Abstract—A well-established technique for reducing embedded processor power is to leverage rigorous static code analysis to enable application-specific hardware customizations. Energy can be saved by leveraging compile-time information to eliminate certain hardware operations. One such research area reduced branch prediction hardware power by statically extracting application control flow information. However, the applicability of these static techniques has stalled with respect to modern smartphones. These high-performance mobile processors inhabit a unique mobile ecosystem domain space. Rather than an application consisting of a monolithic instruction sequence compiled on the target processor, mobile applications are stored in a centralized marketplace and consist of high-level object-oriented code that dynamically binds with device-specific foundation libraries. While this model helps reduce development time and enables applications to be downloaded and run on a variety of device models and operating system versions, it significantly hinders whole application static analysis and application-specific hardware optimizations. This paper addresses the unique challenges of the mobile ecosystem by enabling on-device application analysis to guide reconfiguration of the branch target buffer. Software and hardware customizations are intelligently combined to greatly reduce power dissipation while maintaining or even improving performance.

I. INTRODUCTION

One of the foremost challenges for mobile embedded processors is to provide low power consumption in order to extend battery life. Over the last decade, much prior work has been done to reduce the power consumption of these embedded instruction processors by leveraging rigorous static code analysis to customize the processor microarchitecture in an application-specific manner. These techniques delivered significant energy savings by extracting compile-time information to reduce or even eliminate the need for certain run-time hardware computations. One such research area focused on reducing the power utilization of branch prediction hardware by statically extracting control flow information from an application [1], [2], [3]. However, the applicability of these static application-specific techniques has stalled with respect to modern smartphone and tablet embedded processors. State-of-the-art mobile processors have evolved to the level of having highly dynamic, feature-rich applications comparable to those of interactive desktop programs, providing high-quality visual and auditory experiences. These mobile processors are becoming increasingly complex in order to respond to this more diverse and dynamic application base.

For example, the ARM Cortex-A15 processor, which is commonly incorporated in high-performance smartphones and tablets, employs an out-of-order architecture with a pipeline length of between 15-stages and 24-stages [4]. As can be expected, robust branch prediction is necessary in order to ensure control flow changes do not degrade performance by stalling the pipeline. It is the job of the branch prediction hardware to provide an early prediction on whether a control flow instruction is taken or not, and if it is taken to also provide the target address from which subsequent instructions should be fetched. A direction predictor is used to determine if a control instruction is taken or not and a branch target buffer (BTB) is used to identify control instructions and the target address each of them can redirect the control flow to. However, such prediction hardware has a price in terms of power dissipation. The branch prediction hardware can consume as much as 10% of the total processor power, with the Branch Target Buffer (BTB) accounting for 88% of this power consumption [5]. Unfortunately, the typical static techniques that could help reduce this power overhead are unable to cope with the complex and distributed nature of modern mobile processors.

These high-performance mobile processors inhabit a unique mobile ecosystem domain space. Unlike basic embedded instruction processors, where the entire application is a unified sequence of instructions that is directly compiled for the target processor, mobile applications are distributed and dynamic. Typical mobile applications consist of high-level object-oriented code written by the application developer, which in turn dynamically relies on numerous operating system foundation libraries to provide common user interface frameworks and methods to access the device’s peripherals and sensors. Applications are compiled separately from the operating system foundation libraries and stored in a centralized application marketplace. While this model helps reduce development time and enables applications to be downloaded and run on a variety of device models and operating system versions, it significantly hinders whole application static analysis and application-specific hardware optimizations.
In this paper, we acknowledge the fundamental paradigm shift in the mobile processor domain. Whereas early mobile applications consisted of small and highly-optimized firmware code that was tightly integrated with the physical device, state-of-the-art smartphone and tablet applications are much more complex and abstractly written using high-level object-oriented languages with little knowledge of the underlying physical device properties. Furthermore, this mobile revolution has resulted in the creation of centralized application marketplaces, where applications are developed in a generalized fashion without knowledge of any particular target device, and will instead rely on dynamically-linked device-specific foundation libraries to interact with low-level hardware features on each individual device. This new mobile ecosystem precludes the ability to statically perform high-level interprocedural optimizations, and also results in significantly larger amounts of control flow instructions [6]. Innovative solutions are needed to address the challenges posed by this mobile ecosystem revolution.

