Lightweight Runtime Control Flow Analysis for Adaptive Loop Caching

Marisha Rawlins and Ann Gordon-Ross*
Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611, USA
mrawlins@ufl.edu & ann@ece.ufl.edu
*Also with the NSF Center for High-Performance Reconfigurable Computing

ABSTRACT
Loop caches provide an effective method for decreasing memory hierarchy energy consumption by storing frequently executed code in a more energy efficient structure than the level one cache. However, due to code structure restrictions and/or costly design time pre-analysis efforts, previous loop cache designs are not suitable for all applications and system scenarios. In this paper, we present an adaptive loop cache that is amenable to a wide range of system scenarios, providing an additional 20% average instruction memory hierarchy energy savings (with individual benchmark energy savings as high as 69%) compared to the best previous loop cache design.

Categories and Subject Descriptors
B.3.2 [**Hardware**]: Memory Structures: Design Styles – *cache memories.*

General Terms
Design.

Keywords
Loop cache, low energy, architecture tuning, embedded systems.

1. INTRODUCTION
Since an embedded system's memory hierarchy consumes as much as 50% of total system power [12], much research focuses on reducing the memory hierarchy's power/energy consumption [6] [15]. Several optimization techniques exploit the 90-10 rule (90% of execution time is spent in 10% of the code [14] (critical regions)), such as filter [8][9] and loop caches [4][5][10].

Previous work introduces several loop cache design variations, however, the main purpose of any loop cache is to service as many instruction fetches as possible and to provide tagless cache accesses (eliminating power and time costly cache tag comparisons). Loop caches must also guarantee a 100% hit rate making loop cache design difficult because the instruction fetch location (loop cache or L1 cache) must be determined with 100% certainty before the instruction fetch is issued.

To ensure a 100% hit rate, not all loop cache designs can store all types of critical region code structures. The simplest code structure for a loop cache is straight-line code, or basic blocks, since all instructions are fetched in succession. Since critical region analysis shows that many critical regions are basic blocks [14], Motorola introduced a simple counter-based dynamically loaded tagless loop cache [10] (DLC) to exploit these types of critical regions. The DLC detected small loops using short backwards branch (*sbb*) instructions (any branch instruction with a small negative offset). The DLC was attractive because this optimization was completely transparent, requiring no application designer effort. However, the DLC was not able to store *complex critical regions* consisting of *control of flow changes* (cof) (i.e. taken jumps, branches, subroutine calls, etc.). Complex critical code regions include nested loops and loops with internal cofs (i.e. forward jumps or if/else statements).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GLSVLSI'10, May 16–18, 2010, Providence, Rhode Island, USA.
Copyright 2010 ACM 978-1-4503-0012-4/10/05...$10.00.

The preloaded tagless loop cache [5] (PLC) expanded the loop cache's applicability to a wider range of applications by storing multiple complex critical regions. The PLC used clever *exit bits* to indicate loop *exit conditions* (cofs which exit the PLC) and ensure a 100% hit rate, however, the PLC required an application designer to perform an offline pre-analysis step to determine the critical regions and associated exit bits, which were preloaded into the PLC during system startup making the PLC applicable to only static situations.

Since the *system scenario* (application behavior and desired application designer effort) dictates a loop cache's success and neither the DLC nor the PLC was best for all system scenarios, the hybrid loop cache [4] (HLC) combined the advantages of both the DLC and the PLC. The HLC separated the loop cache into two partitions: a larger PLC partition and a smaller DLC partition (for critical regions not preloaded). Whereas the HLC increased the loop cache's applicability to a larger set of system scenarios, the main disadvantage was that the PLC could not cache complex critical regions that were not preloaded, thus reducing the HLC's functionality to that of a DLC.

In order to address disparate loop cache behavior, we present the adaptive loop cache (ALC), which integrates the PLC's complex functionality with the DLC's runtime flexibility. The PLC adapts to nearly any system scenario via lightweight runtime critical region control flow analysis and dynamic loop cache exit condition evaluation, which eliminates costly designer pre-analysis efforts and the uncertainly in choosing the "best" loop cache design. We evaluate the ALC's system scenario flexibility on several benchmark suites and reveal additional energy savings as high as 69% as compared to previous loop caches.

2. RELATED WORK
Much previous work focused on decreasing memory hierarchy energy consumption by reducing energy consumption of the instruction cache using techniques which replace costly instruction cache accesses with accesses to smaller, more energy efficient structures. The filter cache [9], a direct predecessor of tagless loop caches, reduced L1 instruction cache fetches with an additional small direct-mapped level zero (L0) (Figure 1 (a)) at the expense of an increased cache miss penalty.

