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

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 – The cache hierarchy's large contribution to total microprocessor system power makes caches a good optimization candidate. We propose a single-pass trace-driven cache simulation methodology – T-SPaCS – for a two-level exclusive instruction cache hierarchy. Instead of storing and simulating numerous stacks repeatedly as in direct adaptation of a conventional trace-driven cache simulation to two level caches, T-SPaCS simulates both the level one and level two caches simultaneously using one stack. Experimental results show T-SPaCS efficiently and accurately determines the optimal cache configuration (lowest energy).

I. Introduction

C*ache tuning* as the process of determining the best cache configuration (values for cache parameters such as total size, block size, and associativity) in the *design space* (the collection of all possible cache configurations) for particular application requirements is a prevailing optimization technique. Cache tuning can reduce energy consumption by as much as 40% on average [7][19].

Cache tuning can be applied either at design time (i.e., offline static cache tuning) or during runtime (i.e., online dynamic cache tuning. Static cache tuning is suitable for stable systems with predictable inputs and execution behavior. Designers determine cache parameter values during design time and set these values in synthesizable soft-core processors [1] or hard-core processors [7][19] with configurable caches. Static cache tuning introduces no runtime overhead since designers perform design space exploration prior to system runtime. Alternatively, dynamic cache tuning performs design space exploration during runtime [7] and requires no designer effort. Whereas this method can adaptively react to a changing system environment [5][6], online design space exploration imposes system overheads (e.g., performance, area, power/energy). Additionally, determining when to explore the design space is challenging [5]. In this paper, we focus on static cache tuning.

Most existing offline static cache tuning 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 mathematical models [4], but analytical modeling can be inaccurate. Simulation methods improve cache tuning accuracy by simulating each cache configuration, however, simulation time can be lengthy when iteratively exploring a large design space (even relatively fast simulation methods such as functional simulation instead of cycle accurate simulation can still require lengthy simulation times [7][19]).

Trace-driven cache simulation significantly reduces design exploration time by functionally simulating an application once (one lengthy simulation) to produce a memory reference trace (*access trace*), and then a cache simulator processes (multiple faster simulations) the access trace for each cache configuration. Although access trace files are typically very large with large storage space requirements and slow processing time, approaches such as SimPoint [15], trace sampling, and trace compression reduce these overheads.

Instead of iteratively exploring the design space as is typical with most previous methods, single-pass trace-driven simulation evaluates multiple configurations simultaneously in a single simulation pass [9][13][16][17], achieving simulation speedups on the order of tens [10][18] as compared to iterative simulation. However, all previous methods, to the best of our knowledge, only simulate a single level of cache even though multi-level caches are becoming more common in embedded systems.

Unfortunately, inherent multi-level cache execution characteristics make direct adaptation 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 and produces *filtered traces* for each level two cache (L2). Another words, each unique L1 configuration's misses form a unique filtered trace for L2, each of which must be separately stored (large storage requirements) and processed (long processing time).

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 an exclusive instruction cache. The use of an exclusive cache hierarchy limits storage and processing overheads and enables L1 and L2 to be logically analyzed as one single combined cache. A supplementary processing step extracts the exclusive L2 contents. Our proposed methodology determines the optimal cache configuration (lowest energy) with high simulation speedup and low storage requirements compared to iterative simulation.

II. Related Work

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

Mattson et al. [13] first proposed the stack-based algorithm, wherein a stack data structure stored the access trace. 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 [9] extended the stack-based algorithm to simulate direct-mapped and set-associative caches. Thompson and Smith [17] 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 [16] proposed a tree data structure-based algorithm that provided a maximum 5X simulation speedup. Janapsatya et al. [10] further reduced simulation time using a forest data structure, but increased the storage requirements. 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, however, this method can be difficult to setup in practice and may require large computing resources for large design spaces.

Since these tree-based algorithms and parallel simulations are not amenable to hardware implementation for runtime cache tuning, the stack algorithm is still widely used. Viana et al. [18] proposed SPCE (Single Pass Cache Exploration), which attained speedups as high as 14x compared to previous work and Gordon-Ross et al. [8] architected a hardware version for non-intrusive runtime cache tuning.

