

Low-power branch target buffer for application-specific embedded processors

P. Petrov and A. Orailoglu

Abstract: A methodology for a low-power branch identification mechanism which enables the design of extremely power-efficient branch predictors for embedded processors is presented. The proposed technique utilises application-specific information regarding the control-flow structure of the program's major loops. Such information is used to completely eliminate the power hungry branch target buffer (BTB) lookups which normally occur at every execution cycle. Exact application knowledge regarding the control-flow structure of the program obviates the power expensive BTB operations, thus enabling the utilisation of contemporary branch predictors in high-end, yet power-sensitive embedded processors. The utilisation of exact application knowledge results not only in the complete elimination of the power hungry BTB structure but also in a perfect branch and target address identification. A cost-efficient and programmable hardware architecture for capturing the control-flow structure of the program is presented. The hardware complexity of the proposed architecture is carefully analysed in terms of power, performance and area overhead. The proposed technique delivers power reductions in excess of 90% for a set of representative embedded benchmarks.

1 Introduction

The continuously decreasing feature sizes created by modern technology processes have resulted in proportionally higher silicon integration capabilities. As a consequence, system-on-a-chip design solutions have been widely adopted in many electronic products where embedded processor cores constitute a major building component.

Implementing most of the system functionality in software by utilising embedded processor cores results in sharp reductions in the time-to-market in addition to lower design costs, and easily reprogrammable products. Fundamentally, all these advantages stem from the inherent processor generality as computing devices. However, this same generality of processor cores constitutes a major drawback with respect to their excessive power consumption. However, processor components with all their attendant benefits of flexibility and high volume would be natural candidates for further utilisation if they could be guaranteed not to impose power impediments. Customising the processor to the needs of a particular application or application domain has been a powerful technique to achieve significant power and performance improvements while preserving the inherent processor flexibility as programmable devices.

Energy efficiency has already been well established as an important product quality characteristic. In mobile applications, such as hand-held and wireless devices, not only

does it impact directly on the cost of ownership, but it also may severely undermine the usability and acceptance of the product. Consequently, techniques to minimise the system power consumption are of significant importance in achieving a high product quality.

Due to the ever increasing performance requirements, deeply pipelined architectures have become a standard even in the embedded processor domain. In order to efficiently utilise a deep pipeline, it is of paramount importance that the processor front-end be capable of constantly delivering instructions for decoding and execution. However, changes in the execution flow caused by branch, jump, and subroutine call instructions introduce significant complications to the pipeline flow. For instance, when a branch instruction is fetched, the next instruction to be executed might be either the subsequent instruction or the instruction at the branch destination. The precise information regarding the branch direction and the branch destination are typically available late in the pipelined execution. Consequently, in order to fetch the next instruction, the processor front-end has to wait until the branch is executed before continuing to fetch from the correct location. These pipeline stalls lead to a significant performance degradation, especially in the case of control-dominated applications. To alleviate this problem, branch prediction mechanisms have been proposed for high-end general-purpose processors. However, these mechanisms have not been widely adopted in embedded processors, mainly due to their complexity and associated power consumption. The branch target buffer (BTB) is the main source of power consumption since it is a significantly large cache-like structure that needs to be looked up at every execution cycle. A hit in the BTB indicates that the instruction currently being fetched is a control-altering instruction and according to its type, an appropriate action is to be undertaken.

We intend to propose an extremely power efficient application-specific BTB architecture that can be customised to the particular application needs. Information regarding the program control flow is extracted during the

© IEE, 2004

IEE Proceedings online no. 20041101

doi: 10.1049/ip-cdt:20041101

Paper first received 7th January and in revised form 20th August 2004

P. Petrov is with the ECE Department, University of Maryland at College Park, College Park, MD 20742, USA

A. Orailoglu is with the CSE Department, University of California at San Diego, San Diego, CA 92093, USA

E-mail: ppetrov@ece.umd.edu

compile time and transferred to the proposed application customisable branch target buffer (ACBTB). The ACBTB is a software programmable component that can be reconfigured with the particular application control-flow structure. By utilising precise application knowledge, the ACBTB is accessed only when a branch instruction is to be executed, thus making all the relevant branch information available in an extremely power efficient manner.

The proposed technique is fundamentally a microarchitectural customisation methodology for low-power branch resolution. Application information is extracted during the compile/link time and utilised by a reprogrammable hardware support. In this way, not only is the processor architecture tuned for the particular application, but it is also capable of recustomisation in field in the case of application changes and modifications.

