linear}, B_{hash}, B_{binary} and $B_{pattern}$ implementations of the illustrative examples on XEEMU simulator. These tables use the following metrics to compare the different energy and performance implementations of the illustrative examples:
(i) 'Time' is the total execution time taken of the program in seconds (sec),
(ii) 'Total Energy' is the energy consumed by the program in Joules (J),
(iii) 'BTB Energy' is the energy consumed by the BTB during the execution of the program micro Joules (μJ).

The tables also show the performance and energy gained by B_{hash}, B_{binary} and $B_{pattern}$ implementations with respect to B_{linear} in percentage (%). The tables also compare the following BTB parameters:
(i) 'Total branches' is the total number of branch instructions executed in the program,
(ii) 'Miss prediction taken' is the total number wrong predictions taken by the Branch Prediction Unit (BPU) when a branch takes place,
(iii) 'Miss prediction not taken' is the total number wrong predictions taken by the BPU when no branch takes place,
(iv) 'Non prediction taken' is the total number of branches taken when no predictions are taken by the BPU because the BTB has no entry for the branch history and target addresses of the corresponding branch instructions.

3.1.1. Illustrative Example 1 (EX1)

*EX*1 considers a simple MB which can implemented as *if-then-else* and *switch-case*, as shown in Figures 4(a) and (b), respectively. Here, 'marks' is the index variable that forms the index expression. The matching value set for index variable marks is *value* (marks) = {4, 5, 6, 7, 8, 9, 10}. The GCC compiler xscale-gcc-elf translates the source code in Figure 4(a) to B_{linear} code. For the source code in Figure 4(b) the xscale-gcc-elf generates B_{hash} code.

Table III. *EX*1 results.

*EX*1 code	Metric	Value	Gain (%)	BTB parameter	Value
B_{linear} *at* (v_1, f_1)	Time (sec)	0.0832	–	Total branches	8999968
	Total energy (J)	0.0616	–	Miss predictions taken	56
	BTB energy (μJ)	353.08	–	Miss predictions not taken	18
				Non prediction taken	11
B_{hash} *at* (v_1, f_1)	Time (sec)	0.0450	45.91	Total branches	3000116
	Total energy (J)	0.0338	45.12	Miss predictions taken	36
	BTB energy (μJ)	117.67	66.67	Miss predictions not taken	4
				Non prediction taken	81
B_{hash} *at* (v_5, f_5)	Time (sec)	0.0707	15.02	Total branches	3000116
	Total energy (J)	0.0169	72.56	Miss predictions taken	36
	BTB energy (μJ)	47.71	86.48	Miss predictions not taken	4
				Non prediction taken	81
B_{binary} *at* (v_1, f_1)	Time (sec)	0.0477	42.66	Total branches	7000076
	Total energy (J)	0.0359	41.72	Miss predictions taken	49
	BTB energy (μJ)	274.62	22.22	Miss predictions not taken	12
				Non prediction taken	11
B_{binary} *at* (v_5, f_5)	Time (sec)	0.0750	9.85	Total branches	7000076
	Total energy (J)	0.0179	70.94	Miss predictions taken	49
	BTB energy (μJ)	111.31	68.47	Miss predictions not taken	12
				Non prediction taken	11

Table IV. *EX*2 results.

*EX*2 code	Metric	Value	Gain (%)	BTB parameter	Value
B_{linear} *at* (v_1, f_1)	Time (sec)	0.0560		Total branches	8008955
	Total energy (J)	0.0411	–	Miss predictions taken	1085
	BTB energy (μJ)	314.225	–	Miss predictions not taken	56
				Non prediction taken	11
B_{binary} *at* (v_1, f_1)	Time (sec)	0.0273	51.25	Total branches	5008198
	Total energy (J)	0.0213	48.17	Miss predictions taken	3071
	BTB energy (μJ)	196.578	37.44	Miss predictions not taken	2040
				Non prediction taken	11
B_{binary} *at* (v_5, f_5)	Time (sec)	0.0430	23.21	Total branches	5008198
	Total energy (J)	0.0106	74.20	Miss predictions taken	3071
	BTB energy (μJ)	79.682	74.64	Miss predictions not taken	2040
				Non prediction taken	11

This depends on the ability of the compiler to find a possible hash function. Sometimes it is not possible to find a hash function. However, it is always possible to generate a B_{binary} code for a *MB*. The *MB* in *EX*1 can be translated to B_{binary} code as shown in Figure 19, in Appendix A. Figure 5 shows the binary search tree formed with all possible values to be matched with index variable. B_{binary} is generated by preorder traversal of the binary search tree. For a *MB* with n branch destinations belonging to the class of *EX*1, B_{linear} will take $O(n)$ time to jump to a branch destination. While B_{hash} and B_{binary} will take $O(1)$ and $O(\log_2 n)$ time, respectively. The B_{linear}, B_{hash}, and B_{linear} codes of *EX*1 are shown in the Appendix A. Table III compare the energy and performance of the different implementations of the *EX*1 and show the values of the BTB parameters. It also shows the energy and performance gained by B_{hash} and B_{binary} with respect to B_{linear}. The execution time of B_{linear} at (v_1,f_1) is considered as the *deadline* for B_{hash} and B_{binary} to finish execution. VFS is applied to B_{hash} and B_{binary} to minimize energy consumption.

3.1.2. Illustrative Example 2 (*EX*2)

The *MB* in Figure 6 is an *if-then-else ladder* which performs a two-dimensional range testing. The *if-then-else ladder* contains three branch destinations BD_1, BD_2 and BD_3 for the blocks of code '$z = 1$,' '$z = 2$' and '$z = 3$,' respectively. When none of the conditions in the *if-then-else ladder* are satisfied, the control jumps to a branch destination NEXT. For such *MB* it is hard for a compiler to generate B_{hash} code. Compilers generate B_{linear} code for this type of *MB*, which is inefficient in terms of energy and performance. The present work introduces that it is possible to generate B_{binary} code for such *MB*s. This is done with the help of *k–d* tree.[6] *k–d* tree is a multidimensional binary search tree. The matching value set for index variable 'x' is the $value(x) = \{3, 5, 6, 12, 13, 16\}$. The matching value set for index variable 'y' is $value(y) = \{1, 3, 4, 7, 8, 12\}$. The ordered pair set or point set of matching values is $value(x, y) = \{(3, 1), (3, 3), (5, 1), (5, 3), (6, 4), (6, 7), (12, 4), (12, 7), (13, 8), (13, 12), (16, 8), (16, 12)\}$, as obtained from the source code in Figure 6. The *k–d* tree decomposition for the point set $value(x, y)$ (as shown in Fig. 7) is done with the help of Bentley's approach in Ref. [6]. The resulting *k–d* tree for the point set $value(x, y)$ is shown in Figures 8. In Figure 7 lines l_3, l_7, l_{10} and l_{14} encloses the region related to BD_1. The lines l_9, l_6, l_{13} and l_1 enclose the region related to BD_2. The lines l_8, l_{12}, l_2 and l_4 enclose the region related to BD_3. The rest of the regions are related to NEXT. Each non-leaf node of the *k–d* tree has left and right edges which connects it to its left and right subtrees, respectively.

Table V. *EX*3 results.

*EX*2 code	Metric	Value	Gain (%)	BTB parameter	Value
B_{linear} *at* (v_1, f_1)	Time (sec)	0.7771	–	Total branches	71933373
	Total energy (J)	0.5909	–	Miss predictions taken	3866738
	BTB energy (μJ)	2944.25	–	Miss predictions not taken	3666680
				Non prediction taken	8600003
$B_{pattern}$ *at* (v_1, f_1)	Time (sec)	0.6695	13.84	Total branches	33133409
	Total energy (J)	0.4969	15.90	Miss predictions taken	3466753
	BTB energy (μJ)	1532.97	47.93	Miss predictions not taken	1066672
				Non prediction taken	7866669
$B_{pattern}$ *at* (v_5, f_5)	Time (sec)	0.7364	5.23	Total branches	33133409
	Total energy (J)	0.4246	28.14	Miss predictions taken	3466753
	BTB energy (μJ)	1267.61	56.94	Miss predictions not taken	1066672
				Non prediction taken	7866669

```c
int main()
{
 int s, marks;
 char grade;
 for(s=0;s<1000000;s++)
 {
  marks=s/10;
  if(marks==10)
       grade='X';
  else
  if(marks==9)
       grade='X';
  else
  if(marks==8)
       grade='A';
  else
  if(marks==7)
       grade='B';
  else
  if(marks==6)
       grade='C';
  else
  if(marks==5)
       grade='D';
  else
  if(marks==4)
       grade='P';
  else
       grade='F';
 }
 return 0;
}
```

(a) Source code of MX1 containing if-then-else ladder

```c
int main()
{
 int s,marks;
 char grade;
    for(s=0;s<1000000;s++)
    {
        marks=s/10;
        switch(marks)
         {
          case 10: grade='X';
          break;
          case 9:grade='X';
          break;
          case 8: grade='A';
          break;
          case 7:grade='B';
          break;
          case 6: grade='C';
          break;
          case 5:grade='D';
          break;
          case 4: grade='P';
          break;
          default: grade='F';
         }
    }
 return 0 ;
}
```

(b) Source code of MX1 containing switch-case statement

Fig. 4. Two possible source codes of *EX*1.