Whereas these single-pass cache simulation methodologies (stack- and tree-based) are highly efficient, these methods are limited to a single level of cache. In this paper, 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.

III. Two-Level Cache Characteristics

Since one of the major challenges in two-level single-pass cache simulation is the storage and simulation time required to process each filtered 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, higher level cache's contents are a subset of the lower level cache's (closer to the processor) contents. L1 misses are copied from L2, L2 misses are copied from main memory, and evicted blocks are discarded (without loss of generality we assume the instruction cache has no dirty blocks). In an exclusive hierarchy 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 caches. L1 misses are *moved* from L2 (into L1) and the evicted L1 block is moved to L2. L2 misses are copied from main memory directly to L1 (L2 evictions are discarded). 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 using a stack-based algorithm.

To exemplify the reduced storage and simulation time 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 (Section IV presents stack processing details). In the inclusive hierarchy (a), each cache is processed separately. The L1 stack records the complete access trace, and for each L1 configuration, the unique filtered trace is recorded in an L2 stack. Each L2 stack is separately processed using the same process as for single-level cache simulation. In the exclusive hierarchy (b), only one stack is required since L1 and L2 are treated as one combined cache and are evaluated simultaneously.

This difference in stack processing has a large impact on the storage and simulation time complexities. The inclusive cache hierarchy requires one L1 stack and M L2 stacks (M is the number of L1 configurations) with $O(n)$ stack elements (all stack addresses are unique, thus n is the program size). Therefore the storage and time complexities for an inclusive and exclusive cache is $O((M+1)n)$ and $O(n)$, respectively. The tradeoff for reduced overheads is a design space reduction. For example, even though an exclusive hierarchy requires L1 and L2 block sizes to be equal, previous work [6] shows that for a large design space, several cache configurations offer nearly equal energy and performance, thus this restriction will have a nominal affect on the optimal energy cache configuration.

IV. Two-Level Single-Pass Tuning Methodology

T-SPaCS is suitable for a highly configurable cache hierarchy by simultaneously evaluating size, block size, and associativity. T-SPaCS's output is the miss rates for all cache configurations. When combined with a performance and energy model [7], a system designer can determine an appropriate cache configuration (e.g., highest performance, lowest energy, or pareto-optimal design trading off performance and energy).

First, we present T-SPaCS's functional overview. A single application execution produces the instruction access trace that is processed once using single-pass cache simulation. During simulation, the time ordered sequence of unique addresses is recorded into a stack structure and *stack processing* is done for each number of cache sets in the design space for the combined cache (the combination of size, block size, and associativity determines a configuration's number of cache sets). For each address *Addr* in the access trace and each number of cache sets, stack processing begins at the top of the stack and determines the *conflicts* with *Addr* (previously accessed addresses that map to the same cache set as *Addr* given a particular configuration). Next, a supplementary process categorizes these conflicts as either L1 or L2 conflicts. The number of conflicts dictates the minimum associativity necessary for *Addr* to be a hit. After stack processing, the *stack update* process removes *Addr* from the stack if *Addr* was accessed previously, and then pushes *Addr* on the top of the stack.

The remainder of this section presents T-SPACS's detailed operation, which is based on the stack-based algorithm for single-level cache simulation described in Section IV.A. We extend the methodology to L2 in Section IV.B. Section IV.C

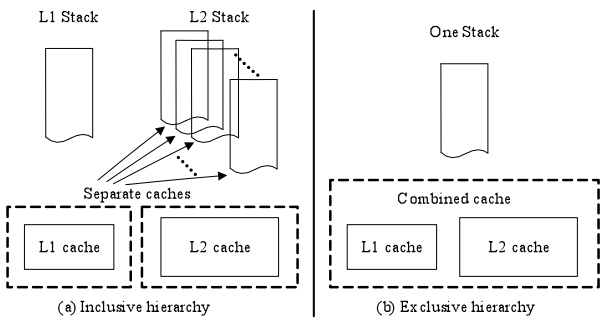

Fig 1: Storage requirements in the stack-based algorithm for a two-level (a) inclusive cache hierarchy and (b) exclusive cache hierarchy.

discusses acceleration strategies to assist stack processing. TABLE 1 provides the reference for notations that will be used throughout the paper.

