T-SPaCS—A Two-Level Single-Pass Cache Simulation Methodology

Wei Zang, *Student Member*, *IEEE*, and Ann Gordon-Ross, *Member*, *IEEE*

Abstract—The cache hierarchy's large contribution to total microprocessor system power makes caches a good optimization candidate. To facilitate a fast design-time cache optimization process, we propose a single-pass trace-driven cache simulation methodology—T-SPaCS—for a two-level exclusive cache hierarchy. Direct adaptation of conventional trace-driven cache simulation to two-level caches requires significant storage and simulation time as numerous stacks record cache access patterns for each level one and level two cache combination and each stack is repeatedly processed. T-SPaCS significantly reduces storage space and simulation time using a set of stacks that only record the complete cache access pattern. Thereby, T-SPaCS simulates all cache configurations for both the level one and level two caches simultaneously in a single pass. Experimental results show that T-SPaCS is 21.02X faster on average than sequential simulation for instruction caches and 33.34X faster for data caches. A simplified, but minimally lossy version of T-SPaCS (simplified-T-SPaCS) increases the average simulation speedup to 30.15X for instruction caches and 41.31X for data caches. We leverage T-SPaCS and simplified-T-SPaCS for determining the lowest energy cache configuration to quantify the effects of lossiness and observe that T-SPaCS and simplified-T-SPaCS still find the lowest energy cache configuration as compared to exact simulation.

Index Terms—Cache memories, low-power design, real-time systems and embedded systems, simulation

1 Introduction

Since the cache hierarchy can consume as much as 50 percent or more of total system power [26], caches are a good candidate for optimization in low-power embedded systems. Research shows that applications have varying cache requirements for optimal energy consumption [36]. Specializing cache parameters, such as total cache size, block size, and associativity to an application's temporal and spatial locality characteristics can reduce energy consumption by as much as 40 percent on average [4], [17]. *Cache tuning*, a prevailing optimization technique, determines the best cache configuration (specific cache parameter values) in the *design space* (all possible cache configurations) for the entire application [14], [36] or each application phase [16], [27].

Cache tuning can be performed at design time—offline static cache tuning—or during runtime—online dynamic cache tuning. Offline static cache tuning is suitable for stable systems with predictable inputs and execution behavior. Since an embedded system usually executes a fixed application or set of similar applications, cache tuning can be specifically applied to optimize the cache for these predictable embedded applications. System designers determine cache parameter values during design time and set these values in synthesizable soft-core processors that provide numerous cache parameters for customization [1], [2], [31]. For systems using hard-core processors [4], [15] that contain configurable caches, the optimal energy cache configurations could still be determined a priori, and the hard-core processors would be configured to the designer-specified parameter values at system startup. Since static cache tuning determines cache parameter values prior to system runtime, static cache tuning introduces no runtime overhead with respect to design space exploration.

Alternatively, online dynamic cache tuning determines the best cache configuration by exploring the design space during runtime [17] and requires no system designer effort. This in-system exploration capability enables dynamic cache tuning to adaptively react to the system's environment and input changes [5], [14], [16], [36]. Dynamic adaptation enables potentially more accurate cache configurations (and thus lower energy consumption) for unpredictable or dynamic applications as compared to static cache tuning. However, online design space exploration may interfere with system behavior and impose system overheads such as performance, area, and power/energy while exploring the design space. Additionally, determining when to apply dynamic cache tuning is challenging, as cache tuning must quickly react to changes in an application's behavior. Inappropriate cache tuning times may result in significantly higher energy consumption than running the system in a fixed base configuration without any cache tuning [14]. Due to these dynamic tuning challenges, in this paper, we focus on static cache tuning.

Most existing offline static methods determine the cache configuration using an analytical model or simulation. Analytical modeling quickly predicts cache performance by analyzing program locality or data reuse patterns using

- *W. Zang is with the Department of Electrical and Computer Engineering, University of Florida, 2841 SW 13th St. Apt. J238, Gainesville, Fl 32608. E-mail: weizang@ufl.edu.*
- *A. Gordon-Ross is with the Department of Electrical and Computer Engineering, University of Florida, PO Box 116200, Gainesville, FL 32611. E-mail: ann@ece.ufl.edu.*

Manuscript received 29 Oct. 2010; revised 7 July 2011; accepted 7 Sept. 2011; published online 30 Sept. 2011.
Recommended for acceptance by L. John.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2010-10-0597.
Digital Object Identifier no. 10.1109/TC.2011.194.

mathematical models [6], [12]. These models are usually built based on program characteristics that are independent of the processor type and operating system. Even though analytical modeling quickly determines the cache configuration, analytical modeling is generally inaccurate and produces suboptimal results. Simulation methods improve cache tuning accuracy by simulating each cache configuration to determine the lowest energy configuration. However, the simulation time required to iteratively explore a large design space is lengthy even though heuristics can be used to prune the design space [17], [36]. Even though relatively faster simulation methods can be used, such as functional simulation as compared to cycle accurate simulation, these methods still require long simulation time [17], [36].

Trace-driven cache simulation significantly reduces design exploration time by functionally simulating an application once to produce a memory reference trace (*access trace*), and then processing the access trace for each cache configuration with a fast cache simulator. The reason for time reduction is that lengthy functional simulation is only employed once and trace simulation is faster than functional simulation. Although access trace files are typically very large with a large storage space requirement and consequently slow processing time, approaches such as SimPoint [28], trace sampling [9], and trace compression [21] reduce these overheads.

Most tuning methods iteratively evaluate the design space, processing only one configuration on each simulation pass [10], resulting in lengthy design space exploration time. Instead of iteratively exploring the design space, single-pass trace-driven simulation evaluates multiple configurations simultaneously in a single simulation pass [19], [24], [29], [32], achieving simulation speedups on the order of tens [20], [33] as compared to iterative simulation. However, all previous single-pass trace-driven methods, to the best of our knowledge, only simulate single-level caches.

However, inherent multilevel cache execution characteristics make direct application of single-level cache single-pass simulation techniques challenging. For example, in a two-level cache hierarchy, the level one cache (L1) *filters* the access trace (all L1 accesses) and produces one *filtered access trace* for each level two cache (L2) (i.e., each unique L1 configuration's misses form a unique filtered access trace for L2). Since the L2 access trace is determined by each L1 configuration, each filtered access trace must be stored and processed separately. In addition, the design space increases approximately exponentially as the number of cache parameters increases because essentially each L2 configuration can be coupled with each L1 configuration, which increases the storage space and processing time requirements.

In this paper, we present for the first time (to the best of our knowledge) a *T*wo-level *S*ingle-*Pa*ss trace-driven *C*ache *S*imulation methodology—T-SPaCS for exclusive instruction and data caches for offline static cache tuning. The designers can employ T-SPaCS a priori at design time to evaluate all cache configurations for a particular embedded application, and then determine the optimal (lowest energy) cache configuration to be used during execution time. We leverage an exclusive cache hierarchy to limit area and processing overheads and enable the L1 and L2 to be logically analyzed as one single cache followed by a supplementary processing step to extract the exclusive L2 contents. Our proposed methodology determines the optimal cache configuration with high simulation speedup and low storage requirements compared to iterative simulation. For reference, Table 1 defines notations used throughout this paper.

TABLE 1
Notational Reference

$\gg$	Bitwise right shift operator.
$\cdot$	Multiplication.
B	Cache block size: $B = 2^b$.
S	Number of sets.
W	Number of ways.
Z	Total cache size: $Z = B \cdot S \cdot W$.
$X_{min/max}$	Subscript min/max represents the minimum /maximum value of X where X can be B, S, W, or Z.
$X^{1/2}$	Superscript 1 or 2 indicates L1 or L2, respectively.
C_B	Number of configurable B in the design space.
C_S	Number of configurable S in the design space.
$\vec{T}$	Instruction trace.
$T[t]$	Trace address.
$A[t]$	Block address of $T[t]$ (i.e., $A[t] = T[t] \gg b$).
$\overrightarrow{K_{i,t}}$	Stack recording unique block addresses of the trace addresses from $T[1]$ to $T[t]$ that have set index i for B and S^1_{min} in MRU order from top to bottom. $i \in [0, S^1_{min} - 1]$.
$K_{i,t}[m]$	The $m-th$ (counting from the stack's top) block address recorded in $\overrightarrow{K_{i,t}}$.
$K_{i,t}[h]$	Most recently previous access to the cache block that $T[t]$ maps to (i.e., $K_{i,t}[h] = A[t]$), acquired by stack processing of $\overrightarrow{K_{i,t}}$.
$\xi(S)$	All conflicts with $T[t]$ under S.
δ	Extracted L2 conflicts.
$\overline{\xi_\alpha}$	Conflicts associated with one of the complementary sets in L1. For a combination of S^1 and S^2, subscript $\alpha \in [1, S^1/S^2 - 1]$.
$\{Y\}$	Collection of Y (Y can be $\xi(S)$, δ, or $\overline{\xi_\alpha}$), listing elements in MRU order.
$\|Y\|$	Cardinality of $\{Y\}$.

2 Related Work

There exists much previous work in single-pass trace-driven cache simulation, with each variation focusing on expanding the design space and reducing the processing time via new data structures and processing techniques.

Mattson et al. [24] first proposed the stack-based algorithm, wherein a stack data structure stored the access trace (Section 4 presents stack processing details). For each access, a stack search determined the minimum cache size necessary for that access to be a hit in a fully associative cache. Hill and Smith [19] extended the stack-based algorithm to simulate direct-mapped and set-associative caches. Thompson and Smith [32] introduced dirty-level analysis and included write-back counts.

To improve the slow processing time required for the stack search (the upper bound on the stack size is the number of unique addresses in the access trace), Sugumar and Abraham [29] proposed a tree data structure-based algorithm to efficiently store and traverse memory references.

The tree-based algorithm provided a maximum 5X improvement in simulation speed. Janapsatya et al. [20] decreased simulation time using a forest data structure, which used linked lists to maintain address tags for set-associative cache analysis. However, the tradeoff for increased processing speed using these tree-based algorithms had a larger storage requirement to store the tree. Furthermore, these complex data structures and corresponding complicated processing operations made tree-based algorithms not amenable to hardware implementation for runtime cache tuning. Due to these drawbacks, the stack algorithm is still widely employed in trace-driven cache simulation methods. Viana et al. [33] proposed SPCE—a stack-based algorithm that evaluated cache size, block size, and associativity simultaneously using simple bit-wise operations. SPCE's attained speedup was as high as 14X compared to previous works. Gordon-Ross et al. [15] designed SPCE's hardware prototype for runtime cache tuning. Whereas these single-pass cache simulation methodologies (stack- and tree-based) are highly efficient, these methods are limited to single-level cache simulation.

Another technique to speed up access trace processing is parallel-distributed simulation, a straightforward technique that simulates different cache configurations using a parallel processor system. Heidelberger and Stone [18] proposed a method that partitioned the memory trace into small parts and simulated each part in parallel. Sugumar [30] proposed a parallel stack search method. Wan et al. [34] developed a GPU-based simulator supporting multilevel cache simulation. Although parallel-distributed simulation is capable of simulating configurable caches in a multilevel hierarchy with any design space size, several factors make parallel simulation difficult in practice, such as large hardware resources, a complex task management scheme, and data transmission between parallel processors.

