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. 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.

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

As $T_{\text{MB}}$, utilizes either a $B_{\text{binary}}$ or a $B_{\text{hash}}$, its execution time is smaller than that of $B_{\text{linear}}$. 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. 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

#### Table I. Voltage-frequency pairs supported by XScale.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$v_i$ (Volt)</th>
<th>$f_i$ (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.5</td>
<td>733</td>
</tr>
<tr>
<td>2</td>
<td>1.4</td>
<td>666</td>
</tr>
<tr>
<td>3</td>
<td>1.3</td>
<td>600</td>
</tr>
<tr>
<td>4</td>
<td>1.2</td>
<td>533</td>
</tr>
<tr>
<td>5</td>
<td>1.1</td>
<td>466</td>
</tr>
<tr>
<td>6</td>
<td>1.0</td>
<td>400</td>
</tr>
<tr>
<td>7</td>
<td>1.0</td>
<td>333</td>
</tr>
<tr>
<td>8</td>
<td>1.0</td>
<td>266</td>
</tr>
<tr>
<td>9</td>
<td>1.0</td>
<td>200</td>
</tr>
</tbody>
</table>
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 respond 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 not taken, 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 $MB_i$ as input and produces $TMB_i$ as output. Figure 3(a) shows the format of an $MB_i$. Here $MB_i$ 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 $MB_i$. 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 $MB_i$ 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 $TMB_i$ 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 $TMB_i$ at $(vf_{min_{vf\_pair}} \leq P)$ will minimize the energy consumed by it. In Figure 3(c) $P[min_{vf\_pair}]$, $P[min_{vf\_pair}-1]$, $P[min_{vf\_pair}-2], \ldots, P[2]$ are the minimum values

![Fig. 3. Format of the multiway branch (MB) and translated multiway branch (TMB) codes.](image)
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 1 the MB can be implemented as \(B_{\text{linear}}\), \(B_{\text{hash}}\) or \(B_{\text{binary}}\). For the illustrative example 2 compilers generate \(B_{\text{linear}}\). It is hard to implement \(B_{\text{hash}}\) for EX2. In this case, \(k-d\) tree\(^6\) is used to implement \(B_{\text{binary}}\) for EX2. The illustrative example 3 EX3 deals with an MB where the index variables and values are strings. The branching takes place on string matching. The compilers generate \(B_{\text{linear}}\) for such MBs. Here it introduces the use of pattern matcher to generate time and energy efficient \(B_{\text{pattern}}\) code for EX3. The assembly language used in this paper is based on the instruction set of XScale processor. The assembly language codes for \(B_{\text{linear}}\) and \(B_{\text{hash}}\) are generated by xscale-gcc-elf compiler. The \(B_{\text{linear}}\) and \(B_{\text{pattern}}\) 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 \(B_{\text{linear}}\), \(B_{\text{hash}}\), \(B_{\text{binary}}\), and \(B_{\text{pattern}}\) 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 (\(\mu\)J).

The tables also show the performance and energy gained by \(B_{\text{hash}}\), \(B_{\text{binary}}\) and \(B_{\text{pattern}}\) implementations with respect to \(B_{\text{linear}}\) 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)

EX1 considers a simple MB which can implemented as \(\text{if-then-else}\) and \(\text{switch-case}\), 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 \(v/lparenorivmin\) \(v/commaorif/rparenori\) \(v/commaorif/rparenori\) \(v/commaorif/rparenori\) \(v/commaorif/rparenori\). For this example, xscale-gcc-elf translates the source code in Figure 4(a) to \(B_{\text{linear}}\) code. For the source code in Figure 4(b) the xscale-gcc-elf generates \(B_{\text{hash}}\) code.
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 \( B_{\text{binary}} \) code for a MB. The MB in EX1 can be translated to \( B_{\text{binary}} \) 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. \( B_{\text{binary}} \) is generated by preorder traversal of the binary search tree. For a MB with \( n \) branch destinations belonging to the class of EX1, \( B_{\text{linear}} \) will take \( O(n) \) time to jump to a branch destination. While \( B_{\text{hash}} \) and \( B_{\text{binary}} \) will take \( O(1) \) and \( O(\log n) \) time, respectively. The \( B_{\text{linear}}, B_{\text{hash}}, \) and \( B_{\text{binary}} \) codes of EX1 are shown in the Appendix A. Table III compare the energy and performance of the different implementations of the EX1 and show the values of the BTB parameters. It also shows the energy and performance gained by \( B_{\text{hash}} \) and \( B_{\text{binary}} \) with respect to \( B_{\text{linear}} \). The execution time of \( B_{\text{linear}} \) at \((v_i, f_1)\) is considered as the deadline for \( B_{\text{hash}} \) and \( B_{\text{binary}} \) to finish execution. VFS is applied to \( B_{\text{hash}} \) and \( B_{\text{binary}} \) to minimize energy consumption.

