A Table-based Method for Single-Pass Cache Optimization

Pablo Viana
Federal University of Alagoas
Arapiraca-AL, Brazil
pablo@lccv.ufal.br

Ann Gordon-Ross
University of Florida
Gainesville-FL, USA
ann@ece.ufl.edu

Edna Barros
Federal University of Pernambuco
Recife-PE, Brazil
ensb@cin.ufpe.br

Frank Vahid
University of California, Riverside
Riverside-CA, USA
vahid@cs.ucr.edu

ABSTRACT

Due to the large contribution of the memory subsystem to total system power, the memory subsystem is highly amenable to customization for reduced power/energy and/or improved performance. Cache parameters such as total size, line size, and associativity can be specialized to the needs of an application for system optimization. In order to determine the best values for cache parameters, most methodologies utilize repetitious application execution to individually analyze each configuration explored. In this paper we propose a simplified yet efficient technique to accurately estimate the miss rate of many different cache configurations in just one single-pass of execution. The approach utilizes simple data structures in the form of a multi-layered table and elementary bitwise operations to capture the locality characteristics of an application's addressing behavior. The proposed technique intends to ease miss rate estimation and reduce cache exploration time.

Categories and Subject Descriptors

B.3 [**Memory Structures**]: Performance Analysis and Design Aids

General Terms

Algorithms

Keywords

Configurable cache tuning, cache optimization, low energy.

1. INTRODUCTION

Optimization of system performance and power/energy consumption is an important step during system design and is accomplished through specialization, or tuning, of the system. Tunable parameters include supply voltage, clock speed, bus width and encoding schemes, etc. Of the many tunable parameters, it is well known that one of the main bottlenecks for system efficiency resides in the memory sub-system (all levels of cache, main memory, buses, etc) [16]. The memory subsystem can attribute to as much as 50% of total system power [1, 17].

Memory subsystem parameters such as total size, line size, and associativity can be tuned to an application's temporal and spatial locality to determine the best cache configuration to meet optimization goals [3]. However, the effectiveness of such tuning is dependent on the ability to determine the best cache configuration to complement an application's memory addressing behavior.

To determine a cache size that yields good performance and low energy for an application, the size must closely reflect the temporal locality needs of an application. It is important to determine how frequently memory addresses are accessed and how long it takes for an executing application to access the same memory reference again. This property is mostly attributed to working-set characteristics such as loop size.

Similarly, the cache line size must closely reflect the spatial locality of an application, which is present in straight-line instruction code and data array accesses. Additionally, associativity must closely reflect the needs of the application.

To determine the best values for these tunable parameters, or best cache configuration, existing cache evaluation techniques include analytical modeling [6, 10] and execution-based evaluation [4] to evaluate the design space. Analytical models evaluate code characteristics and designer annotations to predict an appropriate cache configuration in a very short amount of time, requiring little designer effort. Whereas this method can be accurate, it can be difficult to predict how an application will respond to real-world input stimuli.

A more precise technique is execution-based evaluation. In this technique, an application is typically simulated multiple times, and through the use of a cache simulator, application performance and/or energy are evaluated for each cache configuration explored. Whereas this technique is more accurate than an analytical model, modern embedded systems are becoming more and more complex and simulating these applications for numerous cache configurations can demand a large amount of design time. To accelerate execution-based evaluation, specialized caches have been designed that allow for cache parameters to be varied during runtime [2, 14, 19]. However, due to the intrusive nature of the exploration heuristics, the cache must be physically changed to explore each configuration.

Exploring a large number of cache configurations can potentially significantly adversely effect program execution in terms of energy and performance overhead while exploring poor configurations. To reduce the number of configurations explored, efficient heuristics have been proposed [8, 19] to systematically traverse the configuration space and result in a near-optimal cache configuration while evaluating only a fraction of the design space. However, even though the number of cache configurations is greatly reduced, in some systems, tens of cache configurations may need to be explored, thus still potentially imposing a large overhead and consuming too much exploration time. Exploration time must be quick enough to adapt to rapidly changing resource requirements [7].

Instead of changing the cache configuration numerous times to evaluate each different cache configuration, much information about the memory addressing behavior could be extracted from a single execution of an application independent of the cache configuration. In multi-cache evaluation, multiple cache configurations are evaluated in a single pass of execution [18]. For example, the number of neighboring addresses that are accessed in a short period of time or how often a given address is repeatedly referenced can suggest an application's temporal and spatial locality requirements. If such properties about the spatial and temporal locality of the application are extracted and well correlated in an organized structure, the behavior of many cache configurations can be estimated and appropriate cache parameters can be projected.