We propose a novel methodology to enable application-specific low-power optimizations for the processor control microarchitecture by addressing the unique characteristics of the mobile ecosystem. Using a mobile ecosystem driven approach, analysis and customization of a given application and its shared libraries is performed to enable application-specific microarchitecture optimizations. While an application is still compiled generically at the time it is added to the mobile application marketplace, critical information about the control flow structure of the application is also garnered and encapsulated. Once the application is downloaded onto a device, additional analysis will be performed incorporating the information about the device-specific hardware, operating system version, and foundation libraries. By enabling on-device application analysis and transformation, we are able to leverage comprehensive application-specific information to reconfigure the BTB hardware to save power. Static control flow information is extracted and provided to the hardware microarchitecture in order to enable a power-efficient hybrid branch target buffer, greatly reducing power dissipation while maintaining or even improving performance. We show the implementation of this methodology and provide experimental data taken over a sample of real-world mobile applications to demonstrate the benefits of our approach. Our main contributions are:

- Applying an on-device optimization framework to capture the necessary branch target distances from the distributed application code;
- Identifying application hotspots where complete static branch information is known;
- Development of hybrid branch target buffer architecture to efficiently switch between original BTB and statically-known BTB structures;
- Capturing class inheritance relationships and modifying OS polymorphic resolution logic to handle indirect branch targets.

II. RELATED WORK

In the last six years, the mobile smartphone and tablet space has seen enormous growth in terms of generalized instruction-based processor hardware as well as robust high-level application development ecosystems. Applications are written in object-oriented languages, employing polymorphism. The Branch Target Buffer (BTB) is a critical component for optimization in these systems, since it is accessed every execution cycle to determine if the fetched instruction is a control flow instruction and which target address to continue fetching from if the branch is predicted as taken. A well-functioning BTB ensures better instruction throughput, but a large amount of power is dissipated by the BTB since it is physically similar to a cache and is accessed every cycle.

The authors in [2] proposed the Application Customizable Branch Target Buffer (ACBTB), where static code analysis is used to determine the number of instructions between a given control instruction and the next control instruction (both for the taken and not taken paths). A small hardware table is constructed that holds the target address and these distances to the next control instruction for the taken and not taken paths, as well as the index into the ACBTB for that next control instruction. Thus, when a branch is encountered, the system will know exactly how many instructions will be executed before the next branch. In this interval, no further lookups of the ACBTB take place while non-control instructions are being executed. Once the interval completes, the ACBTB is directly indexed based on the index value of the prior branch, avoiding the overheads of the cache-like lookup for a standard BTB. However, in order for this proposal to work, all control instructions must be statically known so the ACBTB table can be fully populated. The presence of the mobile ecosystem and the large amount of dynamic dispatch and indirect branching hinders statically obtaining a complete control-flow graph.

The authors in [7], [8] proposed an approach to dynamically reduce the number of sets of the BTB using instructions that communicate with special control registers. Using a profile-based approach to identify sections of code where the BTB will likely not benefit from having its full storage capacity, instructions are inserted that reduce the BTB size by a power of 2, saving power by gating the unused portions of the BTB. When more BTB storage is required, similar instructions are inserted which will restore the sets of the BTB which were disabled. One drawback to this approach is that many mobile applications are highly interactive and dependent on user and sensor inputs. Thus, profiling may be unable to accurately represent the exact control flow that will be present on any given execution of the application.

The Shifted-Index BTB proposed in [9] aims to reduce unnecessary accesses to the BTB by more intelligently indexing into the structure. A standard BTB is indexed using the lowest bits of the PC, where each consecutive instruction would result in accessing a different BTB set. The authors propose to instead index using the same number of bits, but shifted a few bits to the left. In this manner, the likelihood of two sequential
BTB accesses mapping to the same set increases. Using this characteristic of BTB indexing, a set buffer is created that stores the last accessed BTB set as well as the index for that set. When looking up subsequent instructions, if the index is the same as what is currently in the set buffer, the set buffer is queried to get the target address instead of the full BTB. In this manner, a large portion of accesses to the BTB can be filtered out to save power.

