A Single-Pass Cache Simulation Methodology for Two-level Unified Caches

Wei Zang and Ann Gordon-Ross*

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

***Abstract*—Cache tuning is the process of determining the optimal cache configuration given an application's requirements for reducing energy consumption and improving performance. As embedded systems trend towards unified second-level caches for improved performance, the need for fast cache tuning methodologies for multi-level cache hierarchies is becoming more critical. In this paper, we present U-SPaCS, a single-pass cache simulation methodology for design-time tuning of two-level cache hierarchies with a unified second-level cache. To afford fast simulation time, U-SPaCS maintains unique cache block addresses in a set of stacks, which enables simulation of all cache configurations for the level one instruction and data caches, and level two unified cache simultaneously in a single pass of an application's time-ordered instruction and data access trace. Experiments show that U-SPaCS can accurately determine the miss rates for a configurable cache design space consisting of 2,187 cache configurations with a 41X speedup in average simulation time as compared to the most widely-used trace-driven cache simulation.**

I. INTRODUCTION

Due to high memory latency and memory bandwidth limitations (i.e., the memory wall), optimizing the cache subsystem is critical for improving overall system performance. In addition, since the cache subsystem generally consumes a large percentage of total microprocessor system energy [16], optimizing the cache subsystem's energy consumption results in significant total system energy savings. Previous research shows that since applications require diverse cache configurations (i.e., specific values for cache parameters such as total size, block size, and associativity), cache parameters can be specialized for optimal performance or energy consumption [21]. *Cache tuning* is a popular cache optimization method that determines the optimal (or near-optimal) cache configuration in the design space (all possible cache configurations) for a particular application.

Given the availability of synthesizable core-based processors with tunable cache parameters [2][3][15], designers can perform design space exploration offline during design time to determine the optimal cache configuration. The designer synthesizes the processor with these optimal cache parameter values and the cache configuration remains fixed during runtime. Since offline cache tuning is done prior to system runtime, there is no runtime overhead (e.g., performance energy) introduced with respect to online design space exploration. Furthermore, since many embedded systems execute a fixed application or fixed set of applications, offline cache tuning is feasible for these systems.

Offline cache tuning evaluates the cache configuration design space using either analytical modeling or simulation to determine the caches' performance (hit/miss statistics). Based on the application's requirements and design constraints, a system performance/energy model is applied to these statistics to determine the optimal performance/energy configuration or a Pareto-optimal configuration trading off energy and performance. Analytical modeling [7] quickly predicts cache performance using mathematical models, but generates only statistically accurate results. Execution-driven cache simulation [4] leverages software, such as an instruction set simulator, to accurately simulate cache performance, but requires lengthy simulation time. Trace-driven cache simulation [5] reduces the simulation time by functionally simulating an application once to generate a memory access trace, and then processes that access trace quickly to determine the caches' performance. However, the simulation time for iteratively exploring a large design space can still be prohibitive, even with pruning heuristics.

Instead of iteratively exploring the design space, single-pass trace-driven cache simulation [10][14][17][18] further reduces the simulation time by evaluating multiple cache configurations simultaneously in a single processing pass of the access trace. All previous works, with the exception of [20], simulated a single level of cache. Whereas these methodologies could be directly applied to a multi-level cache by separately tuning each cache structure, the combination of these independently tuned cache configurations is unlikely to be the optimal cache *hierarchy* configuration due to the dependencies between the cache levels [9]. In multi-level caches, cache hits in higher cache levels (closer to the processor) filter the access trace to lower cache levels. This filtered access trace consists of only the higher level caches' misses. Since there is a unique filtered access trace for every combination of the higher level caches' configurations, the processing time grows exponentially as the number of each level's cache configuration increases. Therefore, developing a time efficient single-pass trace-driven cache simulation methodology is challenging.

T-SPaCS is the only previous work for two-level single-pass trace-driven cache simulation for instruction caches [20]. T-SPaCS leverages an exclusive cache hierarchy's

characteristics to simultaneously evaluate both the level one (L1) cache and the level two (L2) cache using only the complete (un-filtered) access trace in a single processing pass, thereby efficiently and accurately determining the optimal cache configuration. However, T-SPaCS's methodology is not applicable to a two-level cache hierarchy with a unified second-level cache (referred to as a two-level unified cache herein) due to the interlacing of the L1 instruction and data cache misses, which precludes independent processing of the data and instruction access traces. The relative access ordering of the L1 caches' misses to the L2 cache must be maintained, which cannot be captured with T-SPaCS's data structures and algorithms.

In this paper, we present the first, to the best of our knowledge, two-level ***U***nified ***S***ingle-***Pa***ss ***C***ache ***S***imulation methodology—U-SPaCS. We leverage an exclusive cache hierarchy due to the inherent fast runtime complexities afforded in single-pass exclusive cache simulation [20], which are critical for U-SPaCS since our current work focuses on adapting U-SPaCS for dynamic, runtime cache evaluation. The single-pass simulation feature makes U-SPaCS amenable to hardware implementation in dynamic cache tuning without runtime system intrusion [8]. Exclusive caches are applied in modern commercial processors, such as the AMD Athlon and Duron processors [1], and ARM Cortex-A8 and Cortex-A9 processors [3]. Zheng et al. [22] evaluated the exclusive cache performance and concluded that the exclusive cache is suitable for embedded systems with limited on-chip cache area since an exclusive hierarchy can provide a larger effective cache size than an inclusive hierarchy.

Our experiments show that in a design space with 2,187 cache hierarchy configurations, U-SPaCS accurately determines each cache hierarchy configuration's performance and affords a simulation time speedup of 41X on average as compared to the most widely-used trace-driven cache simulation, thereby significantly reducing offline, design-time cache tuning effort.

II. Related Work

Previous works on single-pass trace-driven cache simulation can be classified into two categories based on the data structures and algorithms used to store and process the access trace. Stack-based algorithms leverage a simple stack structure but require lengthy trace processing time and tree/forest-based algorithms leverage complex tree/forest structures and afford quick trace processing time.

Mattson et al. [14] developed the first stack-based algorithm for fully-associative caches, which served as the foundation for all future trace-driven cache simulation research. Hill and Smith [10] extended the algorithm to simulate direct-mapped and set-associative caches, and used the set refinement property to decrease the simulation time. Thompson and Smith [18] simulated write-back caches by introducing dirty-level analysis and write-back counters.

Since most previous works varied only two cache parameters while holding the remaining parameters fixed, Viana et al. [19] proposed SPCE, a stack-based algorithm that simultaneously evaluated all cache configurations with varying size, block size, and associativity in a single pass. Gordon-Ross et al. [8] implemented SPCE in hardware for runtime cache tuning. Zang and Gordon-Ross [20] developed T-SPaCS, which extended the stack-based algorithm for single-pass exclusive two-level instruction cache simulation.

To reduce the processing time, tree/forest structures/algorithms store/traverse the access trace more efficiently than stack-based structures/algorithms. Hill and Smith [10] simulated direct-mapped caches using a tree/forest structure and Janapsatya et al. [11] extended the work to simulate set-associative caches. Sugumar and Abraham [17] recognized that storage space and processing time could be reduced using tree/forest-based structures/algorithms that were specialized for the cache parameters being varied. The authors developed two structures/algorithms to simulate the cache configurations with either a fixed block size or a fixed cache size while varying the remaining two parameters.