2 Branch prediction architecture

Control-altering instructions, such as branches, have long been known to introduce significant performance degradation to pipelined processor architectures. The fundamental problem lies in the inability of the front-end to know in advance the two major branch characteristics: (i) the branch direction; and (ii) the target address. Since these two branch properties are computed late in the pipeline, stalls have to be introduced in the processor front-end before fetching is resumed from the correct program location. Predicting the branch direction while caching the branch target address has been the fundamental objective in many designs of efficient pipelined architectures.

There has been a quite significant research effort within the computer architecture community in the design and implementation of various branch prediction methodologies. There are two major groups of branch predictors: (i) static predictors, and (ii) dynamic predictors. The static branch predictors provide a fixed direction prediction for each branch during program execution. This prediction can simply be a function of the branch destination relative position, i.e. forward or backward, or identified through compiler analysis [1, 2] or profiling [3] and provided as a part of the branch op-code. On the other hand, the dynamic branch predictors rely on run-time program information in order to provide a prediction, which can change throughout the program execution. The fundamental idea on which the dynamic predictors are based consists in the exploitation of the existing correlation between branch outcomes [4, 5]. Recently, an approach for conditional branch folding for embedded applications has been proposed [6]. It is based on the detection and resolution of a subset of short loop branches. In all these prediction architectures, the branch direction is provided as the result of some prediction mechanism, whereas the branch target address is provided by the BTB. A typical architecture of a branch prediction logic is shown in Fig. 1.

Reducing the power consumption associated with the branch prediction logic has been recently identified as an

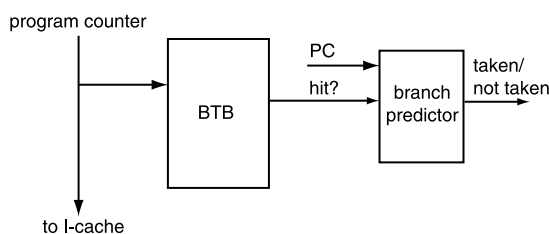


Fig. 1 Basic branch prediction architecture

important problem since its power consumption can easily amount up to 10% of the total power consumption [7, 8]. The most important contributor to this power consumption is the BTB, which is in essence a highly associative cache structure that is looked up at each processor cycle with the program counter. In this work we aim at drastically reducing this power overhead by eliminating the need for cycle-by-cycle BTB access and instead, replace it with a single indexed table accessed once per branch instruction. In [7], a technique for banking the branch prediction tables and the BTB has been proposed. Additionally, a logic is introduced whose purpose is the prediction of when an access to the branch predictor is needed in order to avoid unnecessary lookups. A technique, which selectively updates the branch predictor was proposed in [9]. Using the past branch behaviour, the technique identifies whether there is enough information about the branch for correct prediction, so as to avoid unnecessary branch predictor updates. In [8], a technique for dynamically adjusting the size of the branch prediction table is proposed. By using profiling, the technique scales down or increases the effective size of the prediction tables in accordance with the application demand. In contrast with all these approaches, the technique that we propose eliminates all the accesses to the BTB altogether through the utilisation of application-specific information and replaces it with a directly indexed table accessed once per branch instruction. In this way we hope to achieve drastic power reductions.

The BTB is looked up while the instruction is being fetched. The BTB is a cache-like structure typically comprising of 512 sets with an associativity ranging from two to eight ways [10]. The same program counter (PC) that is used to fetch the current instructions is used to access the BTB. A hit in the BTB indicates that the instruction to be executed is a control-altering instruction. The instruction type, such as branch, jump, function call, or others, is also provided by the BTB. According to the instruction type, a direction prediction or computation is performed. If the instruction type is a conditional branch then the branch predictor is consulted. For branches and direct jumps, the BTB contains the destination address. If the branch is predicted to be taken, the target address provided by the BTB is used as the next PC. It is also possible that the BTB contains some branch prediction information.

Fundamentally, the purpose of the BTB is to provide early branch identification, before the instruction is brought to the front-end, as well as some information regarding the type of the branch. This early branch identification assumes significant additional importance as the typical number of pipeline stages dedicated for instruction fetch increases. If no such identification exists, even a correct branch prediction for a taken branch would lead to a number of stalls that is equal to the number of fetch cycles. Consequently, since no information generally exists regarding the application control-flow structure, the BTB has to be always looked up during the first fetch cycle. As the mapping between the PC and the branch instruction has to be precise, the upper part of the PC is used as a tag to access the BTB. In this way, the BTB architecture entirely resembles a cache architecture, where tag and data arrays are looked up in order to find the correct data, if present in the cache, or alternatively to indicate a miss. It is evident that the BTB operations play a significant role in the total front-end power dissipation if a branch prediction mechanism is to be utilised.