In this paper, we enhance previous works to include two-level cache simulation. We combine a stack-based algorithm [33] to record the memory access trace with a tree-based data structure to support the stack search for processing time acceleration. We propose for the first time, to the best of our knowledge (besides trivial parallel simulation techniques), a single-pass trace-driven cache simulation methodology for two-level caches.

3 Two-Level Cache Characteristics

Two of the major challenges in two-level single-pass cache simulation are the storage and simulation time required to process each filtered access trace. In this section, we motivate our selection of an exclusive cache hierarchy, as opposed to an inclusive cache hierarchy, to address these challenges.

In an inclusive hierarchy with the least recently used (LRU) replacement policy for both L1 and L2, each cache level contains a subset of the contents of the lower level caches (closer to the processor). On an L1 miss and an L2 hit, the cache block is *copied* from L2 to L1. If both L1 and L2 miss, the cache block is copied to both L1 and L2. The evicted LRU blocks are discarded if the blocks are not dirty (assuming the data cache uses write-allocate and write-back policies). Otherwise, the evicted blocks are written back to main memory. In an exclusive hierarchy [35] with LRU for L1 and first-in-first-out (FIFO)-like for L2 (the exclusive hierarchy complicates L2 evictions, making the process similar to FIFO), each cache level's contents are disjoint from the contents of all other cache levels. On an L1 miss and an L2 hit, the cache block is *moved* from L2 to L1, the evicted LRU L1 block is moved to L2, and the evicted oldest block from L2 is discarded if the block is not dirty. When L1 and L2 both miss, the missed block is only fetched into L1 from main memory. This lack of replication across L1 and L2 provides an opportunity to logically view L1 and L2 as one *combined* cache, whose analysis can be processed based solely on the complete access trace. Deriving the L2 miss rate using this combined analysis eliminates the need to store and process each filtered L2 access trace and alters the basic stack-based algorithm processing.

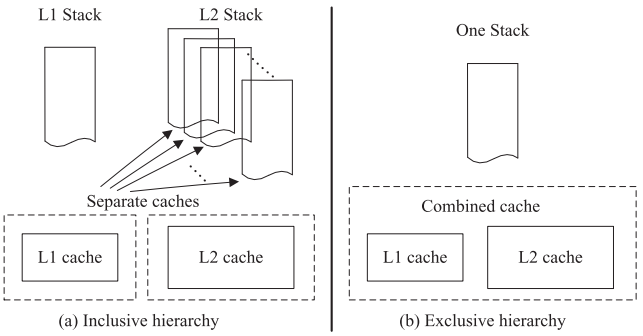

Fig. 1. Storage requirements for the stack-based algorithm for a two-level cache with (a) an inclusive hierarchy where L1 and L2 are processed separately and (b) an exclusive hierarchy where L1 and L2 are treated as one combined cache denoted using the dotted box.

To exemplify the reduced storage requirements afforded by the exclusive hierarchy, Fig. 1 depicts the stack-based algorithm's cache layout view (dotted boxes) and storage requirements for a two-level cache with inclusive (Fig. 1a) and exclusive (Fig. 1b) hierarchies (Section 4 presents stack processing details). More specifically, for the inclusive hierarchy, each cache is processed separately. The complete access trace is recorded in the L1 stack and for each L1 configuration, the filtered access trace is recorded in an L2 stack (one distinct L2 stack is required for each L1 configuration). Each L2 stack is processed separately using the same process as for single-level cache simulation [33]. In the exclusive hierarchy, only one stack is required since L1 and L2 are treated as one combined cache (denoted using the dotted boxes) and are evaluated simultaneously.

This difference in stack processing has a large impact on the storage and time complexity. The inclusive cache hierarchy requires one L1 stack and M L2 stacks, where M is the number of L1 configurations. The L1 stack has a storage complexity of $O(n)$, where n is the number of unique addresses in the access trace. Each of the M L2 stacks has the same storage complexity of $O(n)$ by assuming the worst case where all L1 accesses are misses. The exclusive cache hierarchy requires only one L1 stack to generate both L1 and L2 results. Therefore, the storage and time complexities for two-level inclusive and exclusive caches are $O((M+1)n)$ and $O(n)$, respectively.

The lightweight storage and time complexities of the exclusive cache are important for T-SPaCS since future work will adapt T-SPaCS for dynamic, runtime cache evaluation. The single-pass simulation feature makes T-SPaCS amenable to hardware implementation in dynamic cache tuning without runtime system intrusion [15].

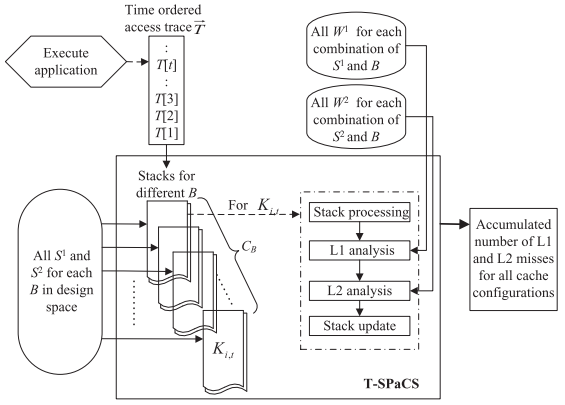

Fig. 2. T-SPaCS's functional overview: Executing the application generates the time-ordered access trace, which is sequentially fed into a different set of stacks based on the trace address's set index for S^1_{min} under different B. T-SPaCS's processing for each trace address consists of four steps (encompassed by the dotted box). After processing the entire access trace, T-SPaCS produces the accumulated number of L1 and L2 misses for all cache configurations. (Refer to Table 1 for notations.)

The tradeoff for the exclusive cache's reduced storage and time complexities is a design space reduction since an exclusive hierarchy requires L1 and L2 block sizes to be equal. However, previous work [16] showed that for a large design space, several cache configurations offer nearly equal energy and performance, thus this restriction will have a nominal effect on the cache tuning.

4 Two-Level Single-Pass Cache Simulation Methodology—T-SPaCS

T-SPaCS is suitable for a highly configurable two-level exclusive cache hierarchy by simultaneously evaluating cache configurations with varying size, block size, and associativity. T-SPaCS's output is the miss rates for all cache configurations. When combining the miss rates with a performance and energy model [17], a system designer can determine an appropriate cache configuration based on the application requirements.

T-SPaCS evaluates (determines a cache hit or miss) a trace address for a particular cache configuration by locating the previously accessed block addresses that map to the same cache set as the evaluated trace address. We refer to these block addresses as *conflicts*. If the number of conflicts is large enough to evict the previously fetched block that the evaluated trace address maps to, the evaluated trace address access results in a cache miss. Thus, T-SPaCS's goal is to simultaneously determine the conflicts with each trace address in L1 and L2 for all cache configurations in the design space.

Fig. 2 illustrates T-SPaCS's functional overview. The application is executed once to produce the time-ordered sequence of accessed addresses, which is denoted by the vector $\vec{T}$, and $T[t], t \in Z^+$ (t is a positive integer) is one element in vector $\vec{T}$ and represents the tth accessed address. The corresponding block address $A[t]$ is calculated by $T[t]/B = T[t] \gg b$, where "$\gg$" is a bitwise right shift operator and $B = 2^b$ is the cache block size. During T-SPaCS's simulation, the time-ordered sequence of unique block addresses that map to the same set with index i for the minimum number of sets S^1_{min} (without loss of generality, we assume $S^1_{min} < S^2_{min}$) is recorded into one stack structure for every cache block size B. Thus, the number of required stack structures for a particular B is equal to S^1_{min} (determined by B, the minimum cache size Z^1_{min}, and the maximum associativity W^1_{max}). In the set of a particular B's stacks, we denote each stack's contents as the vector $\overrightarrow{K_{i,t}}$, after inserting $A[t]$ into the stack. $K_{i,t}[m]$, $m \in Z^+$, is one element in vector $\overrightarrow{K_{i,t}}$, representing the mth uniquely accessed block address (counting starts from the stack's top, thus $K_{i,t}[1]$ is the stack top and represents the address of the most recently accessed cache block that maps to the set with index i for B and S^1_{min}). During T-SPaCS's processing for a particular $T[t]$, the cache configurations with different B are simulated sequentially using the corresponding set of stacks for each evaluated B. Since T-SPaCS's processing of each $T[t]$ for each B is the same, we limit our discussion in the remainder of this paper to T-SPaCS's processing of one arbitrary trace address $T[t]$ for one arbitrary cache block size B ($\forall B \in [B_{min}, B_{max}]$, where B_{min} and B_{max} represent the minimum and maximum block size values, respectively).

T-SPaCS processes each trace address for the set of stack structures for each B using four steps: stack processing, L1 analysis, L2 analysis, and stack update, as shown in Fig. 2. For a trace address $T[t]$ (whose block address is $A[t]$), the set index i is determined through B and S^1_{min} ($i = A[t] mod\ S^1_{min}$) and i locates the stack structure that stores $T[t]$'s conflicts for all possible number of sets $S(\forall S \in [S^1_{min}, S^1_{max}] \cup [S^2_{min}, S^2_{max}]$, where S^1_{min}, S^1_{max}, S^2_{min}, and S^2_{max} represent the minimum and maximum number of cache sets in L1 and L2, respectively). *Stack processing* scans the stack structure $\overrightarrow{K_{i,t-1}}$ to determine whether the block $A[t]$ was recorded (a cache line with that block address has already been fetched). If there exists h satisfying $K_{i,t-1}[h] = A[t]$, the block that $T[t]$ maps to was accessed previously and the most recent access was recorded in the stack as $K_{i,t-1}[h]$. For all S, stack processing scans the stack from $K_{i,t-1}[1]$ to $K_{i,t-1}[h]$ to evaluate the conflicts with $T[t]$. We refer to this process as *conflict evaluation*. The conflicts with $T[t]$ for a particular S are denoted by $\xi(S)$, whose collection is represented by $\{\xi(S)\}$. We note that conflict evaluation is trivial for S^1_{min}, since all the stack addresses in $\overrightarrow{K_{i,t-1}}$ conflict with $T[t]$ for S^1_{min} and thus the conflict evaluation for S^1_{min} simply records the stack addresses in $\overrightarrow{K_{i,t-1}}$ into $\{\xi(S^1_{min})\}$. After identifying these conflicts, *L1 analysis* directly determines $T[t]$ to be an L1 hit/miss based on the number of conflicts for the L1 configurations with S^1. If there is an L1 miss, *L2 analysis* is required. When the particular S^1 and any S^2 ($\forall S^2 \in [S^2_{min}, S^2_{max}]$) are combined to form one two-level cache configuration, L2 analysis categorizes the evaluated conflicts of the combined cache as either L1 or L2 conflicts. Similarly, the number of conflicts in L2 dictates an L2 hit/miss. After stack processing, L1 analysis, and L2 analysis (if necessary) for all cache configurations, the *stack update* removes $K_{i,t-1}[h]$ from $\overrightarrow{K_{i,t-1}}$ if h exists, and then pushes $A[t]$ on the top of $\overrightarrow{K_{i,t-1}}$ to update to $\overrightarrow{K_{i,t}}$. If there is no $K_{i,t-1}[h]$ in $\overrightarrow{K_{i,t-1}}$, the stack update directly pushes $A[t]$ on the top of $\overrightarrow{K_{i,t-1}}$. After the entire $\vec{T}$ is processed, T-SPaCS accumulates the number of L1 and L2 misses for all two-level cache configurations.