A. Stack-based Single-Level Cache Analysis

The single-level cache stack processing algorithm serves as the basis for two-level cache analysis. *Addr*'s presence in a cache set (the set that *Addr* maps to) depends on the cache configuration and the number of conflicts in the stack before $Addr^B$ (previous access to *Addr's* cache block). A stack address A is recorded as a conflict in $\{SConfl\}$ for the cache configuration with block size B ($B = 2^b$) and number of sets S when $(A >> b) \bmod S = (Addr >> b) \bmod S$.

Fig 2 illustrates the stack-based algorithm [18] for processing each *Addr* in the access trace. For every combination of B (state 1) and S (state 2), stack processing determines the conflicts $\{SConfl\}$ in the stack addresses before $Addr^B$ (state 3). If $Addr^B$ is not present in the stack, *Addr* is a compulsory miss (state 5) and L2 analysis for *Addr* is not necessary. Next, the stack update occurs (state 6), and stack processing begins for the next *Addr*. If $Addr^B$ is present in the stack, the number of conflicts $|SConfl|$ dictates the minimum set associativity that yields a hit, thus a hit or miss for each set associativity *SWay* (state 4) can be determined (e.g., $SWay > |SConfl|$ is a hit). Since L1 hits do not require any L2 analysis, an L1 hit ends *Addr*'s processing. After the stack update (state 6), stack processing proceeds to the next *Addr*. If there is an L1 miss, the next subsection describes the supplementary processing for L2 analysis.

B. Stack-based Two-Level Cache Analysis

When using an exclusive hierarchy, L1 and L2 can be treated as one combined cache. The stack processing in Section IV.A produces conflicts for this combined cache for L1 and L2 simultaneously, but to differentiate distinct L1 and L2 conflicts with respect to access order, conflicts are recorded in MRU (most recently used) time order for the L1 conflicts $\{SConfl^1\}$ and the L2 conflicts $\{SConfl^2\}$. Since $\{SConfl^2\}$ contains inclusive L2 conflicts, exclusion requires the removal of the L1 conflicts from $\{SConfl^2\}$ to isolate the exclusive L2 conflicts $\{L2Confl\}$.

If *Addr* results in an L1 conflict miss, the first $SWay^1$ conflicts in $\{SConfl^1\}$ (denoted by $\{SConfl^1\}_{SWay1}$) are present in L1. Stack processing determines the L2 conflicts $\{SConfl^2\}$ for the combined cache with the same block size B as L1/L2 and number of sets S^2. Since $\{SConfl^2\}$ represents conflicts in the combined cache, the conflicts for just L2 $\{L2Confl\}$ are the conflicts remaining after removing $\{SConfl^1\}_{SWay1}$ (effectively removing the L1 conflicts) from $\{SConfl^2\}$. We refer to this supplementary process as the *compare-exclude* operation. The number of conflicts $|L2Confl|$ determines the minimum L2 associativity necessary for *Addr* to be an L2 hit.

TABLE 1: Notational reference

$>>$	Bitwise right shift operator
$B = 2^b$	B = cache block size
S	Number of sets
$SWay$	Number of ways corresponding to S
C	Total cache size. $C = B * S * SWay$
$X_{\min/\max}$	Subscript min/max represents minimum/maximum value of X (X can be B, S, $SWay$ or C).
X^i	Superscript i can be 1 or 2 for L1 or L2, respectively.
$Addr$	Address currently being processed
$Addr^B$	Previous access to *Addr's* cache block (i.e., $(Addr >> b) = (Addr^B >> b)$
$SConfl$	Conflicts under S
$SCompl$	Conflicts associated with all complementary sets
$L2Confl$	Conflicts present in L2
$\{Y\}$	Collection of Y (Y can be $SConfl$, $SCompl$, or $L2Confl$), listing elements in MRU (most recently used) order.
$\|Y\|$	Cardinality of collection $\{Y\}$

Fig 3 depicts the address partitioning for the three possible compare-exclude scenarios: (a) the number of L1 and L2 sets are equal ($S^1 = S^2$ and $\{SConfl^1\}_{SWay1}$ is equal to the first $SWay^1$ elements in $\{SConfl^2\}$), (b) the number of L1 sets is less than the number of L2 sets ($S^1 < S^2$ and $\{SConfl^1\}_{SWay1}$ contains the first few elements in $\{SConfl^2\}$), and (c) the number of L2 sets is less than the number of L1 sets ($S^1 > S^2$ and $\{SConfl^1\}_{SWay1}$ is a subset of $\{SConfl^2\}$). The following subsections detail these three scenarios.