We now propose a methodology for extremely low-power application-specific BTB organisation. Having exact knowledge regarding the program control-flow structure, the

ACBTB can efficiently track the program execution and thus identify in advance the control-altering instructions with no need for cycle-by-cycle lookups.

3 Customisation framework

3.1 Functional overview

A well known rule-of-thumb in computer architecture is that a program spends 90% of its execution cycles on only 10% of its code. As has been observed in their numerous uses the embedded processor applications certainly do not violate this principle. Every application can be easily partitioned into a few major loops or functions that contribute almost all of the execution cycles, while covering a relatively small fraction of the static program code. Illustrative examples are various multimedia applications, which spend most of their execution time within a set of heavily utilised DSP kernels. Furthermore, the various application hot spots are not only quite tractable due to their limited size, but are also extremely independent, thus enabling locally applied and therefore inexpensive optimisation techniques. Consequently, concentrating the optimisation efforts on these application ‘hot spots’ delivers maximum benefits at a minimal compiler and hardware cost.

The proposed customisation methodology is applied independently on the application hot spots, which are identified by performing application profiling or extracted from the algorithmic specifications. The control-flow graph for each hot spot is extracted and analysed during the compile/link time. The application information regarding the control-flow structure is transferred to the ACBTB before entering the hot spot, thus introducing no performance overhead to the program execution time.

3.2 Control-flow information

Since the fundamental purpose of the BTB is to provide early identification of the control-altering instructions as well as certain information regarding their type, application knowledge about the control-flow structure is needed in order to efficiently achieve the BTB objective. The control structure of a program is generally represented by the control flow graph (CFG). The CFG nodes correspond to basic blocks of code whereas the edges represent a possible transfer of execution across the corresponding basic blocks. A basic block is a linear sequence of instructions with single entry and exit points. Consequently, a control-altering instruction, such as a branch instruction, can only be the last instruction of a basic block.

Figure 2a shows a program loop containing two conditional statements. The CFG of this code is shown in Fig. 2b. Basic blocks B2 and B3 correspond to the ‘then’ and ‘else’ clause of the first conditional statement, whereas basic block B5 corresponds to the ‘then’ part of the second conditional statement. Figure 2c shows a possible code layout for the given loop. It is evident that there are two conditional branches, one a direct jump and the other a loop branch. The two branch instructions, branch 1 and branch 2, determine the control change at the end of blocks B1 and B4, respectively. Block B2 ends with a direct unconditional jump, whose purpose is to go directly to the joint point of the first conditional statement.

An important observation that can be made after considering a CFG layout, such as the one in Fig. 2c, is that the distance in terms of the number of instructions between the control-altering instructions is known immediately after compiling the program. For instance, in the case of taking branch1, the next control-altering instruction,

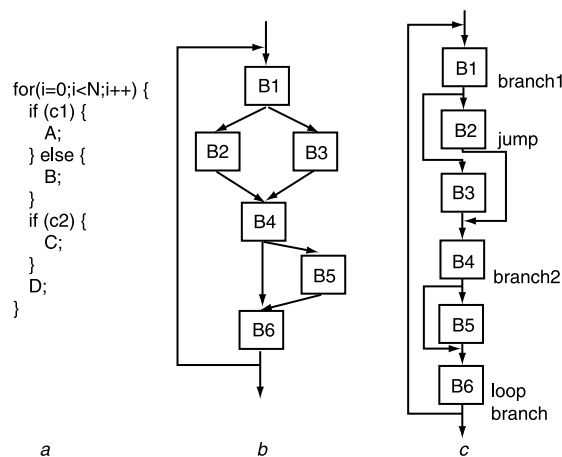


Fig. 2 Control-flow structure

- a A program loop containing two conditional statements
- b The CFG of the code of Fig. 2a
- c Possible code layout

which is the branch2 instruction is to be executed after $|B3| + |B4|$ instructions, where $|B|$ denotes the number of instructions within a basic block B. In the case of the not taking branch1, the subsequent control change happens in $|B2|$ instructions and its cause would be the jump instruction. Evidently, there are two possible execution paths that can be followed after executing the branch1 instruction. The first one goes through blocks B3 and B4 before encountering a subsequent possible control change at instruction branch2, whereas the second path goes through basic block B2, before encountering the jump instruction.

Consequently, by utilising the exact application knowledge regarding the CFG and the distances of all possible execution paths between control-altering instructions, a specialized front-end hardware structure can efficiently track the control-altering instructions by simply counting the instructions executed after the branch; depending on the starting branch direction, the subsequent control instruction can thus be exactly pinpointed. For this type of architecture, no cycle-by-cycle expensive BTB lookups are needed; instead, only a counter logic and single indexing per branch into a special table that contains the CFG information would suffice.