Several techniques introduced methods to reduce or eliminate filter cache misses, such as the tagless hit instruction cache (TH-IC) [8]. The TH-IC eliminated the additional miss penalty by including

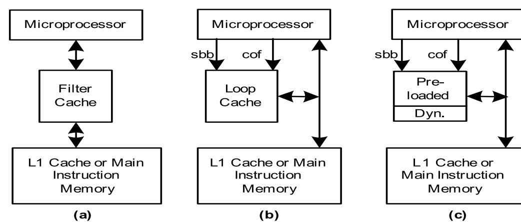

Figure 1: Architectural placement of the (a) filter cache (b) the dynamic, preloaded, and adaptive loop caches (DLC, PLC, and ALC, respectively) and (c) the hybrid loop cache (HLC).

meta-bits (similar in concept to exit bits) with each instruction and cache line to ensure a 100% hit rate, but however, imposed several architectural overheads including a large area overhead for the meta-bits, considerable instruction fetch unit augmentations and microprocessor modifications.

Compared to previous loop cache designs, the TH-IC provided the only advanced mechanism for dynamically caching complex critical regions. In this paper, we introduce the ALC for complex critical region caching without incurring the microprocessor architectural modifications and complex meta-bit invalidation overheads (area and performance) imposed by the TH-IC.

3. PREVIOUS LOOP CACHE DESIGNS

Even though previous loop cache designs provided methodologies suitable for different system scenarios, no single loop cache design is the *best* loop cache design (lowest energy loop cache) for every system scenario. An appropriate loop cache design choice based on the system scenario is critical as an inappropriate loop cache can increase memory hierarchy energy consumption (Section 5.2).

Figure 1 (b) and (c) depict loop cache architectural layout. The loop cache is a small instruction cache placed in parallel with the L1 cache such that *either* the loop cache or the L1 cache services an instruction fetch. Minor microprocessor architectural modifications include two control signals, *sbb* and *cof*, asserted by the instruction decode and branch resolution phases. All loop cache designs may store loops that are larger than the loop cache size. In these cases, given a loop cache of size *M*, the loop cache simply stores the first *M* static instructions.

In the remainder of this section, we provide operational background and architectural fundamentals of previous loop cache designs necessary to build a foundation for our ALC.

3.1 Dynamic Loop Cache (DLC)

The DLC [10] (Figure 1 (b)) has three operational states: idle, fill, and active. On a triggering sbb (i.e. a loop's last branch instruction is taken, returning execution to the loop's first instruction), the DLC transitions from the idle to fill state to store instructions while the L1 cache services instruction fetches on the second loop iteration. On the next iteration, the DLC transitions to the active state and services instruction fetches until the triggering sbb is not taken.

The DLC's main limitation is that it must cache all loop instructions during a single iteration in order to provide a 100% hit rate. If internal loop cofs such as forward jumps are encountered, the DLC returns to the idle stage since the DLC must not contain *gaps* (DLC locations which do not store valid instructions) because these gaps cannot be identified during the active state. DLC advantages include zero application designer effort and excellent performance for suitable system scenarios, but inappropriate system scenarios, such as the presence of forward jumps within a loop and nested loops where the inner loop has few iterations, can cause DLC thrashing. In these situations, the DLC constantly transitions between the idle and fill states, and never transitions to the active state.

3.2 Preloaded Tagless Loop Cache (PLC)

The PLC [5] (Figure 1 (b)) requires additional offline, application designer effort. During a pre-analysis step, application designers identify critical regions and determine corresponding loop exit bit values which are preloaded during system startup and remain fixed.

The PLC contains sets of loop address registers (LARs) indicating the start and end address of each stored critical region, as well as the critical region's location/offset in the PLC. After a cof, if the next instruction's address falls within a PLC's critical region, the PLC transitions from the idle to active state. Two exit bits stored with each PLC instruction provide a seamless transition (no cycle penalties) between the active and idle states. An instruction's exit bits consider cofs and their associated targets and indicate whether the PLC or the L1 cache should service the next instruction fetch based on whether or not the cof is taken or not taken.

The PLC provides higher loop cache access rates for straight-line loops as compared to the DLC (the PLC requires no runtime filling) and can efficiently cache loops that would thrash the DLC. However, PLC drawbacks include limitations on the number of stored critical regions and additional PLC index address translation. Most importantly, the inherent static nature of the PLC makes it unsuitable for dynamic system scenarios.