All previous single-pass trace-driven cache simulation methodologies are only applicable to single-level caches except for T-SPaCS [20], which was developed to simulate two-level instruction caches only. Since T-SPaCS focused only on exclusive replacement operations, the separate L1 and L2 instruction caches could be logically treated as one combined cache. Therefore, by simply maintaining the complete cache access trace, T-SPaCS analyzed the L1 cache and combined L1/L2 cache using standard stack-based trace-driven cache simulation. Then, T-SPaCS leveraged three different inclusion relationships between the L1 cache and combined L1/L2 cache sets and defined three different compare-exclude operations that derived the L2 cache performance.

Even though we leverage some of T-SPaCS fundamental methods, U-SPaCS has several major advances including the ability to simulate two-level unified caches where T-SPaCS was limited to only the instruction cache hierarchy. This broadening of applicability introduces several processing challenges. First, unlike a two-level instruction cache, which includes only one L1 cache and one L2 cache both storing instructions only, in a two-level unified cache, both the L1 instruction and data caches share the L2 unified cache. Thus, all three of these caches cannot logically be considered as one combined cache and U-SPaCS cannot directly leverage the compare-exclude operations developed in T-SPaCS to derive the L2 cache performance. Second, since the L2 cache stores the blocks evicted from both the L1 instruction and data caches and the storage of both caches' evicted blocks are interlaced into the L2 cache, the instruction and data address processing cannot be isolated. This interdependency precludes the use of T-SPaCS to individually process the instruction and data addresses since during the processing of the instruction (or data) addresses, an interlaced data (or instruction) address stored in the L2 cache may affect the instruction (or data) address's L2 hit/miss. Third, the relative access ordering (interlacing) of the cache blocks evicted from the two L1 caches into the L2 cache is critical for L2 cache analysis. Different L1 cache configurations generate different relative

L1 cache eviction orderings to the L2 cache. If there are M L1 instruction cache configurations and N L1 data cache configurations, the blocks evicted from the two L1 caches to the L2 cache have MN unique eviction orderings. Therefore, efficiently maintaining and processing a potentially large number of unique eviction orderings for large L1 cache design spaces is very challenging.

III. Target Architecture Design Space and Cache Configuration Notations

U-SPaCS's target cache architecture is a two-level exclusive unified cache consisting of configurable L1 instruction and data caches and a configurable L2 unified cache, each with independently configurable total size, block size, and associativity. The L1 caches use the least recently used (LRU) replacement policy and the L2 cache uses a first-in-first-out (FIFO)-like replacement policy (the exclusive hierarchy complicates L2 cache evictions, making the L2 cache replacement process similar to FIFO).

In an exclusive cache hierarchy, the caches' contents are disjoint due to moving instead of copying cache blocks. On an L1 cache miss and an L2 cache hit, the cache block is moved from the L2 cache to the L1 cache, the evicted LRU L1 cache block is moved into the L2 cache, and the oldest block in the L2 cache is evicted and discarded (or written back to main memory if the block is dirty and the write-back policy is used). When the L1 and L2 caches both miss, the missed block is only fetched into the L1 cache (on a read miss, or a write miss when the write-allocate policy is used).

The cache hierarchy configuration design space essentially contains all combinations of the three caches' configurable parameter values. However, due to the disjoint cache contents in an exclusive cache hierarchy, all the three caches must be configured with the same block size.

To maintain concise explanations in the remainder of this paper, we introduce several cache configuration notations. We denote the L1 instruction cache configuration as $c^1\_inst(S^1\_inst, W^1\_inst, B)$, where S^1_inst, W^1_inst, and B represent the number of cache sets, associativity, and the block size, respectively. The L1 data cache and L2 cache configurations are similarly denoted as $c^1\_data(S^1\_data, W^1\_data, B)$ and $c^2(S^2, W^2, B)$, respectively, and the cache hierarchy configuration is denoted as $c\_hier(c^1\_inst, c^1\_data, c^2)$. Each cache parameter has a minimum and maximum value, denoted by the subscripts "min" and "max", respectively, and the parameter values increase by powers of two. We use cardinality $|X|$ to denote the number of different values for a cache parameter X. For reference, Table 1 summarizes all notations used throughout this paper.

Table 1: Notation reference

>>	Bitwise right shift operator.
B	Cache block size.
S	Number of sets.
W	Number of ways.
c	Cache configuration: $c(S, W, B)$.
$X^{1/2}$	Superscript 1 or 2 indicates L1 the cache or L2 cache, respectively. X can be B, S, W, or c.
$X_{\min/\max}$	Subscript min or max represents the minimum or maximum value of X, respectively. X can be B, S, W, or c.
X^1_inst	Parameter defined for the L1 instruction cache. X can be B, S, W, or c.
X^1_data	Parameter defined for the L1 data cache. X can be B, S, W, or c.
$\|X\|$	The cardinality of X, which denotes the number of different values for X. X can be B, S, W, or c.
c_hier	Cache hierarchy configuration: $c\_hier(c^1\_inst, c^1\_data, c^2)$.
T	The processed trace address.
A	Block address of T.
K_{i_inst}	Stack recording unique instruction block addresses that have the set index i_inst for B and $S^1_{\min}\_inst$, $i\_inst = A \bmod S^1_{\min}\_inst$.
K_{i_data}	Stack recording unique data block addresses that have the set index i_data for B and $S^1_{\min}\_data$, $i\_data = A \bmod S^1_{\min}\_data$.
$K_{i\_inst}[m]$ $K_{i\_data}[m]$	The m-th (counting from the stack's top) block addresses recorded in K_{i_inst} or K_{i_data}.
$conf\_inst$	Previously accessed instruction block addresses that map to the same L2 cache set as A.
$conf\_data$	Previously accessed data block addresses that map to the same L2 cache set as A.
$L2\_inst$	The block in $conf\_inst$, but has already been evicted from the L1 instruction cache.
$L2\_data$	The block in $conf\_data$, but has already been evicted from the L1 data cache.
$L2conf$	L2 cache conflicts, which determines an L2 hit/miss for the processed address.
$\{Y\}$	The collection of Y. Y can be $conf\_inst$, $conf\_data$, $L2\_inst$, $L2\_data$, or $L2conf$.

IV. Single-Pass Cache Simulation Methodology for Two-level Unified Caches—U-SPaCS

A. Overview

U-SPaCS is a single-pass simulation for configurable two-level exclusive unified cache hierarchies. U-SPaCS simultaneously evaluates the cache hierarchy configurations with varying total size, block size, and associativity. For each cache hierarchy configuration, U-SPaCS outputs each cache's (i.e., the L1 instruction and data caches and the L2 cache) number of misses and the L2 cache's number of write-backs (there is no write-back in the L1 caches since the L1 caches propagate (evict) the dirty blocks to the L2 cache instead of directly writing these blocks back to main memory). Cache tuning combines these outputs with a performance/energy model (e.g., [9]) to determine the optimal hierarchy configuration.

U-SPaCS sequentially processes each time-ordered trace address and evaluates each processed trace address with respect to all previously accessed addresses to determine if the processed address is a cache hit/miss for each cache hierarchy configuration. To classify the processed address T as a cache hit/miss for a particular cache configuration, the number of previously accessed cache blocks that map to the same cache

set as the cache block that contains T, referred to as *conflicts*, must be determined. If the number of conflicts is large enough to evict T's block's previous access, T's current access results in a cache miss.