The left edge either labeled with the symbol '$<$' or with '$\leq$'. The right edge is either labeled with the symbol '$>$'or with '$\geq$'. The left and right edge symbols of a node depend on the source code. For example, the left edge symbol of the node l_1 is '$<$' and its right edge symbol is '$\geq$'. This is because in the source code in Figure 6 there is an expression '$x \geq 6$' and l_1 is the line representing '$x = 6$'. So for any node with '$x \geq 6$' will be in the right subtree of l_1, while nodes with '$x < 6$' will be in the left subtree of l_1. The leaf nodes of the k–d tree contain the branch destinations. The leaf nodes BD_1, BD_2, and BD_3 contain the branch destinations for the blocks of code '$z = 1$,' '$z = 2$' and '$z = 3$,' respectively. The rest of the leaves contain NEXT as branch destination. There are two kinds of non-leaf k–d tree nodes considered in this work. The circular non-leaf nodes are the mandatory nodes required to form the k–d tree. The square non-leaf node ensures that a branch destination is enclosed within the desired region. For example in Figure 7 the lines l_{12}, l_{13} and l_{14} provides enclosure for the regions related to BD_3, BD_2, and BD_1, respectively. To jump to BD_3, the following

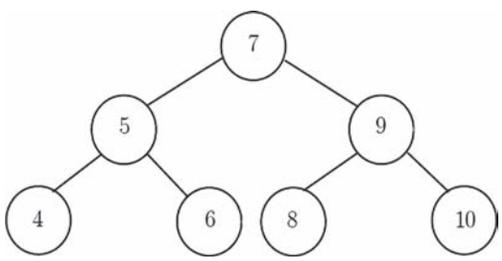

Fig. 5. Binary search tree for *EX*1.

```c
int main()
{
 int x,y,z;
 for(x=0;x<1000;x++)
  {
    for(y=0;y<1000;y++)
     {
      if((x>=13)&&(x<=16)&&(y>=8)&&(y<=12))
        z=1; // BD1
      else
      if((x>=6)&&(x<=12)&&(y>=4)&&(y<=7))
        z=2; // BD2
      else
      if((x>=3)&&(x<=5)&&(y>=1)&&(y<=3))
        z=3; // BD3
     }
  }
return 0;
}
```

Fig. 6. Source code of *EX*2 as *if-then-else ladder*.

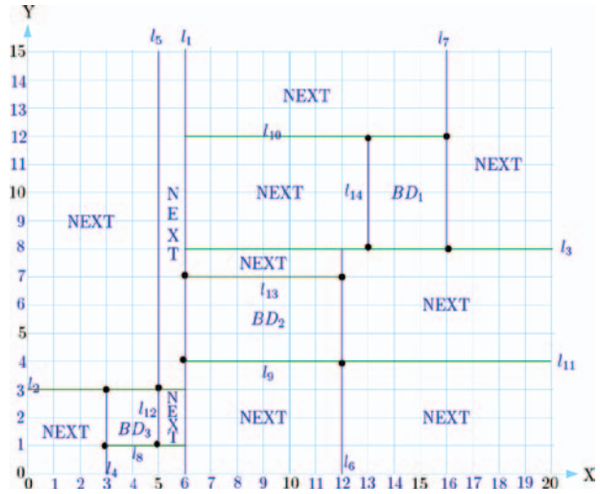

Fig. 7. *k–d* tree decomposition for the point set *value*(x, y).

sequences of conditions are to be satisfied, '$x < 6$,' '$y < 3$,' '$x \geq 3$,' '$y \geq 1$' and '$x \leq 15$'. But, the conditions '$x < 6$' and '$x \leq 5$' are redundant because 'x' is an integer variable. The node l_{12} can be deleted to obtain the modified *k–d* tree in Figure 9. Similarly, l_{13} can also be deleted.

In Figure 8 all the leaves of the right subtrees of the nodes l_2 and l_6 contain NEXT. Each of these subtrees are pruned and replaced with a leaf node containing NEXT as shown in Figure 9. Figure 10 shows two possible assembly language implementations of the *if-then-else ladder* in *EX*2. These assembly language code fragments are written using ARM instruction set. B_{linear} in Figure 10(a) is a brute-force implementation. B_{binary} in Figure 10(b) is obtained by a preorder traversal of the modified *k–d* tree in Figure 9. The preorder traversal algorithm of the *k–d* tree in Figure 9 is shown in Appendix C.

The detailed B_{linear} and B_{binary} implementations of *EX*2 are shown in Figures 20 and 21, respectively, in Appendix B. For a *MB* with n branch destinations belonging to the class of *EX*2 having d distinct index variables in each of the n index expressions, B_{linear} will take $O(2 \times d \times n)$ time to jump to a branch destination. While B_{binary} will take $O(\log_2 n + d + 1)$ time, where d is the dimension of the *k–d* tree. In *EX*2 $d = 2$. Table IV compares the energy and performance of the different implementations of *EX*2 and shows the values of BTB parameters. It also shows the gain achieved by B_{binary} with respect to B_{linear}. The execution time of B_{linear} at (v_1, f_1) is considered as the T_{linear}, which is the deadline for B_{binary} to finish execution. VFS is applied to B_{binary} to minimize energy consumption maintaining the constraint $T_{binary} \leq T_{linear}$. Table IV shows the maximum gain in BTB energy achieved by B_{binary} is 74.64% along with a performance gain of 23.21%, when B_{binary} is executed at (v_5, f_5).

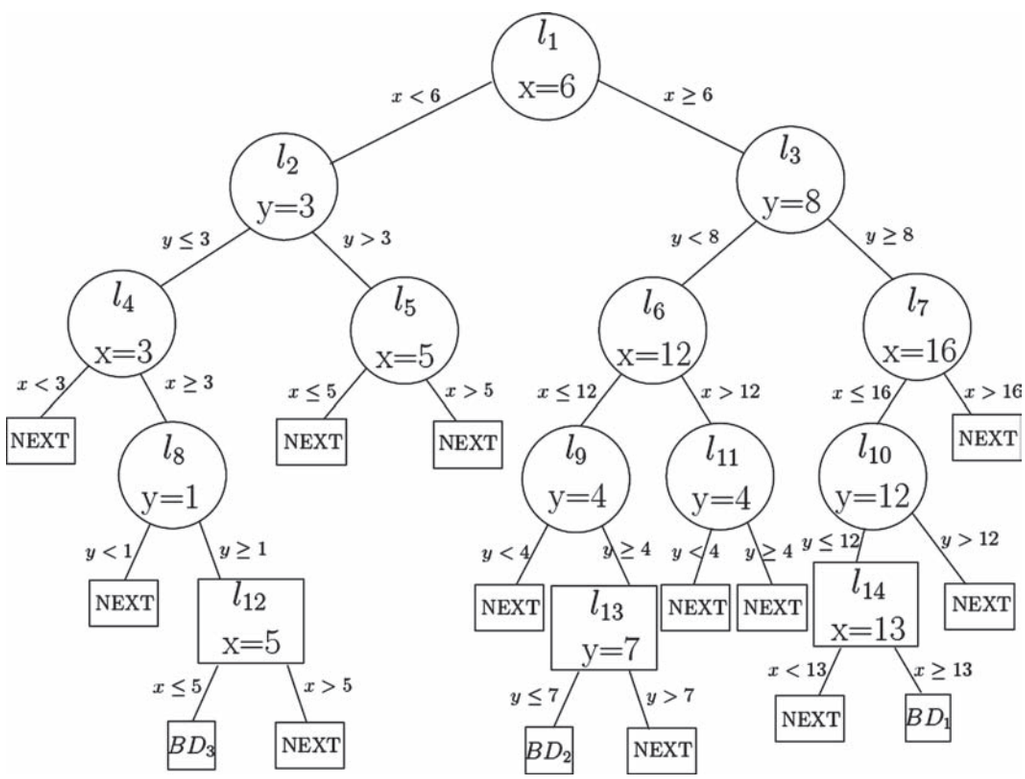

Fig. 8. The resulting *k–d* tree for the point set *value*(x, y).