In this work, we present a simplified, yet efficient way to extract locality properties for an entire cache configuration design space in just one single-pass. SPCE (pronounced spee-cee), our single-pass multi-cache evaluation methodology, utilizes small, compact table structures and elementary bitwise operations consisting of comparisons and shifting to allow us to estimate the cache miss rate for all configurations simultaneously. SPCE provides design time acceleration in a simulation-based environment, but most importantly, we design SPCE with a hardware implementation in mind, providing an important non-intrusive cache exploration alternative for quick runtime exploration. In this paper, we present a detailed algorithm for SPCE's operation and evaluate SPCE in a simulation-based environment compared to a state-of-the-art cache tuning heuristic. In [9], we provide a hardware implementation of SPCE and quantify the importance of such a runtime tuning environment.

This paper is organized as follows: Section 2 discusses related work pertaining to single-pass cache evaluation. Section 3 describes an overview of SPCE. In the Section 4, we present elementary properties for addressing behavior analysis to estimate the cache miss rate of fully-associative caches. Section 5 extends those properties to analyze address conflicts and introduces more advanced concepts to build a multi-layered table for multi-cache evaluation of direct-map caches as well as set-associative caches. Section 6 presents the SPCE algorithm and discusses its implementation details. In Section 7 we validate SPCE with experimental results and finally in Section 8, we conclude the paper and outline future directions for SPCE to a broader domain.

2. RELATED WORK

Much research exists in the area of multi-cache evaluation, however, nearly all existing techniques require multiple passes to explore all configurable parameters or employ large and complex data structures that are not amenable to hardware implementation, thus restricting their applicability to strictly a simulation-based evaluation environment.

Algorithms for single-pass cache simulation tackle the problem of multi-cache evaluation by examining concurrently a set of caches with different sizes during the same execution pass. Research on this issue began in 1970 when Mattson et al. presented an algorithm for simulating fully-associative caches with varying sizes and a fixed block size [15]. The algorithm utilized stack-based simulation and took advantage of the inclusion property.

The inclusion property states that at any time, the contents of a cache are a subset of the contents of a larger cache. Hill and Smith [12] identified the set-refinement property, which extends the inclusion property to study the inclusion effects of cache associativity, and extended the inclusion property for direct-mapped and set-associative caches. Sugumar and Abraham [18] developed algorithms using binomial trees resulting in single-pass schemes 5 times faster than previous approaches. They also proposed another single-pass algorithm to simulate caches with varying block sizes. In [5], Cascaval and Padua proposed a method to estimate cache misses at compile time using a machine independent model based on a stack algorithm.

Most of the existing methods perform very well for a given set of caches with a fixed line size or a fixed total size. Thus, these algorithms can be utilized in simulation-based cache tuning, reducing the number of necessary simulation passes. However, since these methods are unable to evaluate all different cache parameters simultaneously, multiple simulation passes are still required, thus in a runtime tuning environment, cache exploration could be too lengthy. In [13], Janapsatya et al. present a technique to evaluate all different cache parameters simultaneously and, to the best of our knowledge, is the only such technique. They present a tree-based structure consisting of multiple linked lists to keep track of cache statistics for a large design space. Whereas this work most closely resembles our methodology and shows tremendous speedups in simulation time, their methodology was not designed with a hardware implementation in mind. Our methodology utilizes simple array structures, structures that are more amenable to a light-weight hardware implementation.

3. SPCE OVERVIEW

SPCE is a single-pass multi-cache evaluation technique to evaluate all values for all cache parameters (total size, line size and associativity) simultaneously, requiring only one simulation pass. Whereas previous single-pass cache evaluation techniques utilize complex data structures to estimate cache miss values, SPCE utilizes simple table structures and elementary bitwise operations consisting of comparisons and shifting.

Figure 1 illustrates an overview of SPCE's cache evaluation methodology. The running application a_i produces a sequence of instruction addresses T, which can be captured independently of the cache configuration, by using an instruction-set architecture simulator or executable platform model [4]. In this work, we utilize a processor simulator to generate an address trace for post execution processing, but we point out that, given the sheer size of typical address traces, address trace generation may be omitted and instruction addresses may be trapped during simulation and fed to SPCE in parallel.