U-SPaCS is a stack-based trace-driven cache simulator and maintains separate stack structures for the instruction and data addresses. For each cache block size B, U-SPaCS processes the trace address and records the time ordered sequence of unique instruction/data block addresses that map to the same cache set for the minimum number of sets into one instruction/data stack. The total number of required stacks for each B is $S^1_{\min}\_inst$ and $S^1_{\min}\_data$ for the instruction and data stacks, respectively. The instruction stacks are differentiated from each other using the set index for the minimum number of sets, which is denoted by i_inst ($0 \le i\_inst < S^1_{\min}\_inst$). We denote the stacks using K, such that K_{i_inst} denotes one instruction stack where all the stored block addresses have the set index i_inst for the minimum number of sets, and $K_{i\_inst}[m]$ denotes the m-th block address in the stack, where $m = 1$ denotes the stack's top. The data stacks are similarly differentiated.

Since the processing of each trace address for each block size is the same, we discuss U-SPaCS's processing for an arbitrary trace address T and an arbitrary block size B. Additionally, since the processing of an instruction and data address is the same, except for the dirty status and write-backs for data address writes, without loss of generality, we describe U-SPaCS's instruction address processing and discuss additional data address processing operation details in Section IV.F.

Fig. 1 depicts an overview of U-SPaCS's operation. A single execution of the application (using any arbitrary method such as an instruction set simulator) generates the time-ordered access trace of instruction and data addresses, including an additional read/write designation for each data address. As is the case in most offline cache performance analysis techniques, we assume an in-order processor, in which the relative orders of instruction and data addresses in the access trace do not change with different cache configurations. U-SPaCS sequentially processes the trace addresses. Given a particular B, the block address A of a

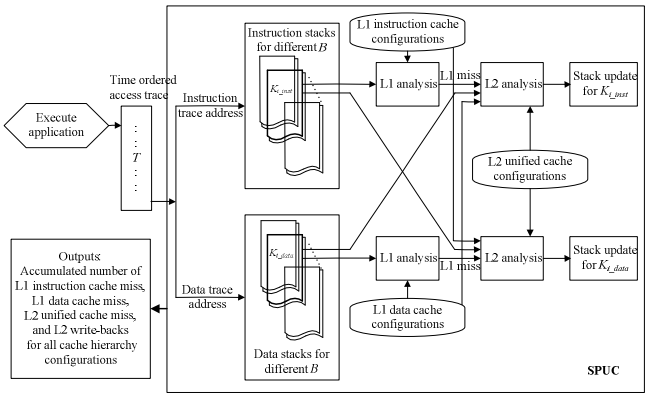

Fig. 1: Overview of U-SPaCS's operation. U-SPaCS sequentially processes the access trace and outputs the number of cache misses and write-backs for all cache hierarchy configurations.

processed instruction address T is $A = T >> (\log_2 B)$, where '>>' is the bitwise right shift operator. A's set index i_inst for the minimum number of sets for the L1 instruction cache is calculated by $i\_inst = A \bmod S^1_{\min}\_inst$, indicating the stack K_{i_inst}, from which the instruction conflicts for all possible S^1_inst and S^2 are determined. Similarly, A's set index i_data for the minimum number of sets for the L1 data cache is calculated by $i\_data = A \bmod S^1_{\min}\_data$, indicating the stack K_{i_data}, from which the data conflicts for all possible S^2 are determined (since T is an instruction address, the data conflicts are only required when analyzing the L2 cache).

When processing T, U-SPaCS first scans K_{i_inst} for T's block address A to determine if A was fetched previously. If A is not located in K_{i_inst}, A is being fetched into the cache for the first time and accessing T results in a compulsory miss for all c_hier and U-SPaCS continues processing the next trace address. If A is located in K_{i_inst} at the stack location h (i.e., $K_{i\_inst}[h]$ is equal to A), A was previously fetched and U-SPaCS begins *L1 analysis*. L1 analysis evaluates the L1 cache conflicts (conflict evaluation) in stack K_{i_inst} between $K_{i\_inst}[1]$ and $K_{i\_inst}[h]$ for all possible S^1_inst to determine if T is a hit/miss for each c^1_inst. For each c^1_inst that results in an L1 cache miss, the c^1_inst is combined with all possible c^1_data and c^2 to form the c_hier, which will be analyzed for L2 hits/misses during *L2 analysis*. L2 analysis performs conflict evaluation using both K_{i_inst} and K_{i_data} for all possible S^2. Next, for each evaluated c_hier, L2 analysis determines the *L2 cache conflicts* for A, which are the conflicts evicted from the L1 instruction and data caches after $K_{i\_inst}[h]$'s eviction from the L1 instruction cache. The number of L2 cache conflicts dictates whether T is an L2 cache hit/miss. After evaluating all c_hier for T, the *stack update process* modifies K_{i_inst} to reflect T's access. If A was previously accessed, the stack update process removes $K_{i\_inst}[h]$ from K_{i_inst} and pushes A onto the top of K_{i_inst}. If A was not previously accessed, the stack update process directly pushes A onto the top of K_{i_inst}.

After processing all of the trace addresses, U-SPaCS outputs the number of L1 instruction cache misses, L1 data cache misses, L2 unified cache misses, and write-backs for all cache hierarchy configurations.

The remainder of this section is organized as follows: Section IV.B presents the stack-based algorithm for L1 analysis and an acceleration strategy for conflict evaluation; Section IV.C presents the processing details for L2 analysis; Section IV.D summarizes U-SPaCS's algorithm for processing an instruction address; Section IV.E describes occupied blank labeling, a method to account for a special L2 cache case introduced by exclusive hierarchy-specific operations; and Section IV.F discusses write-back counting for data addresses.

B. Accelerated First-level Cache Analysis

L1 analysis uses a conventional stack-based algorithm for simulating set-associative caches [10][19], however, since L1 analysis provides a fundamental basis for L2 analysis and L1 cache hits do not require L2 analysis, this section briefly describes the stack-based algorithm for L1 analysis.

L1 analysis evaluates the conflicts in K_{i_inst} for each S^1_inst and only the conflicts accessed after $K_{i\_inst}[h]$ contribute to

$K_{i\_inst}[h]$'s eviction from the cache set. Given a processed instruction address T located in block address A, A was previously stored at stack location h. A stack address $K_{i\_inst}[m]$ where $0 < m < h$ is an L1 cache conflict for S^1_inst if ($K_{i\_inst}[m]$ mod S^1_inst) is equal to (A mod S^1_inst). The number of L1 cache conflicts dictates the minimum associativity that yields a hit, thus if L1 instruction cache associativity W^1_inst is larger than the number of L1 cache conflicts, accessing T results in an L1 instruction cache hit.