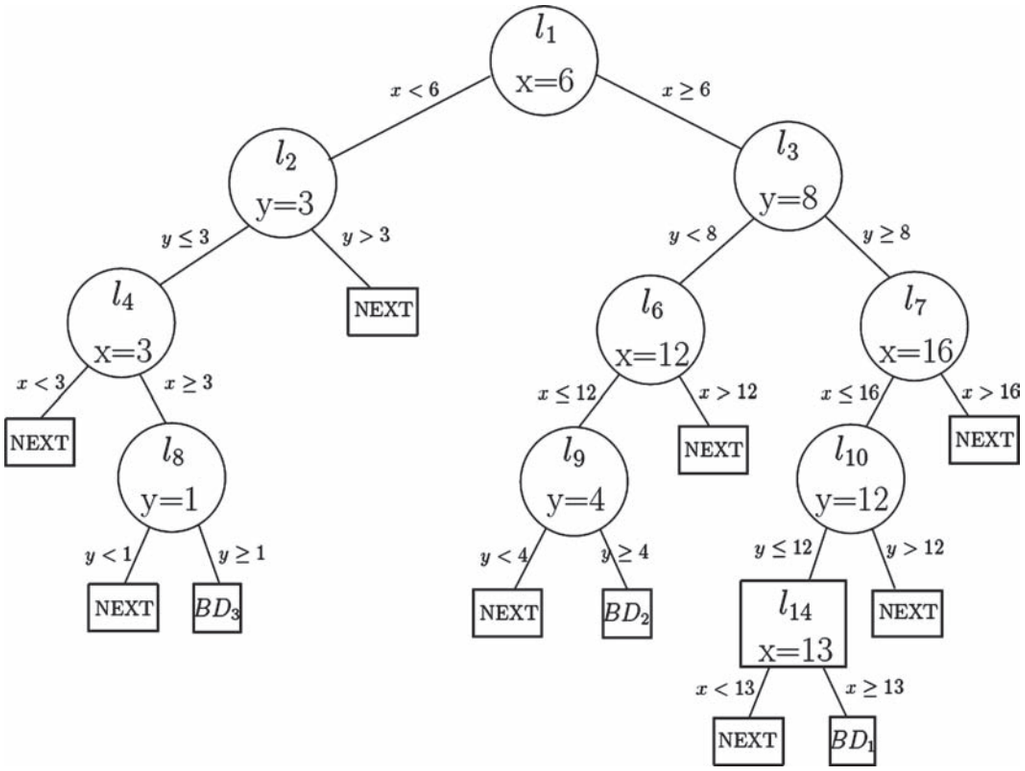

Fig. 9. Modified k–d tree of the for the point set $value(x, y)$.

3.1.3. Illustrative Example 3 (*EX*3)

Programming languages like Ruby provides multiway branch with strings as shown in Figure 11. B_{linear} implementations of these multiway branches are inefficient in terms of time and energy. The pattern matcher in form of a finite state machine in Figure 11 can help to generate $B_{pattern}$ which is both energy and time efficient. The matching value set for index variable 'month' is $value(\text{month}) =$ {"JANUARY," "FEBRUARY," "MARCH," "APRIL," "MAY," "JUNE," "JULY," "AUGUST," "SEPTEMBER," "OCTOBER," "NOVEMBER," "DECEMBER"}. $B_{pattern}$ is generated by breadth first traversal of the pattern matcher graph. $B_{pattern}$ makes use of a data structure called trie (or prefix-tree) to restrict the state transition time while pattern matching to $O(1)$. Table V shows the energy-performance gain, and the BTB parameters of the B_{linear} and $B_{pattern}$, respectively. $B_{pattern}$ takes $O(\psi)$ time to reach a BD, where ψ is the maximum external path length of the pattern matcher graph. The execution time of B_{linear} at (v_1, f_1) is considered as the *deadline* for $B_{pattern}$ to finish execution.

3.2. BTB Energy Reduction Algorithm

This algorithm takes MB_l as input and produces TMB_l as output. The MB_l taken as input is considered to be implemented as B_{linear} code. $B_{translated}$ code in TMB_l is either a B_{hash} code or a B_{binary} code or a $B_{pattern}$ code. After translating the code from B_{linear} code to B_{hash} or B_{binary} or $B_{pattern}$, the algorithm finds the possibility of VFS. On the basis of input dependency of p as shown in Table II the desired VFS algorithm is called. The VFS algorithm scales down the (v, f) to minimize the energy consumed by TMB_l such that $T_{translated} \leq deadline$.

BTB_Energy_Reduction_Algorithm
1. {
2. Given a B_{linear} as input;
3. *if* (B_{linear} can be translated to its equivalent B_{hash}) *then*
4. $B_{translated} := B_{hash}$;
5. *else*
6. *if* (B_{linear} can be translated to its equivalent B_{binary}) *then*
7. $B_{translated} := B_{binary}$;
8. *else*
9. *if* (B_{linear} can be translated to its equivalent $B_{pattern}$) *then*
10. $B_{translated} := B_{pattern}$;
11. *else*
12. goto 17;
13. *if* (p is input dependent) *then*
14. *Call VFS_Algorithm* $B(B_{linear}, B_{translated})$;
15. else
16. *Call VFS_Algorithm* $A(B_{linear}, B_{translated})$;
17. }

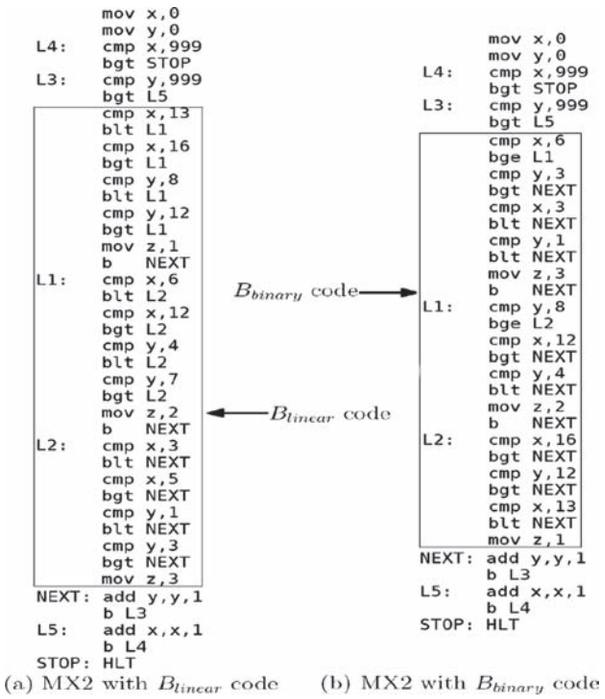

Fig. 10. Assembly language codes representing *if-then-else ladder* in *EX2*.

3.3. VFS_Algorithm

This subsection explains the VFS algorithms in detail. The VFS algorithms find the value of *min_vf_pair*. The *min_vf_pair* is the (v, f) that minimizes the energy consumed by TMB_l. The VFS algorithms calculates the energy overhead ($E_{overhead}$) and time overhead ($t_{overhead}$) due to VFS. They are calculated using the following formulae.[25] Overheads when switching from (v_i, f_i) to (v_w, f_w)

$$E_{overhead}(i, w) = (1 - \mu) \times C \times |V_i^2 - V_w^2| \quad (1)$$

$$t_{overhead}(i, w) = 2 \times \frac{C}{I_{\text{MAX}}} \times |V_i^2 - V_w^2| \quad (2)$$

where, μ is the energy efficiency of the energy regulator which is considered as 90%, C is the voltage regulator's capacitance to be 10 μF, I_{MAX} is the maximum current allowed which is assumed to be 1 A and $1 \leq w \leq 9$. The VFS algorithms make use of a C library function *sprintf* which prints a formatted output to the string S. The subroutine generate code generates the assembly equivalent of the high-level code in S and inserts it to the target program file.

3.3.1. VFS_Algorithm_A

VFS_Algorithm_A finds the possibility of VFS to save energy of TMB_l when p is input independent.

VFS_Algorithm_A(B_{linear}, $B_{translated}$)

1. {
2. *char S*[50];
3. *p:=constant value fixed in compile time;*
4. $T_{linear} := \frac{1}{n} \times \sum_{j=1}^{n} (t_{linear\_j} + t_{branch\_exe\_j}) \times p;$
5. $E_{linear} := \frac{1}{n} \times \sum_{j=1}^{n} (e_{linear\_j} + e_{branch\_exe\_j}) \times p;$
6. $deadline := T_{linear}$;
7. $E_{min} := E_{linear}$;
8. $min\_vf\_pair := 1$;
9. *for*$(i := 1; i \leq 9; i++)$
10. {
11. $T[i] := 2 \times t_{overhead}(1, i) \times \frac{1}{n} \times \sum_{j=1}^{n} (t_{translated\_ij} + t_{branch\_exe\_ij}) \times p;$
12. $E[i] := 2 \times E_{overhead}(1, i) \times \frac{1}{n} \times \sum_{j=1}^{n} (e_{translated\_ij} + e_{branch\_exe\_ij}) \times p;$
13. *if* $(T[i] \leq deadline)$ *then*
14. {
15. *if* $(E[i] < E_{min})$ *then*
16. {

```
names=["XYZ","JANUARY","FEBRUARY","MARCH","APRIL","MAY","JUNE","123","JULY",
"AUGUST","SEPTEMBER","OCTOBER","NOVEMBER","DECEMBER","ABC"]
for s in 0..1000000
   month=names[s%15]
   case month
   when "DECEMBER", "JANUARY", "FEBRUARY"
    season=5  #BD1
   when "MARCH", "APRIL"
    season=1  #BD2
   when "MAY", "JUNE"
    season=2  #BD3
   when "JULY", "AUGUST", "SEPTEMBER"
    season=3 #BD4
   when "OCTOBER", "NOVEMBER"
    season=4  #BD5
   else
    season=0 #BD6
   end
end
```