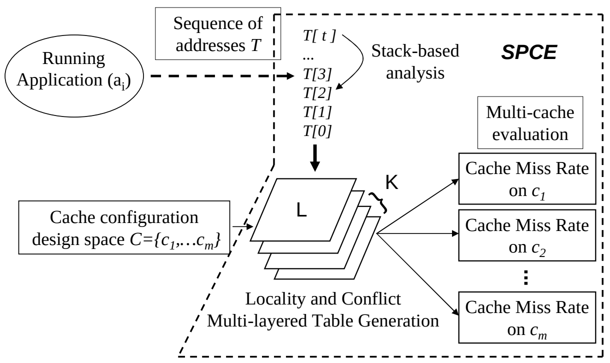
Running Application (a_i)
Sequence of addresses T
$T[t]$
...
$T[3]$
$T[2]$
$T[1]$
$T[0]$
Stack-based analysis
SPCE
Multi-cache evaluation
K
L
Cache configuration design space $C=\{c_1,\ldots c_m\}$
Cache Miss Rate on c_1
Cache Miss Rate on c_2
Cache Miss Rate on c_m
Locality and Conflict Multi-layered Table Generation

Figure 1: SPCE overview: multi-layered table generation for multi-cache evaluation.

SPCE processes the sequence of addresses and analyzes the entire cache configuration design space C consisting of m different configurations. Each configuration is a unique combination of values for line size, associativity, and total size in the design space. SPCE uses a stack structure to store previous addresses for cache hit evaluation. When an address is processed, SPCE scans the stack to determine if the current address has been fetched previously and would perhaps result in a cache hit depending on the cache configuration. After SPCE processes an address, if the address was already present in the stack, the address is removed and pushed onto the top of the stack. If the address was not present in the stack, it is simply pushed onto the top of the stack.

We present details of data generation in sections 4 and 5. The resulting locality information for line size and total cache size analysis populates the table L, while conflict data for associativity analysis populates the multi-layered table K, where each layer represents the associativity levels explored.

After processing all instruction addresses, the cells of the tables L and K store the number of cache hits for the sequence of addresses analyzed for all cache configurations in the space C. By summing up the contents of the tables and subtracting that value from the size of the address trace, it is easy to evaluate cache miss rates for every cache configuration. The miss rates can then be supplied to an energy model for the system to calculate energy consumption for each cache configuration.

4. LOCALITY ANALYSIS

4.1 Definitions

While application a_i executes, the memory hierarchy fulfills successive instruction reference requests. We define the time ordered sequence of referenced addresses by the vector $\vec{T}[t]$, $t \in \mathbb{Z}^+$(t is a positive integer), of length $|\vec{T}|$, such that $\vec{T}[t]$ is the t^{th} address referenced [11].

In a traditional cache mapping, two addresses $\vec{T}[t_i]$ and $\vec{T}[t_i+d]$ belong to the same cache block if, and only if $\frac{\vec{T}[t_i]}{2^b} = \frac{\vec{T}[t_i+d]}{2^b}$, where $\vec{T}[t_i + d]$ is the d^{th} address referenced after $\vec{T}[t_i]$ and 2^b is the cache block size in number of words (words are defined in Bytes).

Since the block size is the number of words, usually a power of 2, it is reasonable to represent it by the power 2^b. We define the operator $\triangleright$ as the bitwise shift operation $\vec{T}[t_i] \triangleright b$ where the address $\vec{T}[t_i]$ is shifted to the right b times. Shifting $\vec{T}[t_i]$ to the right b times is equivalent to dividing $\vec{T}[t_i]$ by 2^b (Equation 1).

$$\vec{T}[t_i] \triangleright b = \frac{\vec{T}[t_i]}{2^b} \qquad (1)$$

Thus, if $\vec{T}[t_i] \triangleright b = \vec{T}[t_i + d] \triangleright b$, then the addresses $\vec{T}[t_i]$ and $\vec{T}[t_i + d]$, are references to the same cache block of 2^b words. Note that for the particular case where $b = 0$, $\vec{T}[t_i]$ and $\vec{T}[t_i + d]$ would correspond to exactly the same address. By considering the probability of frequent accesses to the same block during the execution of a given application, *spatial* and *temporal* locality can be directly correlated as a function of b and d.

We evaluate the locality in the sequence of addresses $\vec{T}[t_i]$ of a running application a_i by counting the occurrences where $\vec{T}[t_i] \triangleright b = \vec{T}[t_i + d] \triangleright b$ and registering it in the cell $L(b, d)$ of the *locality table*. The number of rows and columns of the locality table are defined by the boundaries $1 \leq d \leq dmax$ and $0 \leq b \leq bmax$ respectively (Table 1) where $dmax$ is defined as the total number of available lines in the largest cache of the configuration space, and $bmax$ defines the total number of line sizes available.