This conflict evaluation is very time consuming since each stack address $K_{i\_inst}[m]$ where $0 < m < h$ is evaluated for conflicts for all S^1_inst. Previous works leveraged the set refinement property to accelerate conflict evaluation [10][20] by evaluating each stack address for conflicts for all S^1_inst simultaneously. 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. Therefore, conflict evaluation begins at $S^1_{\min}\_inst$. If $K_{i\_inst}[m]$ is a conflict for a small S^1_inst, $K_{i\_inst}[m]$ is a conflict for the next larger S^1_inst on the condition that the most significant bits in the cache indexes of A and $K_{i\_inst}[m]$ are the same for the next larger S^1_inst. For example, if both A and $K_{i\_inst}[m]$'s cache indexes for the number of four sets are "01", which indicates that $K_{i\_inst}[m]$ is A's conflict for the number of four sets, $K_{i\_inst}[m]$ will be determined as A's conflicts for the number of eight sets as long as the one bits in $K_{i\_inst}[m]$ and A's addresses before the "01" are the same. On the contrary, if $K_{i\_inst}[m]$ is not a conflict for a small S^1_inst, conflict evaluation for all larger S^1_inst is not necessary since the cache indexes of A and $K_{i\_inst}[m]$ for all the larger S^1_inst are different.

In addition to determining the number of conflicts, U-SPaCS stores A's conflict information for all S^1_inst into a layered structure, referred to as the *instruction frame*, with $|S^1\_inst|$ number of layers. The top five layers (rectangles) in Fig. 2 (a) provide an example of the instruction frame built in L1 analysis for the processed block address A = "100110110110" to store A's conflicts for all S^1_inst. Each layer stores the conflict information for each S^1_inst in MRU (most recently used) order to preserve the conflicts' relative access order. The conflict information is recorded by a pointer that points to the conflict's stack location, which will be leveraged in L2 analysis to quickly locate the conflict.

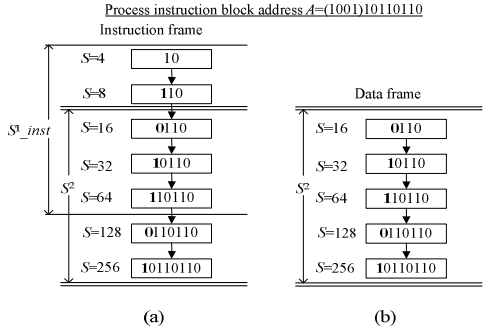

Fig. 2: Sample instruction and data frames. Each rectangle represents one layer that stores the conflict information for A for the layer's corresponding S. The conflict information is recorded using a pointer that points to the conflict's stack location. The number shown in the rectangle indicates the cache index of the recorded conflicts in that layer.

Fig. 3 summarizes the accelerated conflict evaluation algorithm. Given T's block address A and a stack address $K[m]$, conflict evaluation evaluates whether $K[m]$ is A's conflict for S from $S_{\min}$ to $S_{\max}$ (line 1). If the S is the smallest value, all index bits of A and $K[m]$ are compared (lines 7-8). Otherwise, only the most significant bits in the indexes are compared (lines 9-10). If $K[m]$ is A's conflict for S, a pointer that points to $K[m]$ is stored into the corresponding layer (dictated by S) of the instruction or data frame based on the instruction or data stack that $K[m]$ belongs to (line 3). If $K[m]$ is not A's conflict for S, the conflict evaluation for the remaining larger S is not necessary (line 5).

C. Second-level Unified Cache Analysis

If there is an L1 miss, L2 analysis is required. L2 analysis determines the number of L2 cache conflicts *L2conf* for the processed instruction address T. Since the number of *L2conf* dictates an L2 cache hit/miss for T, only the conflicts that contribute to A's eviction from the L2 cache set are counted as *L2conf*. Due to the FIFO-like L2 cache replacement policy, *L2conf* is not only the instruction/data block addresses that maps to the same L2 cache set as A (i.e. A's conflicts for S^2), but also the blocks that were evicted from the L1 instruction/data caches after $K_{i\_inst}[h]$'s eviction from the L1 instruction cache. Therefore, L2 analysis includes two steps. In the first step, A's conflicts for S^2 are determined from the stacks K_{i_inst} and K_{i_data}. We denote the collections of instruction and data conflicts as {*conf_inst*} and {*conf_data*}, respectively. In the second step, L2 analysis isolates the blocks that have been evicted from the L1 caches (whose collections are referred to as {*L2_inst*} and {*L2_data*} for the instruction and data blocks, respectively) from {*conf_inst*} and {*conf_data*} by excluding the blocks that are still stored in the L1 caches. Thereafter, {*L2conf*} is determined from {*L2_inst*} and {*L2_data*}, whose constituting blocks' eviction times are compared with $K_{i\_inst}[h]$'s eviction time from the L1 cache. The following discusses these two steps in detail.

Step One: The first step determines the instruction conflicts {*conf_inst*} from K_{i_inst} and the data conflicts {*conf_data*} from K_{i_data} for all S^2 using conflict evaluation similar to L1

```
   ConflictEvaluation (A, K[m], Smin, Smax):
1  for (S=Smin; S<=Smax; S*2)
2      if (IsConflict (A, K[m], S, Smin))
3          -record a pointer that points to K[m] in the corresponding
           layer (dictated by S) of the instruction or data frame,
           depending on the instruction or data stack that K[m]
           belongs to;
4      else
5          -break;

   //evaluate whether K[m] is A's conflict for S
6  Boolean IsConflict (A, K[m], S, Smin)
7      if (S==Smin)
8          -return ((A mod S) == (K[m] mod S));
9      else
10         -return ((A and (S>>1)) == (K[m] and (S>>1)));
```

Fig. 3: Accelerated conflict evaluation algorithm that evaluates whether the stack address $K[m]$ is A's conflict for all number of sets from $S_{\min}$ to $S_{\max}$.

analysis. However, the L1 cache eviction order does not follow the address accessing order recorded in the stacks and any of the stack addresses could be *L2conf*. For example, a block $K_{i\_inst}[m]$ ($m>h$) that was accessed (stored into the stack) before $K_{i\_inst}[h]$'s access could be evicted from the L1 cache after $K_{i\_inst}[h]$'s eviction from the L1 cache and thereby should be included in {*L2conf*}. This situation occurs when the block $K_{i\_inst}[m]$ ($m>h$) and $K_{i\_inst}[h]$ map to different L1 cache sets but the same L2 cache set. Therefore, step one must evaluate all stack addresses in K_{i_inst} and K_{i_data} for conflicts (this is in contrast to L1 analysis in Section IV.B where only the stack addresses stored after $K_{i\_inst}[h]$'s stack location are evaluated for conflicts).

Since the stacks can be very large, we leverage the acceleration strategy described in L1 analysis to speedup the conflict evaluation for all S^2 and record the conflicts into the layered structures. New layers to store the instruction conflicts for S^2 are added to the instruction frame that was built during L1 analysis (the termination of the conflict evaluation for each $K_{i\_inst}[m]$ ($0 < m < h$) in L1 analysis is correspondingly extended from $S^1_{max}\_inst$ to S^2_{max}), and a new layered structure, referred to as the *data frame*, is built to store data conflicts for each S^2. The total number of layers for the instruction frame is equal to $|S^1\_inst|$ plus $|S^2|$, and the number of layers for the data frame is equal to $|S^2|$.

Fig. 2 depicts sample extended instruction (a) and data frames (b) for a processed address T with block address A = "100110110110" for an arbitrary B, and a design space bounded by $S^1_{min}\_inst = 4$, $S^1_{max}\_inst = 64$, $S^2_{min} = 16$, and $S^2_{max} = 256$. The instruction frame stores A's instruction conflicts for all S from $S^1_{min}\_inst$ to S^2_{max} and the data frame stores A's data conflicts for all S^2 (from S^2_{min} to S^2_{max}). Since S^2_{min} is less than $S^1_{max}\_inst$, some S overlap between the range of S^1_inst and S^2, thus the number of layers in the instruction frame is equal to $|S^1\_inst|$ plus $|S^2|$ minus the overlapped number of S.