Fig. 11. Multiway branch with strings.

```
17.                 E_min := E[i];
18.                 min_vf_pair := i;
19.               }
20.           }
21.       }
22.     if (min_vf_pair > 1) then
23.       {
24.         sprintf(S, "setFrequency(%d); setVoltage
              (%d);" min_vf_pair, min_vf_pair);
25.         generate_code(S);
26.       }
27.     generate_code(B_translated);
28.     if (min_vf_pair > 1) then
29.       {
30.         sprintf(S, "setVoltage(1); setFrequency(1);");
31.         generate_code(S);
32.       }
33. }
```

Here, t_{linear_j} and $t_{branch\_exe\_j}$ are time taken to jump to BD_j and execute BC_j of B_{linear}, respectively, at (v_1, f_1). e_{linear_j} and $e_{branch\_exe\_j}$ are energy consumed to jump to BD_j and execute BC_j of B_{linear}, respectively, at (v_1, f_1). $t_{translated\_ij}$ and $t_{branch\_exe\_ij}$ are time taken to jump to BD_j and execute BC_j of $B_{translated}$, respectively, at (v_i, f_i). $e_{translated\_ij}$ and $e_{branch\_exe\_ij}$ are energy consumed to jump to $BD_j$and execute BC_j of $B_{translated}$, respectively at (v_i, f_i). In steps 4 and 5, T_{linear} and E_{linear} are calculated as p times the average time taken and p times the average energy consumed to jump to BD_j and execute the block of code BC_j at (v_1, f_1). Similarly, for $B_{translated}$, $T[i]$ and $E[i]$ are calculated in steps 11 and 12. The algorithm finds the value of *min_vf_pair*, the (v, f) that will minimize energy consumed by TMB_l and allow the execution of the TMB_l to finish within the deadline.

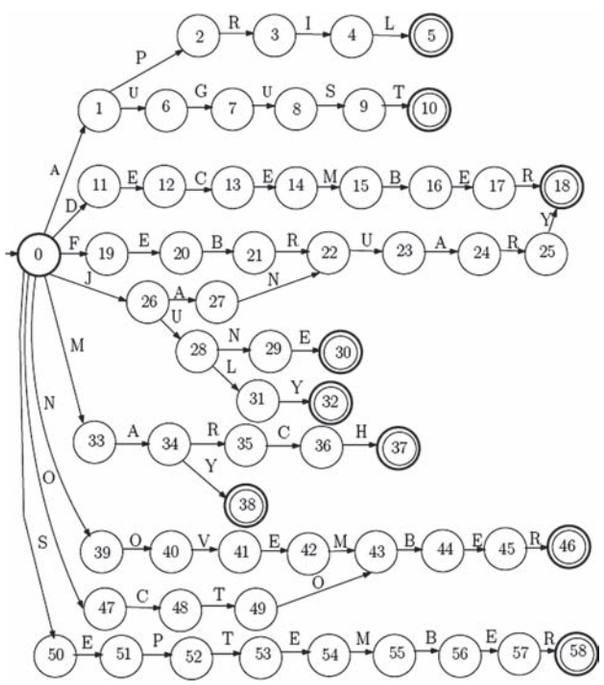

Fig. 12. Pattern matcher for multiway branch with strings (*EX*3).

3.3.2. VFS_Algorithm_B

VFS_Algorithm_B finds the possibility of VFS to save energy of TMB_l when p is dependent. T_{linear}, E_{linear}, $T[i]$ and $E[i]$ are calculated in a similar way as in VFS_Algorithm_A.

```
VFS_Algorithm_B(B_linear, B_translated)
 1. {
 2.   char S[50];
 3.   p := 10^6;
 4.   Linked_List linkedlist := null;
 5.   T_linear := (1/n) × Σ_{j=1}^{n} (t_linear_j + t_branch_exe_j) × p;
 6.   E_linear := (1/n) × Σ_{j=1}^{n} (e_linear_j + e_branch_exe_j) × p;
 7.   deadline := T_linear;
 8.   t_lin_avg := T_linear / p;
 9.   E_min := E_linear;
10.   min_vf_pair := 1;
11.   for(i := 1; i ≤ 9; i++)
12.     {
15.       t_i := (1/n) × Σ_{j=1}^{n} (t_translated_ij + t_branch_exe_ij);
16.       T[i] := 2 × t_overhead(1, i) + t_i × p;
17.       E[i] := 2 × E_overhead(1, i) × (1/n)
            × Σ_{j=1}^{n} (e_translated_ij + e_branch_exe_ij) × p;
18.       if (t_lin_avg > t_i) then
19.         {
20.           P[i] := ⌈ 2 × t_overhead(1, i) / (t_lin_avg − t_i) ⌉;
21.         }
22.       if (T[i] ≤ deadline) then
23.         {
24.           if (E[i] < E_min) then
25.             {
26.               E_min := E[i];
27.               min_vf_pair := i;
28.               if (i > 1) then
29.                 {
30.                   L := create_node();
31.                   L → vf_pair := i;
32.                   linkedlist.addfirst(L);
33.                 }
34.             }
35.         }
36.     }
```

```
37.  for(L := linkedlist.header_node( );
       L ≠ null; L := L → next_node)
38.    {
39.      sprintf(S,"if (p ≥ %d){setFrequency(%d);
           setVoltage(%d); }," P[L → vf_pair],
           L → vf_pair, L → vf_pair);
40.      generate_code(S);
41.      if (L→next node ≠ null) then
42.        {
43.          sprintf(S,"else");
44.          generate_code(S);
45.        }
46.      last_node := L;
47.    }
48.  generate_code(B_translated);
49.  if (min_vf_pair > 1) then
50.    {
51.    sprintf(S,"if (p >= %d){setVoltage(1);
         setFrequency(1); }"; P[last_node→vf_pair]);
52.    generate_code(S);
53.    }
54. }
```

Since, p is input dependent; its value is not known at compile time. The value of p is assigned 10^6 in step 3. Apart from finding *min_vf_pair* the algorithm calculates $P[i]$ for every (v_i, f_i). $P[i]$ is the minimum value of p required to execute TMB_l at (v_i, f_i). The formula for $P[i]$ is derived as follows. Let, t_{lin_avg} be the average execution time of B_{linear}, executed once at (v_1, f_1). Let, t_i be the average execution time of $B_{translated}$, executed once at (v_1, f_1). Steps 8 and 15 of VFS_Algorithm_B calculates t_{lin_avg} and t_i, respectively, when, $t_{lin\_avg} > t_i$. If $B_{translated}$ is executed $P[i]$ times at (v_i, f_i), then the time taken to do this should be atmost that of $P[i]$ time execution of B_{linear} at (v_1, f_1). Considering the overhead of (v, f) scale up and scale down, this can be written as

$$\begin{aligned} & P[i] \times t_i + 2 \times t_{overhead}(1, i) \leq P[i] \times t_{lin\_avg} \\ \Rightarrow\ & P[i] \times (t_{lin\_avg} - t_i) \geq 2 \times t_{overhead}(1, i) \\ \Rightarrow\ & P[i] \geq \frac{2 \times t_{overhead}(1, i)}{(t_{lin\_avg} - t_i)} \end{aligned} \tag{3}$$