To simulate a data cache that uses the write-back policy, the cache block's dirty status must be considered. We track the number of write-backs using a write-avoid counter [32]. In a write-back cache, not all the cache writes result in write-backs to main memory. For example, if a cache block is written to X times, X-1 write-backs to main memory are *avoided* since all writes to the cache block are coalesced and are only written back to memory once when the cache block is evicted. Therefore, during T-SPaCS's data cache processing, if writing $T[t]$ results in an L1/L2 hit, and the stack searched address $K_{i,t-1}[h]$ is dirty, this write is avoided, and the write-avoid counter is incremented. The number of write-backs is equal to the total number of writes minus the number of write-avoids. We use a bit-array attached with each stack address, in which each bit indicates whether the address is dirty for each cache configuration. The bit-array's size is equal to the number of cache configurations with the same B in the design space. The dirty status of the stack address is maintained in the stack update step, which is detailed as follows: writing $T[t]$ sets the newly inserted $A[t]$ as dirty; if reading $T[t]$ results in an L2 miss, $A[t]$ is set as clean, since an L2 miss implies fetching $A[t]$ from main memory, which is always clean; if reading $T[t]$ results in an L1/L2 hit, the dirty status of $A[t]$ is dictated by the dirty status of the removed $K_{i,t-1}[h]$.

We note that although T-SPaCS uses several stack structures to record block-size-specific cache access patterns to simplify conflict determination, the storage, and time complexities are similar to those defined in Section 3. Since one cache block encapsulates multiple addresses, the combined storage space, and thus processing time for all stacks will be similar to the storage space and processing time required by one stack.

The remainder of this section presents T-SPaCS's detailed operations. Since there is no L2 analysis on an L1 hit, we describe the stack-based algorithm for single-level cache simulation in Section 4.1 and extend the methodology to L2 analysis in Section 4.2. Section 4.3 discusses acceleration strategies to assist stack processing.

4.1 Stack-Based Single-Level Cache Analysis

The single-level cache analysis algorithm serves as the basis for two-level cache analysis. In a single-level cache, the presence of an accessed address $T[t]$ in the cache set that $T[t]$ maps to depends on the cache configuration and the number of conflicts in the stack before $K_{i,t-1}[h]$. A stack address $K_{i,t-1}[m]$, $m \in [1, h)$ is recorded as a conflict in $T[t]'s$ conflict collection $\{\xi(S)\}$ for the cache configuration with block size B and number of sets S if and only if:

$$(K_{i,t-1}[m]) mod\ S = (T[t] \gg b) mod\ S, m \in [1, h). \quad (1)$$

Fig. 3 illustrates the algorithm for L1 analysis for processing an arbitrary $T[t]$ in the access trace. For each B's corresponding stack $\overrightarrow{K_{i,t-1}}$ (state 1) with every S^1 (state 2), stack processing determines the conflicts $\{\xi(S^1)\}$ in the stack addresses before $K_{i,t-1}[h]$ (state 3). If $K_{i,t-1}[h]$ is not present in the stack, $T[t]$ is a compulsory miss (state 5) and L2 analysis for $T[t]$ is not necessary. In this case, the stack update occurs (state 6) and the processing proceeds to the next address $T[t+1]$. If $K_{i,t-1}[h]$ is present in the stack,

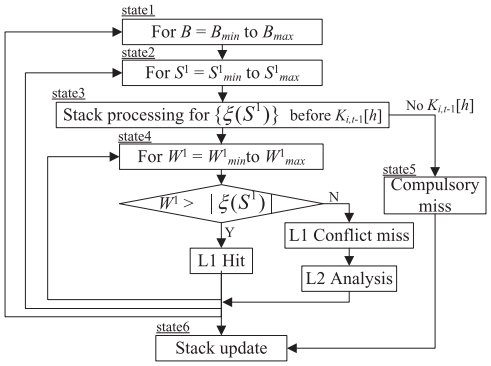

Fig. 3. Algorithm for L1 analysis for $T[t]$.

the number of conflicts $|\xi(S^1)|$ dictates the minimum set associativity that yields a hit, thus a hit or miss for each set associativity W^1 (state 4) can be determined (e.g., $W^1 > |\xi(S^1)|$ is a hit). L1 hits do not require any L2 analysis, and L1 misses require additional L2 analysis for all possible L2 configurations (Section 4.2). After $T[t]$'s evaluation for all S^1 and W^1 combinations (and additional L2 analysis if necessary), stack update (state 6) completes $T[t]$'s processing and the processing proceeds to the next address $T[t+1]$.

4.2 Stack-Based Two-Level Cache Analysis

When using an exclusive hierarchy, L1 and L2 can be treated as one combined cache. The conflict evaluation for the combined cache is processed using the L2 configurations in order to extract the L2 conflicts. We represent the conflicts for the combined cache as $\{\xi(S^2)\}$ (for cache configurations of S^2). Similar to the process for identifying $\{\xi(S^1)\}$, $\{\xi(S^2)\}$ can also be identified using conflict evaluation from $K_{i,t-1}[1]$ to $K_{i,t-1}[h]$ in stack processing. $\{\xi(S^2)\}$ consists of the conflicts in both L1 and L2. Since $\{\xi(S^2)\}$ contains inclusive L2 conflicts, exclusion requires the removal of the L1 conflicts from $\{\xi(S^2)\}$ to isolate the exclusive L2 conflicts, whose collection is denoted by $\{\delta\}$. The number of L2 conflicts $|\delta|$ determines the minimum L2 associativity necessary for $T[t]$ to be an L2 hit. Due to the LRU replacement policy for L1 and FIFO for L2, stack processing orders the conflicts in the conflict collections $\{\xi(S^1)\}$ and $\{\xi(S^2)\}$ in MRU (most recently used) time order to facilitate L1 and L2 analysis.

As shown in Fig. 3, when $T[t]$ results in an L1 conflict miss for a particular L1 configuration with B, S^1, and W^1, the first W^1 conflicts $\{\xi(S^1)\}_{W^1}$ in $\{\xi(S^1)\}$ are the blocks present in L1. The subsequent L2 analysis will evaluate all possible L2 configurations (with the same B as L1). For each S^2 ($\forall S^2 \in [S^2_{min}, S^2_{max}]$), the L2 analysis consists of three steps: 1) stack processing to determine $\{\xi(S^2)\}$ (as described in Section 4.1); 2) removing $\{\xi(S^1)\}_{W^1}$ (effectively removing the L1 blocks) from $\{\xi(S^2)\}$ to deduce $\{\delta\}$, which we refer to as the *compare-exclude* operation; and 3) evaluate $T[t]$ as an L2 hit/miss for each set associativity W^2 based on $|\delta|$.

The compare-exclude operation is divided into three scenarios based on three different inclusion relationships between $\{\xi(S^1)\}$ and $\{\xi(S^2)\}$. Fig. 4 depicts the cache addressing formats of L1 and L2, in which k and l represent the number of L1 and L2 index bits, respectively. The three

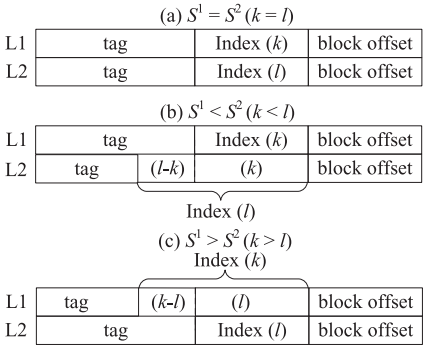

Fig. 4. Cache addressing formats for the compare-exclude operation scenarios: (a) $S^1 = S^2$, (b) $S^1 < S^2$, and (c) $S^1 > S^2$, in which k and l represent the number of L1 and L2 index bits, respectively.

compare-exclude scenarios are: 1) the number of L1 and L2 sets are equal ($S^1 = S^2$ and $\{\xi(S^1)\}_{W^1}$ is the first W^1 conflicts in $\{\xi(S^2)\}$); 2) the number of L1 sets is less than the number of L2 sets ($S^1 < S^2$ and $\{\xi(S^1)\}_{W^1}$ contains several conflicts in $\{\xi(S^2)\}$); 3) the number of L2 sets is less than the number of L1 sets ($S^1 > S^2$ and $\{\xi(S^1)\}_{W^1}$ is a subset of $\{\xi(S^2)\}$).

4.2.1 Compare-Exclude Scenario: $S^1 = S^2$

For L1 and L2 configurations with the same B and S values (Fig. 4a), $\{\xi(S^2)\}$ is the same as $\{\xi(S^1)\}$. Thus, stack processing for $\{\xi(S^2)\}$ is not necessary and $\{\xi(S^1)\}$ can be directly applied to deduce $\{\delta\}$. In this scenario, the conflicts in $\{\xi(S^1)\}$ are divided into two categories: the first W^1 conflicts are present in L1 and the remaining conflicts form $\{\delta\}$. The condition that $W^2 > |\delta|$ indicates that $K_{i,t-1}[h]$ has not been evicted from L2 by the following accessed blocks and $T[t]$ results in an L2 hit. For example, if $|\xi(S^1)| = 5$ and $W^1 = 2$, the first two conflicts in $\{\xi(S^1)\}$ are present in L1 and the remaining conflicts compose $\{\delta\}$. Therefore, $|\delta| = 3$ and the L2 configurations with associativities greater than 3 result in an L2 hit.

4.2.2 Compare-Exclude Scenario: $S^1 < S^2$

As depicted in Fig. 4b, the number of L1 index bits k is less than the number of L2 index bits l in the $S^1 < S^2$ scenario. The evicted cache blocks from one L1 set map to multiple (multiple of two) L2 sets. We refer to these multiple L2 sets as the *affinity group* associated with one L1 set, and the number of L2 sets in the affinity group is equal to S^2/S^1. The cache indexes for addressing one L1 set and the multiple L2 sets in the affinity group retain the following relationship: the least significant k index bits in L1 and L2 are equal and the L2 index's most significant $(l - k)$ bits are all values from all "0"s incremented to all "1."s

Since the stack processing for $\{\xi(S^2)\}$ begins from the stack's top, $\{\xi(S^2)\}$ determined by the L2 index of $T[t]$ contains some conflicts that are still present in L1. The collection of these conflicts is the subset of $\{\xi(S^1)\}_{W^1}$ with the same L2 index as $T[t]$, and thereby can be determined by the intersection of $\{\xi(S^1)\}_{W^1}$ and $\{\xi(S^2)\}$. After removing these intersecting conflicts, the remaining conflicts in $\{\xi(S^2)\}$ are the L2 conflicts $\{\delta\}$:

$$\{\delta\} = \{\xi(S^2)\} - \{\xi(S^1)\}_{W^1} \cap \{\xi(S^2)\}. \quad (2)$$

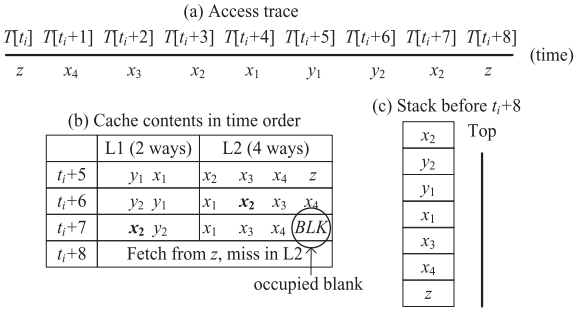

Fig. 5. Special case when $S^1 < S^2$ where fetching x_2 from L2 results in an occupied blank (BLK).