Step Two: Step two determines {*L2conf*} from {*conf_inst*} and {*conf_data*}. Since the conflicts for S^2 could include conflicts (block addresses) currently stored in the L1 caches, step two excludes (removes) the conflicts still stored in the L1 caches from {*conf_inst*} and {*conf_data*}, and the remaining conflicts, {*L2_inst*} and {*L2_data*}, respectively, are the conflicts stored exclusively in the L2 cache. However, not all the conflicts in {*L2_inst*} and {*L2_data*} can be classified as T's L2 conflicts {*L2conf*} since {*L2conf*} must only be the conflicts that were evicted from the L1 caches after $K_{i\_inst}[h]$ was evicted from the L1 instruction cache.

However, the blocks' relative stack locations are not sufficient to determine the blocks' eviction orders since the stack only stores the blocks' latest access and maintains no previous access history. Therefore, each stack location includes an array to explicitly record that block's eviction time from the L1 cache. Each array element corresponds to the eviction time for each L1 cache configuration and the array size is equal to the associated cache's (instruction or data) design space size. Each time that a block is evicted from the L1 cache, the currently processed address's trace order value is added to the evicted block's array to indicate the block's eviction time. We begin counting the trace order from "1" such that an eviction time array value of "0" indicates that the block is currently stored in the L1 cache for the corresponding L1 cache configuration.

If there is an L1 cache miss determined for T during the L1 analysis, the W^1_inst-th L1 cache conflict, which is acquired from the instruction frame layer's MRU ordering, is the evicted block from the L1 cache after fetching A. Thus, T's trace order is assigned to the corresponding (dictated by the c^1_inst) element in the eviction time array of the W^1_inst-th L1 cache conflict. Since an eviction time value of "0" implies that the block is located in the L1 cache, during the stack update process, all of the elements in the eviction time array of the newly pushed address A are cleared to "0".

By maintaining the eviction time array, *L2conf* can be easily determined using the eviction times of the conflicts in {*conf_inst*} and {*conf_data*}. First, {*L2_inst*} and {*L2_data*} are derived from {*conf_inst*} and {*conf_data*} by excluding the conflicts with an eviction time value of "0". Next, {*L2conf*} is determined by selecting the conflicts from {*L2_inst*} and {*L2_data*} with the condition that the conflict's eviction time is larger than the eviction time of $K_{i\_inst}[h]$.

D. U-SPaCS's Processing Algorithm

In this section, we will summarize U-SPaCS's processing algorithm for an instruction trace address T and a particular block size B. In order to support any arbitrary configurable B, during each T's processing, U-SPaCS simply repeats this algorithm for each B.

Fig. 4 depicts U-SPaCS's algorithm for processing an instruction address T for a particular B. First, A, *i_inst*, and *i_data* are calculated (lines 1-3). Next, U-SPaCS searches the stack K_{i_inst} to determine whether the block A was accessed before (line 4). If there is no h such that $K_{i\_inst}[h]$ is equal to A, accessing T results in a compulsory cache miss for all of the cache hierarchy configurations (lines 5-6), T's processing is finished (line 7), and the stack update process pushes A onto the top of K_{i_inst} (line 51). If there exists h such that $K_{i\_inst}[h]$ is equal to A, U-SPaCS begins L1 analysis and performs conflict evaluation for the stack addresses $K_{i\_inst}[m]$, where $0 < m < h$, for all S from $S^1_{min}\_inst$ to S^2_{max} and stores the conflict information into the corresponding layers (dictated by S) of the instruction frame (lines 9-10). For each S^1_inst, the conflicts stored in the corresponding layer of the instruction frame are all of the L1 cache conflicts (lines 11-12). Therefore, all c^1_inst with W^1_inst larger than the number of L1 cache conflicts result in an L1 cache hit, and thereby, all the *c_hier* with these c^1_inst also result in an L1 cache hit (lines 13-16). Otherwise, c^1_inst results in an L1 cache miss and the c^1_inst is recorded into an L1 cache miss list for future reference (lines 17-19). Since T's conflict cache miss results in a block eviction after fetching A into the L1 instruction cache, U-SPaCS locates the evicted block (which is the W^1_inst-th L1 cache conflict according to the MRU order) and set T's trace access order to the value of the corresponding element (dictated by c^1_inst) in the evicted block's eviction time array (lines 20-21).

As long as there is at least one L1 cache miss recorded in the L1 cache miss list, U-SPaCS performs L2 analysis (line 22). L2 analysis continually evaluates the conflicts for the stack addresses $K_{i\_inst}[m]$, where $m > h$, for all S^2 and stores the conflict information in the corresponding layers (dictated by S^2) of the instruction frame. L2 analysis similarly evaluates all stack addresses in K_{i_data} for all S^2 and stores the conflict information in the corresponding layers (dictated by S^2) of the data frame (lines 23-26). After conflict evaluation for both the instruction and data stacks, for each S^2, the conflicts in the corresponding layer (dictated by S^2) in the instruction and data frames form {*conf_inst*} and {*conf_data*}, respectively (lines 28-29). For each c^1_inst that resulted in an L1 cache miss (line 30), all *c_hier* that is composed of c^1_inst and all combinations of every S^2 (line 27), c^1_data (line 37), and W^2 (line 43) are analyzed for L2 cache hits/misses. The sub-collection of all the instruction conflicts in {*L2conf*} is derived from {*conf_inst*} by excluding those *conf_inst*, whose eviction time array's corresponding element (dictated by c^1_inst) value is less than (including the value "0") the value of $K_{i\_inst}[h]$'s (lines 31-36). Thereafter, the sub-collection of all the data conflicts in {*L2conf*} is derived from {*conf_data*} in the same way, by comparing the eviction time array's corresponding element (dictated by c^1_data) value of *conf_data* and $K_{i\_inst}[h]$ (lines 38-42). With the total number of derived *L2conf*, the *c_hier* with different W^2 generates the L2 hit/miss results (lines 43-47). After L1 and L2 analysis, U-SPaCS performs the stack update process for *T*, which removes $K_{i\_inst}[h]$ from the stack (lines 49-50) and pushes *A* onto the top of K_{i_inst} (line 51). All of the elements in the eviction time array of the newly pushed stack address are cleared to "0" (line 52), and the instruction and data frames built for *T*'s processing are freed/cleared (line 53).