A similar approach to filtering unnecessary BTB accesses is presented in [10]. By delaying the BTB lookup one cycle, they propose using the branch direction predictor to skip BTB accesses when the direction is predicted as not taken. If the branch is not going to be taken, there is no need to determine the target address. To help counteract the performance lost due to delaying BTB target address lookup by one cycle, a small filter BTB is introduced that is accessed in parallel to the direction prediction and will be used instead of the BTB if the PC address is found within it. In this manner, they claim that accesses to the original BTB are reduced, greatly reducing the dynamic power consumption while incurring a small performance penalty.

Although there has been prior work leveraging domain-specific and operating system techniques [11], research examining application-specific microarchitectural optimizations for mobile processors is lacking. The domain of mobile processors introduces unique challenges related to mitigating BTB power overhead. Since mobile applications are highly interactive and frequently utilize polymorphism both within the application’s local source code and externally into common foundation library code, a solution that can statically analyze the combination of the original application code and its interaction with common device-specific libraries becomes necessary.

III. MOTIVATION

It is common knowledge that control-altering instructions such as branches, jumps, and function calls can severely degrade the performance of instruction pipelines. Fundamentally, the throughput benefit of instruction pipelining hinges on the ability to keep feeding new instructions into the pipeline. However, the nature of a control-altering instruction is to inform the pipeline to begin fetching instructions from a new target location. The ability to fully resolve the control-altering instruction and determine from where to begin fetching typically occurs late in the pipeline, degrading the pipeline’s throughput. In order to surmount this limitation, branch prediction hardware is commonly employed.

The goal of branch prediction is to provide an early decision on whether a control-altering event will occur and what the new target instruction address will be. All branch prediction hardware essentially provides two pieces of information: the branch direction and the target address. The direction is often predicted using correlation between previous branch outcomes, including both local and global pattern history. On the other hand, the target address is typically provided by a Branch Target Buffer (BTB). A typical branch prediction architecture is shown in Figure 1.

![Typical Branch Prediction Architecture](image)

The BTB is a cache-like structure which is indexed using the same program counter (PC) that is used to fetch the current instruction [12]. A hit in the BTB indicates that the instruction to be executed is a control-altering instruction. The BTB entry also includes the destination address and a type field that describes the kind of control-altering instruction (branch, jump, function call, unconditional/conditional, etc.). For conditional branches, the branch direction predictor is consulted to determine whether or not the condition is true. If the condition is met or the instruction type is unconditional, the target address provided by the BTB is used as the next PC address.

As one can see, the BTB helps identify branch instructions before the instruction is even decoded. This early branch identification becomes even more paramount when the number of fetch pipeline stages grows. Without the BTB’s early fetch-time identification, even a correct branch prediction would need to stall the pipeline the same number of cycles as fetch stages to allow the decoding and identification of control-altering instructions. Given this, the BTB has to be queried during every instruction’s first fetch cycle. Since the mapping between the PC and the branch instruction needs to be precise, the lower bits of the PC are used to index into the BTB and the upper bits are used as a tag during lookup. In this respect, the BTB architecture entirely resembles a standard cache architecture, where both the tag and data elements are looked up in order to find the correct data or indicate a miss. Thus, for every instruction executed within the processor, a large amount of power is dissipated by having to query the BTB to determine if the instruction is control-altering and what its target address is. As mentioned earlier, the branch prediction hardware can consume as much as 10% of the total processor power, and the BTB in particular accounts for 88% of this power dissipation [5].

There are two major limitations to utilizing purely static application-specific techniques to reduce BTB power in the more complex mobile smartphone and tablet domain. First, all control instructions must be statically known so the hardware table can be fully populated, yet the presence of the distributed mobile software ecosystem and the large amount of dynamic
dispatch and indirect branching precludes the static retrieval of the complete control-flow graph. Second, it has been observed that mobile applications can commonly contain tens of thousands of control instructions, and storing all of these instructions within a fixed-size hardware structure would have a prohibitive area cost.