### 3.1.2. Illustrative Example 2 (EX2)

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 \( B_{\text{hash}} \) code. Compilers generate \( B_{\text{linear}} \) 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 \( B_{\text{linear}} \) code for such MBs. This is done with the help of k-d tree. k-d tree is a multidimensional binary search tree. The matching value set for index variable ‘x’ is the \( \text{value}(x) = \{3, 5, 6, 12, 13, 16\} \). The matching value set for index variable ‘y’ \( \text{value}(y) = \{1, 3, 4, 7, 8, 12\} \). The ordered pair set or point set of matching values is \( \text{value}(x, y) = \{(3, 1), (5, 3), (5, 1), (5, 3), (6, 4), (6, 7), (12, 4), (12, 7), (13, 8), (13, 12), (16, 8), (16, 12)\} \), as obtained from the source code in Figure 6. The k-d tree decomposition for the point set \( \text{value}(x, y) \) (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 \( \text{value}(x, y) \) is shown in Figures 8. In Figure 7 lines \( l_1, l_2, l_{10} \) and \( l_{14} \) encloses the region related to \( BD_1 \). The lines \( l_3, l_4, l_5 \) and \( l_6 \) enclose the region related to \( BD_2 \). The lines \( l_7, l_{12}, l_8 \) and \( l_9 \) enclose the region related to \( BD_3 \). The rest of the regions are related to NEXT.

### Table IV. EX2 results.

<table>
<thead>
<tr>
<th>EX2 code</th>
<th>Metric</th>
<th>Value</th>
<th>Gain (%)</th>
<th>BTB parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B_{\text{linear}} ) at ((v_i, f_1))</td>
<td>Time (sec)</td>
<td>0.0560</td>
<td>–</td>
<td>Total branches</td>
<td>8008955</td>
</tr>
<tr>
<td></td>
<td>Total energy (J)</td>
<td>0.0411</td>
<td>–</td>
<td>Miss predictions taken</td>
<td>1085</td>
</tr>
<tr>
<td></td>
<td>BTB energy ((\mu J))</td>
<td>314.225</td>
<td>–</td>
<td>Miss predictions not taken</td>
<td>56</td>
</tr>
<tr>
<td>( B_{\text{linear}} ) at ((v_i, f_1))</td>
<td>Time (sec)</td>
<td>0.0273</td>
<td>51.25</td>
<td>Total branches</td>
<td>5008198</td>
</tr>
<tr>
<td></td>
<td>Total energy (J)</td>
<td>0.0213</td>
<td>48.17</td>
<td>Miss predictions taken</td>
<td>3071</td>
</tr>
<tr>
<td></td>
<td>BTB energy ((\mu J))</td>
<td>196.578</td>
<td>37.44</td>
<td>Miss predictions not taken</td>
<td>2040</td>
</tr>
<tr>
<td>( B_{\text{linear}} ) at ((v_i, f_1))</td>
<td>Time (sec)</td>
<td>0.0430</td>
<td>23.21</td>
<td>Total branches</td>
<td>5008198</td>
</tr>
<tr>
<td></td>
<td>Total energy (J)</td>
<td>0.0106</td>
<td>74.20</td>
<td>Miss predictions taken</td>
<td>3071</td>
</tr>
<tr>
<td></td>
<td>BTB energy ((\mu J))</td>
<td>79.682</td>
<td>74.64</td>
<td>Miss predictions not taken</td>
<td>2040</td>
</tr>
</tbody>
</table>