E. Occupied Blank Labeling

In an inclusive cache hierarchy and assuming no cache coherency evictions, a cache block remains valid until the application terminates or a process switch occurs. However, in the exclusive cache hierarchy, the eviction process can lead to a valid cache block being switched to the invalid state. This special case occurs when the number of L1 cache sets is less than the number of L2 cache sets, such that the L1 cache blocks evicted from a single L1 cache set will map into several different L2 cache sets. For example, if the L1 cache has four sets and the L2 cache has eight sets, the evicted blocks from the L1 cache set with index "01" could be stored into either of the L2 cache sets with indexes "001" and "101". If *T* maps to the L1 cache set with index "01" and the L2 cache set with index "001" and accessing *T* results in an L1 cache miss and an L2 cache hit, the block *A* (that *T* locates in) will be fetched from the L2 cache into the L1 cache. To keep the exclusive cache hierarchy, the block in the L2 cache set (with index "001") used to store *A* is invalidated. If fetching *A* into the L1 cache set (with index "01") evicts a block and the evicted block maps to the same L2 cache set (with index "001") as *A*, the evicted block will be moved from the L1 cache set with index "01" to the L2 cache set with index "001" and the L2 cache way with the invalid block will be occupied by a valid

```
    Process instruction address (T, B):
1   -A = T >> log2B;
2   -i_inst = A mod S1min_inst;
3   -i_data = A mod S1min_data;
4   -search for Ki_inst[h] that satisfying (Ki_inst[h]==A) in stack Ki_inst;
5   if (Ki_inst[h] is not searched)
6   |  -all c_hier results in a miss; // compulsory miss of T
7   |  -goto END_PROCESSING;
8   else
    |  //L1 cache analysis
9   |  for (m = 1; m < h; m++)
10  |      -ConflictEvaluation (A, Ki_inst[m], S1min_inst, S2max);
    |
11  |  for (S1_inst = S1min_inst, S1_inst<= S1max_inst; S1_inst*2)
12  |  |  -check the number of L1 cache conflicts recorded in corresponding layer
    |  |  (dictated by S1_inst) of the instruction frame;
13  |  |  for (W1_inst = W1min_inst, W1_inst <= W1max_inst; W1_inst*2)
14  |  |  |  if (W1_inst > the number of L1 cache conflicts)
15  |  |  |  |  -c1_inst(S1_inst, W1_inst, B) results in a L1 cache hit;
16  |  |  |  |  -each c_hier including the c1_inst results in an L1 cache hit;
17  |  |  |  else
18  |  |  |  |  -c1_inst(S1_inst, W1_inst, B) results in an L1 cache miss;
19  |  |  |  |  -record the c1_inst in a L1 cache miss list;
    |  |  |  |  // set the eviction time for the L1 evicted block
20  |  |  |  |  -L1_eviction = the W1_inst-th L1 cache conflict, which is
    |  |  |  |   determined from the corresponding layer (dictated by S1_inst)
    |  |  |  |   of the instruction frame;
21  |  |  |  |  -the corresponding element (dictated by c1_inst) in the
    |  |  |  |   eviction time array of L1_eviction = T's trace order;
    |  |
    |  //L2 cache analysis
22  |  if (L1 cache miss list != NULL)  // there is L1 cache misses
23  |  |  for (m > h; Ki_inst[m] !=NULL; m++)
24  |  |     -ConflictEvaluation (A, Ki_inst[m], S2min, S2max);
25  |  |  for (m = 1; Ki_data[m] !=NULL; m++)
26  |  |     -ConflictEvaluation (A, Ki_data[m], S2min, S2max);
    |  |
27  |  |  for (S2 = S2min, S2<= S2max; S2*2)
28  |  |  |  -the conflicts recorded in the corresponding layer (dictated by S2)
    |  |  |   of the instruction frame form {conf_inst};
29  |  |  |  -the conflicts recorded in the corresponding layer (dictated by S2)
    |  |  |   of the data frame form {conf_data};
30  |  |  |  for (each c1_inst in the L1 cache miss list)
31  |  |  |  |  -A_EvictionTime = the eviction time array's corresponding
    |  |  |  |   element (dictated by c1_inst) value of Ki_inst[h];
32  |  |  |  |  for (each conf_inst in {conf_inst})
33  |  |  |  |  |  -chk_EvictionTime = the eviction time array's corresponding
    |  |  |  |  |   element (dictated by c1_inst) value of conf_inst;
34  |  |  |  |  |  if (chk_EvictionTime < A_EvictionTime)
35  |  |  |  |  |     -exclude the conf_inst from the collection {conf_inst};
36  |  |  |  |  -{L2conf} = {conf_inst};
37  |  |  |  |  for (each c1_data)
38  |  |  |  |  |  for (each conf_data in {conf_data})
39  |  |  |  |  |  |  -chk_EvictionTime = the eviction time array's corresponding
    |  |  |  |  |  |   element (dictated by c1_data) value of conf_data;
40  |  |  |  |  |  |  if (chk_EvictionTime < A_EvictionTime)
41  |  |  |  |  |  |     -exclude the conf_data from the collection {conf_data};
42  |  |  |  |  |  -{L2conf} = {L2conf} ∪ {conf_data};
43  |  |  |  |  |  for (W2 = W2min, W2 <= W2max; W2*2)
44  |  |  |  |  |  |  if (W2 > the number of L2conf)
45  |  |  |  |  |  |  |  -c_hier(c1_inst, c1_data, c2(S2, W2, B)) results in an
    |  |  |  |  |  |  |   L2 cache hit (and L1 cache miss);
46  |  |  |  |  |  |  else
47  |  |  |  |  |  |  |  -c_hier(c1_inst, c1_data, c2(S2, W2, B)) results in an
    |  |  |  |  |  |  |   L2 cache miss  (and L1 cache miss);

48  END_PROCESSING:
    //stack update process for T
49  if (Ki_inst[h] was searched)
50  |  -remove Ki_inst[h] from the stack Ki_inst;
51  -push A to the top of Ki_inst as Ki_inst[1];
52  -clear all the elements in Ki_inst[1]'s eviction time array to "0";
53  -free the instruction frame and data frame;
```

Fig. 4: U-SPaCS's algorithm for processing an instruction address *T* for a particular *B*.

block. However, if the evicted block maps to the L2 cache set with index "101", which is different from the set that *A* is fetched from, the invalid block is still in the L2 cache set with index "001". The contents of this invalid block may have been involved in evicting a block that was previously stored in the L2 cache (i.e., the blocks that were stored into the L2 cache set before *A* was stored in this L2 cache set), but was not tracked by the L2 analysis. Thus, for later access of the already evicted block from the L2 cache set, U-SPaCS might introduce an incorrect L2 cache hit/miss classification.

T-SPaCS referred to the invalid blocks as *occupied blanks* (BLK) and used *BLK labeling* to keep track of the BLK [20]. T-SPaCS's results revealed that even though the BLK introduced an average miss rate error of 0.71%, T-SPaCS still determined the optimal cache configurations for all of the studied benchmarks and therefore, BLK labeling was not necessary. However, since our experimental results revealed that the induced miss rate error for a two-level unified cache was as high as 20% and this error caused cache tuning to incorrectly determine the optimal cache configuration, we integrate *BLK labeling* [20] into U-SPaCS.

A bit-array associated with each stack address is used to label the BLK. The bit-array's size is equal to the number of *c_hier* with the same *B*. A 'set' bit in the array denotes that a BLK follows the block for the corresponding *c_hier*. If accessing *T* results in an L2 cache hit, and the evicted block from the L1 cache does not map to the same L2 cache set as *T*, a BLK is generated in the L2 cache. BLK labeling determines the W^2-th (MRU ordering) conflict stored in the L2 cache by sorting the conflicts in {*L2store_inst*} and {*L2store_data*} based on the conflicts' eviction times, and then sets the BLK label in the corresponding element (dictated by *c_hier*) of the bit-array for the W^2-th conflict. L2 analysis is augmented by including this BLK label examination. If there is a 'set' label for any L2 cache conflict, $K_{i\_inst}[h]$ has already been evicted from the L2 cache, and thereby, accessing *T* results in an L2 cache miss even though the number of L2 cache conflicts is less than W^2.