The locality table $L(b, d)$ represents a structured abstraction of a sequence of addresses $\vec{T}$ and can be built by evaluating the variables b and d when $\vec{T}[t_i] \triangleright b = \vec{T}[t_i + d] \triangleright b$, as the application runs. It is important to note that when evaluating $\vec{T}[t_i] \triangleright b = \vec{T}[t_i + d] \triangleright b$, it is not sufficient to simply count the number of references d which occur between $\vec{T}[t_i]$ and $\vec{T}[t_i + d]$.

If any reference $\vec{T}[t_i + j]$ for $1 \leq j < d$ results in a cache hit, that reference would not displace any cache line. Thus, in this situation, $\vec{T}[t_i] \triangleright b = \vec{T}[t_i + d + 1] \triangleright b$ would result in a cache hit as well. As this point we redefine d as the *delay* or the number of unique cache references occurring between any two references where $\vec{T}[t_i] \triangleright b = \vec{T}[t_i + d] \triangleright b$.

Table 1: Locality table $L(b, d)$

$L(0,1)$	$L(1,1)$	$L(2,1)$	...	$L(bmax,1)$
$L(0,2)$	$L(1,2)$	$L(2,2)$	...	$L(bmax,2)$
$L(0,3)$	$L(1,3)$	$L(2,3)$	...	$L(bmax,3)$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$L(0,dmax)$	$L(1,dmax)$	$L(2,dmax)$	...	$L(bmax,dmax)$

4.2 Fully-associative Cache Miss Rate

In a *fully-associative* cache, references can be mapped to any cache line. For this reason, a cache with n lines, using an optimal (OPT) or near optimal (LRU) replacement policy stores any address reference $\vec{T}[t_i]$ until address $\vec{T}[t_i + n]$ is referenced, where n represents the number of cache misses occurring between reference $\vec{T}[t_i]$ and reference $\vec{T}[t_i + n]$.

A fully-associative cache configuration is defined by the notation $c_j(b, n)$, where b defines the line size in terms of words, and n the total number of lines in the cache.

We know that $L(b, d)$ gives us the number of occurrences $\vec{T}[t_i] \triangleright b = \vec{T}[t_i + d] \triangleright b$ in a sequence of addresses $\vec{T}$. Thus, $L(b, d)$ indicates how often the same block is regularly accessed after d references to other addresses.

We conclude that a fully associative cache configuration $c_j(b, n)$, composed of n lines with 2^b words per line, is able to hold any reference $\vec{T}[t_i]$ as long as it is repeatedly accessed, yielding $L(b, d)$ cache hits, $\forall d \leq n$. (Equation 2).

$$Cache\_hit[c_j(b, n)] = \sum_{d=1}^{n} L(b, d) \qquad (2)$$

We can estimate the cache miss rate of a given cache configuration $c_j(b, n)$ for a sequence $\vec{T}$ by subtracting the result of $Cache\_hit$ from the total size of the sequence of addresses $|\vec{T}|$ (Equation 3):

$$Miss\_rate[c_j(b, n)] = \frac{|\vec{T}| - Cache\_hit[c_j(b, n)]}{|\vec{T}|} \qquad (3)$$

Any fully-associative cache configuration $c_j(b, n)$ within the design space defined by the boundaries $0 \leq b \leq bmax$ and $1 \leq d \leq dmax$ can be estimated by using the Locality Table L. Thus, just one single-pass simulation of the application (or trace $\vec{T}$) is necessary to generate the entire contents of L.

5. DIRECT-MAPPED AND SET-ASSOCIATIVE CACHES

5.1 Multi-layered Conflict Table

The locality table $L(b, d)$ presented in Section 4 composes an efficient way to estimate the cache miss rate of fully-associative caches. However, due to the high cost in terms of area and energy consumption for fully-associative caches, most real-world cache devices are built as direct-map or set-associative structures.

For these cache structures, the locality table $L(b, d)$ can not be used to estimate cache misses, since mapping conflicts for addresses that map to the same cache line are not considered. Miss rate estimation for such cache structures must take into account the evaluation of address mapping conflicts.

Here, we define s as the number of sets independent of the associativity. A direct-mapped cache can be considered a particular case of a set-associative cache (set size is 1 line). In this case, the number of sets is equal to the number of cache lines ($s = n$).

Two distinct addresses $\vec{T}[t_i]$ and $\vec{T}[t_i+d]$ suffer from an address mapping conflict if $\vec{T}[t_i] \triangleright b$ divided by s and $\vec{T}[t_i + d] \triangleright b$ divided by s results in the same remainder (Equation 4).

$$(\vec{T}[t_i] \triangleright b) \bmod s = (\vec{T}[t_i + d] \triangleright b) \bmod s \Longrightarrow Conflict \quad (4)$$

In fully-associative caches ($s = 1$), the probability of a given reference ($\vec{T}[t_i] \triangleright b$) being present in the cache is dependent on the number of cache lines and how long it takes until the same block is referenced again ($d \leq n$).