However, Fig. 5 illustrates a special case that must be considered in this scenario. Fig. 5a shows a time-ordered access trace segment from $T[t_i]$ to $T[t_i + 8]$, $t_i \in Z^+$. We assume the following: $T[t_i] = T[t_i + 8]$; $T[t_i + 3] = T[t_i + 7]$; and the other addresses map to different unique cache blocks. In these blocks, $T[t_i]$ maps to the same cache set as $T[t_i + 1]$, $T[t_i + 2]$, $T[t_i + 3]$, and $T[t_i + 4]$ under both S^1 and S^2, while $T[t_i]$ maps to the same cache set as $T[t_i + 5]$ and $T[t_i + 6]$ under S^1 but not S^2. For simplification, we represent $T[t_i]$ to $T[t_i + 8]$ with block addresses: z, x_4, x_3, x_2, x_1, y_1, y_2, x_2, and z, respectively, as shown in Fig. 5a. For $W^1 = 2$ and $W^2 = 4$, Fig. 5b shows the L1 and L2 set contents at the time points $t_i + 5$, $t_i + 6$, and $t_i + 7$. Fig. 5c shows the stack contents before $t_i + 8$. According to the cache set contents, accessing $T[t_i + 8]$ with z results in L1 and L2 misses. Stack processing for $T[t_i + 8]$ produces the conflicts $\{\xi(S^1)\}_2 = \{x_2, y_2\}$ and $\{\xi(S^2)\} = \{x_2, x_1, x_3, x_4\}$. The compare-exclude operation produces the conflicts $\{\delta\} = \{x_1, x_3, x_4\}$. Since $|\delta| = 3$ and $W^2 = 4$, $T[t_i + 8]$ is incorrectly classified as a hit.

To explain this incorrect classification, we note that accessing $T[t_i + 7]$ moves x_2 from L2 to L1, leaving an empty way in L2—an *occupied blank* (*BLK*), as shown in Fig. 5b. The occupied blank occurs because at $t_i + 7$, y_1 was evicted from L1 to accommodate x_2, but y_1 maps to a different L2 set than the set that x_2 maps to. The occupied blank means that x_2 was in L2 and involved in evicting z from L2 (at $t_i + 6$), thus x_2 should be counted as a conflict for $T[t_i + 8]$ in $\{\delta\}$.

In order to account for the occupied blank, *occupied blank labeling* is a supplemental process that labels occupied blanks using a bit-array associated with each stack address. The bit-array size is equal to the number of cache configurations with the same B in the design space. A "set" bit indicates that an occupied blank follows the labeled block in the corresponding cache configuration. The compare-exclude operation is augmented to include blank label examination. If the label associated with the last conflict in $\{\delta\}$ is set, there is an occupied blank behind the last conflict, which means the last conflict is the LRU block present in the L2 set and the block that $T[t]$ maps to has already been evicted. As a result, $T[t]$ is classified as an L2 miss even though $|\delta| < W^2$ in this case.

The occupied blank is introduced when there is an L2 hit, such as the example described in Fig. 5b (i.e., the L2 hit for $T[t_i + 7]$ results in a *BLK* by fetching x_2 into L1). Therefore, occupied blank labeling must proceed whenever the

processed address $T[t]$ results in an L2 hit. We refer to the previously fetched block that $T[t]$ maps to as $K_{i,t-1}[h]$ following our previous notation. There are two cases for occupied blank labeling: 1) when $K_{i,t-1}[h]$ is the W^2-*th* (MRU order) block in the L2 set (i.e., $|\delta| = W^2 - 1$), the last conflict in $\{\delta\}$ is labeled as an occupied blank since $K_{i,t-1}[h]$ will be fetched into L1 after accessing $T[t]$; and 2) when $K_{i,t-1}[h]$ is not the W^2-*th* block in the L2 set (i.e., $|\delta| < W^2 - 1$), stack processing must continue after $K_{i,t-1}[h]$ to locate and label the W^2-*th* block in the L2 set. The example shown in Fig. 5b follows the second case. Since x_2 is not the W^2-*th* block (MRU order) in L2 at $t_i + 6$, stack processing continues and labels the W^2-*th* block x_4 to indicate a *BLK* after x_4 in processing $T[t_i + 7]$.

4.2.3 Compare-Exclude Scenario: $S^1 > S^2$

In the $S^1 > S^2$ scenario (Fig. 4c), blocks evicted from multiple L1 sets map to one L2 set. Similarly to the $S^1 < S^2$ scenario, one L2 set corresponds to an affinity group in L1 and the number of L1 sets in the affinity group is equal to S^1/S^2. The L2 set that $T[t]$ maps to has an affinity group consisting of multiple L1 sets, and $T[t]$ maps to one of the L1 sets. We denote these multiple L1 sets in the affinity group, excluding the set that $T[t]$ maps to, as the *complementary* sets of $T[t]$. The conflicts associated with one of the complementary sets are denoted by $\overline{\xi_\alpha}$, where the subscript $\alpha \in [1, S^1/S^2 - 1]$ differentiates between each of the complementary sets. The collection of conflicts for the α complementary set is denoted by $\{\overline{\xi_\alpha}\}$, whose cardinality is denoted by $|\overline{\xi_\alpha}|$. In Fig. 4c, $T[t]$'s address uses k bits for the L1 index and l bits for the L2 index ($k > l$). The complementary set's indexes in L1 can be composed by joining the least significant l bits with each combination of "0"s and "1"s for the most significant $(k - l)$ bits excluding the combination associated with $T[t]$'s L1 index. For example, if $T[t]$'s indexes are "101101" for L1 and "1101" for L2, the collection of $\{\overline{\xi_\alpha}\}$ for all α will include all conflicts associated with {"001101," "011101," "111101"}.

The conflicts in $\{\xi(S^2)\}$, while still present in L1, include both the L1 conflicts in the set that $T[t]$ maps to ($\{\xi(S^1)\}_{W^1}$) and the L1 conflicts associated with the complimentary sets ($\{\overline{\xi_\alpha}\}$). Stack processing determines these additional conflicts by simply considering the complementary set's indexes. Therefore, the compare-exclude operation in this scenario produces

$$\{\delta\} = \{\xi(S^2)\} - \{\xi(S^1)\}_{W^1} - \sum_{\alpha=1}^{S^1/S^2-1} \{\overline{\xi_\alpha}\}_{W^1}, \qquad (3)$$

where $\{\overline{\xi_\alpha}\}_{W^1}$ represents the first W^1 conflicts (MRU order) in $\{\overline{\xi_\alpha}\}$.

4.3 Acceleration Strategies

During T-SPaCS's processing, stack processing is the most time consuming operation. For every $T[t]$ in the access trace, stack processing repeatedly evaluates all stack addresses $K_{i,t-1}[m]$ $(\forall m \in [1, h))$ for conflicts with every number of cache sets in the design space. If C_s denotes the total number of configurable S^1 and S^2 in the design space, the conflict evaluation's complexity is $O(C_s \cdot h)$ (without considering the complementary sets) for each $T[t]$ unless $T[t]$ results in L1 hits for all L1 configurations (in which case, the complexity is $O(C_{s^1} \cdot h)$, where C_{s^1} is the total number of configurable S^1 in the L1 design space only). To reduce the conflict evaluation runtime, stack processing can be accelerated using the *set refinement property* [19]. The set refinement property states that the blocks that map to the same set in larger caches also map to the same set in smaller caches. The set refinement property can be leveraged by processing S from smallest to largest and a stack address $K_{i,t-1}[m]$ is evaluated for conflicts with $T[t]$ only if $T[t]$ conflicts with $K_{i,t-1}[m]$ for a smaller S. Alternatively, processing S from largest to smallest leverages the set refinement property. However, since most stack addresses are not conflicts even for small S in a typical application, starting from S_{min} reduces more trivial conflict evaluations.

This acceleration strategy can be applied to both stack processing steps: 1) determining all conflicts $\{\xi(S^1)\}$, $\{\xi(S^2)\}$, and $\{\overline{\xi_\alpha}\}$ for all possible S^1 and S^2 using a tree data structure (tree-assisted acceleration in Section 4.3.1); and 2) occupied blank labeling using an array data structure (array-assisted acceleration in Section 4.3.2).

4.3.1 Tree-Assisted Acceleration

When processing $T[t]$ for an arbitrary B with stack $\overrightarrow{K_{i,t-1}}$ on an L1 miss, the compare-exclude operation compares the conflicts in $\{\xi(S^1)\}$ for each L1 configuration with the conflicts in $\{\xi(S^2)\}$ for all possible L2 configurations. An efficient method to determine these conflicts is to determine the conflicts for all possible S initially and store these conflicts in a tree structure for later reference. We note that this data structure is not a traditional tree structure, but is a hierarchical representation that we refer to as a tree for simplicity.

The tree structure stores $T[t]$'s conflicts and the conflicts associated with the complimentary sets for all S with the same B. Each tree level corresponds to a different S, with S increasing from root to leaf (higher level to lower level) by powers of two. Tree nodes store the conflict information and the maximum L1 and L2 associativities dictate the maximum number of conflicts stored at each node (*conflict storage*). Every conflict is represented by the conflict's block address and a pointer to the block's stack location, which assists in occupied blank labeling since the blank labels (linked using the stack address) of the recorded conflicts will be examined to correct the compare-exclude results in the $S^1 < S^2$ scenario.

We accelerate stack processing by determining all conflicts for all S simultaneously. We denote all S from the minimum S_{min} to the maximum S_{max} with a subscript β, where β is an integer satisfying $\beta \in [1, \log_2(S_{max}/S_{min}) + 1]$ such that the βth tree level corresponds to S_β and the number of tree levels is $\log_2(S_{max}/S_{min}) + 1$. Due to the set refinement property, evaluating the conflicts for $K_{i,t-1}[m](\forall m \in [1, h))$ with $T[t]$ for a particular $S_{\beta_i}(\beta_i \in [2, \log_2(S_{max}/S_{min}) + 1])$ depends on whether $K_{i,t-1}[m]$ conflicts with $T[t]$ for $S_{\beta_i - 1}$. When $K_{i,t-1}[m]$ is $T[t]$'s conflict for $S_{\beta_i - 1}$, $K_{i,t-1}[m]$ will be $T[t]$'s conflict for S_{β_i} on the condition that the most significant bit in the indexes of both $T[t]$ and $K_{i,t-1}[m]$ under S_{β_i} are the same. On the contrary, if $K_{i,t-1}[m]$ is not $T[t]$'s conflict for $S_{\beta_i - 1}$, the indexes of $T[t]$ and $K_{i,t-1}[m]$ under all larger S_β $(\forall \beta \in [\beta_i, \log_2(S_{max}/S_{min}) + 1])$ are also different.