a. Compare-Exclude Scenario 1: $S^1 = S^2$

For the same B and S values in L1 and L2, *Addr's* conflicts are divided into two categories: the first $SWay^1$ conflicts are present in L1 and the remaining conflicts form $\{L2Confl\}$. If $SWay^2 > |L2Confl|$, $Addr^B$ has not been evicted from L2, and *Addr* results in an L2 hit. For example, if the number of *Addr's* conflicts is 5 and $SWay^1 = 2$, the first two conflicts are present in L1 and the remaining conflicts compose $\{L2Confl\}$. Therefore, $|L2Confl| = 3$ and L2 configurations with associativities greater than 3 result in an L2 hit.

b. Compare-Exclude Scenario 2: $S^1 < S^2$

As depicted in Fig 3 (b), the number of L1 index bits k is less than the number of L2 index bits l and the L1/L2 address's least significant k index bits are equal and the L2 address's highest significant $(l-k)$ index bits are arbitrary binary values. Essentially, the evicted cache blocks from one L1 set will be moved into multiple L2 sets. Therefore, some $\{SConfl^2\}$ conflicts are still present in L1 and these conflicts are the intersection of $\{SConfl^1\}_{SWay1}$ and $\{SConfl^2\}$. After removing these intersecting conflicts, the remaining conflicts in $\{SConfl^2\}$ are the L2 conflicts $\{L2Confl\}$ (i.e., $\{L2Confl\} = \{SConfl^2\} - [\{SConfl^1\}_{SWay1} \cap \{SConfl^2\}]$).

However, Fig 4 illustrates a special case that must be considered in this scenario. Fig 4 (a) shows the time ordered access trace where time t_1, t_2, t_3, and t_4 represent four

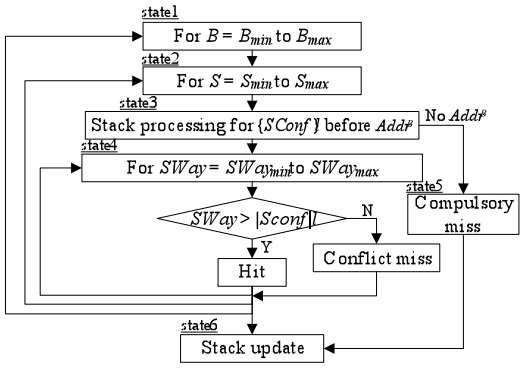

Fig 2: Flow chart for single-level cache simulation

evaluation points. We assume the following: all addresses (represented by *Z*, *X4*, *X3*, *X2*, *X1*, *y1*, *y2*, *X2,* and *Z*) are in different cache blocks; *Z*, *X4*, *X3*, *X2,* and *X1* map to the same cache set under both S^1 and S^2; and *Z*, *y1,* and *y2* map to the same cache set under S^1 but not S^2. For $SWay^1 = 2$ and $SWay^2 = 4$, Fig 4 (b) shows the L1 and L2 cache set contents at t_1, t_2, and t_3 and Fig 4 (c) shows the stack contents at t_4. From the cache set contents, accessing *Z* at t_4 results in an L1 and an L2 miss. Stack processing for *Z* determines the conflicts $\{SConfl^1\}_{SWay1} = \{X2, y2\}$ and $\{SConfl^2\} = \{X2, X1, X3, X4\}$. Thus the compare-exclude operation produces the conflicts $\{L2Confl\} = \{X1, X3, X4\}$. Since $|L2Confl| = 3$ and $SWay^2 = 4$, *Z* is incorrectly classified as a hit.

To explain this incorrect classification, we note that accessing *X2* at t_3 moves *X2* from L2 to L1, leaving an empty way in L2 – an *occupied blank* (*BLK*). The occupied blank occurs because at t_3, *y1* was evicted from L1 to accommodate *X2*, but *y1* maps to a different L2 set than *X2*. The occupied blank means that *X2* was in L2 and caused *Z* to be evicted from L2 (at t_2), thus *X2* should be counted as a conflict in $\{L2Confl\}$.