3.3 Hybrid Tagless Loop Cache (HLC)

The HLC [4] (Figure 1(c)) leverages the advantages of both the DLC and the PLC to increase system scenario amenability. The HLC partitions the loop cache in to a larger PLC partition and a smaller DLC partition. During execution, the HLC first checks the PLC's LARs on any cof. If this check fails, and the cof is a triggering sbb, the DLC partition begins filling.

Whereas the HLC appears to provide the best of both techniques, one main disadvantage is that if the application behavior changes or an application designer does not perform the necessary pre-analysis, the PLC is not used (but still expends static energy and increases DLC dynamic energy) and HLC operation reduces to a DLC.

4. THE ADAPTIVE LOOP CACHE (ALC)

The ALC's novelty combines the DLC's dynamic flexibility with the PLC's ability to cache complex critical regions using lightweight runtime control flow analysis. Thus, the ALC's system scenario amenability spans that of the DLC and the PLC.

The ALC is highly flexible and suitable for all system scenarios and eliminates the costly pre-analysis step required to cache complex critical regions in the PLC/HLC while accurately identifying critical regions using actual operating inputs during runtime. Additionally, whereas the PLC and HLC can be useful in system scenarios where application behavior is static, this inflexibility makes the PLC and HLC unsuitable for system scenarios where behavior changes due to application phase changes [6], input vectors changes [3], or application updates.

Like the DLC, the ALC is initially filled during the loop's second iteration while instructions are fetched from the L1 cache and the ALC supplies the processor with instructions for the remaining iterations. Each ALC entry contains an instruction and the corresponding exit bits indicating whether the next instruction

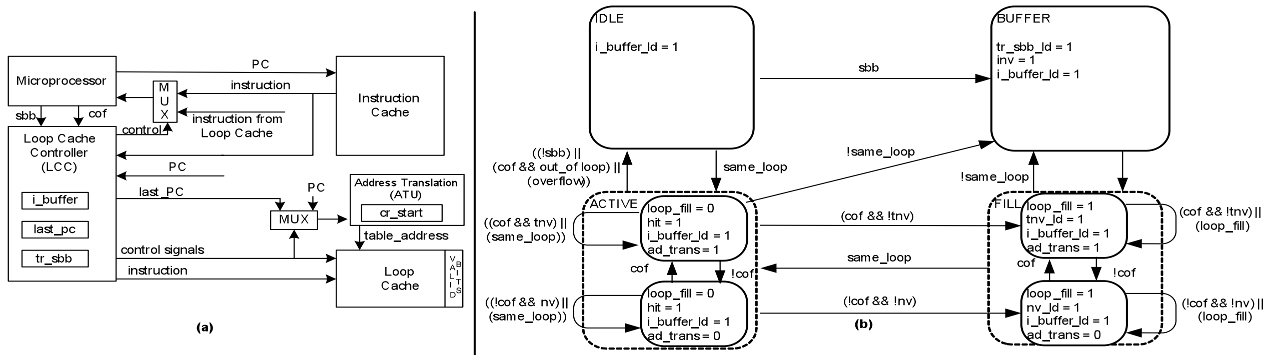

Figure 2: The adaptive loop cache (ALC) (a) architecture and (b) loop cache controller (LCC) state machine operation. Runtime control flow analysis occurs in the *fill* state.

should exit the loop cache and fetch from the L1 cache or continue fetching from the loop cache. The exit bits are critical for transferring control from the ALC to the L1 cache without a cycle penalty as the ALC must guarantee a 100% hit rate to not affect system performance.

In the fill state, the ALC buffers an instruction during runtime to evaluate the exit bits. The ALC uses the current instruction (supplied by the L1 cache) and the previously buffered instruction to determine the exit bits then writes the previously buffered instruction and its corresponding exit bits to the loop cache. This continues until the sbb is taken again (the ALC then transitions to the active state), a new loop is encountered (the ALC is filled with the new loop), or the sbb is not taken (the loop is no longer being executed).

Unlike the DLC, the ALC caches complex critical regions containing branches. In the active state, the ALC supplies the processor with instructions and uses the exit bits to determine the location of the next instruction fetch. The ALC supplies the next instruction for straight-line code or when a cof is encountered but not taken and the *next_valid* exit bit is set, or when a cof encountered is taken and the *taken_next_valid* exit bit is set. Otherwise, the L1 cache supplies the instruction. If execution remains within the same loop, the ALC returns to the filling state to fill the gap left during the initial filling cycle (the L1 cache continues serving instruction fetches) then transitions to the active state if the sbb is taken again.