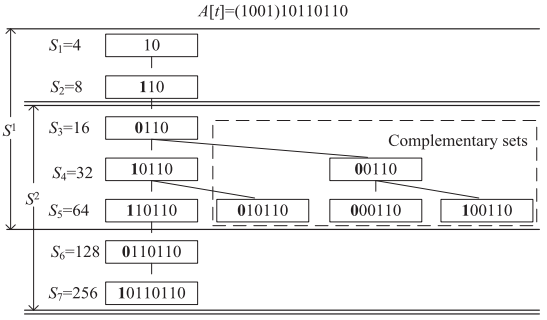

Fig. 6. A sample tree structure where rectangles correspond to tree nodes and values in the nodes are the indexes of the recorded conflicts.

When the combined L1 and L2 configurations satisfy the $S^1 > S^2$ scenario, the compare-exclude operation should exclude the conflicts associated with the L1 complementary sets as well as the L1 conflicts. Therefore, for each S_β in the L1 design space that is larger than S^2_{min}, the additional conflicts with each of the complimentary sets $\{\overline{\xi_\alpha}\}$ must also be searched and recorded in the βth tree level as one node. The number of nodes at each tree level is dictated by the number of complementary sets required for S_β. More specifically, if $\exists S_{\beta_i} \in [S^1_{min}, S^1_{max}]$ and $S_{\beta_i} > S^2_{min}$, the number of nodes in the β_ith tree level is $S_{\beta i}/S^2_{min}$. When S_{β_i} is combined with an S^2 other than S^2_{min} while still complying with the condition that $S_{\beta_i} > S^2$, the conflicts with the complementary sets can be determined by selecting the corresponding nodes in the β_ith tree level based on the difference between the number of index bits for S_{β_i} and S^2.

Fig. 6 provides a sample tree structure for a processed address $T[t]$ with block address $A[t] =$ "100110110110". The configurable number of sets in the design space for the certain B are bounded by $S^1_{min} = 4$, $S^1_{max} = 64$, $S^2_{min} = 16$, and $S^2_{max} = 256$. Rectangles correspond to tree nodes and values in the nodes correspond to the indexes of the recorded conflicts. For levels S_1, S_2, S_3, S_6, and S_7, only one node is required in each level to record the conflicts with $T[t]$. For levels S_4 and S_5, the complementary set's conflicts must be recorded since S_4 and S_5 are larger than S^2_{min} (i.e., S_3). When $S^1 = S_5$ and $S^2 = S_3$, the conflicts selected from all the four nodes in the fifth level will be excluded during the compare-exclude operation. When $S^1 = S_5$ and $S^2 = S_4$, only the conflicts in the first two nodes will be excluded.

Fig. 7 summarizes the tree-assisted stack processing acceleration algorithm. For each $T[t]$ and B, the tree contents are cleared and S_{start} is initialized to S_1 (line 1). For each stack address $K_{i,t-1}[m]$ $(\forall m \in [1, h))$ (lines 2-19), conflict evaluation determines the conflicts with $T[t]$ or the complementary sets from S_{start} to S_{max} (lines 3-13). If the complementary set's conflicts are not required for the βth tree level, $K_{i,t-1}[m]$ is directly evaluated for conflict with $T[t]$ for S_β by comparing the most significant index bits of $T[t]$ and $K_{i,t-1}[m]$ under S_β (since S increases by powers of two) (lines 4-6). If $K_{i,t-1}[m]$ conflicts with $T[t]$ for S_β, $K_{i,t-1}[m]$ is recorded into the node in the βth tree level (lines 7-8); otherwise conflict evaluation for $K_{i,t-1}[m]$ is terminated (larger S_β conflict evaluations are not necessary) (lines 9-10). If the complimentary set's conflicts are required in the βth tree level, $K_{i,t-1}[m]$ is stored into a node in the βth tree level based on the most $\log_2(S_\beta/S^2_{min})$ significant index bits (lines 11-13). In this situation, only checking the most $\log_2(S_\beta/S^2_{min})$ significant bits requires the condition that $K_{i,t-1}[m]$ conflicts with $T[t]$ for S^2_{min} when processing proceeds to line 12. This condition is guaranteed by the constraint in changing S_{start} in line 18. Since the required number of conflicts stored in each node is limited by the L1 and L2 associativities, if all nodes in the S_{start} level are full after processing $K_{i,t-1}[m]$ for all S_β $(\forall S_\beta \in [S_{start}, S_{max}])$, searching for more conflicts in the S_{start} level is trivial. Therefore, the value for S_{start} may change (line 14). If the complementary set's conflicts are not needed for the level with $S_{start+1}$, S_{start} is updated by $S_{start+1}$ (lines 15-16); otherwise conflict evaluation always starts from S^2_{min} (lines 17-18) since the evaluation of the conflicts for S^2_{min} is a prior requirement for determining the complimentary set's conflicts.

```
   Stack processing for each T[t] and B:
1  -Clear tree contents; S_start = S_1;
2  for (m=1; m<h; m++)    //check from K_{i,t-1}[1] to K_{i,t-1}[h]
3     for (S_β=S_start; S_β<=S_max; S_β·2)
4        if (complementary sets's conflicts are not required in the β-th level)
5              -evaluate the conflict of K_{i,t-1}[m] for S_β when S_β != S^1_min; otherwise
6               directly record K_{i,t-1}[m] in the 1-th level;
7              if (conflict)
8                 -record into the node in the β-th level;
9              else
10                -go to END_m;
11       else
12          -check the most log2(S_β/S^2_min) significant bits in index and store into the
13           corresponding node in the β-th level;
14    if (all nodes in the start-th level are full)
15       if ( complementary sets's conflicts are not required in the (start+1)-th level)
16          -S_start = S_start+1;
17       else
18          -S_start=S^2_min;
19    END_m;
```

Fig. 7. Tree-assisted stack processing acceleration algorithm.

Since only one tree is built and the tree's contents are cleared for every $T[t]$ and B, the storage space required by the tree is minimal as compared to the stack structures.

4.3.2 Array-Assisted Acceleration

Stack processing for $T[t]$ is limited to only evaluating the stack addresses before $K_{i,t-1}[h]$ except during occupied blank labeling for the $S^1 < S^2$ scenario. In this scenario, additional stack processing after $K_{i,t-1}[h]$ is required if $K_{i,t-1}[h]$ is not the W^2th (MRU order) block in the L2 set. Since this additional stack processing may be required for all possible S^2, we propose array-assisted acceleration for this additional stack processing.

An array Γ, whose size (number of elements) is dictated by the maximum L2 associativity W^2_{max}, records additional conflicts in the stack after $K_{i,t-1}[h]$. Each element stores the information for one conflict using the same format as the tree structure nodes in Section 4.3.1.

Fig. 8 depicts the array-assisted stack processing acceleration algorithm. When $T[t]$ results in an L1 miss for a particular L1 configuration, Γ's contents are cleared (line 3) and all possible L2 configurations for all combinations of S^2 and W^2 are analyzed by the compare-exclude operation (lines 4-6). If there is a hit in an L2 configuration and $K_{i,t-1}[h]$ is not the W^2th block in the L2 set, additional stack processing is required to determine and label the W^2th block (line 7). If Γ is empty, this L2 configuration is the first configuration to

```
   For processed T[t] with certain B, S^1, W^1:
1  -L1 analysis;
2  if (L1 miss)
3     -clear Γ;
4     for (S^2=S^2_min; S^2<=S^2_max; S^2·2)
5        for (W^2=W^2_max;W^2>=1; W^2/2)
6           -L2 analysis;
7           if (W^2-th block in L2 is required)
8              if (Γ is NULL)
9                 -stack processing for additional conflicts with S^2 until the
10                 W^2-th block is determined;
11                -store all searched additional conflicts in Γ;
12             else
13                if (the W^2-th block is firstly required for the certain S^2)
14                   -evaluate conflicts in Γ for S^2;
15                   -continue stack processing to determine the W^2-th
16                     block if the conflicts searched in Γ are not enough;
17                   -replace Γ contents with all obtained additional
18                     conflicts for S^2;
19                else
20                   -determine the W^2-th block in Γ directly;
```

Fig. 8. Array-assisted stack processing acceleration algorithm.

require the W^2th block and stack processing determines the additional conflicts after $K_{i,t-1}[h]$ for S^2, and records these additional conflicts in Γ (lines 8-11). If Γ is not empty, one of two situations occurs. 1) If the W^2th block is first required for that particular S^2 (lines 13-18), the additional conflicts can be obtained by evaluating Γ's elements (line 14) since Γ already stores the conflicts for the previously processed smaller S^2 and contains the first several conflicts for larger S^2 (according to the set refinement property). If the conflicts determined in Γ are not enough to determine the W^2th block for the new S^2, additional stack processing continually evaluates the stack addresses until the W^2th block is determined (lines 15-16), and all additional conflicts for the new S^2 will replace Γ's contents (lines 17-18). 2) If the additional conflicts for that particular S^2 with previously processed larger W^2 were determined and stored in Γ, the W^2th block for smaller W^2 can be directly determined in Γ (lines 19-20).

Since Γ's contents are updated for each $T[t]$ with each specific cache configuration, only one array is required and the storage overhead for array-assisted acceleration is negligible as compared to the stack structures.

5 Experimental Results and Analysis

We verified T-SPaCS using the 15 benchmarks from EEMBC benchmark suite [11], five arbitrarily selected benchmarks from the Powerstone benchmark suite [23], and four arbitrarily selected benchmarks from the MediaBench benchmark suite [22]. We gathered the access trace for each benchmark by modifying "sim-cache" in SimpleScalar 3.0d [7] and these traces served as input to T-SPaCS. For comparison, we modified the widely used trace-driven cache simulator Dinero IV [10] to simulate both two-level exclusive instruction and data caches, respectively, for each benchmark.

We used the same design space for the two-level configurable cache hierarchy as in [17]. The design space consisted of 243 configurations by varying (in increments of powers of 2) the L1 size from 2 to 8 Kbytes, the L2 size from 16 to 64 Kbytes, the L1 and L2 associativities from direct-mapped to 4-way, and the cache block size from 16 to 64 bytes. We note that we selected this design space for comparison convenience and T-SPaCS itself does not impose any restriction on the configurable cache parameters, and is thus valid for any design space.

In order to determine T-SPaCS's accuracy and efficiency, we gathered the cache miss rates for all 243 configurations using the modified Dinero, which produces exact results, and T-SPaCS, then evaluated the margin of errors in T-SPaCS with respect to the exact miss rate and the optimal (lowest) energy cache.

5.1 Miss Rate Accuracy

We compared the miss rates determined by T-SPaCS with the exact miss rates determined by the modified Dinero for each benchmark. The results showed that for both instruction and data caches, T-SPaCS's L1 miss rates were 100 percent accurate for all configurations and the L2 miss rates were 100 percent accurate for 240 out of 243 configurations, which accounts for 99 percent of the design space. For each benchmark, we calculated the average and standard deviation of miss rate errors across the three inaccurate cache configurations. For the instruction cache, across all benchmarks, the maximum values for the average and standard deviation of miss rate errors were 1.16 and 0.64 percent, respectively. For the data cache, across all benchmarks, the maximum values for the average and standard deviation of miss rate errors were 0.69 and 0.32 percent, respectively. Since inaccurate miss rates result in inaccurate write-back rates in the data cache, the maximum values for the average and standard deviation of write-back rate errors across all benchmarks were only 0.15 and 0.07 percent, respectively.