To account for the occupied blank in an L2 hit, *occupied blank labeling*, as a supplemental process, is applied to label occupied blanks using a bit-array (whose size is the number of cache configurations) associated with each stack address. A set bit indicates that an occupied blank follows that address in the corresponding cache configuration. Stack processing evaluates the blank labels while processing *Addr* in L2 analysis. If the label associated with the last conflict in $\{L2Confl\}$ is set (i.e., the occupied blank behind the last conflict means *Addr*'s block has already been evicted from L2) then *Addr* results in an L2 miss regardless of the condition that $|L2Confl| < SWay^2$.

c. Compare-Exclude Scenario 3: $S^1 > S^2$

In this scenario, blocks evicted from multiple L1 sets will map to the same L2 set. We refer to these multiple L1 sets, excluding the set that *Addr* maps to, as the *complementary* sets and $\{SCompl\}$ denotes the collection of blocks in all complementary sets. In Fig 3 (c), *Addr's* index has *k* bits for S^1 and *l* bits for S^2. The complementary set's indexes can be determined 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 *Addr's* S^1 index. For example, if *Addr's* index is "101101" for S^1 and "1101" for S^2, then $\{SCompl\}$ will include all conflicts associated with sets {"001101", "011101", "111101"}.

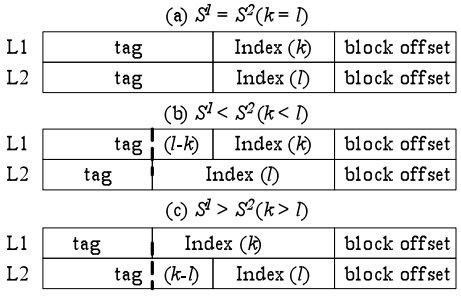

Fig 3: Cache addressing scenarios for a two level cache where *k* and *l* represent the number of L1 and L2 index bits, respectively.

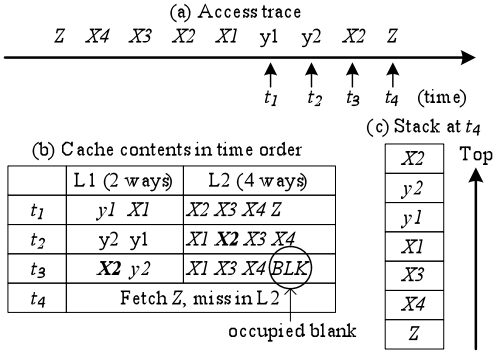

Fig 4: Special case when $S^1 < S^2$ and fetching *X2* from L2 results in an occupied blank (BLK).

Therefore, the L1 conflicts included in $\{SConfl^2\}$ contain the L1 conficts in the set that *Addr* maps to ($\{SConfl^1\}_{SWay1}$) and the L1 conflicts associated with $\{SCompl\}$ (the number of L1 conflicts in each complementary set is limited by *SWay1*). Stack processing can determine these additional conflicts by simply considering the complementary set's indexes. Therefore, the compare-exclude operation in this scenario produces: $\{L2Confl\} = \{SConfl^2\} - \{SConfl^1\}_{SWay1} - \{SCompl\}$.

C. Acceleration Strategies

In single-pass cache evaluation, stack processing is the most time consuming operation. We leverage the *set refinement property* [9] to accelerate stack processing by processing the number of sets *S* from smallest to largest. In this manner, a stack address only needs to be evaluated for conflicts with *Addr* if *Addr* conflicts with that stack address for a smaller *S*. We leverage this acceleration when determining all conflicts $\{SConfl^1\}$, $\{SConfl^2\}$, and $\{SCompl\}$ for all *S* using a tree data structure.

When processing *Addr* for an arbitrary *B*, the conflicts in $\{SConfl^1\}$ for one L1 configuration will be compared with the conflicts in $\{SConfl^2\}$ for all L2 configurations when there is a L1 miss. An efficient method to determine these conflicts would be to determine the conflicts for all possible *S* initially and store the conflicts in a tree structure[1] for later reference.