The obtained expression of $P[i]$ is the minimum value of p required to execute TMB_l at (v_i, f_i). In other words, if the value of p is obtained at runtime and $p \geq P[i]$ then TMB_l can be executed at (v_i, f_i). The algorithm also generates a linked list as shown in Figure 13(b). Each node of the linked list is an instance of a node type structure as shown in Figure 13(a). The *vf_pair* field of the header node of the linked list contains the *min_vf_pair*. The nodes of the linked list are ordered by the value of *vf_pair* field, as *min_vf_pair*, *min_vf_pair-1*, *min_vf_pair*-2, ..., and 2, where, 1< *min_vf_pair* ≤ 9. The linked list is arranged in such a manner because $P[min\_vf\_pair] > P[min\_vf\_pair - 1] > P[min\_vf\_pair - 2] > \cdots > P[2] \geq 1$. The reason behind this is, as (v_i, f_i) decreases, $P[i]$ increases. After the formation of the linked list the algorithm generates the TMB_l shown in Figure 3(b). The utility of the VFS_Algorithm_B is explained with the help of an *MB* and its equivalent *TMB*, as shown in Figure 14. Figure 14 considers *MB* and its equivalent *TMB*, where p is input dependent. The *MB* in Figure 14(a) contains an *if-then-else ladder* for which compiler generates B_{linear}. *TMB* in Figure 14(b) contains a *switch-case* for which compiler generates B_{hash}. For simplicity, the codes for *MB* and *TMB* in Figure 14 are shown in high level language. For the TMB in Figure 14 VFS_Algorithm_B generates the linked list shown in Figure 15. The linked list informs that the TMB can be run at four different (v, f) pairs other than (v_1, f_1), depending on the value of p. The value of *min_vf_pair* is 5. The value of t_{lin_avg} is 0.0832 μsec. Table VI shows the values of t_i, T_{ov_i} required to calculate $P[i]$. $T_{ov\_i}(= 2 \times t_{overhead}(1, i))$ is the time taken to scale down from (v_1, f_1) to (v_i, f_i) and scale up from (v_i, f_i) to (v_1, f_1). Table VI also shows the values of $T_{MB(v_1, f_1, P[i])}$, $E_{MB(v_1, f_1, P[i])}$, $T_{TMB(v_i, f_i, P[i])}$, and $E_{TMB(v_i, f_i, P[i])}$. These values are obtained for experimental verification of VFS_Algorithm_B. $T_{MB(v_1, f_1, P[i])}$ and $E_{MB(v_1, f_1, P[i])}$ are the time taken and energy consumed, respectively, by the *MB* in Figure 14(a), when it is executed at (v_1, f_1) and $p = P[i]$. $T_{TMB(v_i, f_i, P[i])}$ and $E_{TMB(v_i, f_i, P[i])}$ are time taken and energy consumed, respectively, by *TMB* in Figure 14(b), when it is executed at (v_i, f_i) and $p = P[i]$. $T_{MB(v_1, f_1, P[i])}$

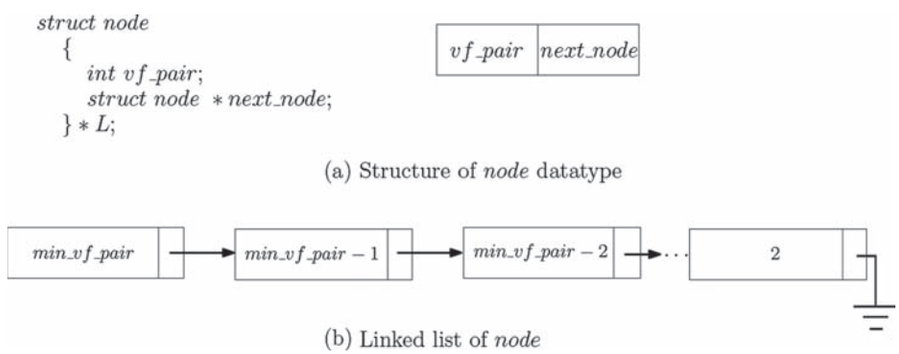

Fig. 13. Linked list created by VFS_Algorithm_B.

Fig. 14. High level representation of an *MB* and its equivalent *TMB*, where p is input dependent.

is the deadline for *TMB* in Figure 14(b). In Table VI $T_{TMB(v_i,f_i,P[i])} \leq T_{MB(v_1,f_1,P[i])}$ for each i $(2 \leq i \leq 5)$. This ensures the utility of VFS_Algorithm_B. The VFS algorithms can save more energy, when the delays of blocks of code at all the branch destinations are equal and the blocks of code contain few branch instructions.

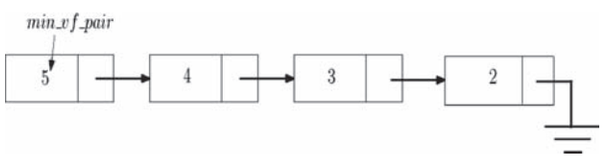

Fig. 15. Linked list created by VFS_Algorithm_B for the *TMB* in Figure 14(b).

4. EXPERIMENT AND RESULT

The proposed scheme is evaluated on eight benchmark programs on XEEMU simulator.[7,8] XEEMU simulates Intel's XScale processor. Since there does not exist standard benchmark programs involving *MB*, several representative examples in which *MB* are possible are considered as synthetic benchmarks. These synthetic benchmark programs impose the workload on the branch prediction unit causing BTB access, which implies their utility for testing the proposed work. This section explains the experimental procedure along with the analysis of the experimental results.

4.1. Experiment

The benchmark programs in Table VII contain one or more *MB*s. Each *MB* belongs to the class of the illustrative

Table VI. The time and energy values of *mb* and *tmb* in Figure 14.

i	t_i (μsec)	T_{ov_i} (μsec)	$P[i]$	$T_{MB(v_1,f_1,P[i])}$ (μsec)	$E_{MB(v_1,f_1,P[i])}$ (μJ)	$T_{TMB(v_i,f_i,P[i])}$ (μsec)	$E_{TMB(v_i,f_i,P[i])}$ (μJ)
2	0.0495	60.49	1795	158.51	115.11	156.69	58.11
3	0.0550	60.46	2144	186.63	136.34	185.27	60.38
4	0.0619	60.48	2840	242.90	178.67	242.24	65.57
5	0.0707	60.53	4843	406.21	300.50	405.15	86.87

Table VII. Benchmark programs.

Benchmark	Description	Source
PL	Evaluates proposition logic formula	cse.iitkgp.ac.in/∼spyne/benchmarks
LR	A shift-reduce bottom up parser	cse.iitkgp.ac.in/∼spyne/benchmarks
GUI	A GUI controller of a database system	cse.iitkgp.ac.in/∼spyne/benchmarks
P'man	Pacman, a computer game	www.javaboutique.internet.com/PacMan
Chess	A computer game	www.caspercomsci.com/pages/javasource.htm
B'ship	Battleship, a computer game	www.caspercomsci.com/pages/javasource.htm
M'Conv	The mode converter, does base conversion of numbers	www.caspercomsci.com/pages/javasource.htm
B'Jack	The black Jack, a computer game	www.caspercomsci.com/pages/javasource.htm

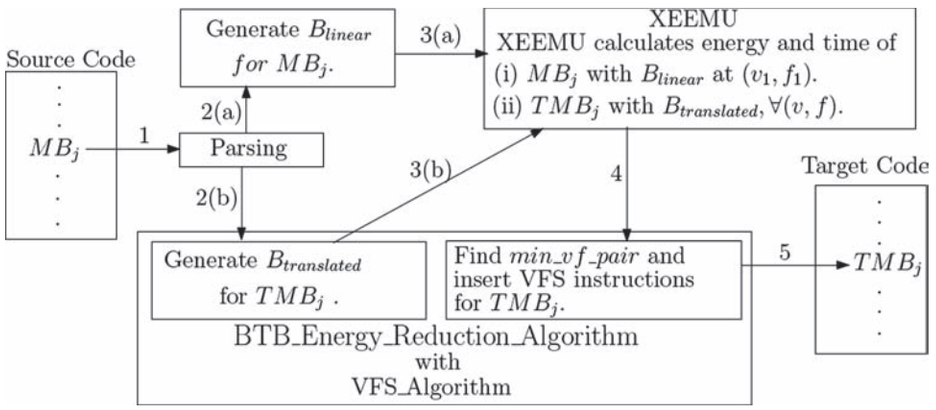

Fig. 16. Experimental setup.

examples as discussed in Section 3.2. All the energy and performance values in this work are measured in XEEMU. The translated codes are written using ARM instruction set. All the programs are run on XEEMU which simulates Intel's XScale processor. XScale has a 128-entry BTB.[26] Each entry contains the address of a branch instruction, the target address associated with the branch instruction, and a previous history of the branch being taken or not-taken.

Table VIII. Benchmark result.

Benchmark	Metric	Naïve code	Translated code	Gain (%)	Type of k–d tree	#k–d tree	Size range
PL	Time (sec)	1.7213	1.589	7.68	1D	16	5–11
	Total energy (J)	1.243	1.153	7.24	2D	–	–
	BTB energy (μJ)	5746	1187	79.34	3D	–	–
LR	Time (sec)	1.1119	0.998	10.24	1D	10	3–11
	Total energy (J)	0.8368	0.7824	6.50	2D	–	–
	BTB energy (μJ)	3380	989	70.73	3D	–	–
GUI	Time (sec)	1.1504	1.1297	1.79	1D	5	5–10
	Total energy (J)	1.8761	1.6572	11.66	2D	2	10–35
	BTB energy (μJ)	3753	1023	72.74	3D	4	4–12
P'man	Time (sec)	2.175	1.962	9.79	1D	1	7
	Total energy (J)	1.875	1.1787	4.69	2D	1	4
	BTB energy (μJ)	7054	2987	57.65	3D	–	–
Chess	Time (sec)	3.154	2.942	6.72	1D	–	–
	Total energy (J)	1.725	1.597	7.42	2D	3	3–8
	BTB energy (μJ)	9432	4763	49.50	3D	–	–
B'Ship	Time (sec)	3.187	3.082	3.29	1D	–	–
	Total energy (J)	1.472	1.386	5.84	2D	1	3
	BTB energy (μJ)	5087	3986	21.64	3D	2	2
M'Conv	Time (sec)	0.482	0.374	22.40	1D	2	26–32
	Total energy (J)	0.6427	0.6182	3.81	2D	–	–
	BTB energy (μJ)	4876	1928	60.45	3D	–	–
B'Jack	Time (sec)	3.257	3.162	2.91	1D	3	3–13
	Total energy (J)	1.876	1.677	10.6	2D	1	14
	BTB energy (μJ)	14872	6748	54.62	3D	–	–

The history is recorded as one of four states: strongly taken, weakly taken, weakly not-taken, or strongly not-taken. If the address of the branch instruction hits in the BTB and its history is strongly or weakly taken, the instruction at the branch target address is fetched; if its history is strongly or weakly not-taken, the next sequential instruction is fetched. In either case the history is updated. Each BTB access and update of its state causes energy consumption. The experimental setup in Figure 16 shows the proposed scheme in a sequence right from syntactical analysis (parsing) to translated multiway branch generation.