For the three inaccurate configurations, multiple L1 sets in one affinity group corresponded to one L2 set (i.e., scenario $S^1 > S^2$ in Section 4.2.3). In this scenario, the eviction order of blocks from the different L1 sets to the same L2 set does not follow the memory access order and the blocks in the L2 set are disordered. When we determine the conflicts of $T[t]$ in L2, only the blocks that are evicted into L2 after $K_{i,t-1}[h]$ affect $K_{i,t-1}[h]$'s eviction from L2. Since the stack structure only records the latest memory access order, the historical eviction order of the blocks from multiple L1 sets to the same L2 set cannot be obtained from the stack. Therefore, the blocks in $\{\delta\}$ generated by the compare-exclude operation are not guaranteed to be the blocks present in the L2 set. However, inaccurate $\{\delta\}$ does not necessarily produce an incorrect cache hit/miss classification since a cache miss is determined when $|\delta| >= W^2$. Only when the inaccurate $|\delta|$'s error is larger than the difference between W^2 and accurate $|\delta|$, the cache hit/miss classification will be affected. Our experimental results showed that the effect of errors in $|\delta|$ on miss rate estimation was nominal.

5.2 Optimal Cache Configurations

Since low energy/power consumption is a critical optimization for both embedded systems and desktop computers, we evaluated T-SPaCS's ability to determine the optimal (lowest energy) cache configuration. We expanded a two-level inclusive cache hierarchy energy model [17] to include evicted block write energy, and determined the energy

consumption for each cache configuration using the following energy model:

total_energy = static_energy + dynamic_energy
dynamic_energy = L1_dynamic_energy
+ L2_dynamic_energy + offchip_access_energy
+ (miss_cycles . CPU_stall_energy)
L1_dynamic_energy = L1_accesses . L1_per_read_energy
L2_dynamic_energy = (L2_hits . L2_per_read_energy)
+ (L2_misses . L2_per_tag_read_energy)
+ (L1_evicts . L2_per_write_energy)
offchip_access_energy = L2_misses.
memory_per_read_energy
+ write-backs . memory_per_write_energy
miss_cycles = L1_miss_cycles + L2_miss_cycles
L1_miss_cycles = L1_misses . L1_miss_latency
+ (L1_misses . blocksize. L1_bandwidth)
L2_miss_cycles = L2_misses . L2_miss_latency
+ (L2_misses . blocksize. L2_bandwidth)
static_energy = total_cycles . static_energy_per_cycle
static_energy_per_cycle = energy_per_Kbyte
. cache_size_in_Kbytes
energy_per_Kbyte
= ((dynamic_energy_of_base_cache)*20%)
/ base_cache_size_in_Kbytes

We used T-SPaCS and modified Dinero for a two-level exclusive cache to determine *L1_accesses*, *L1_misses*, *L2_hits*, *L2_misses*, *L1_evicts*, and *write-backs*. We obtained dynamic cache and memory read/write energy using CACTI 6.5 [8] for 0.09-micron technology and the cache static energy consumption accounted for 20 percent of the total cache energy [17]. We assumed the *CPU_stall_energy* to be 20 percent [17] of a 0.09-micron ARM1156 microprocessor [3], and estimated bandwidth and latency based on a reasonable system architecture: an L2 fetch is 4 times longer than an L1 fetch; a main memory fetch is 20 times longer than an L2 fetch; and the memory throughput is 50 percent of the latency [17].

We applied this energy model to the miss rates determined by T-SPaCS and the exact miss rates determined by the modified Dinero. The calculated results showed that the optimal cache configurations determined by T-SPaCS were exactly the same as those determined by Dinero for all benchmarks. Table 2 shows the optimal instruction and data cache configurations for each benchmark. Despite incorrect miss rates for the three configurations where $S^1 > S^2$, the errors were too small to affect the determined optimal cache configurations. Considering that the number of configurations with $S^1 > S^2$ generally occupies a small percentage (3 out of 243 (1 percent) in our experiment) of the design space since caches' sizes are technically limited by $Z^1 < Z^2$, and the introduced errors do not affect the determined optimal cache configuration, there is no need to eliminate the small miss rate errors since doing so would significantly increase the simulation time.

Fig. 9 depicts the normalized energy savings for the optimal cache configurations compared to a base cache configuration for each benchmark. The base cache configuration represents a configuration that may be commonly found on a platform intended to run benchmarks similar to those we studied. The base cache configuration was an 8 Kbyte L1 cache with a 32-byte block size and 4-way set associativity and a 64 Kbyte L2 cache with a 32-byte block size and 4-way set associativity. Fig. 9 shows average and maximum energy savings of 22 and 46 percent, respectively, for instruction caches, and average and maximum energy savings of 26 and 48 percent, respectively, for data caches.

To further corroborate the significance of two-level cache tuning over single-level cache tuning in embedded systems intended for low power, we compared the energy savings using the two-level optimal cache configurations with the energy savings using single-level optimal cache configurations. The single-level configurable cache design space

TABLE 2
Optimal Instruction and Data Cache Configurations (L1 and L2 Configurations for Instruction Caches (I) and Data Caches (D) Are Listed as the Total Size in kbytes (kB) Followed by the Block Size in Bytes (B) Followed by the Associativity in Ways (W))

bcnt	I: L1_2kB_64B_1W_L2_16kB_64B_4W D: L1_2kB_64B_1W_L2_16kB_64B_4W
bilv	I: L1_4kB_64B_4W_L2_16kB_64B_2W D: L1_2kB_64B_1W_L2_16kB_64B_4W
blit	I: L1_2kB_64B_4W_L2_16kB_64B_2W D: L1_8kB_32B_4W_L2_16kB_32B_2W
brev	I: L1_4kB_64B_4W_L2_16kB_64B_4W D: L1_2kB_64B_1W_L2_16kB_64B_1W
fir	I: L1_4kB_64B_4W_L2_16kB_64B_2W D: L1_2kB_64B_1W_L2_16kB_64B_1W
A2TIME01	I: L1_8kB_64B_4W_L2_32kB_64B_4W D: L1_4kB_64B_4W_L2_16kB_64B_1W
AIFFTR01	I: L1_2kB_32B_2W_L2_32kB_32B_4W D: L1_8kB_64B_4W_L2_64kB_64B_2W
AIFIRF01	I: L1_4kB_32B_4W_L2_16kB_32B_4W D: L1_2kB_64B_4W_L2_16kB_64B_1W
AIIFFT01	I: L1_2kB_32B_2W_L2_32kB_32B_4W D: L1_8kB_64B_4W_L2_64kB_64B_4W
BaseFP01	I: L1_4kB_64B_4W_L2_64kB_64B_4W D: L1_4kB_64B_4W_L2_32kB_64B_1W
BITMNP01	I: L1_4kB_64B_4W_L2_16kB_64B_4W D: L1_4kB_64B_4W_L2_16kB_64B_1W
CACHEB01	I: L1_8kB_64B_4W_L2_32kB_64B_4W D: L1_8kB_32B_4W_L2_16kB_32B_4W
CANRDR01	I: L1_4kB_64B_4W_L2_32kB_64B_4W D: L1_8kB_64B_4W_L2_16kB_64B_4W
IDCTRN01	I: L1_8kB_64B_4W_L2_16kB_64B_4W D: L1_4kB_64B_4W_L2_16kB_64B_2W
IIRFLT01	I: L1_8kB_32B_1W_L2_16kB_32B_4W D: L1_2kB_64B_4W_L2_16kB_64B_1W
PNTRCH01	I: L1_2kB_32B_1W_L2_16kB_32B_4W D: L1_8kB_64B_4W_L2_16kB_64B_4W
PUWMOD01	I: L1_4kB_64B_4W_L2_32kB_64B_4W D: L1_2kB_64B_4W_L2_16kB_64B_1W
RSPEED01	I: L1_4kB_64B_4W_L2_32kB_64B_4W D: L1_4kB_64B_4W_L2_16kB_64B_1W
TBLOOK01	I: L1_4kB_64B_4W_L2_64kB_64B_4W D: L1_8kB_64B_2W_L2_16kB_64B_4W
TTSPRK01	I: L1_8kB_16B_4W_L2_16kB_16B_4W D: L1_8kB_64B_4W_L2_16kB_64B_4W
epic	I: L1_2kB_32B_2W_L2_32kB_32B_4W D: L1_8kB_64B_2W_L2_64kB_64B_1W
jpegencode	I: L1_8kB_64B_4W_L2_32kB_64B_4W D: L1_8kB_64B_4W_L2_64kB_64B_1W
mpeg2decode	I: L1_4kB_16B_4W_L2_32kB_16B_4W D: L1_8kB_64B_4W_L2_64kB_64B_4W
pegwitencode	I: L1_8kB_32B_1W_L2_16kB_32B_4W D: L1_8kB_16B_2W_L2_64kB_16B_4W

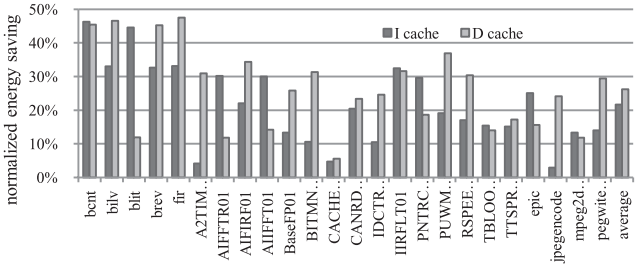

Fig. 9. Energy savings for the optimal instruction and data cache configurations normalized to the base cache configuration.

consisted of the same L1 configurations as in the two-level configurable cache. We determined the optimal single-level cache configurations using an exhaustive search. Fig. 10 depicts the energy consumption of the two-level optimal cache configurations normalized to the energy consumption of the single-level optimal cache configurations. The results indicate that for instruction caches, six of the 24 benchmarks consumed less energy using a single-level cache as compared to a two-level cache, while the remaining 18 benchmarks showed increased energy savings using two-level caches; for data caches, 16 of the 24 benchmarks presented increased energy savings using two-level caches. On average, over all benchmarks, the two-level optimal instruction caches consumed 28 percent less energy than the single-level optimal instruction caches, and the two-level optimal data caches consumed 22 percent less energy than the single-level optimal data caches.

5.3 Simulation Time Efficiency

To illustrate T-SPaCS's efficiency, we compared the simulation time required for T-SPaCS to simultaneously evaluate all 243 configurations with the simulation time required to sequentially simulate all 243 configurations with the modified Dinero. We tabulated the *user time* reported from the Linux *time* command for the simulations running on a Red Hat Linux Server v5.2 with a 2.66 GHz processor and 4 Gigabytes of RAM.

To verify the speedup improvement obtained using our acceleration strategies (Section 4.3), we simulated the benchmarks using two T-SPaCS versions: T-SPaCS without acceleration and T-SPaCS with acceleration.