Since we do not restrict the ALC to caching only loops with straight-line code, it is possible to have gaps (invalid instructions) in the ALC (note that these invalid instructions will never be fetched from the ALC since the program counter is used to access the correct ALC instruction). For example, if a loop contains an if/else cof instruction, the instructions belonging to either the if clause or the else clause will be initially cached leaving a gap in the ALC. However, since the loop cache returns to the filling state when gaps are encountered, it is possible for the ALC to contain instructions from both the if and else clause and therefore, for the cof's *next_valid* and *taken_next_valid* exit bits to be set simultaneously. If the sbb is not taken again or if a cof causes a jump to a target outside of the ALC range, the ALC transitions to the idle state while the L1 cache supplies instructions.

4.1 ALC Architectural Layout

The ALC's architectural placement and use of the *sbb* and *cof* microprocessor signals is identical to the DLC and PLC (Figure 1 (b)). Figure 2 (a) depicts the detailed architectural layout of the ALC. The ALC contains a *loop cache controller* (LCC) to orchestrate loop cache operation, the *loop cache* to store instructions, and an *address translation* unit (ATU) for loop cache indexing.

The LCC contains three internal registers, *tr_sbb, last_PC,* and *i_buffer*, which collectively enable complex critical region caching and provide support for loop cache gaps. *Tr_sbb* stores the currently cached critical region's triggering sbb address, thus the *tr_sbb* effectively identifies the currently cached critical region. Therefore, when the microprocessor asserts *sbb*, the LCC uses the new triggering sbb's address to determine if it corresponds to the currently cached critical region or if execution has entered a new critical region. If the new triggering sbb address matches *tr_sbb,* the LCC asserts an internal signal, *same_loop*, which enables the ALC to resume fetching the currently cached critical region (for critical regions larger than the loop cache). *I_buffer* and *last_PC* assist in control flow analysis by buffering the previously fetched instruction and address while the next instruction fetch location is evaluated. Special *valid bits* appended to each instruction in the loop cache store control flow analysis information.

4.2 ALC Functionality for Critical Regions with Forward Branches

Complete ALC operation requires cooperation between the LCC and the ATU in order to fill the loop cache and service instruction fetches. In this section, we describe ALC functionality necessary for caching complex critical regions with forward branches.

4.2.1 Address Translation Unit (ATU)

In order to cache complex critical regions, maintain tagless loop cache accesses, and use indexing counters, the ATU translates the current instruction address into a loop cache index. During straight-line code, the ATU uses an indexing counter to step through the loop cache. However, when a cof occurs, the ATU must translate the instruction's address to the loop cache index. The ATU contains an internal register, *cr_start*, which stores the critical region's first instruction address. When the LCC asserts *ad_trans* indicating a cof, the ATU subtracts *cr_start* from the current instruction address (last_PC during loop cache filling or PC during loop cache fetching) to obtain the loop cache index. This new loop cache index becomes the indexing counter, which is incremented for each subsequent straight-line instruction access.

4.2.2 Loop Cache Controller (LCC) and Runtime Control Flow Analysis

Figure 2 (b) depicts LCC state machine operation. In the idle and active states, instructions are fetched from the L1 cache and the loop cache, respectively. In the buffer and fill states, instructions are fetched from the L1 cache and written to the loop cache after runtime control flow analysis.

ALC operation begins in the idle state, with two possible exit transitions. If *same_loop* is asserted (corresponding to a triggering sbb for a currently cached critical region, Section 4.1), the LCC transitions to the active state. If *sbb* is asserted and *same_loop* is not asserted, the triggering sbb corresponds to a new critical region and the LCC transitions to the buffer state.

The LCC spends one cycle in the buffer state to prepare the loop cache for filling by resetting all *valid bits* to invalidate the currently stored critical region. The LCC asserts *tr_sbb_ld* to store the triggering sbb address in *tr_sbb* and asserts *i_buffer_ld* to buffer the current instruction and address into *i_buffer* and *last_PC*, respectively.

On the next clock cycle, the LCC transitions to the fill state and performs an initial runtime control flow analysis pass to dynamically determine the value of the two *valid bits, next_valid* and *taken_next_valid* (*nv* and *tnv*, respectively, in Figure 2(b)), which store loop exit condition information for each instruction. *Valid bits* (similar to the PLCs exit bits) indicate whether the L1 cache or loop cache should service the next instruction fetch.