The tree structure stores *Addr*'s conflicts for every *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). Therefore, a stack address can only be a conflict for *S* if the stack address is a conflict in the next higher level. For S^1 larger than S^2_{min}, the additional conflicts in the compliment sets $\{SCompl\}$ must also be searched and recorded (Section IV.B.c). Since conflicts with the same index in $\{SCompl\}$ are recorded using one node in the same level as S^1, the number of nodes at each level is dictated by the number of compliment sets required for that *S*. Nodes store conflict information and the maximum L1/L2 associativity dictates the maximum number of conflicts required at each node.

The tree assisted acceleration algorithm for each *Addr* and an arbitrary *B* can be summarized in four steps. Step 1) Clear the tree contents and set $S_{start} = S_{min}$. Step 2) Begin stack

[1] This data structure is not a traditional tree structure, but is instead a hierarchical representation that we refer to as a tree for simplicity.

processing for *Addr* from the S_{start} level. For each stack address *A* that conflicts with *Addr* or is a conflict associated with a complement set (when additional compliment set conflicts are required) at an arbitrary level *S*, record this conflict and continue to evaluate for conflicts in the next lower level until S_{max} level, then proceed to the next *A*. If *A* does not conflict with *Addr* at an arbitrary level *S*, proceed to the next *A* directly without evaluating for conflicts in lower levels. Step 3) If all nodes at the S_{start} level are full, update $S_{start} = S_{start} * 2$. Step 4) Stack processing ends for *Addr* if either $Addr^B$ is found or all nodes in the tree are full.

Since only one tree is required (the contents are cleared for each *Addr* processing under each *B*), the storage space for the tree is minimal as compared to the stack structure.

V. Experimental Results and Analysis

We verified T-SPaCS using the EEMBC [3], Powerstone [12], and MediaBench benchmark suites [11] (benchmarks were arbitrarily selected from each suite). We gathered the access traces using 'sim-fast' in SimpleScalar 3.0d [14]. For comparison, we modified 'sim-cache' to simulate an exclusive hierarchy to produce the *exact* miss rates. The design space (modeled after [7]) 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/L2 associativities from direct-mapped to 4-way, and the cache block size from 16 to 64 bytes. We point out that T-SPaCS is not limited to this design space, and is 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 SimpleScalar 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.

A. Miss Rate Accuracy

T-SPaCS's L1 miss rates as compared to the exact miss rates were 100% accurate and the L2 miss rates were 100% accurate for 240 configurations (99% of the design space). Across all 24 benchmarks, the maximum values of average miss rate error, standard deviation, and maximum absolute miss rate error for the three inaccurate configurations were 1.16%, 0.64%, and 1.55%, respectively.

The three inaccurate configurations had $S^1 > S^2$ (Section IV.B.c). In this scenario, the eviction order of blocks from different L1 sets to the same L2 set does not follow the memory access order. Only the blocks that are moved into L2 after $Addr^B$ affect $Addr^B$'s eviction from L2. Since the stack structure only records the latest memory access order, the eviction order of the blocks from multiple L1 sets to the same L2 set cannot be recorded. Therefore, the blocks in {*L2Confl*} generated by the compare-exclude operation are not guaranteed to be the blocks present in L2. However, inaccurate |*L2Confl*| does not necessarily produce an incorrect cache hit/miss determination since a cache miss is determined when $|L2Confl| >= SWay^2$. If the inaccurate |*L2Confl*|'s error is larger than the difference between $SWay^2$ and the accurate |*L2Confl*|, the cache hit/miss determination will alter. Our experimental results showed that the effect of introduced errors in |*L2Confl*| on miss rate estimation was nominal.

B. Optimal Cache Configuration

We expanded the inclusive two-level cache hierarchy energy model [7] (see reference for details) to include evicted block write energy. In the calculation of both dynamic and static energy consumption, we obtained dynamic cache and memory fetch energy using CACTI 6.5 [2] for 0.09-micron technology, CPU stall energy from a 0.09-micron MIPS microprocessor, and assumed cache static energy consumption accounted for 10% of the total cache energy [7]. We estimated bandwidth and latency based on a reasonable system architecture: an L2 fetch is four times longer than an L1 fetch; a main memory fetch is ten times longer than an L2 fetch; and the memory throughput is 50% of the latency [7].