The actual simulation times for evaluating all 243 instruction cache configurations for a single benchmark ranged from 9.2 to 419.5 minutes for Dinero, 41 to 1,409 seconds for T-SPaCS without acceleration, and 24 to 1,108 seconds for

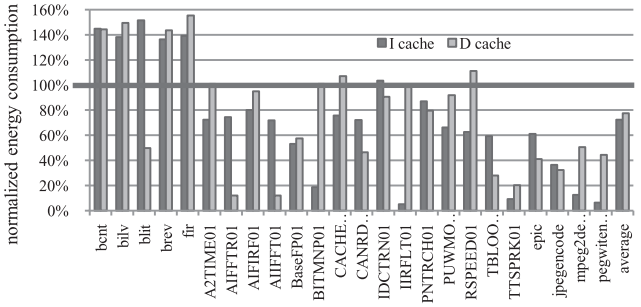

Fig. 10. Energy consumption of the two-level optimal cache configurations normalized to the energy consumption of the single-level optimal cache configurations.

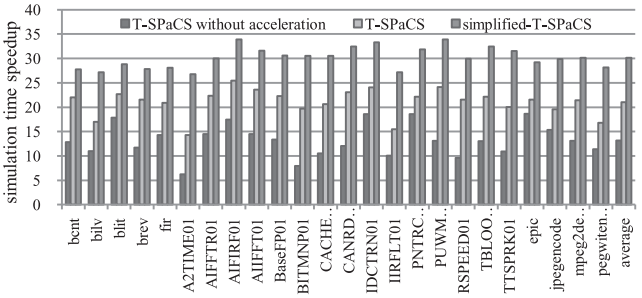

Fig. 11. Simulation time speedup of T-SPaCS without acceleration, T-SPaCS, and simplified-T-SPaCS compared to the modified Dinero for instruction caches.

T-SPaCS with acceleration. For all simulators, the benchmarks that required the most and least simulation times were *blit* and *BaseFP01*, respectively. Fig. 11 shows the simulation speedups as compared to the modified Dinero for instruction caches. T-SPaCS with acceleration (second bar) achieved maximum and average speedups of 25.42X and 21.02X, respectively, which improved the speedup of T-SPaCS without acceleration (first bar) by 6.79X and 7.84X, respectively.

The actual simulation times for evaluating all 243 data cache configurations for a single benchmark ranged from 14.6 to 804.3 minutes for Dinero, 22 to 1,774 seconds for T-SPaCS without acceleration, and 19 to 1,499 seconds for T-SPaCS with acceleration. For all simulators, the benchmark that required the most simulation time was *blit* and the benchmark that required the least simulation time was *jpegencode* for Dinero and *TBLOOK01* for both versions of T-SPaCS. Fig. 12 shows the simulation speedups for data caches. T-SPaCS with acceleration (second bar) acquired maximum and average speedups of 46.8X and 33.34X, respectively, which reduced the simulation time by 6.52X and 2.07X as compared to T-SPaCS without acceleration (first bar), respectively. We note that two benchmarks in Fig. 12 did not show acceleration improvement, which was due to the fact that our acceleration strategies reduced the simulation time using the set refinement property. The conflict determinations are omitted for the larger S if the stack address is not a conflict for a small S. Thus, in a rare case that a large amount of stack addresses are the conflicts for most S in the design space, there will be no significant speedup obtained by using our acceleration strategy. Due to the processing overhead introduced by acceleration, the total simulation time with acceleration can be longer than the simulation time without acceleration.

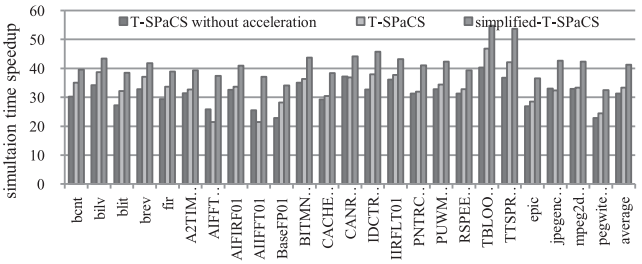

Fig. 12. Simulation time speedup of T-SPaCS without acceleration, T-SPaCS, and simplified-T-SPaCS compared to the modified Dinero for data caches.

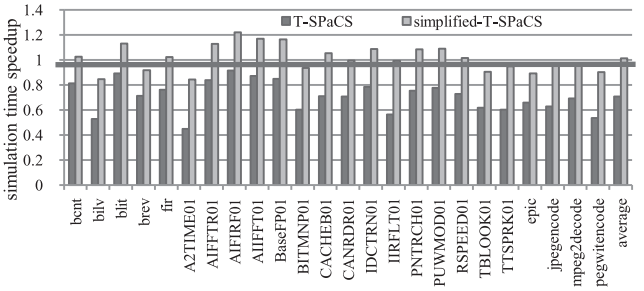

Fig. 13. Simulation time speedup of T-SPaCS and simplified-T-SPaCS compared to TCaT for instruction caches.

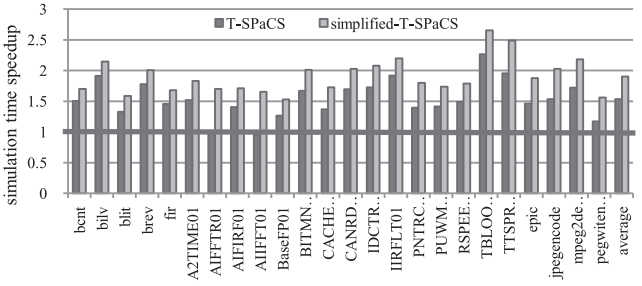

Fig. 14. Simulation time speedup of T-SPaCS and simplified-T-SPaCS as compared to TCaT for data caches.

To avoid the miss rate error introduced during the $S^1 < S^2$ scenario (Section 4.2.2), we supplemented the compare-exclude operation with occupied blank labeling for each L2 hit. Experiments revealed that occupied blank labeling accounted for a large portion of T-SPaCS's simulation time even when leveraging the array-assisted acceleration (Section 4.3.2). Therefore, we evaluated a simplified version of T-SPaCS (simplified-T-SPaCS) by removing occupied blank labeling. The measured simulation times for evaluating all 243 configurations for a single benchmark using simplified-T-SPaCS ranged from 18 (*BaseFP01*) to 873 (*blit*) seconds for the instruction caches and from 16 (*TBLOOK01*) to 1,254 (*blit*) seconds for the data caches. Fig. 11 shows the simulation time speedups obtained by simplified-T-SPaCS for each benchmark (third bar) as compared to the modified Dinero for instruction caches. Simplified-T-SPaCS's maximum and average speedups were increased to 33.92X and 30.15X, respectively. Fig. 12 depicts the simulation time speedups of simplified-T-SPaCS (third bar) for data caches, and the maximum and average speedups were 54.71X and 41.31X, respectively.

The tradeoff for the increased simulation speedups of simplified-T-SPaCS is additional L2 miss rate errors for the 228 configurations where $S^1 < S^2$. In order to quantify the degradation in the miss rate accuracy without occupied blank labeling, we counted the number of occurrences of occupied blanks and the number of inaccurate L2 hit/miss classifications without labeling the occupied blanks. Averaged across all benchmarks, occupied blanks accounted for 92 percent of the L2 hits (only L2 hits introduce occupied blanks) for instruction caches and 90 percent of the L2 hits for data caches. However, the average number of L2 hit/miss classifications corrected by occupied blank labeling was only 0.47 percent of the occurrences of occupied blanks for instruction caches and 0.52 percent for data caches. We further calculated the average and standard deviation of miss rate errors across the 228 inaccurate cache configurations for each benchmark. Across all benchmarks, the maximum values of the average and standard deviation of miss rate errors were 0.71 and 0.90 percent, respectively, for instruction caches. For data caches, the maximum values of the average and standard deviation of miss rate errors were 1.02 and 1.65 percent, respectively. We also examined the maximum values for the average and standard deviation of write-back rate errors across all benchmarks, which were 0.13 and 0.14 percent, respectively. Furthermore, we determined the optimal cache configurations using simplified-T-SPaCS. Results revealed that even with the inaccurate miss rates, simplified-T-SPaCS produced identical optimal configurations as those determined using the exact miss rates for both instruction and data caches.

Therefore, simplified-T-SPaCS is an ideal choice for cache tuning due to simplified-T-SPaCS's competitively fast simulation time and accurate optimal configuration determination. Alternatively, T-SPaCS is suitable in situations that require more accurate cache miss rate estimation (e.g., performance analysis) while still providing simulation speedup.

5.4 Comparison with TCaT

To further verify T-SPaCS's and simplified-T-SPaCS's efficiency, we compared to a state-of-the-art two-level cache tuner, TCaT [13]. TCaT is an efficient heuristic that determines the optimal energy cache configuration using an interlaced exploration methodology. Since TCaT sequentially simulates the design space using SimpleScalar's [7] "sim-cache" [13], we modified "sim-cache" to simulate a two-level exclusive cache. Even though TCaT sequentially simulates the cache configurations, TCaT only simulates 6.5 percent of the configurations on average and can thus determine the optimal energy cache configuration quickly. Fig. 13 shows the simulation time speedup of T-SPaCS (first bar) and simplified-T-SPaCS (second bar) as compared to TCaT for instruction caches. The results indicated that T-SPaCS required more simulation time than TCaT for all the 24 benchmarks. For simplified-T-SPaCS, results revealed that 12 benchmarks required less simulation time than TCaT. There was a maximum speedup of 1.22X, and the average simulation time of simplified-T-SPaCS was approximately equal to TCaT. Fig. 14 shows the simulation time speedup obtained by T-SPaCS (first bar) and simplified-T-SPaCS (second bar) as compared to TCaT for data caches. The results revealed that T-SPaCS simulated 22 benchmarks faster than TCaT, and simplified-T-SPaCS simulated all 24 benchmarks faster than TCaT. The average speedups of T-SPaCS and simplified-T-SPaCS were 1.54X and 1.90X, respectively.

Even though TCaT is generally faster than T-SPaCS, since TCaT is an inexact heuristic, TCaT trades off fast simulation time for reduced accuracy and TCaT is unable to determine the optimal energy cache configuration for all benchmarks. We refer to the cache configuration determined by TCaT as TCaT's configuration, which may be suboptimum. Fig. 15 compares the normalized (normalize to the base cache) energy savings between the optimal cache configurations (first bar) and TCaT's configurations (second

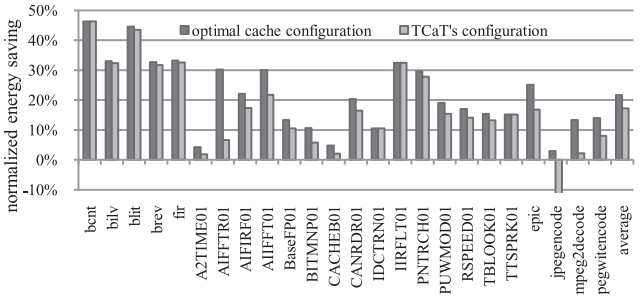

Fig. 15. The comparison of the normalized energy savings between the optimal cache configurations and TCaT's configurations for instruction caches.