Valid bits are critical to loop cache operation and maintain a 100% hit rate as *valid bits* indentify cofs with targets outside of the loop cache and allow gaps within the loop cache. During straight-line code execution in the fill state (including untaken branch instructions), control flow analysis sets the *next_valid* bit for these instructions indicating that the next sequential instruction is stored in the loop cache. If a cof occurs and the cof's target is within the loop cache bounds, control flow analysis sets the *taken_*n*ext_valid* bit for this instruction indicating that when the cof occurs, the target instruction is stored in the loop cache. Note that at this point, control flow analysis sets only one *valid bit*, leaving the other *valid bit* unset. Since the initial control flow analysis pass operates on one loop iteration, both *valid bits* cannot be set. In order to fill in unset *valid bits* and loop cache gaps, the LCC may return to the fill state for additional control flow analysis passes.

During the fill state, *i_buffer* serves two purposes. First, *i_buffer* enables control flow analysis to compare the previous instruction executed with the subsequent instruction executed to evaluate loop exit conditions. The second purpose significantly reduces loop cache writes and eliminates the need for a dual-ported loop cache. Previous loop cache designs simultaneously write instructions as the instructions are fetched from the L1 cache. However, since the ALC's control flow analysis determines an instruction's *valid bits* after the next instruction fetch, each loop cache instruction would require two updates: one update to write the current instruction and one update to write the previous instruction's *valid bits*. Furthermore, these updates refer to two different loop cache lines, resulting in two writes per clock cycle (necessitating a dual-ported loop cache). In order to avoid a dual-ported loop cache, *i_buffer* buffers the previous instruction during control flow analysis such that each loop cache instruction only requires one loop cache update. The loop cache is *backfilled* with the previous instruction and associated *valid bits* while the current instruction is fetched from the L1 cache (backfilling the loop cache and fetching from the L1 cache take one cycle). Because *i_buffer* is read at the beginning of and written at the end of the clock cycle, *i_buffer* always latches the current instruction after the previous instruction is sent to the loop cache.

The LCC continues this backfilling process until one of several conditions is met. If a new triggering sbb is taken (*same_loop* is not asserted), the LCC transitions to the buffer state to prepare for the new loop. If execution reaches the end of the loop, and the loop's triggering sbb (*tr_sbb*) is taken again (*same_loop* is asserted), the LCC transitions to the active state (the LCC backfilled the last loop instruction at the end of the loop's first iteration, thus the last instruction is actually the first instruction written to the loop cache).

In the active state, the LCC deasserts *loop_fill* to allow the loop cache to service instruction fetches (in a single cycle). The LCC buffers the instruction fetched from the loop cache and associated *valid bits* in *i_buffer* to prepare for additional control flow analysis passes. The LCC analyzes *valid bits* to determine the next instruction's fetch location. If the current instruction is not a cof and *next_valid* is set or if the current instruction is a cof and *taken_next_valid* is set, the LCC remains in the active state. Otherwise, the L1 cache must service the next instruction fetch and the LCC transitions out of the active state.

There are several LCC transitions out of the active state. If the *next_valid* bit is not set and the instruction corresponds to the last instruction in the loop cache (*overflow* is asserted), the LCC transitions to the idle state. Otherwise, the unset *next_valid* bit indicates a loop cache gap and the LCC transitions to the fill state to perform another control flow analysis pass. If the LCC detects a new loop (a triggering sbb is taken and is not *tr_sbb*), the LCC transitions to the buffer state to prepare the loop cache for a new loop. If the current loop's triggering sbb (*tr_sbb*) is not taken, the LCC transitions to the idle state. Lastly, if a cof's target (identified during instruction decode) is outside the loop range (past the triggering sbb) the controller's *out_of_loop* signal is asserted and the controller transitions to the idle state.

4.3 ALC Functionality for Critical Regions with Nested Loops

We modified the LCC described in Section 4.2 to include functionality for caching critical regions with nested loops. However, experimental results showed no average improvement and only little improvement in loop cache access rates and energy savings for a very small number of benchmarks.

5. EXPERIMENTAL RESULTS

5.1 Experimental Setup

To determine the number of instructions fetched from the loop cache and calculate energy savings for the DLC, PLC, HLC, and ALC, we executed 31 benchmarks from the EEMBC [2], MiBench [7], and Powerstone [11] benchmark suites (all benchmarks were run to completion, however, due to incorrect execution not related to the loop caches, we could not evaluate the complete suites). We implemented each loop cache design in SimpleScalar [1] using the PISA instruction set (64 bit instructions and 32 bit addresses). We evaluated small loop cache sizes ranging from 4 to 256 entries Since [4] suggested that the HLC's ideal DLC partition size was 32 entries, we present HLC results for 64-, 128-, and 256-entry loop caches only.