### Table V. EX3 results.

<table>
<thead>
<tr>
<th>EX2 code</th>
<th>Metric</th>
<th>Value</th>
<th>Gain (%)</th>
<th>BTB parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B_{\text{linear}} ) at ((v_i, f_1))</td>
<td>Time (sec)</td>
<td>0.7771</td>
<td>–</td>
<td>Total branches</td>
<td>7193373</td>
</tr>
<tr>
<td></td>
<td>Total energy (J)</td>
<td>0.5900</td>
<td>–</td>
<td>Miss predictions taken</td>
<td>3866738</td>
</tr>
<tr>
<td></td>
<td>BTB energy ((\mu J))</td>
<td>2944.25</td>
<td>–</td>
<td>Miss predictions not taken</td>
<td>3666680</td>
</tr>
<tr>
<td>( B_{\text{linear}} ) at ((v_i, f_1))</td>
<td>Time (sec)</td>
<td>0.6695</td>
<td>13.84</td>
<td>Total branches</td>
<td>33133409</td>
</tr>
<tr>
<td></td>
<td>Total energy (J)</td>
<td>0.4969</td>
<td>15.90</td>
<td>Miss predictions taken</td>
<td>3466753</td>
</tr>
<tr>
<td></td>
<td>BTB energy ((\mu J))</td>
<td>1532.97</td>
<td>47.93</td>
<td>Miss predictions not taken</td>
<td>1066672</td>
</tr>
<tr>
<td>( B_{\text{linear}} ) at ((v_i, f_1))</td>
<td>Time (sec)</td>
<td>0.7364</td>
<td>5.23</td>
<td>Total branches</td>
<td>33133409</td>
</tr>
<tr>
<td></td>
<td>Total energy (J)</td>
<td>0.4246</td>
<td>28.14</td>
<td>Miss predictions taken</td>
<td>3466753</td>
</tr>
<tr>
<td></td>
<td>BTB energy ((\mu J))</td>
<td>1267.61</td>
<td>56.94</td>
<td>Miss predictions not taken</td>
<td>1066672</td>
</tr>
<tr>
<td>( B_{\text{linear}} ) at ((v_i, f_1))</td>
<td>Time (sec)</td>
<td>0.0273</td>
<td>51.25</td>
<td>Total branches</td>
<td>1532.97</td>
</tr>
<tr>
<td></td>
<td>Total energy (J)</td>
<td>0.0213</td>
<td>48.17</td>
<td>Miss predictions taken</td>
<td>3071</td>
</tr>
<tr>
<td></td>
<td>BTB energy ((\mu J))</td>
<td>196.578</td>
<td>37.44</td>
<td>Miss predictions not taken</td>
<td>2040</td>
</tr>
</tbody>
</table>
Sumanta Pyne and Ajit Pal

Branch Target Buffer Energy Reduction Through Efficient Multiway Branch Translation Techniques

Fig. 4. Two possible source codes of EX1.

The left edge either labeled with the symbol ‘<’ or with ‘≤’. The right edge is either labeled with the symbol ‘>’ or with ‘≥’. 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 ‘≥’. 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 \( BD_1 \), the following

Fig. 5. Binary search tree for EX1.

Fig. 6. Source code of EX2 as if-then-else ladder.
sequences of conditions are to be satisfied, ‘\(x < 6\),’ ‘\(y < 3\),’ ‘\(x \geq 3\),’ ‘\(y \geq 1\)’ and ‘\(x \leq 5\).’ 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 \(l_1\) and \(l_6\) contain \textsc{next}. Each of these subtrees are pruned and replaced with a leaf node containing \textsc{next} as shown in Figure 9. Figure 10 shows two possible assembly language implementations of the \textit{if-then-else ladder} in \textit{EX2}. These assembly language code fragments are written using ARM instruction set. \(B_{\text{linear}}\) in Figure 10(a) is a brute-force implementation. \(B_{\text{binary}}\) 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 \(B_{\text{linear}}\) and \(B_{\text{binary}}\) implementations of \textit{EX2} are shown in Figures 20 and 21, respectively, in Appendix B. For a \(MB\) with \(n\) branch destinations belonging to the class of \textit{EX2} having \(d\) distinct index variables in each of the \(n\) index expressions, \(B_{\text{linear}}\) will take \(O(2 \times d \times n)\) time to jump to a branch destination. While \(B_{\text{binary}}\) will take \(O(\log_2 n + d + 1)\) time, where \(d\) is the dimension of the \(k-d\) tree. In \textit{EX2} \(d = 2\). Table IV compares the energy and performance of the different implementations of \textit{EX2} and shows the values of BTB parameters. It also shows the gain achieved by \(B_{\text{binary}}\) with respect to \(B_{\text{linear}}\). The execution time of \(B_{\text{linear}}\) at \((v_1, f_1)\) is considered as the \(T_{\text{linear}}\), which is the deadline for \(B_{\text{binary}}\) to finish execution. VFS is applied to \(B_{\text{binary}}\) to minimize energy consumption maintaining the constraint \(T_{\text{binary}} \leq T_{\text{linear}}\). Table IV shows the maximum gain in BTB energy achieved by \(B_{\text{binary}}\) is 74.64% along with a performance gain of 23.21%, when \(B_{\text{binary}}\) is executed at \((v_5, f_5)\).
3.1.3. Illustrative Example 3 (EX3)