In direct-mapped and set-associative caches, the probability of a given reference being present in the cache depends on the number of cache sets (s) and how many cache conflicts occur before the same block is referenced again. If the level of associativity is higher than the number of conflicts, the reference will still be in the cache (Hit). For the analysis of cache conflicts, we propose the set of table layers K_α, denoted as a *conflict table*, which is composed of α layers, one for each associativity explored, as illustrated in Table 2.

Table 2: Conflict table $K_\alpha(b, s)$ for $\alpha = 1$ and 2

$K_1(0,1)$	$K_1(1,1)$	$K_1(2,1)$	...	$K_1(bmax,1)$
$K_1(0,2)$	$K_1(1,2)$	$K_1(2,2)$	...	$K_1(bmax,2)$
$K_1(0,4)$	$K_1(1,4)$	$K_1(2,4)$	...	$K_1(bmax,4)$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$K_1(0,smax)$	$K_1(1,smax)$	$K_1(2,smax)$	...	$K_1(bmax,smax)$

$K_2(0,1)$	$K_2(1,1)$	$K_2(2,1)$	...	$K_2(bmax,1)$
$K_2(0,2)$	$K_2(1,2)$	$K_2(2,2)$	...	$K_2(bmax,2)$
$K_2(0,4)$	$K_2(1,4)$	$K_2(2,4)$	...	$K_2(bmax,4)$
$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$K_2(0,smax)$	$K_2(1,smax)$	$K_2(2,smax)$	...	$K_2(bmax,smax)$

SPCE builds the conflict table K_α for a running application by analyzing the sequence of addresses $\vec{T}[t]$ and counting the number of occurrences (mapping conflicts) of distinct address blocks mapped to the same cache line. The number of lines in each layer α of the conflict table is defined by the maximum number of sets $s = smax$. For any cache configuration (b, s), Equation 4 computes the number of mapping conflicts (x) between $\vec{T}[t_i] \triangleright b$ and $\vec{T}[t_i+d] \triangleright b$, which defines the appropriate layer α to fill in $K_\alpha(b, s)$. The layer α is determined by rounding up the value of $x+1$ to the next power of 2 (Equation 5).

$$\alpha = 2^{\lceil \log_2(x+1) \rceil} \quad (5)$$

5.2 Miss Rate Estimation

We designed the conflict table $K_\alpha(b, s)$ in a such way that layer α refers to the associativity of a given cache configuration. Layer $\alpha = 1$, for example, characterizes the addressing behavior of a direct-map cache, layer $\alpha = 2$ characterizes a two-way set-associative cache, layer $\alpha = 4$ characterizes a four-way cache, and so on.

At the end of simulation, the value stored in each element of the table $K_\alpha(b, s)$ indicates how many times the same block (size 2^b) is repeatedly referenced and results in a hit. Depending on the cache mapping, which is defined by the number of sets (s), the lowest associativity level which guarantees $K_\alpha(b, s)$ cache hits is given by α.

A given cache configuration with level of associativity w is capable of overcoming no more than $w - 1$ mapping conflicts. Thus, the number of cache hits is determined by summing up the cache hits from layer $\alpha = 1$ up to its respective layer $\alpha = w$, where w refers to the associativity. From now on, each cache configuration will be defined by $c_j(w, b, s)$ (Equation 6).

$$Cache\_hit[c_j(w, b, s)] = \sum_{\alpha=1}^{w} K_\alpha(b, s) \quad (6)$$

6. ALGORITHM IMPLEMENTATION

Figure 3 shows the SPCE algorithm to build the multi-layered conflict table. A stack keeps track of the sequence of previously accessed addresses and is repeatedly scanned to evaluate which configurations would result in that access being a "cache hit."

```
process(ADDR)
{
  ADDR = ADDR >> W               //shift out word offset
  for B = BMAX downto BMIN{          //for each line size
     //shift out block offset
     BASE_ADDR = ADDR >> B
     //scan stack
     WAS_FOUND = lookup_stack(BASE_ADDR)
     if(WAS_FOUND){
       for S = SMIN to SMAX{           //for each set size
          //scan stack looking for set conflicts
          NUM_CONFLICTS = count_conflicts(S, BASE_ADDR)
          if(NUM_CONFLICTS <= AMAX){
            //mark the appropriate table level
            ALPHA = roundup(NUM_CONFLICTS)
            update_table(ALPHA, S, B)
          }
       } // end for S = SMIN to SMAX
     }
  } // end for B = BMAX downto BMIN
  //push or move addr to top
  update_stack(WAS_FOUND, ADDR)
}
```

Figure 2: The SPCE algorithm.