IV. METHODOLOGY

One of the primary challenges to statically analyzing application control flow on mobile processors is the vast diversity of hardware, operating system versions, and consumer applications. Mobile applications are developed in a loosely general-purpose fashion and the compiled code is uploaded into a central marketplace (e.g. Apple AppStore or Windows Phone Store). This application is then downloaded onto numerous devices, each of which may vary in terms of foundation library routines. This one-to-many relationship is ideal from a scalability and validation perspective; however it also impedes many global compile-time analysis techniques.

In order to accomplish the goal of reducing mobile processor power consumption due to BTB accesses, a novel on-device BTB power optimization framework is proposed. Our BTB optimization framework will utilize the software framework proposed in [13], where high-level application and OS library metadata about class hierarchy relationships is extracted by the mobile ecosystem compiler toolchain to enable on-device interprocedural code optimizations. We extend this software framework to also extract control flow distance relationships and method implementation metadata to enable BTB-specific optimizations. This metadata is included in the application bundle that is uploaded onto the marketplace. Once an application is downloaded onto a given device, an on-device code analysis and transformation process will occur. Combining the metadata from both the OS and the application, a complete control flow graph and object-oriented hierarchy view becomes available. A high-level overview of this framework is shown in Figure 2.

Once this on-device static analysis and code transformation is complete, common static optimization techniques can be enabled. For this paper, we will use the Application Customizable BTB (ACBTB) static optimization technique described in [2] to demonstrate our methodology. However, it should be noted that our methodology can generally be applied to other static techniques as well. By analyzing the control flow graph of the entire program, the target addresses of all statically resolved branches will be known. In order to handle the large quantity of control instructions in mobile applications, large hotspots will each have separate static branch target distance contents, and a hybrid architecture that automatically switches between the BTB and a hotspot-localized ACBTB is proposed. Additionally, metadata is added to the virtual function tables generated by the compiler for dynamic dispatch. Using this metadata, all indirect branches caused by polymorphism can also be resolved in a just-in-time fashion.

A. Extracting Source Code Metadata

The first step of this framework is to capture high-level information related to the program control flow and object-oriented hierarchies that would otherwise be unavailable by the time the application arrives on the mobile device. When a developer creates an application and compiles it for inclusion in the mobile marketplace, the critical information related to all control flow and class hierarchies is captured.

At the time the application is compiled, information on the complete control flow graph and class hierarchy is not present due to much functionality coming from foundation library code. So, in order to allow control flow and class hierarchy analysis to occur at a later time, the application compiler toolchain is augmented to extract the known control flow and class hierarchy information and store it as part of the application deliverable, in a fashion similar to what the authors described in [13]. In the similar vein, the same metadata information is captured for the vendor-provided operating system foundation library classes and stored for later reference. Once the application is downloaded onto the phone or tablet, the combined metadata from the application and operating system will provide the complete view of all class hierarchies.

B. On-Device Analysis and Optimization

Once the particular application and foundation library are both present on a given physical device, full program interprocedural static analysis and optimizations can be performed. Unused library code is pruned away and only those library routines used by the application are statically linked into the new application binary. While the resulting binary will be relatively larger than the original application code, it is important to note that instruction code as a whole is quite small compared to other data such as bitmaps, sound files, and databases embedded in the application package, so the impact on overall storage space is trivial. On the other hand,
the benefit is a much more compact instruction space that will reduce sparsity and enable an efficient utilization of the control hardware structures.

The goal of this stage is to generate a static control flow graph of the entire program. For every statically resolvable control-flow instruction (branch, jump, function call, etc.), we can determine both the taken and not taken target instruction addresses. For those control-flow instructions that rely on indirect target address (typically due to polymorphism), only the not taken target is known a priori. For the taken case, the destination address will only be known with certainty at run time when the branch is executed. Thus, a run-time mechanism will be required to help resolve these cases of indeterminism in our proposed architecture.