F. Write-back Counting

The processing for data and instruction addresses is essentially the same, except that data address processing must consider the data write-backs. Assuming a write-back policy (the write-through policy requires no special processing), we distinguish between writes that propagate to lower levels of cache (write-backs of dirty blocks due to evictions) and writes that *avoid* propagation to lower levels of cache. During L1/L2 analysis, U-SPaCS records the number of write-avoids to calculate the number of write-backs [18] where the number of write-backs is equal to the total number of writes minus the number of write-avoids.

Each data stack address includes a bit-array to indicate the dirty status of the block for all *c_hier*. The bit-arrays' size is equal to the number of *c_hier* with the same *B*. During the stack update process, when the processed data address *T* is a write, all elements in the bit-array of the newly pushed *A* on the stack are set as 'dirty'. When *T* is a read, all elements in the bit-array of the newly pushed *A* on the stack retain the same contents as the removed block $K_{i\_data}[h]$, except *T*'s reading results in an L2 cache miss. On an L2 cache miss, *A*'s dirty status is set as 'clean' for the corresponding *c_hier* since an L2 cache miss implies that *A* is fetched from main memory. In L1/L2 analysis, if writing *T* results in an L1/L2 cache hit and $K_{i\_data}[h]$ is dirty, the write's propagation to memory is avoided and the number of write-avoids is incremented.

V. Experimental Results and Analysis

We verified the cache hierarchy miss and write-back rates outputted by U-SPaCS and examined U-SPaCS's simulation time efficiency using the entire EEMBC [6] benchmark suite, five arbitrarily selected benchmarks from Powerstone [13], and four arbitrarily selected benchmarks from MediaBench [12] (due to incorrect execution, we could not evaluate the complete Powerstone and MediaBench suites). We generated the access traces using SimpleScalar's [4] sim-cache module and compared U-SPaCS with the widely-used trace-driven cache simulator Dinero IV [5]. We modified Dinero to simulate an exclusive cache hierarchy.

We leveraged the same configurable L1 instruction cache, L1 data cache, and L2 unified cache design space as in [9] ([9] showed that this design space provided an appropriate variation in cache configurations for similar benchmarks). The L1 instruction and data cache sizes ranged from 2 to 8 Kbytes, the L2 cache size ranged from 16 to 64 Kbytes, the L1/L2 cache associativities ranged from direct-mapped to 4-way, and the L1/L2 cache block sizes ranged from 16 to 64 bytes. Given this configurability and considering the block size restriction (Section III), the total number of cache hierarchy configurations was 2,187.

A. Accuracy Evaluation

We verified U-SPaCS's accuracy by comparing U-SPaCS's cache miss and write-back rates with Dinero's exact cache miss and write-back rates for each benchmark. U-SPaCS's miss rates for all of the caches and the write-back rates for the L2 caches for all of the cache hierarchy configurations were 100% identical to Dinero's. Since U-SPaCS provides accurate results, cache tuning can always determine the optimal cache configuration considering the application requirements and/or design constraints.

B. Simulation Time Evaluation

Since we are the first to propose a single-pass trace-driven cache simulation for two-level unified caches, there is no prior work to directly compare to. Therefore, we quantified U-SPaCS's simulation efficiency by comparing U-SPaCS's total simulation time to simultaneously evaluate the entire design space with the simulation time required by the most widely-used trace-driven cache simulator, Dinero, to iteratively evaluate the design space. We tabulated the *user time* reported from the Linux *time* command for the simulations running on a Red Hat Linux Server version 5.2 with a 2.66 GHz processor and 4 gigabytes of RAM. Fig. 5 depicts U-SPaCS's simulation time speedup as compared to Dinero. U-SPaCS's simulation speedups reached as high as 72X, with an average speedup of 41X.

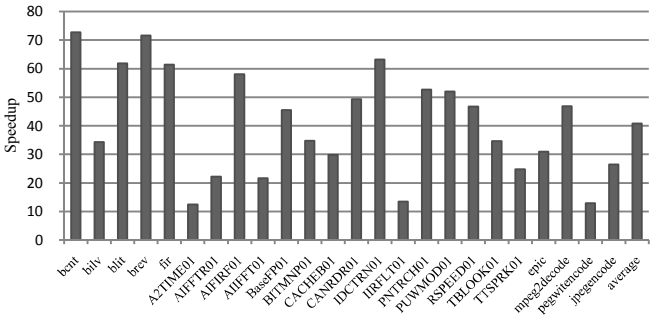

Fig. 5: U-SPaCS's simulation time speedup as compared to Dinero.

Since the speedups varied significantly across different applications, we further analyzed the results to determine the cause of this wide variation. The differentiating factor between applications was the access trace length (number of addresses in the access trace file). Fig. 6 depicts the logarithmic-scaled simulation time (in seconds) for Dinero and U-SPaCS, and logarithmic-scaled speedup of U-SPaCS as compared to Dinero with respect to increasing access trace length (logarithmic-scaled). Each graph point represents a benchmark such that vertically correlated points represent the benchmarks' performance for both simulation methods and the resulting speedup. The results indicated that Dinero's simulation time increased linearly as the access trace increased due to Dinero's constant simulation time for each trace address. U-SPaCS's simulation time does not strictly follow this linear increasing relationship because U-SPaCS's simulation time also depends on the instruction and data stacks' sizes and the L1 caches' miss rates. The stacks' sizes dictate the complexity of the conflict evaluation during L1 and L2 analysis. Furthermore, only L1 cache misses require L2 analysis, which is lengthy as compared to L1 analysis. The combination of application-specific behavior and the cache hierarchy configuration dictates the stacks' sizes and cache miss rates, thus resulting in the observed speedup variations.

Since U-SPaCS's simulation time increased as the L1 cache miss rates and stacks' sizes increased, thereby decreasing U-SPaCS's speedup, we evaluated the relationship between the L1 cache miss rates and the stacks' sizes with U-SPaCS's

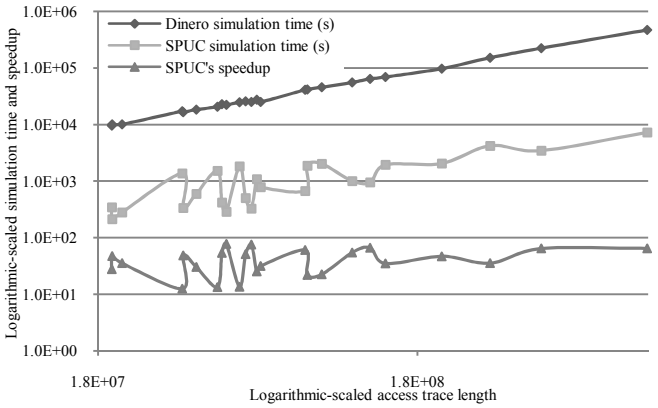

Fig. 6: The logarithmic-scaled simulation time (in seconds) for Dinero and U-SPaCS and the logarithmic scaled speedup attained by U-SPaCS as compared to Dinero with respect to increasing access trace length.