3.3 Tracking instruction execution

In order to keep track of the program execution between control-altering instructions, a mechanism capable of exactly identifying when a specified number of instructions has been executed is needed. Due to stringent power constraints, embedded processor architectures are designed as single and in-order issue pipelines. Superscalar engines capable of multiple instruction issue and dynamic reordering for fine-grained instruction level parallelism (ILP) exploitation are common in the domain of high-end workstations and server processors. The significant complexity in terms of power consumption of this type of superscalar engine precludes its utilisation as an embedded processor. Nonetheless, our proposed approach is readily applicable to such processors since it simply relies on identifying the number of instructions fetched. The only modification to the proposed architecture is that instead of assuming that a single instruction is being fetched we need to identify the number of instructions fetched and to account for this number when counting the instructions preceding the next branch.

Therefore, we focus our attention on single, in-order issue pipelines, including VLIW processors, a very common architecture in the domain of multimedia and DSP

applications. In the case of a processor architecture with a fixed number of instructions fetched and issued, VLIW being one of them, simply counting the number of fetched instructions, or alternatively, the PC updates, can provide an exact identification of the number of instructions being fetched following a certain branch. Naturally, in the case of branch mispredictions, the number of misfetched and subsequently flushed instructions needs to be accounted for and subtracted.

4 The ACBTB

4.1 Architecture overview

The hardware support for the proposed customisable BTB needs to be able to utilise application-specific information regarding the control-flow structure of the program. More specifically, for each control-altering instruction, information is needed about the distances to all subsequent control instructions on the possible control paths. For instance, handling a conditional branch necessitates two pieces of information for the cases of taken and not-taken directions. For each of these cases, both the distance to the next control instruction and the index for accessing the information of the subsequent control instruction are needed. This application information is stored in the ACBTB table.

Figure 3 shows the architecture of the proposed low-power customisable BTB architecture. The (ACBTB) table contains all the relevant application information regarding the control-flow structure. The ACBTB is directly indexed and contains an entry per control-altering instruction. The NT_D and NT_I fields contain data with respect to the branch not-taken execution path. The NT_D field specifies the number of instructions in the fall through branch path to be executed before a control-altering instruction is encountered whereas the NT_I field has the index to the ACBTB entry that contains the information about the subsequent control-altering instruction. The pair of T_D and T_I fields contains similar information but with respect to the branch taken execution path. The index fields NT_I and T_I are essentially pointers to ACBTB entries corresponding to the subsequent control instructions. By utilising such pointers, an efficient access to the ACBTB can be performed with no need of associative lookups. For the case of the taken branch or direct jump instructions, the target address is needed in order to continue fetching from the correct instruction location; this target address is provided by the TA field. Finally, the Type field contains information regarding the type of the control instruction, i.e. conditional branch, jump, functional call, etc. This field may be additionally used to provide extra branch information that can facilitate the branch prediction process. For instance, it can specify a

static branch prediction direction based on program analysis and profile runs.

After a branch is executed, the CNT counter register is loaded with the value of NT_D or T_D depending on its direction. At the same time the IND register is loaded with the ACBTB index corresponding to the next control-altering instruction. In the subsequent execution cycles, the only activity performed by the proposed architecture is to decrement with each fetched instruction the value of the CNT register. Note that the decrement value does not necessarily correspond to the number of fetched instructions as it has to reflect the fetch bandwidth. For instance, in the case of a VLIW processor where a packet of instructions is fetched instead of a single instruction, the distance between control instructions would be measured in packets. A zero value in the counter CNT indicates that a control-altering instruction is about to be fetched. This event, detected by the zero comparator at the counter output, triggers an access to the ACBTB in order to read the branch properties stored in the table.

Fundamentally, the ACBTB table contains the control-flow structure of the application. More specifically, this information is reflected by the NT_D, T_D, and Type fields, where the first two specify the distance to the next control-altering instruction depending on the branch direction outcome whereas the last one specifies the instruction type. The index fields, NT_I and T_I, facilitate an easy and efficacious access to the ACBTB. By utilising these two fields, the ACBTB entry corresponding to the subsequent control-altering instruction can be accessed not only quite efficiently, but also in a timely matter without the introduction of any performance overhead whatsoever. We elaborate on this issue in Section 4.4.

4.2 Transferring the branch information

Most of the execution time of practically any application is concentrated within a few heavily utilised application loops or functions, generally referred to as 'hot spots'. In the proposed application customisation methodology, the application is initially profiled and its hot spots identified. The control-flow structure of each such spot is analysed and the distances between the control-altering instructions on all the execution paths extracted. Typically, the static code size of a hot spot is relatively small which in turn results in a relatively small number of control-altering instructions that need to be targeted by our approach. This analysis is performed statically after compiling and linking the program since the information regarding the control-flow structure is available at this stage.