C. Generation of Application-Specific Custom BTB Tables

In order to demonstrate our application-specific custom BTB microarchitecture, the nomenclature of [2] is used and shown in Figure 3. The ACBTB table contains all the relevant application information regarding control-flow structure. This table is directly indexed and contains an entry for each control-altering instruction. The $NT_D$ field specifies the number of instructions in the not-taken branch path to be executed before a subsequent control-altering instruction is encountered. The $NT_I$ field specifies the index to the ACBTB entry that contains the information about this subsequent control-altering instruction that will next appear in the not-taken path. Similarly, the $T_D$ and $T_I$ fields represent the distance and index of the subsequent control-altering instruction in the taken branch path. The $TA$ field specifies the target address that should be used to update the PC if the branch is taken. Lastly, the $Type$ field contains information about the control-altering instruction which helps facilitate the branch prediction process.

After a branch is executed, the $IND$ index register is updated to store the value of the $NT_I$ or $T_I$ field, depending on if the branch was not taken or taken, respectively. The $IND$ index register will be used to index into the ACBTB when the next branch instruction is encountered. Similarly, the $CNT$ counter register is loaded with the value of the $NT_D$ or $T_D$ field based on the branch direction. In subsequent execution cycles, the only activity performed by the architecture is to decrement the value of $CNT$ with each fetched instruction. To support multiple issue, superscalar processors, the value by which $CNT$ is decremented will correspond to the number of fetched instructions during each cycle. Once the $CNT$ counter reaches zero, a control-altering instruction is about to be fetched, which triggers an access to the ACBTB using the $IND$ index register to read the properties for that instruction.

Once the software analysis and optimization process described in the previous section is completed, the metadata describing the ACBTB table structure will be provided to the underlying hardware microarchitecture, as shown in Figure 4. When the application is loaded by the operating system, the metadata will also be populated into a memory location that the hardware can read from to populate the ACBTB table. Since the ACBTB contains a limited amount of storage capacity for describing control-altering instructions, not every branch instruction can be stored concurrently. In order to account for this, a basic minimum-cut hotspot analysis will identify partitions of the program flow where all control-altering instructions within that partition can fit within the ACBTB. Multiple instances of the ACBTB memory will be conveyed in the metadata, and instructions will be injected at the entry points of each hotspot to load the ACBTB table from memory. To identify when the control flow reaches an exit point within an ACBTB hotspot, the current ACBTB entry will have the corresponding $NT_I$ or $T_I$ field set to the special index value of zero.

Figure 5 shows the proposed hybrid architecture, with the ACBTB added to the prediction path. When the ACBTB is enabled within a hotspot, it will provide both identification of control-altering instructions and the target address for taken branches. During this time, the ACBTB will cause the standard
BTB to be clock- and voltage-gated, completely disabling the structure and saving both dynamic and leakage power. When the control flow exits an ACBTB-enabled hotspot, the ACBTB will disable itself and re-enable the standard BTB structure. Once the control flow encounters another hotspot, the ACBTB table will be populated from memory and re-enabled. One additional optimization is to detect if the same ACBTB hotspot is re-entered without any intervening hotspot overwriting the ACBTB table, in which case the loading from memory can be skipped since the data is already present in the ACBTB.

D. Handling Mobile Software Polymorphism

The optimality of any application-specific control hardware heavily depends on the amount of statically known control flow actions. Unfortunately, mobile applications contain a large amount of polymorphism that makes statically determining the actual destination of a branch challenging. Interprocedural optimizations can help increase the amount of statically-known control flow instructions whenever possible.

The remaining polymorphic calls after this optimization will rely on indirect (or register-based) control flow, where the taken target address is dynamically determined and not known at compile time. In order to handle such cases, the IND and CNT registers are dynamically set at the point when the target destination is determined. The challenge is to determine what the appropriate values for IND and CNT are for all possible destinations. Luckily, the run-time environment present in mobile devices can help overcome this challenge. It is typical for the mobile operating system to provide a centralized library routine to handle polymorphic dynamic dispatch. When the high-level language calls a method in a polymorphic fashion, the compiler will generate the corresponding call to the library routine which will resolve the polymorphic method call and then branch to that method by doing an indirect branch on a register populated with the method’s address.