4.2. Result

The Table VIII shows the comparison of energy, performance and the maximum gain achieved by the translated code. The metrics 'Time,' 'Total Energy' and 'BTB Energy' are same as in Tables III–V in Section 3.1. Naïve Code is the code generated by the compiler xscale-elf-gcc, which contains mainly B_{linear} implementations of *MBs*. Translated code contains possible B_{hash} and B_{linear} as *TMB*s. The performance gain lies within a range of 1 to 22%. The gain in total energy lies within a range of 3 to 12%. The gain in BTB energy lies within a range of 21 to 80%. The size of a k–d tree is the number of nodes in it. For a particular multiway branch, the size of a k–d tree depends on the number of branch destinations (n) and the dimension (d) of the k–d tree. For a multiway branch with n branch destinations and index expression values as d-dimensional discrete points, a k–d tree will have n nodes. For a multiway branch with n branch destinations and index expression values as d-dimensional ranges, a k–d tree will have $2^d \times n$ nodes. The programs with more number of k–d trees with larger size achieve better energy and performance gain. Table VIII also keeps an account of the of the k–d trees formed during the code translation of the benchmark programs. It shows the number of k–d trees (#k–d tree) belonging to different dimensions (Type of k–d tree) and the range of their size (Size range). Here, the 'Type of k–d tree' is either 1D (one-dimensional), 2D (two-dimensional) or 3D (three dimensional).

5. CONCLUSION AND FUTURE WORK

The present work reduces energy consumption for BTB access by translating multiway branch with VFS. The translated multiway branch also improves the performance of the program. It first transforms the multiway branch and then applies VFS to scale down the (v, f) to minimize energy consumed by *MB* under the execution time constraint. It introduces the use of k–d tree and pattern matcher to generate efficient code for multiway branch when hashing is not applicable. A wide range of illustrative examples and benchmark programs are used to highlight the efficacy of the approach. The energy savings ranges from 21 to 80% with performance improvement ranging from 1 to 22%. The total energy is reduced within a range of 3 to 12%. As in the present work, the access to BTB is reduced; the future work will concentrate on reducing runtime leakage energy of BTB when it is not in use. We have restricted the index variables and matching values to integers and strings. The work may be extended to consider real numbers. There are *if-then-else ladders* where the index expressions are formed with several index variables, and the conditions are separated by several logical or conditional operators. The future work will also investigate on efficient translation of such *MB*s.

APPENDIX

A. B_{linear}, B_{hash} and B_{binary} Implementations of *EX*1

The B_{linear} and B_{hash} codes of *EX*1 are generated by xscale-elf-gcc compiler. Step 30 of B_{hash} code of *EX*1 in Figure 18 shows the application of hashing. The B_{binary} code of *EX*1 is generated by preorder traversal of the binary search tree in Figure 5.

B. B_{linear} and B_{binary} Implementations of *EX*2

The B_{linear} code of *EX*2 is generated by xscale-elf-gcc compiler. The B_{binary} code of *EX*2 is generated by preorder traversal of the k–d tree in Figure 8. Appendix C illustrates the algorithm for preorder traversal of K–d tree.

C. B_{binary} Code Generation from *K–d* Tree

C.1. Structure of the K–d Tree Node

```
struct Kd_Tree_Node
{
  char variable_name[20];
  int value;
  char left_edge_symbol, right_edge_symbol;
  boolean left_tree_visited, right_tree_visited;
  Kd_Tree_Node *left_child, *right_child;
};
```

C.2. Code Generation

```
B_binary_code_generation_from_Kd_Tree(Kd_Tree_node
*root)

1. {
2.   for(all nodes q in the K–d tree)
3.   {
4.   q → left_tree_visited := false;
5.   q → right_tree_visited := false;
6.   }
7.   Preorder_Traversal_Kd_Tree(root,1);
8. }
```

```
1.  .file "exp1_linear.s"
2.  .text
3.  .align 2
4.  .global main
5.  .type main, %function
6.  main:
7.  @ args = 0, pretend = 0, frame =
    12
8.  @ frame_needed = 1,
    uses_anonymous_args = 0
9.  mov ip, sp
10. stmfd sp!, {fp, ip, lr, pc}
11. sub fp, ip, #4
12. sub sp, sp, #12
13. mov r3, #0
14. str r3, [fp, #-16]
15. .L2:
16. ldr r2, [fp, #-16]
17. ldr r3, .L19
18. cmp r2, r3
19. bgt .L3
20. ldr r1, [fp, #-16]
21. ldr r3, .L19+4
22. smull r2, r3, r1, r3
23. mov r2, r3, asr #2
24. mov r3, r1, asr #31
25. rsb r3, r3, r2
26. str r3, [fp, #-20]
27. ldr r3, [fp, #-20]
28. cmp r3, #10
29. bne .L5
30. mov r3, #88
31. strb r3, [fp, #-21]
32. b .L4
33. .L5:
34. ldr r3, [fp, #-20]
35. cmp r3, #9
36. bne .L7
37. mov r3, #88
38. strb r3, [fp, #-21]
39. b .L4
40. .L7:
41. ldr r3, [fp, #-20]
42. cmp r3, #8
43. bne .L9
44. mov r3, #65
45. strb r3, [fp, #-21]
46. b .L4
47. .L9:
48. ldr r3, [fp, #-20]
49. cmp r3, #7
50. bne .L11
51. mov r3, #66
52. strb r3, [fp, #-21]
53. b .L4
54. .L11:
55. ldr r3, [fp, #-20]
56. cmp r3, #6
57. bne .L13
58. mov r3, #67
59. strb r3, [fp, #-21]
60. b .L4
61. .L13:
62. ldr r3, [fp, #-20]
63. cmp r3, #5
64. bne .L15
65. mov r3, #68
66. strb r3, [fp, #-21]
67. b .L4
68. .L15:
69. ldr r3, [fp, #-20]
70. cmp r3, #4
71. bne .L17
72. mov r3, #80
73. strb r3, [fp, #-21]
74. b .L4
75. .L17:
76. mov r3, #70
77. strb r3, [fp, #-21]
78. .L4:
79. ldr r3, [fp, #-16]
80. add r3, r3, #1
81. str r3, [fp, #-16]
82. b .L2
83. .L3:
84. mov r3, #0
85. mov r0, r3
86. sub sp, fp, #12
87. ldmfd sp, {fp, sp, pc}
88. .L20:
89. .align 2
90. .L19:
91. .word 999999
92. .word 1717986919
93. .size main, .-main
94. .ident "GCC: (GNU) 3.4"
```

Fig. 17. B_{linear} code of *EX*1.

```
1.  .file "exp1_hash.s"
2.  .text
3.  .align 2
4.  .global main
5.  .type main, %function
6.  main:
7.  @ args = 0, pretend = 0,
    frame = 12
8.  @ frame_needed = 1,
    uses_anonymous_args = 0
9.  mov ip, sp
10. stmfd sp!, {fp, ip, lr, pc}
11. sub fp, ip, #4
12. sub sp, sp, #12
13. mov r3, #0
14. str r3, [fp, #-16]
15. .L2:
16. ldr r2, [fp, #-16]
17. ldr r3, .L15
18. cmp r2, r3
19. bgt .L3
20. ldr r1, [fp, #-16]
21. ldr r3, .L15+4
22. smull r2, r3, r1, r3
23. mov r2, r3, asr #2
24. mov r3, r1, asr #31
25. rsb r3, r3, r2
26. str r3, [fp, #-20]
27. ldr r3, [fp, #-20]
28. sub r3, r3, #4
29. cmp r3, #6
30. ldrls pc, [pc, r3, asl #2]
31. b .L13
32. .p2align 2
33. .L14:
34. .word .L12
35. .word .L11
36. .word .L10
37. .word .L9
38. .word .L8
39. .word .L7
40. .word .L6
41. .L6:
42. mov r3, #88
43. strb r3, [fp, #-21]
44. b .L4
45. .L7:
46. mov r3, #88
47. strb r3, [fp, #-21]
48. b .L4
49. .L8:
50. mov r3, #65
51. strb r3, [fp, #-21]
52. b .L4
53. .L9:
54. mov r3, #66
55. strb r3, [fp, #-21]
56. b .L4
57. .L10:
58. mov r3, #67
59. strb r3, [fp, #-21]
60. b .L4
61. .L11:
62. mov r3, #68
63. strb r3, [fp, #-21]
64. b .L4
65. .L12:
66. mov r3, #80
67. strb r3, [fp, #-21]
68. b .L4
69. .L13:
70. mov r3, #70
71. strb r3, [fp, #-21]
72. .L4:
73. ldr r3, [fp, #-16]
74. add r3, r3, #1
75. str r3, [fp, #-16]
76. b .L2
77. .L3:
78. mov r3, #0
79. mov r0, r3
80. sub sp, fp, #12
81. ldmfd sp, {fp, sp, pc}
82. .L16:
83. .align 2
84. .L15:
85. .word 999999
86. .word 1717986919
87. .size main, .-main
88. .ident "GCC: (GNU) 3.4.3"
```