To calculate energy consumption for each loop cache configuration, we augmented the energy model adopted by Zhang et al. [15] and Gordon-Ross et al. [4] to include loop cache fill and fetch operations as shown here (IC = instruction cache, LC = loop cache):

total energy =IC_energy + LC_energy
IC_energy = IC_fill_energy + IC_dynamic_energy + IC_static_energy
*IC_fill_energy = ((IC_misses * (IC_linesize / wordsize) * mem_energy_perword))+ cpu_stall_energy*
*IC_dynamic_energy = IC_accesses * IC_access_energy*

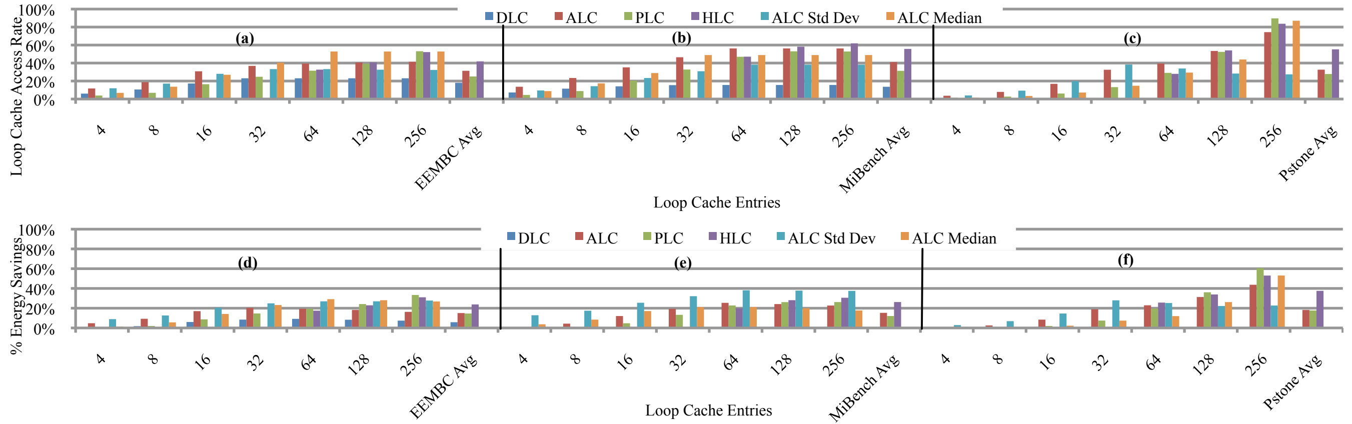

Figure 3: Experimental results for the DLC, ALC, PLC, and HLC for varying loop cache sizes (entries) showing percentage of instruction fetches resulting in loop cache hits (access rate) averaged for the (a) EEMBC, (b) MiBench, and (c) Powerstone benchmark suites and energy savings (normalized to the base system) averaged for the (d) EEMBC, (e) MiBench, and (f) Powerstone benchmark suites.

$$IC_static_energy = (IC_misses * miss_latency_cycles) + (IC_accesses - IC.misses) + LC_hits) * 0.15 * IC_access_energy$$

$$LC_energy = ((LC_hits + LC_fills) * LC_access_energy) + LC_static_energy$$

$$LC_static_energy = (IC_misses * miss_latency_cycles) + (IC_accesses - IC.misses) + LC_hits) * 0.15 * LC_access_energy$$

We gathered *IC_accesses*, *LC_hits*, and *LC_fills* cache statistics using SimpleScalar. We used CACTI [13] to determine dynamic cache energy dissipation for 0.18um technology. Since the loop cache is tagless, we obtained energy consumption for same sized direct-mapped caches with an 8-byte line size (corresponding to one loop cache entry for one instruction) and removed the tag energy. We assumed static energy per clock cycle for the instruction and loop caches as 15% of their respective dynamic access energies.

In order to compare to a system with no loop cache, we define a *base system* configuration consisting of an 8 KB L1 instruction cache with 4-way set associativity and a 32-byte line size – a configuration shown in [15] to perform well for a variety of benchmarks on several embedded microprocessors. To calculate energy savings, we normalize the energy consumption of a system with a loop cache and the L1 base cache to the base system with no loop cache.

5.2 Results and System Scenario Analysis

Figure 3 depicts experimental results comparing the DLC, ALC, PLC, and HLC loop cache designs for various loop cache sizes in number of entries (*entries* on the x-axis). In order to show different loop cache design trends across different benchmark suites, we present results averaging the EEMBC, MiBench, and Powerstone benchmark suites separately.