For example, the iOS `objc_msgSend` routine provides a centralized location where all polymorphic method calls are resolved [14]. This code will identify the appropriate method implementation, populate the ip intra-procedure call scratch register with the method’s address, and then execute an unconditional register-based branch on that register (bx ip). Using this observation, the compiler logic that generates the virtual function address tables used during polymorphism is modified to also include the T_I and T_D values for each method. T_D will simply be the number of instructions between the start of the method and the first control-altering instruction, while T_I will be the index of that control-altering instruction within the ACBTB. Both of these are known statically for each individual method. Using this information, the library code for polymorphic calls can be modified to enable the ACBTB. Before each indirect branch instruction (bx), when the target ip register is read from the virtual function tables, the corresponding T_I and T_D values will also be loaded into the IND and CNT registers, respectively.

V. Experimental Results

In order to assess the benefits from this proposed framework, an actual commercial mobile processor is utilized. The target mobile device chosen is an iPhone 4S (dual ARM Cortex-A9 800MHz processor, 512MB DRAM) running the iOS 5.1.1 operating system (Darwin Kernel 11.0.0). A set of eight open-source, interactive iOS applications are used to benchmark the optimizations. A listing of these benchmarks and their respective descriptions is provided in Table I.

Each application is compiled using version 4.1.1 of Apple’s Xcode integrated development environment. There are three copies of the compiled application. One copy of the compiled application is the baseline without leveraging the proposed framework. Another version contains the annotated binary code leveraging the hybrid control architecture without any software optimizations. The third version is based on first optimizing the software to replace polymorphic call sites with static function calls, and then annotating the binary code to use the hybrid control architecture. Table II lists the number of control hotspots that were generated for both the unoptimized and optimized versions of the software code.

In order to evaluate the corresponding BTB power improvement for this proposal, all three versions of the application will need to be executed on the physical device. This is necessary, as the vast majority of mobile applications are highly interactive and require touchscreen stimuli in order to realistically function. The selected benchmarks are no exception to this characteristic. By running the application directly on the phone and interacting with the application, the dynamic control flow will closely represent typical usage. An additional benefit of

---

### Table I

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubbsy</td>
<td>Graphical world-based game (Cocos2D)</td>
</tr>
<tr>
<td>Canabalt</td>
<td>Popular run and jump game</td>
</tr>
<tr>
<td>DOOM Classic</td>
<td>3D first person shooter</td>
</tr>
<tr>
<td>Gorillas</td>
<td>Turn-based angle shooter (Cocos2D)</td>
</tr>
<tr>
<td>iLabyrinth</td>
<td>Puzzle navigation game (Cocos2D)</td>
</tr>
<tr>
<td>Molecules</td>
<td>3D molecule modeling and manipulation</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>Online encyclopedia reader</td>
</tr>
<tr>
<td>Wolfenstein 3D</td>
<td>3D first person shooter</td>
</tr>
</tbody>
</table>
TABLE II
NUMBER OF CONTROL HOTSPOTS FOR EACH BENCHMARK

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># Hotspots w/o Code Optim.</th>
<th># Hotspots w/ Code Optim.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubbsy</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>Canabalt</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>DOOM Classic</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>Gorillas</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>iLabyrinth</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>Molecules</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Wolfenstein 3D</td>
<td>9</td>
<td>8</td>
</tr>
</tbody>
</table>

running the application code directly on the smartphone is that the correctness of any code transformations are validated since they are being executed directly on the processor.

The GNU gcov test coverage tool is leveraged to instrument executed code and to capture the dynamic execution frequencies of each line of source, as well as how many call and branch instructions occurred. In particular, both the settings of Generate Test Coverage Files and Instrument Program Flow are enabled in the Xcode compiler, and the GCOV_PREFIX and GCOV_PREFIX_STRIP environment variables are set to redirect the generated output profiles for all source files into a central path. As a result, when the application is executed on the phone, a corresponding .gcda coverage file is created for every source file. These coverage files are then downloaded from the phone and analyzed within an environment that simulates the instruction trace accessing the BTB and/or ACBTB.

To generate the dynamic instruction profiles for the benchmarks, all three versions of the applications are run natively on the phone for a period of 5 minutes of active and constant usage. As we are interactively using the application, the exact user input and resulting control flow will vary between every single run. In order to help reduce any outliers and noise, this process of running the application for 5 minutes is repeated 10 times to generate an average for each application.