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

3.2. BTB Energy Reduction Algorithm

This algorithm takes \(MB_i\) as input and produces \(TMB_i\) as output. The \(MB_i\) taken as input is considered to be implemented as \(B_{\text{linear}}\) code. \(B_{\text{translated}}\) code in \(TMB_i\) is either a \(B_{\text{hash}}\) code or a \(B_{\text{binary}}\) code or a \(B_{\text{pattern}}\) code. After translating the code from \(B_{\text{linear}}\) code to \(B_{\text{hash}}\) or \(B_{\text{binary}}\) or \(B_{\text{pattern}}\), 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 \(TMB_i\) such that \(T_{\text{translated}} \leq \text{deadline}\).

**BTB_Energy_Reduction_Algorithm**

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

Fig. 9. Modified \(k-d\) tree of the for the point set \(value(x, y)\).
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 \(TMB\). The VFS algorithms calculate the energy overhead \(E_{\text{overhead}}\) and time overhead \(t_{\text{overhead}}\) due to VFS. They are calculated using the following formulae.\(^{25}\)

Overheads when switching from \((v_i, f_i)\) to \((v_w, f_w)\)

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

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

where, \(\mu\) is the energy efficiency of the energy regulator which is considered as 90\%, \(C\) is the voltage regulator’s capacitance to be 10 \(\mu\)F, \(I_{\text{MAX}}\) 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 \texttt{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 \(TMB\) when \(p\) is input independent.

VFS_Algorithm_A\((B_{\text{linear}}, B_{\text{translated}})\)

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

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

Fig. 11. Multiway branch with strings.
17. \[ E_{\text{min}} := E[i]; \]
18. \[ \text{min}_{\text{vf pair}} := i; \]
19. \}
20. \}
21. \}
22. if (\text{min}_{\text{vf pair}} > 1) then
23. \}
24. \}
25. \}
26. \}
27. \}
28. if (\text{min}_{\text{vf pair}} > 1) then
29. \}
30. \}
31. \}
32. \}
33. \}

Here, \( t_{\text{linear},j} \) and \( t_{\text{branch},j} \) are time taken to jump to BD\(_j\) and execute BC\(_i\) of B\(_{\text{linear}}\), respectively, at \((v_i, f_i)\). \( e_{\text{linear},j} \) and \( e_{\text{branch},j} \) are energy consumed to jump to BD\(_j\) and execute BC\(_i\) of B\(_{\text{linear}}\), respectively, at \((v_i, f_i)\). \( t_{\text{translated},ij} \) and \( t_{\text{branch},exe,ij} \) are time taken to jump to BD\(_j\) and execute BC\(_i\) of B\(_{\text{translated}}\), respectively, at \((v_i, f_i)\). \( e_{\text{translated},ij} \) and \( e_{\text{branch},exe,ij} \) are energy consumed to jump to BD\(_j\) and execute BC\(_i\) of B\(_{\text{translated}}\), respectively at \((v_i, f_i)\). In steps 4 and 5, \( T_{\text{linear}} \) and \( E_{\text{linear}} \) 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\(_i\) at \((v_i, f_i)\). Similarly, for B\(_{\text{translated}}\), \( T[i] \) and \( E[i] \) are calculated in steps 11 and 12. The algorithm finds the value of \( \text{min}_{\text{vf pair}} \), the \((v, f)\) that will minimize energy consumed by TMB\(_i\) and allow the execution of the TMB\(_i\) to finish within the deadline.