Figure 3 (a), (b), and (c) depict the percentage of instruction fetches resulting in loop cache hits (loop cache access rate) and reveal that our ALC always out performs the DLC for all three benchmark suites and all loop cache sizes. This result is expected since the DLC caches a subset of the loops that the ALC caches. These figures also reveal the DLC's ineffectiveness during particular system scenarios. For the Powerstone benchmark suite and all loop cache sizes (Figure 3 (c)), the DLC never resulted in loop cache hits due to the absence of straight-line loops (we verified this using a loop analysis tool [14]). Average loop cache access rate improvements for the ALC compared to the DLC reach as high as 18.43%, 40.63%, and 74.34% for EEMBC, MiBench, and Powerstone benchmark suites, respectively.

For individual benchmarks, the ALC loop cache access rate reaches as high as 97.91% for the EEMBC benchmark suite and as high as 99.74% and 99.49% for the MiBench and Powerstone benchmark suites, respectively. These high access rates and DLC's ineffectiveness for certain application scenarios results in loop cache access rate improvements as high as 97.82% for the ALC compared to the DLC for EEMBC's PNTRCH01 benchmark and as high as 99.70% and 99.48% for MiBench's CRC and Powerstone's blit benchmark, respectively.

Figure 3 (a), (b), and (c) also show that for the EEMBC, MiBench, and Powerstone benchmark suites, respectively, on average, our ALC either out performs or performs as well as the PLC and HLC for loop cache sizes up to 128 entries. In addition, Figure 3(b) shows that for the MiBench suite, on average, the ALC always out performs the PLC and the difference between the ALC and HLC loop cache access rate is only 5% for the a 256 entry loop cache. Thus, the ALC is ideal for sized constrained applications. Increasing the loop cache size to 256 entries results in the PLC/HLC storing almost all of the application's critical regions, thus after this point the PLC/HLC outperforms the ALC on average since the PLC/HLC requires no runtime filling cycles. However, since the ALC outperforms the PLC/HLC for small loop caches, this shows that the ALC's filling overhead is minimal and the ALC's inability to cache subroutines and nested loops does not noticeably affect performance.

Evaluating individual benchmarks reveals that the ALC consistently outperforms the PLC for applications with a large critical region or many critical regions, which would require prohibitively large loop caches, such as with MiBench. For EEMBC's TBLOOK01 benchmark, the ALC out performs both the PLC and HLC for all loop cache sizes since a 256-entry PLC (a 224-entry PLC partition in the case of the HLC) was not large enough to store a single critical region in its entirety. Although these larger PLCs/HLCs could potentially incur more loop cache hits, in some cases the additional loop cache hits would not compensate for the increased energy consumption incurred by fetching from a larger loop cache, thus increasing overall energy

consumption. In addition, highly size constrained system scenarios may not afford the larger loop caches required for optimal PLC/HLC energy savings or performance.

We further note that certain system scenarios may result in the PLC outperforming the ALC for all loop cache sizes. In this scenario, critical loops are executed frequently but only iterate a few times successively i.e. enough iterations to fill the ALC but not enough iterations to take advantage of fetching from the ALC (e.g. EEMBC's A2TIME01 benchmark). However, we point out that even in these scenarios, the flexibility of the ALC (no designer pre-analysis effort and the ability to conform to changes in application behavior and phases) may still outweigh the increased PLC performance.

Figure 3 (d), (e), and (f) depict average memory hierarchy energy savings for the EEMBC, MiBench, and Powerstone benchmarks, respectively. As expected, for all three benchmark suites, our ALC outperforms the DLC due to increased ALC hits for a same sized DLC. Average ALC energy savings improvements over the DLC reach as high as 12%, 26%, and 49% for the EEMBC, MiBench, and Powerstone benchmark suites, respectively, and reach as high as 69% for Powerstone's blit benchmark. For the Powerstone benchmark suite, the DLC always increases energy consumption compared to the base system because thrashing results in no loop cache accesses. The PLC and ALC can also result in negative energy savings for the 4-entry loop cache since a 4-entry loop cache can be too small to incur enough loop cache accesses needed to translate into energy savings.

Figure 3 (d), (e), and (f) show that our ALC saves more energy than the PLC for loop cache sizes less than 64 entries. We point out that on average the ALC does not outperform the PLC/HLC for larger loop cache sizes due to the absence of PLC/HLC (PLC partition) filling (however, individual benchmarks show additional energy savings for the ALC compared to the PLC as high as 53% for EEMBC's PNTRCH01 benchmark with a 64-entry loop cache.