Transferring the control-flow information to the ACBTB table is performed by having this table accessible by software. This can be easily done, for instance, by mapping

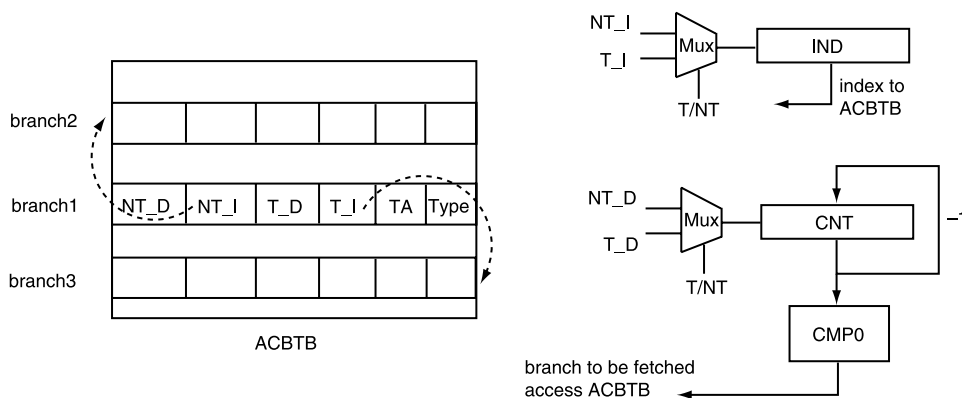


Fig. 3 Customisable BTB architecture

the ACBTB contents to the processor address space. Prior to entering any application hot spot, a sequence of instructions responsible for storing the branch information to the ACBTB will be executed. This setup sequence of instructions is inserted by the compiler after extracting the control-flow structure information. As the setup code is executed only once prior to entering a hot spot, the introduced performance overhead is practically nonexistent.

In most cases, the ACBTB setup code execution can be done only once as a part of the global program setup being performed immediately after loading the program and before transferring control to the application's main function. Such a global ACBTB initialisation policy can be achieved through the utilisation of several ACBTB tables with identical structures. At any instant only one of these tables would be active, thus introducing no additional power overhead. The required ACBTB size is quite small and does not exceed 64 entries; thus no area overhead is introduced since the ACBTB size is directly proportional to the number of control-altering instructions within each of the application hot spots. In Section 5 we provide a quantitative analysis on this issue and provide suggestions for the typical size and number of the ACBTB tables.

4.3 Handling indirect jumps

Most of the control-altering instructions, such as conditional branches, direct jumps, direct function calls, and return instructions, have a static destination address that can be identified after linking the program. These instruction types are efficiently handled by the proposed ACBTB architecture. Even although infrequent, there exist other types of control-altering instructions, namely indirect jumps and indirect function calls. With these types of instructions, the target address is not readily available after linking, but instead it is computed during the run-time, loaded into a register, which in turn is used by the indirect jump or indirect function call as a destination specifier. These instructions are typically used to compile virtual function calls in object-oriented programming languages or to optimise long and regular switch statements. All possible destination addresses are normally stored by the compiler in a special table which is accessed just before the indirect jump or function call in order to obtain the proper destination address.

Since the target address of an indirect jump or call can vary amongst a large set of possible values and is available only after reading its destination register, these control-altering instructions are difficult to handle and introduce pipeline stalls before clearing the target address. In order to handle these instructions within our customisation framework, the contents of the CNT and the IND registers need to be updated by software whereas the target address is loaded into the instruction's destination register. A table containing the appropriate values of the CNT and IND registers for all possible destinations can be constructed by the compiler in a manner similar to the destination address table, which the compiler generates when using indirect control-altering instructions. Subsequently, two instructions for updating the CNT and IND registers are inserted just before the indirect jump and after the load to the jump destination register. The two pipeline cycles taken to execute these instructions are well hidden by the latency of the destination register load instruction, thus introducing no performance overhead. In this case, the CNT and the IND registers need to be made controllable by software; a trivial task for almost any embedded processor architecture.

An alternative approach for handling the indirect jump instructions, is to introduce a different hardware table, the

indirect branch identification table (IBIT), whose purpose would be to capture all the possible CNT and IND pairs for the indirect destinations. This table is looked up with the PC of the branch destination instruction, and contains the pair of CNT and IND values for that particular branch direction. The content of the IBIT table is written by software prior to entering the application hot spot. A setup code, generated by the compiler, fills in the IBIT entries with their actual values for all possible branch directions. This is possible since the compiler generates structures such as jump tables and virtual function tables to efficiently identify the jump/call direction run-time. While generating these standard data structures, the compiler has a complete knowledge in terms of control-flow structure for the subsequent branches. Note that the IBIT table would be accessed only once when an indirect jump or call instruction is encountered, resulting in an extremely insignificant power overhead. By utilising an IBIT table, the program code within the application hot spot will be left intact, thus introducing no additional instruction execution or performance overhead.