### 3.3.2. VFS Algorithm B

VFS Algorithm B finds the possibility of VFS to save energy of TMB\(_i\) when \( p \) is dependent. \( T_{\text{linear}} \), \( E_{\text{linear}} \), \( T[i] \) and \( E[i] \) are calculated in a similar way as in VFS Algorithm A.

**VFS Algorithm B**

\[
\begin{align*}
1. & \quad \{ \\
2. & \quad \text{char S[50];} \\
3. & \quad p := 10^6; \\
4. & \quad \text{Linked List linkedlist := null;} \\
5. & \quad T_{\text{linear}} := \frac{1}{n} \sum_{j=1}^{n} (t_{\text{linear},j} + t_{\text{branch,exe},j}) \times p; \\
6. & \quad E_{\text{linear}} := \frac{1}{n} \sum_{j=1}^{n} (e_{\text{linear},j} + e_{\text{branch,exe},j}) \times p; \\
7. & \quad \text{deadline := } T_{\text{linear}}; \\
8. & \quad t_{\text{lin,avg}} := \frac{p}{T_{\text{linear}}}; \\
9. & \quad E_{\text{min}} := E_{\text{linear}}; \\
10. & \quad \text{min}_{\text{vf pair}} := 1; \\
11. & \quad \text{for}(i := 1; i \leq 9; i++) \\
12. & \quad \{ \\
13. & \quad t_i := \frac{1}{n} \sum_{j=1}^{n} (t_{\text{translated},ij} + t_{\text{branch,exe,ij}}); \\
14. & \quad T[i] := 2 \times t_{\text{overhead}}(1,i) + t_i \times p; \\
15. & \quad E[i] := 2 \times E_{\text{overhead}}(1,i) \times \frac{1}{n} \\
16. & \quad \times \sum_{j=1}^{n} (e_{\text{translated},ij} + e_{\text{branch,exe,ij}}) \times p; \\
17. & \quad \text{if } (t_{\text{lin,avg}} > t_i) \text{ then} \\
18. & \quad \{ \\
19. & \quad P[i] := \left[ \frac{2 \times t_{\text{overhead}}(1,i)}{t_{\text{lin,avg}} - t_i} \right]; \\
20. & \quad \} \\
21. & \quad \text{if } (T[i] \leq \text{deadline}) \text{ then} \\
22. & \quad \{ \\
23. & \quad \text{if } (E[i] < E_{\text{min}}) \text{ then} \\
24. & \quad \{ \\
25. & \quad E_{\text{min}} := E[i]; \\
26. & \quad \text{min}_{\text{vf pair}} := i; \\
27. & \quad \text{if } (i > 1) \text{ then} \\
28. & \quad \{ \\
29. & \quad L := \text{create_node}(); \\
30. & \quad L \rightarrow \text{vf pair} := i; \\
31. & \quad \text{linkedlist.addfirst}(L); \\
32. & \quad \} \\
33. & \quad \} \\
34. & \quad \} \\
35. & \quad \} \\
36. & \quad \} 
\end{align*}
\]

Fig. 12. Pattern matcher for multiway branch with strings (EX3).
37. for(L := linkedlist.header_node();
    L != null; L := L -> next_node)
    {
        sprintf(S,"if (p >= %d)\{setFrequency(%d);
        setVoltage(%d)\} \{P[L] \rightarrow vf_pair\},
        \Leftrightarrow L \rightarrow vf_pair, L \rightarrow vf_pair\};
        generate_code(S);
        if (L -> next_node != null) then
            {
                sprintf(S,"else\} \{P[last_node] \rightarrow vf_pair\};
                generate_code(S);
            }
        last_node := L;
        generate_code(B_tranional);
        if (min_vf_pair > 1) then
            {
                sprintf(S,"if (p >=%d)\{setVoltage(1);
                setFrequency(1)\}\{P[min_vf_pair] \rightarrow vf_pair\};
                generate_code(S);
            }
        }
    }