We applied this energy model to both T-SPaCS's and the exact miss rates and observed that the optimal energy configurations were identical, even with the three inaccurate configuration miss rates.

C. 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 SimpleScalar. The simulation times were measured on a Linux workstation with a 2.66 GHz processor and 4 gigabytes of RAM using the user time reported by the time command.

Fig 5 shows the simulation speedup obtained by T-SPaCS for each benchmark (first bar). T-SPaCS achieved maximum and average speedups of 17.96X and 8.02X, respectively.

Since one of T-SPaCS's most time consuming operations is occupied blank labeling (Section IV.B.b), we removed the occupied blank labeling operation in a simplified version of T-SPaCS (simplified-T-SPaCS). Fig 5 reveals that simplified-T-SPaCS's maximum and average speedups were increased to 24.69X and 15.48X, respectively.

The tradeoff for increased simulation speedup was L2 miss rate errors for an additional 228 configurations where $S^1 < S^2$. Across all 24 benchmarks, the maximum values of average miss rate error, standard deviation, and maximum absolute miss rate error for the 228 inaccurate configurations were 0.71%, 0.90%, and 3.35%, respectively. However, even with this error, simplified-T-SPaCS produced identical optimal energy configurations as the exact miss rates.

Therefore, simplified-T-SPaCS is an ideal choice for cache

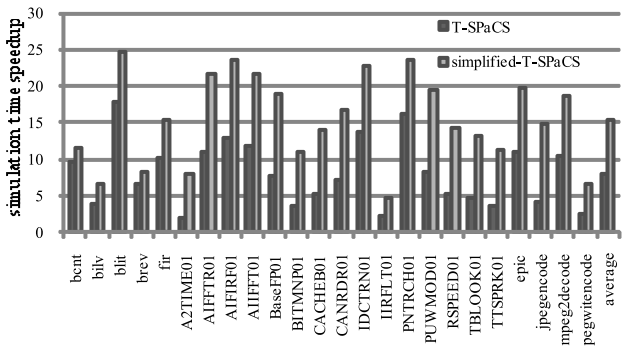

Fig 5: Simulation time speedup of T-SPaCS and simplified-T-SPaCS compared to SimpleScalar

tuning due to simplified-T-SPaCS's competitively fast simulation time and accurate optimal energy configuration determination. Alternatively, T-SPaCS is suitable to situations that require more accurate cache miss rates (e.g., performance analysis) while still providing simulation speedup.

VI. 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 an exclusive instruction cache hierarchy that uses 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 through sequential simulation. On average, T-SPaCS is 8.02X faster than sequential simulation and produces 100% accurate miss rates for 99% of the design space. A simplified version of T-SPaCS (simplified-T-SPaCS) increases average simulation speedup to 15.48X at the expense of inaccurate miss rates for 95% of the design space. However, even with these miss rate errors (maximum of only 3.35%), both T-SPaCS and simplified-T-SPaCS determined accurate optimal energy configurations, thereby facilitating rapid design space exploration for cache tuning. Our future work includes extending T-SPaCS to data and unified cache simulation and generalizing to an N-level cache.

Acknowledgements

This work was supported by the National Science Foundation (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 National Science Foundation.

References

[1] Arc International, http://www.arccores.com.

[2] CACTI, http://www.hpl.hp.com/research/cacti/.

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

[4] 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.

[5] A. Gordon-Ross and F. Vahid, "A self-tuning configurable cache," IEEE Design Automation Conference, 2007.

[6] A. Gordon-Ross, J. Lau, and B. Calder, "Phase-Based cache reconfiguration for highly-configurable two-level cache hierarchy," ACM Great Lakes Symposium on VLSI, 2008.

[7] 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.

[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, Automation and Test in Europe (DATE), Apr. 2007.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] SimpleScalar LLC, http://www.simplescalar.com/.

[15] SimPoint, http://cseweb.ucsd.edu/~calder/simpoint/.

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

[17] 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.

[18] 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.

[19] C. Zhang, F. Vahid and R. Lysecky, "A self-tuning cache architecture for embedded systems," ACM Trans. on Embedded Comput. Systems, Vol. 3, pp. 407-425, 2004.