Fig. 18. B_{hash} code of *EX*1.

```
1. .file "exp1_binary.s"
2. .text
3. .align 2
4. .global main
5. .type main, %function
6. main:
7. @ args = 0, pretend = 0, frame =
   12
8. @ frame_needed = 1,
   uses_anonymous_args = 0
9. mov ip, sp
10. stmfd sp!, {fp, ip, lr, pc}
11. sub fp, ip, #4
12. sub sp, sp, #12
13. mov r3, #0
14. str r3, [fp, #-16]
15. .L2:
16. ldr r2, [fp, #-16]
17. ldr r3, .L19
18. cmp r2, r3
19. bgt .L3
20. ldr r1, [fp, #-16]
21. ldr r3, .L19+4
22. smull r2, r3, r1, r3
23. mov r2, r3, asr #2
24. mov r3, r1, asr #31
25. rsb r3, r3, r2
26. str r3, [fp, #-20]
27. ldr r3, [fp, #-20]
28. cmp r3,#7
29. blt .L5
30. bgt .L6
31. b .L12
32. .L5:
33. cmp r3,#5
34. blt .L7
35. bgt .L8
36. b .L13
37. .L7:
38. cmp r3,#4
39. blt .L9
40. bgt .L4
41. b .L14
42. .L8:
43. cmp r3,#6
44. bne .L4
45. b .L15
46. .L6:
47. cmp r3,#9
48. blt .L10
49. bgt .L11
50. b .L16
51. .L10:
52. cmp r3,#8
53. bne .L4
54. b .L17
55. .L11:
56. cmp r3,#10
57. bne .L4
58. b .L16
59. .L12:
60. mov r3, #66
61. strb r3, [fp, #-21]
62. b .L4
63. .L13:
64. mov r3, #68
65. strb r3, [fp, #-21]
66. b .L4
67. .L14:
68. mov r3, #80
69. strb r3, [fp, #-21]
70. b .L4
71. .L15:
72. mov r3, #67
73. strb r3, [fp, #-21]
74. b .L4
75. .L16:
76. mov r3, #88
77. strb r3, [fp, #-21]
78. b .L4
79. .L17:
80. mov r3, #65
81. strb r3, [fp, #-21]
82. b .L4
83. .L9:
84. mov r3, #70
85. strb r3, [fp, #-21]
86. .L4:
87. ldr r3, [fp, #-16]
88. add r3, r3, #1
89. str r3, [fp, #-16]
90. b .L2
91. .L3:
92. mov r3, #0
93. mov r0, r3
94. sub sp, fp, #12
95. ldmfd sp, {fp, sp, pc}
96. .L20:
97. .align 2
98. .L19:
99. .word 999999
100.      .word 1717986919
101.      .size main, .-main
102.      .ident "GCC: (GNU) 3.4.3"
```

Fig. 19. B_{binary} code of *EX*1.

```
1. .file "exp5_linear.s"
2. .text
3. .align 2
4. .global main
5. .type main, %function
6. main:
7. @ args = 0, pretend = 0, frame = 12
8. @ frame_needed = 1,
   uses_anonymous_args = 0
9. mov ip, sp
10. stmfd sp!, {fp, ip, lr, pc}
11. sub fp, ip, #4
12. sub sp, sp, #12
13. mov r3, #0
14. str r3, [fp, #-16]
15. .L2:
16. ldr r2, [fp, #-16]
17. mov r3, #996
18. add r3, r3, #3
19. cmp r2, r3
20. bgt .L3
21. mov r3, #0
22. str r3, [fp, #-20]
23. .L5:
24. ldr r2, [fp, #-20]
25. mov r3, #996
26. add r3, r3, #3
27. cmp r2, r3
28. bgt .L4
29. ldr r3, [fp, #-16]
30. cmp r3, #12
31. ble .L8
32. ldr r3, [fp, #-16]
33. cmp r3, #16
34. bgt .L8
35. ldr r3, [fp, #-20]
36. cmp r3, #7
37. ble .L8
38. ldr r3, [fp, #-20]
39. cmp r3, #12
40. bgt .L8
41. mov r3, #1
42. str r3, [fp, #-24]
43. b .L7
44. .L8:
45. ldr r3, [fp, #-16]
46. cmp r3, #5
47. ble .L10
48. ldr r3, [fp, #-16]
49. cmp r3, #12
50. bgt .L10
51. ldr r3, [fp, #-20]
52. cmp r3, #3
53. ble .L10
54. ldr r3, [fp, #-20]
55. cmp r3, #7
56. bgt .L10
57. mov r3, #2
58. str r3, [fp, #-24]
59. b .L7
60. .L10:
61. ldr r3, [fp, #-16]
62. cmp r3, #2
63. ble .L7
64. ldr r3, [fp, #-16]
65. cmp r3, #5
66. bgt .L7
67. ldr r3, [fp, #-20]
68. cmp r3, #0
69. ble .L7
70. ldr r3, [fp, #-20]
71. cmp r3, #3
72. bgt .L7
73. mov r3, #3
74. str r3, [fp, #-24]
75. .L7:
76. ldr r3, [fp, #-20]
77. add r3, r3, #1
78. str r3, [fp, #-20]
79. b .L5
80. .L4:
81. ldr r3, [fp, #-16]
82. add r3, r3, #1
83. str r3, [fp, #-16]
84. b .L2
85. .L3:
86. mov r3, #0
87. mov r0, r3
88. sub sp, fp, #12
89. ldmfd sp, {fp, sp, pc}
90. .size main, .-main
91. .ident "GCC: (GNU) 3.4.3"
```

Fig. 20. B_{linear} code of *EX*2.

```
1.  .file "exp5_binary.s"
2.  .text
3.  .align 2
4.  .global main
5.  .type main, %function
6.  main:
7.  @ args = 0, pretend = 0, frame =12
8.  @ frame_needed = 1,
    uses_anonymous_args = 0
9.  mov ip, sp
10. stmfd sp!, {fp, ip, lr, pc}
11. sub fp, ip, #4
12. sub sp, sp, #12
13. mov r3, #0
14. str r3, [fp, #-16]
15. .L2:
16. ldr r2, [fp, #-16]
17. mov r3, #996
18. add r3, r3, #3
19. cmp r2, r3
20. bgt .L3
21. mov r3, #0
22. str r3, [fp, #-20]
23. .L5:
24. ldr r2, [fp, #-20]
25. mov r3, #996
26. add r3, r3, #3
27. cmp r2, r3
28. bgt .L4
29. ldr r2, [fp, #-16]
30. ldr r3, [fp, #-20]
31. cmp r2, #6
32. bge .L9
33. .L8:
34. cmp r3, #3
35. bgt .L7
36. .L10:
37. cmp r2, #3
38. blt .L7
39. .L11:
40. cmp r3, #1
41. blt .L7
42. mov r3, #3
43. str r3,[fp, #-24]
44. b .L7
45. .L9:
46. cmp r3, #8
47. bge .L13
48. .L12:
49. cmp r2, #12
50. bgt .L7
51. .L14:
52. cmp r3, #4
53. blt .L7
54. mov r3, #2
55. str r3, [fp, #-24]
56. b .L7
57. .L13:
58. cmp r2, #16
59. bgt .L7
60. .L15:
61. cmp r3, #12
62. bgt .L7
63. cmp r2, #13
64. blt .L7
65. mov r3, #1
66. str r3, [fp, #-24]
67. .L7:
68. ldr r3, [fp, #-20]
69. add r3, r3, #1
70. str r3, [fp, #-20]
71. b .L5
72. .L4:
73. ldr r3, [fp, #-16]
74. add r3, r3, #1
75. str r3, [fp, #-16]
76. b .L2
77. .L3:
78. mov r3, #0
79. mov r0, r3
80. sub sp, fp, #12
81. ldmfd sp, {fp, sp, pc}
82. .size main, .-main
83. .ident "GCC: (GNU) 3.4.3"
```

Fig. 21. B_{binary} code of *EX*2.