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_j). P[i] is the minimum value of p required to execute TMB_i at (v_i, f_j). 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_i, f_j). Steps 8 and 15 of VFS_Algorithm_B calculates t_{lin_avg} and t_i, respectively, when, t_{lin_avg} > t_i. If B_tranional is executed P[i] times at (v_i, f_j), then the time taken to do this should be at most that of P[i] time execution of B_{linear} at (v_i, f_j). Considering the overhead of (v_i, f_j) scale up and scale down, this can be written as

\[
P[i] \times t_i + 2 \times t_{overhead}(1, i) \leq P[i] \times t_{lin_avg}
\]

\[
P[i] \times (t_{lin_avg} - t_i) \geq 2 \times t_{overhead}(1, i)
\]

\[
P[i] \geq \frac{2 \times t_{overhead}(1, i)}{(t_{lin_avg} - t_i)}
\]

(3)

The obtained expression of P[i] is the minimum value of p required to execute TMB_i at (v_i, f_j). In other words, if the value of p is obtained at runtime and p \geq P[i] then TMB_i can be executed at (v_i, f_j). 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] > ... > P[2] \geq 1. The reason behind this is, as (v_i, f_j) decreases, P[i] increases. After the formation of the linked list the algorithm generates the TMB_i 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_tranional. TMB in Figure 14(b) contains a switch-case for which compiler generates B_tranional. 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_i, f_j) 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 \mu s. Table VI shows the values of t_i. T_{MB}(v_i, f_j) required to calculate P[i], T_{ov}(v_i, f_j) is the time taken to scale down from (v_i, f_j) to (v_i, f_j) and scale up from (v_i, f_j) to (v_i, f_j). Table VI also shows the values of T_{MB}(v_i, f_j), E_{MB}(v_i, f_j), T_{ov}(v_i, f_j), and E_{ov}(v_i, f_j). These values are obtained for experimental verification of VFS_Algorithm_B. T_{MB}(v_i, f_j) and E_{MB}(v_i, f_j) are the time taken and energy consumed, respectively, by MB in Figure 14(a), when it is executed at (v_i, f_j) and p = P[i]. T_{ov}(v_i, f_j) and E_{ov}(v_i, f_j) are time taken and energy consumed, respectively, by TMB in Figure 14(b), when it is executed at (v_i, f_j) and p = P[i]. T_{ov}(v_i, f_j)
Sumanta Pyne and Ajit Pal  
Branch Target Buffer Energy Reduction Through Efficient Multiway Branch Translation Techniques

is the deadline for $TMB$ in Figure 14(b). In Table VI $T_{MB(i),i,P(i)} \leq T_{MB(i),i,P(i)}$ 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.](image)

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. 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.](image)

<table>
<thead>
<tr>
<th>$i$</th>
<th>$t_i$ ($\mu$sec)</th>
<th>$T_{ov,i}$ ($\mu$sec)</th>
<th>$P[i]$</th>
<th>$T_{MB(i),i,P[i]}$ ($\mu$sec)</th>
<th>$E_{MB(i),i,P[i]}$ ($\mu$J)</th>
<th>$T_{TMB(i),i,P[i]}$ ($\mu$sec)</th>
<th>$E_{TMB(i),i,P[i]}$ ($\mu$J)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.0495</td>
<td>60.49</td>
<td>1795</td>
<td>158.51</td>
<td>115.11</td>
<td>156.69</td>
<td>58.11</td>
</tr>
<tr>
<td>3</td>
<td>0.0550</td>
<td>60.46</td>
<td>2144</td>
<td>186.63</td>
<td>136.34</td>
<td>185.27</td>
<td>60.38</td>
</tr>
<tr>
<td>4</td>
<td>0.0619</td>
<td>60.48</td>
<td>2840</td>
<td>242.90</td>
<td>178.67</td>
<td>242.24</td>
<td>65.57</td>
</tr>
<tr>
<td>5</td>
<td>0.0707</td>
<td>60.53</td>
<td>4843</td>
<td>406.21</td>
<td>300.50</td>
<td>405.15</td>
<td>86.87</td>
</tr>
</tbody>
</table>
Branch Target Buffer Energy Reduction Through Efficient Multiway Branch Translation Techniques  