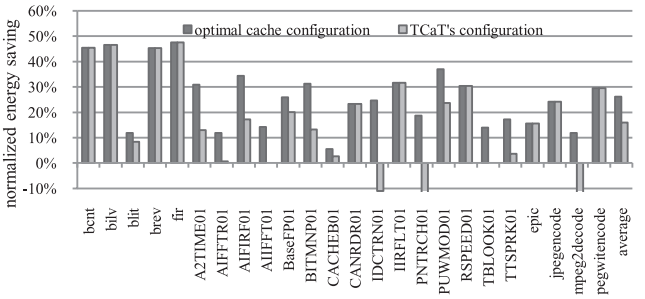

Fig. 16. The comparison of normalized energy savings between the optimal cache configurations and TCaT's configurations for data caches.

bar) for instruction caches. For four benchmarks, TCaT's configurations were the same as the optimal cache configurations. TCaT's configurations consumed 24 percent more energy than the optimal cache configurations in the worst case, and the average degradation in energy saving for TCaT's configurations across all the 24 benchmarks was 4 percent. Fig. 16 provides the normalized energy savings between the optimal cache configurations (first bar) and TCaT's configurations (second bar) for data caches. TCaT's configurations were the same as the optimal cache configurations for ten benchmarks. However, in the worst case, TCaT's configuration consumed 47 percent more energy than the optimal cache configuration, and the average degradation in energy saving for TCaT's configurations across all the 24 benchmarks was 10 percent.

These results suggest that TCaT is less effective than originally reported in [13]. To explain this discrepancy, we point out that TCaT was originally designed for a two-level inclusive cache and the adaptation to an exclusive cache hierarchy explains the relatively poor performance. Therefore, despite T-SPaCS's generally longer simulation time and simplified-T-SPaCS's similar or slightly better simulation time as compared to TCaT, T-SPaCS, and simplified-T-SPaCS estimate the miss rates for all cache configurations in the design space accurately enough to determine the optimal cache configuration.

6 Conclusions and Future Work

In this paper, we presented T-SPaCS—a *T*wo-level *S*ingle-*Pa*ss trace-driven *C*ache *S*imulation methodology for exclusive instruction and data cache hierarchies by using a stack-based algorithm to simulate both the level one and level two caches simultaneously. T-SPaCS reduces the storage and time complexity required for simulating two-level caches as compared to direct adaptation of existing single-pass cache simulation methods to two level caches using sequential simulation. T-SPaCS produces 100 percent accurate results for 99 percent of the design space, and the average simulation time speedups compared to sequential simulation time for instruction and data caches are 21.02X and 33.34X, respectively. A simplified version of T-SPaCS (simplified-T-SPaCS) increases the average simulation speedup to 30.15X for instruction caches and 41.31X for data caches, at the expense of inaccurate miss rates for 95 percent of the design space. However, even with these miss rate errors, both T-SPaCS and simplified-T-SPaCS are still able to accurately determine the optimal energy configuration for all studied benchmarks, thereby facilitating rapid design space exploration for cache tuning.

T-SPaCS is designed to simulate instruction and data caches. In unified cache simulation, the relative ordering of cache misses in separate level one instruction and level one data caches must be maintained to simulate the unified level two cache. Unfortunately, this required history of the relative access ordering cannot be captured given the current data structures and algorithms in T-SPaCS. Therefore, single-pass trace-driven unified cache simulation will be significantly different than T-SPaCS. Our future work includes extending T-SPaCS to unified cache simulation and hardware implementation for dynamic cache tuning.

Acknowledgments

This work was supported by the US National Science Foundation (NSF) (CNS-0953447). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the US National Science Foundation.

References

[1] Altera, "Nios Embedded Processor System Development," http://www.altera.com/corporate/news_room/releases/products/nrnios_delivers_goods.html, 2012.
[2] Arc Int'l, http://www.arccores.com, 2012.
[3] ARM 1156 Processor, http://www.arm.com/products/processors/classic/arm11, 2012.
[4] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures," *Proc. IEEE/ACM 33rd Ann. Int'l Symp. Microarchitecture,* pp. 245-257, Dec. 2000.
[5] S. Banerjee, G. Surendra, and S.K. Nandy, "Program Phase Directed Dynamic Cache Way Reconfiguration for Power Efficiency," *Proc. Asia and South Pacific Design Automation Conf.,* pp. 884-889, Jan. 2007.
[6] M. Brehob and R.J. Enbody, "An Analytical Model of Locality and Caching," technical report, Michigan State Univ., 1996.
[7] D. Burger, T. Austin, and S. Bennet, "Evaluating Future Microprocessors: The Simplescalar Toolset," Technical Report CS-TR-1308, Computer Science Department, Univ. of Wisconsin-Madison, July 2000.
[8] CACTI, http://www.hpl.hp.com/research/cacti/, 2012.
[9] T.M. Conte, M.A. Hirsch, and W.W. Hwu, "Combining Trace Sampling with Single Pass Methods for Efficient Cache Simulation," *IEEE Trans. Computers,* vol. 47, no. 6, pp. 714-720, June 1998.
[10] Dinero IV Trace-Driven Uniprocessor Cache Simulator, http://pages.cs.wisc.edu/~markhill/DineroIV/, 2012.
[11] EEMBC, the Embedded Microprocessor Benchmark Consortium, www.eembc.org, 2012.

[12] A. Ghosh and T. Givargis, "Cache Optimization for Embedded Processor Cores: An Analytical Approach," *ACM Trans. Design Automation of Electronic Systems,* vol. 9, no. 4, pp. 419-440, Oct. 2004.
[13] A. Gordon-Ross, F. Vahid, and N. Dutt, "Automatic Tuning of Two-Level Caches to Embedded Applications," *Proc. IEEE/ACM Design Automation and Test in Europe Conf. and Exhibition,* pp. 208-213, Feb. 2004.
[14] A. Gordon-Ross and F. Vahid, "A Self-Tuning Configurable Cache," *Proc. IEEE Design Automation Conf.,* pp. 234-237, July 2007.
[15] A. Gordon-Ross, P. Viana, F. Vahid, W. Najjar, and E. Barros, "A One-Shot Configurable-Cache Tuner for Improved Energy and Performance," *Proc. IEEE/ACM Design, Automation and Test in Europe Conf. Exhibition,* pp. 1-6, Apr. 2007.
[16] A. Gordon-Ross, J. Lau, and B. Calder, "Phase-Based Cache Reconfiguration for Highly-Configurable Two-Level Cache Hierarchy," *Proc. ACM 18th Great Lakes Symp. VLSI,* pp. 323-337, May 2008.
[17] A. Gordon-Ross, F. Vahid, and N. Dutt, "Fast Configurable-Cache Tuning with a Unified Second-Level Cache," *IEEE Tran. VLSI Systems,* vol. 17, no. 1, pp. 80-91, Jan. 2009.
[18] P. Heidelberger and H.S. Stone, "Parallel Trace-driven Cache Simulation by Time Partitioning," *Proc. Winter Simulation Conf.,* pp. 734-737, Dec. 1990.
[19] M.D. Hill and A.J. Smith, "Evaluating Associativity in CPU Caches," *IEEE Trans. Computers,* vol. 38, no. 12, pp. 1612-1630, Dec. 1989.
[20] A. Janapsatya, A. Lgnjatović, and S. Parameswaran, "Finding Optimal L1 Cache Cinfiguration for Embedded Systems," *Proc. Asia and South Pacific Design Automation Conf.,* Jan. 2006.
[21] A. Janapsatya, A. Lgnjatović, S. Parameswaran, and J. Henkel, "Instruction Trace Compression for Rapid Instruction Cache Simulation," *Proc. Conf. Design, Automation and Test in Europe,* pp. 1-6, Apr. 2007.
[22] C. Lee, M. Potkonjak, and W.H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communication Systems," *Proc. 30th Ann. Int'l Symp. Microarchitecture,* pp. 330-335, Dec. 1997.
[23] A. Malik, W. Moyer, and D. Cermak, "A Low Power Unified Cache Architecture Providing Power and Performance Flexibility," *Proc. Int'l Symp. Low Power Electronics and Design,* pp. 241-243, 2000.
[24] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems J.,* vol. 9, no. 2, pp. 78-117, 1970.
[25] MIPS32 4KE Family, http://www.mips.com/products/cores/32-64-bit-cores/, 2012.
[26] S. Segars, "Low Power Design Techniques for Micropocessors," *Proc. Int'l Solid State Circuit Conf.,* Feb. 2001.
[27] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and Exploiting Program Phases," *Proc. IEEE Micro: Top Picks from Computer Architecture Conf.,* pp. 84-93, Dec. 2003.
[28] SimPoint, http://cseweb.ucsd.edu/~calder/simpoint/, 2012.
[29] R. Sugumar and S. Abraham, "Efficient Simulation of Multiple Cache Configurations Using Binomial Trees," technical report, 1991.
[30] R.A. Sugumar, "Multi-Reconfiguration Simulation Algorithms for the Evaluation of Computer Architecture Designs," PhD thesis, Univ. of Michigan, Ann Arbor, Michigan, 1993.
[31] Tensilica, Xtensa Processor Generator, http://www.tensilica.com/, 2012.
[32] J.G. Thompson and A.J. Smith, "Efficient (stack) Algorithms for Analysis of Write-Back and Sector Memories," *ACM Trans. Computer Systems,* vol. 7, no. 1, pp. 78-117, 1989.
[33] P. Viana, A. Gordon-Ross, E. Baros, and F. Vahid, "A Table-Based Method for Single-Pass Cache Optimization," *Proc. ACM Great Lakes Symp. VLSI (GLSVLSI),* May 2008.
[34] H. Wan, X. Gao, X. Long, and Z. Wang, "GCSim: A GPU-Based Trace-Driven Simulator for Multi-level Cache," *Proc. Advanced Parallel Processing Technologies,* pp. 177-190, 2009.
[35] Z. Ying, B.T. Davis, and M. Jordan, "Performance Evaluation of Exclusive Cache Hierarchies," *Proc. IEEE Int'l Symp. Performance Analysis of Systems and Software,* pp. 89-96, 2004.
[36] C. Zhang, F. Vahid, and R. Lysecky, "A Self-Tuning Cache Architecture for Embedded Systems," *ACM Trans. Embedded Computing Systems,* vol. 3, no. 2, pp. 407-425, May 2004.

Wei Zang received the BS and MS degrees from the Zhejiang University, Hangzhou, China, in 2006 and 2008, respectively, and is currently working toward the PhD degree in electrical and computer engineering in the University of Florida, Gainesville. Her research interests include low-power embedded system design, design automation with an emphasis on cache reconfiguration, cache partitioning on multicore platforms, and energy-aware scheduling on real-time systems. She is a student member of the IEEE.

Ann Gordon-Ross (M'00) received the BS and PhD degrees in computer science and engineering from the University of California, Riverside, in 2000 and 2007, respectively. She is currently an assistant professor of electrical and computer engineering at the University of Florida, and is a member of the US National Science Foundation (NSF) Center for High Performance Reconfigurable Computing (CHREC) at the University of Florida. She is also the faculty advisor for the Women in Electrical and Computer Engineering (WECE) and the Phi Sigma Rho National Society for Women in Engineering and Engineering Technology. She received her CAREER award from the US National Science Foundation in 2010 and Best Paper awards at the Great Lakes Symposium on VLSI (GLSVLSI) in 2010 and the IARIA International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM) in 2010. Her research interests include embedded systems, computer architecture, low-power design, reconfigurable computing, dynamic optimizations, hardware design, real-time systems, and multicore platforms. She is a member of the IEEE.