4.4 Power and performance analysis

The ACBTB table is accessed only once per branch instruction and this access is performed during the first processor fetch stage, in parallel with accessing the instruction memory hierarchy. Therefore, the access time to the ACBTB is effectively hidden and introduces no pipeline timing constraints. Similarly, decrementing the CNT register happens during the same time in parallel with the first fetch stage. Since the number of ACBTB entries is extremely small, the access time can easily fit within a single cycle even for deeply pipelined front-ends. The same does not hold for the large general-purpose BTBs, which are organised as caches and tagged with the PC.

Moreover, not only are there no timing or performance bottlenecks introduced, but performance can typically be improved compared to a general-purpose BTB. Since there is a dedicated ACBTB entry for each control-altering instruction, no conflicts can exist and 100% branch information is achieved. The effects of BTB conflicts and the consequent performance improvements in the case of the proposed customisation methodology are evaluated in Section 5.

In terms of power savings, reductions of several orders of magnitudes compared to a general-purpose BTB are achieved. The fundamental reason for this dramatic reduction is the elimination of the cycle-by-cycle lookups into the BTB. All these lookups together with their tag operations are replaced by a single access per branch to the extremely small ACBTB table. Also, the width of the ACBTB entries is quite small. The index fields are 6 bits wide, assuming 64 entries and the NT_D and T_D fields can be implemented in 8-9 bits at a maximum. The complete elimination of the cycle-by-cycle accesses to the BTB and replacing them instead with a single direct mapped indexing per branch instruction is the fundamental explanation of the reported significant power reductions. The access to the ACBTB is performed only once for a branch instruction that requires a target address. For instance, in the case of conditional instructions no BTB lookups are performed since they do not require a target address.

Interestingly, the index field in the case of the not-taken branch, NT_I, can be completely eliminated from the ACBTB as long as the control-altering instructions are mapped to the ACBTB following their linear order from the program code. In this way the ACBTB entry for the subsequent control-altering instruction in the case of a

not-taken branch is simply the next ACBTB entry and the IND register simply needs to be incremented. Furthermore, even if several ACBTB tables are utilised across the entire application run for efficient coverage of multiple hot spots, there would be only one active ACBTB with the remaining ones being turned off. Complete quantitative power evaluation of the proposed approach is presented in the following Section.

5 Experimental results

We have performed a set of experimental studies in order to evaluate the benefits of the proposed approach. The benchmarks were initially profiled and the hot spots identified. The control-flow structure of the hot spots was then analysed by manually instrumenting the assembly code and performing SimpleScalar [11] simulations. We have used the functional simulator with a branch prediction model. Since the utility of our approach does not depend on any other microarchitectural components than the branch predictor, we have chosen this simulator to focus on the branch target buffer operations. Such a simulation environment corresponds to an in-order pipelined processor. Since the proposed methodology is independent of the rest of the processor architecture, we only report power saving results on the BTB. Depending on the processor configuration, the contribution of the branch prediction logic to the total power consumption can vary, but as reported by other researches [7, 8], it can sometimes be more than 10%.

The following set of benchmarks was used in our experimental evaluation. The three speech coders: *adpcm*, *g721*, and *gsm*, both encoder and decoder parts, the *epic*, *jpeg*, *mp3*, *mpeg* multimedia applications for image, audio, and video processing. All these benchmarks, apart from the *mp3*, constitute the Mediabench [12] collection of multimedia applications. The *mp3* benchmark is a publicly available mp3 codec engine [13].

Table 1 lists the basic characteristics of the application benchmarks used. The hot spots column presents the number of hot spots for each benchmark. The *adpcm* coders consist of a single hot spot, which practically corresponds to the entire application. For all the remaining benchmarks, the set of hot spots covers more than 90% of the execution time. The subsequent three columns illustrate the number of conditional branches, direct jumps, and indirect jumps per application hot spot.