Both the PLC and HLC require designer pre-analysis effort to achieve energy savings – we reiterate that the HLC is a combination of a fixed size (32-entry) DLC partition and a PLC partition that can be increased in size to obtain similar results as an HLC with 64, 128, and 256 entries. Thus, any energy savings obtained by increasing the HLC size is due to increasing the HLC's PLC partition, and therefore requiring designer effort to achieve these savings. Without designer effort, the HLC would be reduced to only a 32entry DLC, which cannot match the energy savings of the ALC. The ALC's strength is that the ALC can provide energy savings near that of the PLC/HLC *without* the application designer's pre-analysis effort and *without* requiring a static system scenario. Since static pre-analysis is not acceptable for applications with changing behavior, the PLC and HLC are not effective for every system scenario.

Figure 3 (d) shows a decrease in energy savings for the ALC with a 256-entry loop cache. At this point, the additional loop cache hits for increased loop cache sizes does not outweigh the additional energy required for the larger loop cache. (Figure 3 (a) shows that the loop cache hit rate for the ALC levels off at 64 entries). Additional experiments run for loop cache sizes greater than 256 entries revealed a negligible increase in loop cache energy savings or a decrease in energy savings for all loop cache designs.

6. CONCLUSIONS

In this paper, we introduced the adaptive loop cache (ALC) – a dynamic tagless loop cache that combines the flexibility of the dynamic loop cache (DLC) with the preloaded loop cache's (PLC's) ability to cache complex critical regions. The ALC significantly increases system scenario amenability compared to previous loop cache designs using a lightweight runtime control flow analysis technique to dynamically cache complex critical regions in an area efficient manner. Furthermore, the ALC eliminates the need for the costly designer pre-analysis effort required for the PLC and HLC, making the ALC the most appropriate design for applications with changing behavior. The ALC increases the average loop cache access rate and average energy savings by as much as 74% and 20%, respectively, compared to previous loop cache designs. Future work includes investigating benefits gained by combining the ALC with other dynamic energy saving techniques such as cache configuration.

7. REFERENCES

[1] Burger, D., Austin, T., Bennet, S. Evaluating Future Microprocessors: The SimpleScalar ToolSet. University of Wisconsin-Madison. Computer Science Department. Tech. Report CS-TR-1308, July 1996.

[2] EEMBC. http://www.eembc.org/.

[3] Eeckhout, L., Vandierendonck, H., De Bosschere, K. Workload design: Selecting Representative Program-input Pairs. 2002 International Conference on Parallel Architectures and Compilation Techniques.

[4] Gordon-Ross, A. and Vahid, F. Dynamic Loop Caching Meets Preloaded Loop Caching – A Hybrid Approach. IEEE International Conference on Computer Design (ICCD), 2002.

[5] Gordon-Ross, A., Cotterell, and Vahid, F. Exploiting Fixed Programs in Embedded Systems: A Loop Cache Example. Computer Architecture Letters, Volume 1, January 2002.

[6] Gordon-Ross, A., Lau, J., and Calder, B. Phase-based Cache Reconfiguration for a Highly-configurable Two-level Cache Hierarchy. 18th ACM Great Lakes Symposium on VLSI . GLSVLSI '08.

[7] Guthaus, M.R., Ringenberg, J.S., Ernst,D., Austin, T.M., Mudge, T., Brown, R.B. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. IEEE 4th Annual Workshop on Workload Characterization, December 2001.

[8] Hines, S., Whalley, D., and Tyson, G. Guaranteeing Hits to Improve the Efficiency of a Small Instruction Cache. IEEE/ACM International Symposium on Microarchitecture 2007.

[9] Kin, J., Gupta, M., and Mangione-Smith, W. H. The Filter Cache: an Energy Efficient Memory Structure. ACM/IEEE International Symposium on Microarchitecture 1997

[10] Lee, L. H., Moyer, W., Arends, J. Low cost Embedded Program Loop Caching – Revisited. University of Michigan Technical Report Number CSE-TR-411-99, December 1999.

[11] Scott, J.,Lee, L., Arends, J., Moyer, B. Designing the Low- Power M~CORE Architecture. International Symposium on Computer Architecture Power Driven Microarchitecture Workshop, July 1998.

[12] Segars, S. Low Power Design for Microprocessors. International Solid State Circuit Conference, February 2001.

[13] Shivakumar, P., G., Jouppi, N.P. Cacti3.0: an Integrated Cache Timing and Power Model. COMPAQ Western Research Lab, 2001.

[14] Villarreal, J., R. Lysecky, S. Cotterell, and F. Vahid. A Study on the Loop Behavior of Embedded Programs. Technical Report UCR-CSE-01-03, University of California, Riverside, 2002.

[15] Zhang, C., F. Vahid, W. Najjar. A highly-configurable Cache Architecture for Embedded Systems. 30th Annual International Symposium on Computer Architecture, June 2000