After collecting and analyzing all the gcov coverage files, a simulation of BTB and ACBTB accesses is conducted, keeping track of access counts, BTB miss rates, and time spent using the power-efficient ACBTB instead of the standard BTB. For the standard BTB, we used a 1024-set, 2-way set-associative configuration. For the ACBTB, a 1024-entry configuration was chosen. We used CACTI [15] and eCACTI [16] to estimate both the dynamic and static power consumption for the standard BTB as well as the ACBTB structures, assuming a 32nm feature size and leakage temperature of 85°C. Furthermore, we take into account the additional power incurred from the additional logic proposed by the ACBTB architecture including the control logic. Using this information, we are able to determine the total branch target prediction power improvement across the entire run-time for each of the aforementioned benchmarks, shown in Figure 6.

As one can see, our proposal without the polymorphic code optimization is able to reduce a significant amount of the power dissipation due to branch target address prediction. The geometric mean reduction in BTB power usage is a 90.2%. Our more aggressive proposal leveraging polymorphic call optimization in conjunction with the hybrid architecture is able to achieve an even greater average reduction of 93.7%. It is important to note that this large amount of power savings is obtained without any loss of performance. In fact, when polymorphic call optimizations are employed, performance will often improve as fewer instructions need to be executed in order to resolve dynamic dispatch calls.

Another analysis that can be done is to observe how much time the standard BTB is still utilized during program execution. Due to the limited size of the ACBTB, whenever the program is not within a ACBTB-enabled hotspot we have to fall back to the original BTB hardware. Figure 7 shows the percentage of executed cycles where the standard BTB was turned off (voltage gated). In the case of using the hybrid architecture without code optimization, on average 95.9% of the time the standard BTB was turned off and did not

Fig. 6. Overall Branch Target Prediction Subsystem Power Improvement

Fig. 7. Percent of Execution Cycles Where Standard BTB is Turned Off

2015 33rd IEEE International Conference on Computer Design (ICCD)
need to be queried for every single fetched instruction. When polymorphic code optimization was also used, the amount of time the more power-efficient ACBTB was active increased to an astonishing 99.1%. As one can see, leveraging our on-device code analysis and optimization framework facilitates even better results in the hybrid architecture.

Lastly, one of the additional benefits from utilizing the ACBTB instead of the BTB is that the ACBTB has no conflicts and will always result in a BTB hit (since all branches within the ACBTB are statically known). While standard BTB miss rates are not large, they still can degrade performance by limiting instruction pipeline throughput. By accessing the standard BTB only outside of the ACBTB hotspots, the standard BTB storage cache is less saturated and will have fewer conflict misses. Figure 8 demonstrates the reduction in target address lookup misses. As one would expect, when the proposed architecture is leveraged, the number of branch miss cycles is reduced by an average of 26.4%. When polymorphic method call optimization is added, the cache miss cycle reduction becomes 49.0%.

VI. CONCLUSIONS

High-performance mobile processors, such as those used in modern smartphones and tablets, are leveraging increasingly deep and complex pipelines. In order to ameliorate the impact of control flow changes on the pipeline, branch prediction hardware is also becoming standard in these devices. Unfortunately, branch prediction accounts for up to 10% of the power consumed by the processor, and the BTB in particular is the largest consumer. Efforts to utilize static optimization techniques are hindered by the mobile ecosystem. These mobile applications continue to leverage high-level, object-oriented programming languages, relying on polymorphism and dynamic dispatch. Furthermore, mobile operating systems continue to add robust foundation libraries in order to abstract and simplify the creation of complex UI-driven interactive applications. These aspects impede many compile time optimization techniques.

A novel framework for enabling on-device application analysis and optimization has been presented. While complete information is not present during initial compilation, critical information related to control flow and class method hierarchies is extracted during initial application compilation and conveyed along with the application binary when downloaded onto a device. A post-processing of the application binary can then take place on the device, leveraging the complete knowledge of the entire program space. In this fashion, static optimization techniques, such as the ACBTB architecture, can now be enabled on mobile devices. The benefits of such an approach have been demonstrated by using numerous highly-interactive mobile applications. The proposed framework provides noteworthy reductions in BTB power and has significant implications for mobile processors, especially high-performance, power-sensitive smartphones.

REFERENCES