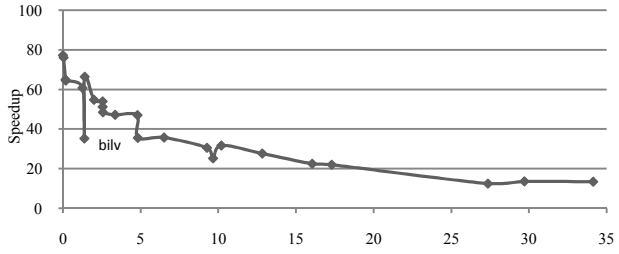

Fig. 7: U-SPaCS's speedup with respect to the product of the average L1 cache miss rate and the average stack size.

speedup. Fig. 7 plots U-SPaCS's speedup for each benchmark (graph point) with respect to the product of the benchmarks' average L1 miss rate and average stack size. For each benchmark, the average L1 miss rate was the summation of the average L1 instruction/data cache miss rates, averaged across all L1 instruction/data cache configurations, and weighted by the percentage of instruction/data accesses in the access trace. Since the stacks' sizes increase during U-SPaCS's processing, we recorded the corresponding stacks' sizes for each trace address' processing and generated a histogram for all stack sizes. Based on the histogram, we calculated the average stack size. The results in Fig. 7 verified that the speedup generally decreased as the product of the average L1 cache miss rate and average stack size increased. One outlying point, the *bilv* benchmark, did not match this decreasing speedup trend due to a relatively higher average L1 cache miss rate as compared to the neighboring graph points. This outlying point revealed that the L1 cache miss rate had a larger impact on the speedup than the stack size. Therefore, we re-plotted U-SPaCS's speedup with respect to the product of the square of the average L1 cache miss rate and average stack size in Fig. 8. With a higher importance placed on the L1 cache miss rate, all benchmark points follow the decreasing speedup trend.

VI. Conclusions and Future Work

In this paper, we presented U-SPaCS, which is the first, to the best of our knowledge, single-pass cache simulation methodology for two-level unified caches. U-SPaCS simultaneously evaluates all cache hierarchy configurations using a stack-based algorithm to store and evaluate uniquely accessed addresses with additionally recorded per-block eviction time information. Experimental results indicated that U-SPaCS's cache miss rates and write-backs were 100% accurate for all cache configurations and the average

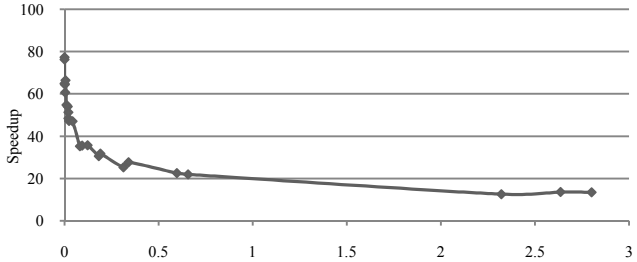

Fig. 8: U-SPaCS's speedup with respect to the product of the square of the average L1 cache miss rate and the average stack size.

simulation time speedup was 41X as compared to the most widely-used trace-driven cache simulator. Our future work includes generalizing U-SPaCS to simulate cache hierarchies with an arbitrary number of cache levels and extending U-SPaCS to support cache simulation in multi-core architectures. Additionally, we are exploring leveraging U-SPaCS for dynamic runtime cache tuning using either custom hardware embedded within a microprocessor or a field-programmable gate array (FPGA) coprocessor. Such non-intrusive runtime cache tuning is especially critical for environments and applications with strict timing deadlines, such as real-time systems, or highly power-constrained environments, such as aerospace applications.

ACKNOWLEDGEMENTS

This work was supported by the National Science Foundation (CNS-0953447) and (ECCS-0901706). 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 National Science Foundation.

REFERENCES

[1] AMD embedded processors, http://www.amd.com/us/products/embedded/processors/Pages/embedded-processors.aspx.

[2] Arc cores, http://www.synopsys.com/IP/ConfigurableCores/Pages/default.aspx.

[3] ARM, http://www.arm.com/products/processors/.

[4] D. Burger, T. Austin and S. Bennet, "Evaluating Future Microprocessors: the Simplescalar Toolset," *Technical Report, CS-TR-1308*, Jul. 2000.

[5] Dinero IV Trace-Driven Uniprocessor Cache Simulator, http://pages.cs.wisc.edu/~markhill/DineroIV/.

[6] EEMBC, http://www.eembc.org.

[7] A. Ghosh and T. Givargis, "Cache optimization for embedded processor cores: an analytical approach," *ACM Trans. on Design Automation of Electronic Systems*, Vol. 9, pp. 419-440, 2004.

[8] A. Gordon-Ross, P. Viana, F. Vahid, W. Najjar. and E. Barros, "A One-Shot Configurable-Cache Tuner for Improved Energy and Performance," *IEEE/ACM Design, DATE*, Apr. 2007.

[9] A. Gordon-Ross, F. Vahid, and N. Dutt, "Fast configurable-cache tuning with a unified second-level cache," *IEEE Trans. on Very Large Scale Integration Systems*. Vol. 17, pp. 80-91, 2009.

[10] M. D. Hill, and A. J. Smith, "Evaluating associativity in CPU caches," *IEEE Trans. Comput.*, Vol. 38, pp. 1612-1630, 1989.

[11] A. Janapsatya, A. Lgnjatović, and S. Parameswaran, "Finding optimal L1 cache configuration for embedded systems," *Asia and South Pacific Design Automation Conference*, 2006.

[12] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communication systems," *Proc. 30th Annual International Symposium on Microarchitecture*, 1997.

[13] A. Malik, W. Moyer, and D. Cermak, "A low power unified cache architecture providing power and performance flexibility," *Intl. Symposium on Low Power Electronics and Design*, 2000.

[14] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," IBM Systems Journal, Vol. 9, pp. 78-117, 1970.

[15] MIPS32 4KTM Processor Core Family Software User's Manual, http://d3s.mff.cuni.cz/~ceres/sch/osy/download/MIPS32-4K-Manual.pdf. 2001.

[16] S. Segars, "Low Power Design Techniques for Micropocessors," *International Solid State Circuit Conference*, Feb. 2001.

[17] R. Sugumar, and S. Abraham, "Efficient simulation of multiple cache configurations using binomial trees," *Tech. Report*, 1991.

[18] J. G. Thompson and A. J. Smith, "Efficient (stack) algorithms for analysis of write-back and sector memories," *ACM Trans. on Computer Systems*, Vol. 7, pp. 78-117, 1989.

[19] P. Viana, A. Gordon-Ross, E. Baros and F. Vahid, "A table-based method for single-Pass cache optimization," *ACM Great Lakes Symposium on VLSI*, 2008.

[20] W. Zang and A. Gordon-Ross, "T-SPaCS – a Two-level Single-pass Cache Simulation Methodology," *Proc. 16th Asia and South Pacific Design Automation Conference*, Jan. 2011.

[21] C. Zhang, F. Vahid and R. Lysecky, "A Self-tuning Cache Architecture for Embedded Systems," ACM Trans. on Embedded Computing Systems, Vol. 3, No. 2, pp. 407-425, May 2004.

[22] Y. Zheng, B.T. Davis, and M. Jordan, "Performance Evaluation of Exclusive Cache Hierarchies," *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2004.