Sumanta Pyne and Ajit Pal

Table VII. Benchmark programs.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL</td>
<td>Evaluates proposition logic formula</td>
<td>cse.iitkgp.ac.in/~spyne/benchmarks</td>
</tr>
<tr>
<td>LR</td>
<td>A shift-reduce bottom up parser</td>
<td>cse.iitkgp.ac.in/~spyne/benchmarks</td>
</tr>
<tr>
<td>GUI</td>
<td>A GUI controller of a database system</td>
<td><a href="http://www.javaboutique.internet.com/PacMan">www.javaboutique.internet.com/PacMan</a></td>
</tr>
<tr>
<td>P’man</td>
<td>Pacman, a computer game</td>
<td><a href="http://www.caspercomsci.com/pages/javasource.htm">www.caspercomsci.com/pages/javasource.htm</a></td>
</tr>
<tr>
<td>Chess</td>
<td>A computer game</td>
<td><a href="http://www.caspercomsci.com/pages/javasource.htm">www.caspercomsci.com/pages/javasource.htm</a></td>
</tr>
<tr>
<td>B’ship</td>
<td>Battleship, a computer game</td>
<td><a href="http://www.caspercomsci.com/pages/javasource.htm">www.caspercomsci.com/pages/javasource.htm</a></td>
</tr>
<tr>
<td>M’Conv</td>
<td>The mode converter, does base conversion of numbers</td>
<td><a href="http://www.caspercomsci.com/pages/javasource.htm">www.caspercomsci.com/pages/javasource.htm</a></td>
</tr>
<tr>
<td>B’Jack</td>
<td>The black Jack, a computer game</td>
<td><a href="http://www.caspercomsci.com/pages/javasource.htm">www.caspercomsci.com/pages/javasource.htm</a></td>
</tr>
</tbody>
</table>

Table VIII. Benchmark result.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Metric</th>
<th>Naïve code</th>
<th>Translated code</th>
<th>Gain (%)</th>
<th>Type of k–d tree</th>
<th>#k–d tree</th>
<th>Size range</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL</td>
<td>Time (sec)</td>
<td>1.7213</td>
<td>1.589</td>
<td>7.68</td>
<td>1D</td>
<td>16</td>
<td>5–11</td>
</tr>
<tr>
<td></td>
<td>Total energy (J)</td>
<td>1.243</td>
<td>1.153</td>
<td>7.24</td>
<td>2D</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>BTB energy (μJ)</td>
<td>5746</td>
<td>1187</td>
<td>79.34</td>
<td>3D</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>LR</td>
<td>Time (sec)</td>
<td>1.1119</td>
<td>0.998</td>
<td>10.24</td>
<td>1D</td>
<td>10</td>
<td>3–11</td>
</tr>
<tr>
<td></td>
<td>Total energy (J)</td>
<td>0.8386</td>
<td>0.7824</td>
<td>6.50</td>
<td>2D</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>BTB energy (μJ)</td>
<td>3380</td>
<td>989</td>
<td>70.73</td>
<td>3D</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>GUI</td>
<td>Time (sec)</td>
<td>1.1504</td>
<td>1.1297</td>
<td>1.79</td>
<td>1D</td>
<td>5</td>
<td>5–10</td>
</tr>
<tr>
<td></td>
<td>Total energy (J)</td>
<td>1.8761</td>
<td>1.6572</td>
<td>11.66</td>
<td>2D</td>
<td>2</td>
<td>10–35</td>
</tr>
<tr>
<td></td>
<td>BTB energy (μJ)</td>
<td>3753</td>
<td>1023</td>
<td>72.74</td>
<td>3D</td>
<td>4</td>
<td>4–12</td>
</tr>
<tr>
<td>P’man</td>
<td>Time (sec)</td>
<td>2.175</td>
<td>1.962</td>
<td>9.79</td>
<td>1D</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Total energy (J)</td>
<td>2.185</td>
<td>1.1787</td>
<td>4.69</td>
<td>2D</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>BTB energy (μJ)</td>
<td>7054</td>
<td>2987</td>
<td>57.65</td>
<td>3D</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Chess</td>
<td>Time (sec)</td>
<td>3.154</td>
<td>2.942</td>
<td>6.72</td>
<td>1D</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Total energy (J)</td>
<td>1.725</td>
<td>1.589</td>
<td>7.42</td>
<td>2D</td>
<td>3</td>
<td>3–8</td>
</tr>
<tr>
<td></td>
<td>BTB energy (μJ)</td>
<td>9432</td>
<td>4763</td>
<td>49.50</td>
<td>3D</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>B’Ship</td>
<td>Time (sec)</td>
<td>3.187</td>
<td>3.082</td>
<td>3.29</td>
<td>1D</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>Total energy (J)</td>
<td>1.472</td>
<td>1.386</td>
<td>5.84</td>
<td>2D</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>BTB energy (μJ)</td>
<td>5087</td>
<td>3986</td>
<td>21.64</td>
<td>3D</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>M’Conv</td>
<td>Time (sec)</td>
<td>0.482</td>
<td>0.374</td>
<td>22.40</td>
<td>1D</td>
<td>2</td>
<td>26–32</td>
</tr>
<tr>
<td></td>
<td>Total energy (J)</td>
<td>0.6427</td>
<td>0.6182</td>
<td>3.81</td>
<td>2D</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>BTB energy (μJ)</td>
<td>4876</td>
<td>1928</td>
<td>60.45</td>
<td>3D</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>B’Jack</td>
<td>Time (sec)</td>
<td>3.327</td>
<td>3.162</td>
<td>2.91</td>
<td>1D</td>
<td>3</td>
<td>3–13</td>
</tr>
<tr>
<td></td>
<td>Total energy (J)</td>
<td>1.876</td>
<td>1.677</td>
<td>10.6</td>
<td>2D</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>BTB energy (μJ)</td>
<td>14872</td>
<td>6748</td>
<td>54.62</td>
<td>3D</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