When the address is found in the stack, SPCE analyzes the number of set conflicts to determine the minimum set-associativity to yield a hit. The occurrences are registered into the appropriate layer of the conflict table by incrementing the value of its cell. After hit determination, if the address was found in the stack, the address is moved to the top of the stack. Otherwise, the new address is pushed onto the stack. Taking advantage of the inclusion property of a larger cache line size over a smaller size, the outer loop responsible for the line size exploration was intentionally implemented with a decrementing counter to optimize stack scanning.

Pre-defining the bounds of the tables specify the particular design space of cache configurations explored. The number of columns in each table layer is defined by the minimum and maximum number of words per line (2^{bmin} and 2^{bmax}, respectively). The number of lines in the tables are defined by the minimum and maximum number of cache sets ($smin$ and $smax$, respectively) in the cache configuration space. The number of table layers is dependent on the highest associativity level considered ($amax$).

Each cache configuration in the design space is defined by the parameters w, b, and s (associativity, line size, and number of sets, respectively), which determine their corresponding cells in the multi-layered table. The number of misses for any cache configuration can be easily calculated by summing up the table cells for the given configuration to determine the number of cache hits and subtracting the amount from the total number of addresses.

7. EXPERIMENTAL RESULTS

7.1 Setup

We implemented SPCE as a standalone C++ program to process an instruction address trace file. We gathered instruction address traces for 9 arbitrarily chosen applications from Motorola's PowerStone benchmark suite [14] using a version of SimpleScalar modified to output instruction traces. SimpleScalar is a 64-bit architecture, thus each address will be shifted by 8 to remove the word offset.

Although SPCE does not impose any restriction on the parameters of the configurable cache architecture, in order to determine both the accuracy and simulation time speedup compared to a state-of-the-art cache tuning heuristic, we adopted the configurable cache architecture presented by Zhang et al [20]. Zhang's system architecture consists of a cache hierarchy with separate level one instruction and data caches, both connected to the main memory and a cache tuner connected to both caches.

Using specialized configuration registers, each cache offers configurable size, line size, and associativity. To offer configurable size and associativity, each cache is composed of four configurable banks each of which acts as a way in the cache. The base cache is a 4-way set-associative cache. The ways/banks may be selectively disabled or enabled to offer configurable size. Additionally, ways may be logically concatenated to offer direct-mapped, 2-way, and 4-way set-associativities.

Given the bank layout of the cache, some size and associativity combinations are not feasible. The cache offers a base physical line size of 16 bytes with configurability to 32 and 64 bytes by fetching/probing subsequent lines in the cache. According to the hardware layout verification presented by Zhang et al. in [20], for their configurable cache, the configurability does not impact access time.

The configurable cache offers the set of $m = 18$ distinct configurations shown in Table 3, where each configuration is designated by a value c_j. For example, a 4-kByte direct-mapped cache with a 32-byte line size is designated as c_8. These designations will be used throughout the rest of this paper to identify each particular cache configuration.

The values of w, b, and s characterize the configurations with respect to SPCE and are determined as follows: w refers to the associativity of the configuration, which for the configurable cache utilized, is limited to directed-map, 2-way, and 4-way set-associative caches, defined by the values $w = 1$, 2, and 4 respectively.

c_j	description	(w, b, s)	**Cache_hit**$[c_j]$
c_1	directed, 2kB, 16B/line	1, 1, 128	$K_1(1, 128)$
c_2	directed, 4kB, 16B/line	1, 1, 256	$K_1(1, 256)$
c_3	2-way, 4kB, 16B/line	2, 1, 128	$K_{1+2}(1, 128)$
c_4	directed, 8kB, 16B/line	1, 1, 512	$K_1(1, 512)$
c_5	2-way, 8kB, 16B/line	2, 1, 256	$K_{1+2}(1, 256)$
c_6	4-way, 8kB, 16B/line	4, 1, 128	$K_{1+2+4}(1, 128)$
c_7	directed, 2kB, 32B/line	1, 2, 64	$K_1(2, 64)$
c_8	directed, 4kB, 32B/line	1, 2, 128	$K_1(2, 128)$
c_9	2-way, 4kB, 32B/line	2, 2, 64	$K_{1+2}(2, 64)$
c_{10}	directed, 8kB, 32B/line	1, 2, 256	$K_1(2, 256)$
c_{11}	2-way, 8kB, 32B/line	2, 2, 128	$K_{1+2}(2, 128)$
c_{12}	4-way, 8kB, 32B/line	4, 2, 64	$K_{1+2+4}(2, 64)$
c_{13}	directed, 2kB, 64B/line	1, 3, 32	$K_1(3, 32)$
c_{14}	directed, 4kB, 64B/line	1, 3, 64	$K_1(3, 64)$
c_{15}	2-way, 4kB, 64B/line	2, 3, 32	$K_{1+2}(3, 32)$
c_{16}	directed, 8kB, 64B/line	1, 3, 128	$K_1(3, 128)$
c_{17}	2-way, 8kB, 64B/line	2, 3, 64	$K_{1+2}(3, 64)$
c_{18}	4-way, 8kB, 64B/line	4, 3, 32	$K_{1+2+4}(3, 32)$