C.3. Preorder Traversal of k–d Tree

```
Preorder_Traversal_Kd_Tree(Kd_Tree_node *root, int
label)
1.  {
2.    if(root!=null) then
3.      {
4.        if(root is a non leaf node) then
5.          {
6.            sprintf(S, "cmp %s, %d,"
                root → variable_name, root → value);
                write(S);
7.            if(root → right_edge_symbol = '>') then
8.              sprintf(S1, "bgt");
9.            else
10.             sprintf(S1, "bge");
11.           if(all leaf nodes of root node's right
                subtree contain NEXT) then
12.           {
13.             sprintf(S, "%s NEXT," S1);
14.             root->right_tree_visited := true;
15.           }
16.         else
17.           sprintf(S, "%s L%d," S1, 2 * label+1);
18.         write(S);
19.       if(root->left_edge_symbol = '<') then
20.         sprintf(S1, "blt");
21.       else
22.         sprintf(S1, "ble");
23.       if(all leaf nodes of root node's left subtree
            contain NEXT) then
24.       {
25.         sprintf(S, "%s NEXT," S1);
26.         root->left_tree_visited: = true;
27.       }
28.     else
29.       sprintf(S, "%s L%d," S1, 2*label);
30.     write(S);
31.   }
32. else
33.   {
34.     if(root node do not contain NEXT) then
35.       {
36.         sprintf(S, "L%d:," label); write(S);
37.         generate code for the content in the
              leaf node and write it;
38.       }
39.   }
40.   if(root → left_tree_visited = false) then
41.     Preorder_Traversal_Kd_Tree(root →
          left_child, 2*label);
42.   sprintf("L%d:," 2*label + 1);
43.   if(root → right_tree_visited = false) then
44.     Preorder_Traversal_Kd_Tree(root →
          right_child, 2*label + 1);
45. }
46. }
```

References

1. V. Tewari, S. Malik, and A. Wolfe, Compilation techniques for low energy: An overview, *Proceedings of Symposium on Low-Power Electronics*, San Diego, CA, October **(1994)**.
2. J. L. Hennessy and D. A. Patterson, Computer Architecture, A Quantitative Approach, 4th edn., Elsevier.
3. R. A. Sayle, A superoptimizer analysis of multiway branch code generation, *Proceedings of the GCC Developers Summit*, Ottawa, Ontario, Canada, June **(2008)**.
4. H. G. Dietz, Coding multiway branches using customized hash functions, ECE technical reports, School of Electrical Engineering, Purdue University **(1992)**.
5. G.-R. Uh and D. Whalley, Effectively exploiting indirect jumps. *J. Software-Practice and Experience* 3, 1061 **(1999)**.
6. J. L. Bentley, Multidimensional binary search trees used for associative searching, *Proceedings of the ACM's George E. Forsythe Student Paper Competition* **(1975)**.
7. Z. Herczeg, A. Kiss, D. Schmidt, N. Wehn, and T. Gyimothy, XEEMU: An improved xscale power simulator, *Proceedings of the PATMOS*, LNCS **(2007)**.
8. Z. Herczeg, A. Kiss, D. Schmidt, N. Wehn, and T. Gyimothy, Energy simulation of embedded XScale systems with XEEMU, *J. Embedded Computing—PATMOS* 2007 *Selected Papers on Low Power Electronics Archive* **(2009)**, Vol. 3.
9. K. J. Deris and A. Baniasadi, SABA: A zero timing overhead power-aware BTB for high-performance processors, *Proceedings of the Workshop on Unique Chips and Systems* (*UCAS-2*). *in conjunction with IEEE Intel Symp. Performance Analysis of System and Software*, March **(2006)**.
10. B. Zamani, E. Adeli, H. Gharedaghi, and M. Soryani, A novel approach for branch buffer consuming power reduction, *Proceedings of the International Conference on Computer and Electrical Engineering* **(2008)**.
11. W. Shi, T. Zhang, and S. Pande, Static techniques to improve power efficiency of branch predictors, *Proceedings of the ACSAC*, LNCS **(2004)**.
12. Y.-C. Hu, W.-H. Chiao, J. J.-J. Shann, and C.-P. Chung, Low-power branch prediction, *Proceedings of the CDES* **(2005)**.
13. R. Kahn and S. Weiss, Thrifty BTB: A comprehensive solution for dynamic power reduction in branch target buffers, Microprocessors and Microsystems, Elesvier **(2008)**, Vol. 32, pp. 425–436.
14. R. Kahn and S. Weiss, Reducing leakage power with BTB access prediction, Integration, the VLSI journal, Elsevier **(2010)**. Vol. 43, pp. 49–57.
15. N. Levison and S. Weiss, Branch target buffer design for embedded processors, Microprocessors and Microsystems, Elsevier **(2010)**., Vol. 34, pp. 215–227.
16. A. Baniasadi and A. Moshovos, Branch predictor prediction: A power-aware branch predictor for high-performance processors, *Proceedings of the International Conference on Computer Design* **(2002)**.
17. H. Sato and T. Sato, A static and dynamic energy reduction technique for *i*-cache and BTB in embedded processors, *Proceedings of the Asia and South Pacific Design Automation Conference* **(2004)**.
18. N. Tomas, J. Sahuquillo, S. Petit, and P. Lopez, Reducing the number of bits in the BTB to attack the branch predictor hot-spot, *Proceedings of the* 14*th International Euro-Par conference on Parallel Processing* **(2008)**.
19. N. Levison and S. Weiss, Low power branch prediction for embedded application processors, *Proceedings of the ISLPED*, Austin, Texas, USA, August **(2010)**.
20. K. J. Deris and A. Baniasadi, Branchless cycle prediction for embedded processors, *Proceedings of the SAC*, Dijon, France, April **(2006)**.
21. C. H. Kim, S. W. Chung, and C. S. Jhon, A power-aware branch predictor by accessing BTB selectively. *J. Computer Science and Technology* 20, 607 **(2005)**.
22. M. Briejer, C. Meenderinck, and B. Juurlink, Extending the cell SPE with energy efficient branch prediction, *Proceedings of the* 16*th international Euro-Par Conference on Parallel Processing* **(2010)**.
23. H. Yang, G. R. Gao, A. Marquez, G. Cai, and Z. Hu, Power and energy impact by loop transformations, *Proceedings of the Workshop on Compilers and Operating Systems for Low Power, Parallel Architecture and Compilation Techniques* **(2001)**.
24. Y. K. Zhu, G. Magklis, Michael, L. Scott, C. Ding, and D. H. Albonesi, The energy impact of aggressive loop fusion, *Proceedings of the* 13*th International Conference on Parallel Architecture and Compilation Techniques* **(2004)**.
25. T. Burd and R. Brodersen, Design issues for dynamic voltage scaling, *Proceedings of the International Symposium on Low Power Electronics and Design*, June **(2000)**.
26. Technical Summary of Intel XScale Microarchitecture, http://int.xscale-freak.com/XSDoc/PXA27X/XScaleDatasheet4.pdf.

Sumanta Pyne

Sumanta Pyne *is a research scholar in the Department of Computer Science and Engineering at Indian Institute of Technology, Kharagpur, since January 2010. He is pursuing Ph.D. under the guidance of Professor Ajit Pal. His current area of research is Power Aware Software. His research interests include Low Power issues of Multicore Processors, Computer Architecture, Compilers, Network Mobility. Prior to joining IIT Kharagpur, Sumanta did is Master of Engineering in Computer Science and Engineering in the year 2009 from Bengal Engineering and Science University, Shibpur (formerly Bengal Engineering College, Shibpur). He did his schooling from Birla High School (formerly Hindi High School) graduated from Meghnad Saha Institute of Technology in the year 2005. He started his professional career as a programmer at Hi-Q Solutions, Kolkata and then worked as a lecturer at Techno India College of Technology, Kolkata.*

Ajit Pal

Ajit Pal *is currently a Professor in the Department of Computer Science and Engineering at Indian Institute of Technology Kharagpur. He received his M.Tech. and Ph.D. degrees for the Institute of Radio Physics and Electronics, Calcutta University in 1971 and 1976, respectively. Before joining IITKGP in the year 1982, he was with Indian Statistical Institute (ISI), Calcutta, Indian Telephone Industries (ITI), Naini and Defense Electronics Research Laboratory (DLRL), Hyderabad in various capacities. He became full Professor in 1988 and served as Head of Computer Center from 1993 to 1995 and Head of the Computer Science and Engineering Department from 1995 to 1998. His research interests include Embedded Systems, Low-power VLSI Circuits, Sensor Networks and Optical Communication. He is the principal investigator of several Sponsored Research Projects including 'Low Power Circuits' sponsored by Intel, USA and 'Formal methods for power intent verification,' sponsored by Synopsis (India) Pvt. Ltd. He has over 135 publications in reputed journals and conference proceedings and two books entitled 'Microprocessors: Principles and Applications' published by TMH (1990) and 'Microcontrollers: Principles and Applications' published by PHI (2011). He is the Fellow of the IETE, India and Senior Member of the IEEE, USA.*