Table 2 lists some branch execution characteristics of the benchmarks running on a baseline architecture. The IPB column lists the average instruction per branch ratio. The IPB value is a numerical characteristic of the branch

instruction density. For five of the six benchmarks, the IPB value is in the range of 3.5 to 6, which is a typical situation for real applications. The significantly higher IPB value of 20.61 for the *gsm* encoder is due to the existence of a basic block of size slightly more than 300 instructions performing an important computation with fixed point numbers. The second column, denoted as *Dir* lists the accuracy of predicting the branch directions (as a percentage) by utilising a general-purpose branch predictor. The *gshare* branch-predictor [4] was used with a global history register of size ten. The last four columns illustrate the BTB performance for four different BTB organisations. The first two columns correspond to BTBs with 128 entries, and the last two to BTBs with 64 entries. In each pair the first column represents a four-way, whereas the second one an eight-way set associative organisation. The numerical values in these columns illustrate the hit ratio (as a percentage) of the BTB. In a perfect situation with no BTB conflicts, this ratio should be equal to the direction prediction ratio. In practice, as can be seen from the Table, a difference exists which is due to branches predicted correctly by the direction predictor, yet not found in the BTB when attempting to obtain their target address. For the relatively small *adpcm* applications, practically all the correctly predicted branches are found in the BTB. For the remaining, more complex benchmarks, one can observe that there might be a sizable difference between correctly predicted branches and branches found in the BTB. For all these cases, the proposed application-specific branch target buffer methodology provides complete branch resolution, thus eliminating the performance penalties associated with the misses in the general-purpose BTB.

Finally, we have evaluated the precise power reductions for all the application benchmarks. We have used the CACTI [14] tool, a cache architecture exploration tool which provides the designer with detailed timing, area, and power characteristics for a given cache configuration, to obtain detailed power models for the BTB and ACBTB memory structures involved in our study. Since the general-purpose BTB architectures are implemented as standard cache structures, it has proved straightforward to utilise CACTI to provide exact power numbers for the four BTB baseline architectures that we compare to, i.e. 128 and 64 entry BTBs with four- and eight-way set associativity. Power characteristics for a 64 entry ACBTB were extracted from CACTI through modelling of a direct-mapped cache structure and subtracting the power consumption of the tag array and the comparator cells. The line width for all the BTB and the ACBTB memory structures was fixed at 8 bytes, the minimal word size allowed by CACTI, which is

Table 1: Benchmark characteristics

	Hot spots	Conditional branches	Direct jumps	Indirect jumps
<i>adp_e</i>	1	14	3	0
<i>adp_d</i>	1	11	2	0
<i>g721_e</i>	3	7, 43, 8	4, 15, 2	0, 0, 0
<i>g721_d</i>	2	7, 43	4, 17	0, 0
<i>gsm_e</i>	5	22, 5, 3, 9, 22	7, 0, 1, 3, 4	0, 0, 0, 0
<i>gsm_d</i>	2	8, 4	3, 0	0, 0
<i>epic</i>	1	97	15	14
<i>jpeg</i>	2	74, 514	10, 122	12, 79
<i>mp3</i>	5	275, 335, 32, 68, 321	86, 119, 15, 33, 143	20, 49, 8, 10, 48
<i>mpeg</i>	3	172, 20, 32	68, 6, 7	15, 2, 4

Table 2: BTB characteristics

	IPB	Dir, %	128-4	128-8	64-4	64-8
adp_e	3.49	79.14	79.13	79.13	79.13	79.13
adp_d	4.55	87.92	87.90	87.90	87.90	87.90
g721_e	4.48	89.88	86.01	88.10	77.92	76.03
g721_d	4.35	91.91	86.66	89.93	78.65	78.42
gsm_e	20.61	92.41	89.97	89.98	88.62	88.28
gsm_d	5.75	98.30	97.48	97.48	97.15	97.22
epic	6.79	94.96	94.93	94.94	94.92	94.92
jpeg	6.80	93.79	93.65	93.66	93.03	93.55
mp3	8.88	94.11	93.50	93.52	93.17	93.17
mpeg	5.87	80.76	80.73	80.74	80.62	80.63

Table 3: BTB and ACBTB power consumption in millijoules

	128-4	128-8	64-4	64-8	ACBTB	Improvement, %
adp_e	5.11	9.16	5.03	9.21	0.35	93.06
adp_d	4.17	7.47	4.11	7.51	0.30	92.81
g721_e	160.91	288.29	158.42	290.00	9.03	94.30
g721_d	153.04	274.19	150.69	275.82	8.56	94.32
gsm_e	162.44	291.03	159.94	292.76	1.44	99.10
gsm_d	49.02	87.82	48.26	88.34	2.10	95.65
epic	40.44	72.45	39.81	72.88	6.99	82.42
jpeg	12.42	22.26	12.23	22.39	1.86	84.80
mp3	601.46	1077.57	592.19	1083.98	31.49	94.68
mpeg	868.79	1556.53	855.41	1565.79	107.21	87.47

about the right amount of storage for both the BTBs and the ACBTB. Since the power characteristics reported by CACTI reflect the energy dissipated per access to the cache structure, we have utilised the SimpleScalar simulator to evaluate the total power consumption of the proposed ACBTB and the baseline BTBs. Note that the power numbers that we report on are for a technology process with a 0.18 μm feature size. An access to the BTB was triggered per each instruction executed, while an access to the ACBTB was accounted for only in the case of control-altering instruction execution.