Table 3: Configuration space for a single-level cache

The value of b depends on the word size (8-bytes) and the respective line size in words. For example, a line size of 16-bytes comprises 2 words (2 × 8-bytes). If $2^b = 2$-words, then $b = 1$. Likewise, 32-bytes is $b = 2$, and 64-bytes is $b = 3$. Finally, the number of sets (s) is determined by dividing the total size by the line size and then by the associativity (w). In the configuration c_{18}, for example, 8-kBytes divided by 64-bytes gives us 128 lines, or 32 sets of 4 lines each ($s = 32$).

Since 64-bytes is the largest block size in the design space utilized, it corresponds to 8 words or $bmax = 3$. The value of $smax$ is defined by the configuration with the maximum number of sets in the design space. It corresponds to configuration c_4 in Table 3 (8-kByte directed-map cache with 16-bytes per line), which give us $smax = 512$. Actually, the multi-layered table limited by the bounds above ($w = \{1, 2, 4\} \times b = \{1, 2, 3\} \times s = \{32, 64, 128, 256, 512\}$) would be able to evaluate 45 different cache configurations, of which the design space of 18 configurations defined in the Table 3 is a subset of.

The right-hand column of the Table 3 shows how SPCE extracts the number of hits from the multi-layered conflict table K, for every c_j in the design space. The number of cache misses can be determined by subtracting the result from the trace size.

In order to validate our results for the whole space C, we determined cache miss rates for our suite of benchmarks with SPCE and also with a very popular trace-driven cache simulator (DineroIV). Then, we estimated the total energy consumed $e(c_j, a_i)$ for running each benchmark a_i with the architecture configuration c_j. We adopted the same energy model as utilized by Zhang, so that our results can be accurately compared with his state-of-the-art tuning heuristic. The energy model calculates total energy consumption for each cache configuration by calculating both static and dynamic energy of the cache, main memory energy, and CPU stall energy. We refer the reader to [19] for a detailed description of energy calculation.

7.2 Results

To validate SPCE, we compared the miss rates generated by SPCE with the miss rates generated by DineroIV for the 45 cache configurations supported by the multi-layered conflict table K. We found the miss rate estimation to be identical. Table 4 shows simulation time speedups as high as 20.77 and an average of 14.88 for cache evaluation. The variation in speedups for each application is due to differences in locality.

	Simulation time (sec)		
benchmark	SPCE	DineroIV	Speedup
bcnt	45	360	8
binary	1	30	30
brev	79	405	5.12
fir	1	7	7
g3fax	13	270	20.77
matmul	1	7	7
pocsag	1	7	7
ucbqsort	126	1890	15
epic	840	13500	16.07
average	123	1830.67	14.88

Table 4: Speedup obtained by using SPCE to evaluate cache miss rate vs. DineroIV

We then applied Zhang's heuristic to the reduced cache configuration space of 18 configurations to determine the optimal cache configuration c_{oi} and optimal energy consumption $e(c_{oi}, a_i)$ for each application a_i to compare simulation speedups obtained by using SPCE as opposed to a tuning heuristic. For each benchmark, we determined how many configurations would be explored by Zhang's heuristic and multiplied that by the time it takes to simulate the corresponding benchmark once. Table 5 shows the simulation time speedup obtained by using SPCE to determine the optimal configuration compared to executing Zhang's heuristic. Due to the limited space, discussion of the tuning heuristic is outside the scope of this paper and we refer the reader to [19] for more details.