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. 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.
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. Naive Code is the code generated by the compiler xscale-elf-gcc, which contains mainly B_{hash} implementations of MBs. Translated code contains possible B_{hash} and B_{linear} as TMBs. 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 MBs.

APPENDIX

A. B_{linear}, B_{hash} and B_{binary} Implementations of EX1

The B_{linear} and B_{hash} codes of EX1 are generated by xscale-elf-gcc compiler. Step 30 of B_{hash} code of EX1 in Figure 18 shows the application of hashing. The B_{binary} code of EX1 is generated by preorder traversal of the binary search tree in Figure 5.

B. B_{linear} and B_{binary} Implementations of EX2

The B_{linear} code of EX2 is generated by xscale-elf-gcc compiler. The B_{binary} code of EX2 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. }
```
Branch Target Buffer Energy Reduction Through Efficient Multiway Branch Translation Techniques

Sumanta Pyne and Ajit Pal

16

J. Low Power Electron. 8, 1–20, 2012
Sumanta Pyne and Ajit Pal
Branch Target Buffer Energy Reduction Through Efficient Multiway Branch Translation Techniques

Fig. 19. $B_{binary}$ code of $EX_1$.

Fig. 20. $B_{linear}$ 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, “L%d:,” label); write(S);
14.               }
15.             if(all leaf nodes of root node’s left subtree contain NEXT) then
16.               {
17.                 sprintf(S, “L%d:,” label); write(S);
18.               }
19.             if(root = left_tree_visited = true) then
20.               Preorder_Traversal_Kd_Tree(root = left_child, 2*label);
21.             else
22.               Preorder_Traversal_Kd_Tree(root = right_child, 2*label+1);
23.         } else
24.         if(all leaf nodes of root node’s left subtree contain NEXT) then
25.             {
26.               sprintf(S, “%s NEXT,” S1);
27.               root = left_tree_visited = true;
28.             }
29.         else
30.         {
31.             sprintf(S, “%s L%d,” S1, 2*label);
32.             write(S);
33.             if(root node do not contain NEXT) then
34.             {
35.               sprintf(S, “L%d:,” label); write(S);
36.               generate code for the content in the leaf node and write it;
37.             }
38.         }
39.     }
40.   }
41. }
42. }
43. }
44. }
45. }
46. }

Branch Target Buffer Energy Reduction Through Efficient Multiway Branch Translation Techniques

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 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 Mehdad 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.

References

4. H. G. Dietz, Coding multiway branches using customized hash functions, ECE technical reports, School of Electrical Engineering, Purdue University (1992).
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 IIT KGP 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.