Table 3 presents the total power consumption numbers in millijoules for the four baseline BTB organisations and the proposed ACBTB architecture. The power consumption numbers presented in this Table correspond to the entire application, including the hot spots and the remaining parts of the benchmarks. The first four columns correspond to the baseline BTBs and the fifth column shows the total power consumption of the ACBTB architecture. One can immediately notice the orders of magnitude reduction in terms of power compared to the general-purpose BTB organisations. The last column in Table 3 shows the achieved power improvements as a percentage. The percentage reduction is computed by comparing the ACBTB power consumption to the BTB architecture with minimal power consumption among the baseline BTBs, i.e. the 64 entry, four-way set

associative BTB. In all the cases, improvements higher than 90% were achieved and for the gsm benchmarks the improvements exceed 95%.

6 Conclusions

We have presented a methodology for application-specific low-power branch target buffer organisation. Application information regarding the control-flow structure of the program is extracted during the compile/link time and utilised by an efficient customisable branch resolution microarchitecture. The extremely power expensive cycle-by-cycle BTB lookups are thus completely eliminated and replaced instead by a single access per branch to a directly indexed and application customisable branch target table. Not only are significant levels of power reductions achieved, but also 100% branch resolution is guaranteed, thus further improving on the lower hit ratio of the general-purpose BTBs. The power efficient and reprogrammable hardware architecture proposed enables the integration of the proposed customisation methodology into a wide range of embedded processor architectures and its subsequent utilisation in a post-design/manufacturing fashion for a broad spectrum of power frugal electronic products.

7 Acknowledgment

This work is supported by NSF grant 0082825.

8 References

- Ball, T., and Larus, J.R.: 'Branch prediction for free'. Proc. Conf. on Programming Language Design and Implementation, June 1993, pp. 300–313
- Young, C., and Smith, M.D.: 'Static correlated branch prediction', *ACM Trans. Program. Lang. Syst.*, 1999, **21**, pp. 111–159
- Fisher, J.A., and Freudenberger, S.M.: 'Predicting conditional branch directions from previous runs of a program'. Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Systems, October 1992, pp. 85–95
- Pan, S.T., So, K., and Rahmeh, J.T.: 'Improving the accuracy of dynamic branch prediction using branch correlation'. Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Systems, October 1992, pp. 76–84
- McFarling, S.: 'Combining branch predictors'. Technical Report TN-36, Western Research Laboratory, DEC, June 1993
- Lee, L.H., Scott, J., Moyer, B., and Arends, J.: 'Low-cost branch folding for embedded applications with small tight loops'. Proc. 32nd Int. Symp. on Microarchitecture, November 1999, pp. 103–111
- Parikh, D., Skadron, K., Zhang, Y., Barcella, M., and Stan, M.: 'Power issues related to branch prediction'. Proc. Int. Symp. on High-performance Computer Architecture, Cambridge, MA, USA, Feb. 2002, pp. 233–244
- Chaver, D., Pinuel, L., Prineto, M., Tirado, F., and Huang, M.: 'Branch prediction on demand: an energy-efficient solution'. Proc. Int. Symp. on High-performance Computer Architecture, Aug. 2003, pp. 25–27
- Baniasadi, A.: 'Power-aware branch predictor update for high-performance processors'. Proc. Int. Workshop on Power and Timing Modeling, Optimization and Simulation, September 2003, pp. 420–429
- Perleberg, C., and Smith, A.J.: 'Branch target buffer design and optimization', *IEEE Trans. Comput.*, 1993, **42**, (4), pp. 396–412
- Austin, T., Larson, E., and Ernst, D.: 'SimpleScalar: An infrastructure for computer system modeling', *Computer*, 2002, **35**, (2), pp. 59–67
- Lee, C., Potkonjak, M., and Mangione-Smith, W.H.: 'MediaBench: A tool for evaluating and synthesizing multimedia and communications systems'. Proc. 30th Int. Symp. on Microarchitecture, December 1997, pp. 330–335
- Available at <http://lame.sourceforge.net>
- Shivakumar, P., and Jouppi, N.: 'CACTI 3.0: An integrated cache timing, power and area model'. Technical report, Western Research Lab, 2001