			Simulation time (secs)		
bmark	c_{oi}	$e(c_{oi}, a_i)$	SPCE	Heuristic	Speedup
bcnt	c_7	0.001663	45	41	0.91
binary	c_7	0.000145	1	2.7	2.7
brev	c_{14}	0.003202	79	54	0.68
fir	c_9	0.000101	1	1	1
g3fax	c_{14}	0.002074	13	36	2.77
matmul	c_1	0.000071	1	0.5	0.5
pocsag	c_9	0.000174	1	1	1
ucbqsort	c_{14}	0.015402	126	252	2
epic	c_1	0.089339	840	1800	2.14
		average	123	243	1.97

Table 5: Speedup obtained by using SPCE to determine the optimal cache configuration vs. a state-of-the-art cache tuning heuristic

8. CONCLUSION

In this paper we introduce SPCE as a technique to evaluate an entire configurable cache design space with just one single pass, eliminating multiple costly simulation passes as with previous methods. The proposed technique exploits fundamental characteristics of locality and conflicts in the application behavior to gather cache statistics and store them in a very compact data structure. SPCE facilitates in both ease of cache miss rate estimation and reduction in simulation time. We devised SPCE with hardware implementation in mind to facilitate in non-intrusive runtime cache configuration exploration. Our future work includes extending the design space exploration to consider a second level of cache.

9. ACKNOWLEDGEMENTS

We would like to thank NSF and CNPq for supporting this International Cooperation (Proc. 490778/2006-2). We gratefully acknowledge their contribution to the development of the research.

10. REFERENCES

[1] N. B. A. Milenkovic, M. Milenkovic. A performance evaluation of memory hierarchy in embedded systems. In Proceedings of the 35th Southeastern Symposium on System Theory, pages 427–431, March 2003.

[2] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. Journal of Instruction-Level Parallelism, 2, May 2000.

[3] E. Berg and E. Hagersten. StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2004).

[4] G. Braun, A. Wieferink, O. Schliebusch, R. Leupers, H. Meyr, and A. Nohl. Processor/memory co-exploration on multiple abstraction levels. In Proc. of the Design, Automation and Test in Europe (DATE'03), March 2003.

[5] C. Cascaval and D. A. Padua. Estimating cache misses and locality using stack distances. In ICS '03: Proceedings of the 17th annual international conference on Supercomputing, pages 150–159, 2003.

[6] A. Ghosh and T. Givargis. Analytical design space exploration of caches for embedded systems. In Proc. of the Design, Automation and Test in Europe (DATE'03), March 2003.

[7] A. Gordon-Ross and F. Vahid. A self-tuning configurable cache. In DAC '07: Proceedings of the 44th annual conference on Design automation, pages 234–237, New York, NY, USA, 2007. ACM.

[8] A. Gordon-Ross, F. Vahid, and N. Dutt. Fast configurable-cache tuning with a unified second-level cache. In ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design, pages 323–326, 2005.

[9] A. Gordon-Ross, P. Viana, F. Vahid, W. Najjar, and E. Barros. A one-shot configurable-cache tuner for improved energy and performance. In Proceedings of the Design, Automation and Test in Europe(DATE), New York, NY, USA, 2007. ACM Press.

[10] S. Gosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. In In the Proceedings of the Workshop on Interaction between Compilers and Computer Architectures (HPCA-3), 1997.

[11] K. Grimsrud, J. Archibald, R. Frost, and B. Nelson. Locality as a visualization tool. IEEE Trans. Comput., 45(11):1319–1326, 1996.

[12] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. IEEE Trans. Comput., 38(12):1612–1630, 1989.

[13] A. Janapsatya, A. Ignjatović, and S. Parameswaran. Finding optimal l1 cache configuration for embedded systems. In ASP-DAC '06: Proceedings of the 2006 conference on Asia South Pacific design automation, pages 796–801, Piscataway, NJ, USA, 2006. IEEE Press.

[14] A. Malik, B. Moyer, and D. Cermak. A low power unified cache architecture providing power and performance flexibility. In ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design, pages 241–243, 2000.

[15] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. IBM Systems Journal, 9(2):78–117, 1970.

[16] P. R. Panda, A. Nicolau, and N. Dutt. Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[17] S. Segars. Low-power design techniques for microprocessors. In ISSCC'01: International Solid State Circuits Conference, San Francisco, CA, USA, 2001. IEEE Computer Society.

[18] R. Sugumar and S. Abraham. Efficient simulation of multiple cache configurations using binomial trees. In Technical Report CSE-TR-111-91., 1991.

[19] C. Zhang, F. Vahid, and R. Lysecky. A self-tuning cache architecture for embedded systems. In Proc. of the Design, Automation and Test in Europe(DATE'04), February 2004.

[20] C. Zhang, F. Vahid, and W. Najjar. A highly configurable cache for low energy embedded systems. Trans. on Embedded Computing Sys., 4(2):363–387, 2005.