Caches As Filters: A Framework for the Analysis of Caching Systems

Dee A. B. Weikle[†], Sally A. McKee[‡], Kevin Skadron[†], Wm.A. Wulf[†]

[†]Department of Computer Science
University of Virginia
151 Engineer's Way, PO Box 400740
Charlottesville, VA 22904-4740
{daw4q|skadron@virginia.edu, wwulf@nae.edu}

[‡]Department of Computer Science
University of Utah
50 S. Central Campus Dr., #3190
Salt Lake City, UT 84112-9205
{sam@cs.utah.edu}

Abstract

This paper introduces a new analytical framework for analyzing and designing caches. It consists of four major parts: *TSpec notation*, into which reference traces can be transformed; *equivalence classes*, which abstract away chance effects of address bindings and specific inputs; the *functional filter model*, which operates on TSpec traces and provides a formal description of cache operation; and *new metrics*, which evaluate cache performance. This paper gives an overview of TSpec notation and equivalence classes, and then illustrates how the functional filter model can be used to derive better understanding of cache behavior.

1. Introduction

The work of today's cache designer is becoming increasingly difficult. It is well-accepted that there is a processor-memory performance gap that must be compensated for with the caching system [4, 10, 17]. Every time there is an increase in the speed of a microprocessor, the cache and corresponding memory system must be redesigned to feed the increased need for instructions and data to operate on. There continues to be a constant level of research and improvement to cache functionality, but such research typically focuses more on improvements to the cache system itself and less on the process or underlying theory behind cache design. The most common approach is to modify the cache hierarchy and then simply judge that design by running benchmarks through a simulator to determine hit rates or average memory access times. Occasionally in the past and more often now, researchers are taking a different approach and attempting to design not just a better cache, but better ways to design and analyze caches through new models or measures [1, 3, 5, 7, 8, 9, 12, 13, 14, 16].

This paper describes an analytical framework for cache design. There are four major components that form the framework, each of which is a contribution on its own. First, the *TSpec notation* is a more formal way for researchers to communicate with clarity about memory references generated by a processor. Second, the concept of an *equivalence class* of memory references provides an abstraction for eliminating the random address placement effects of actions such as declarations, certain compiler and linker decisions, heap allocation or specific inputs. Third, the *functional cache filter model* uses the TSpec notation and equivalence class concept to allow designers to more clearly understand the effects of cache systems on particular memory references. Fourth, *new metrics* provide more insight into cache design than current measures such as hit rate or average memory access time. This last aspect is beyond the scope of this paper, but introduces two new measures, *instantaneous hit-rate* and *instantaneous locality*. Interested readers should see [16].

2. Proposed Approach

The analysis approach discussed here is inspired by viewing a cache as a filter. As depicted in Figure 1, a cache filters out the references that hit and transforms an input set of references into another, hopefully sparser, output set. Thus, designing memory hierarchies can be seen as akin to designing a compound optical lens: no single lens has all the desired properties, but by cascading several lenses, optical designers can achieve amazing acuity. Likewise, we can view a cache as a filter that transforms an input sequence of data references into an output sequence representing a subset of its input. By composing a series of such caches, as many references as possible are filtered from the request string before it is presented to main memory. To get the best overall performance, the goal of a particular level of cache is not only to filter out the most references, but to filter out those that the next level cannot capture.

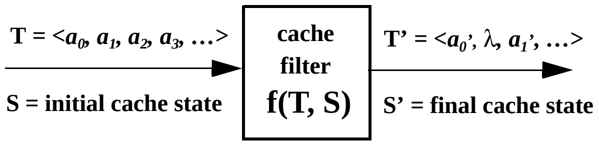

Figure 1: The Cache Filter Model

In the following analysis, we define a *reference string* to be the list of addresses (read or write) presented to the memory system, and denote it as a sequence, <a_0, a_1, a_2, a_3, a_4, …>. The subscript indicates the *position* in the reference string, and is only loosely related to wall-clock time. At first it may seem that a_x and the filtered a_x' will always be the same address. In most cases, this is true, but if an entire line is fetched to fill the cache, the order and value of the address may change. In this paper the terms *reference string*, *reference sequence*, and *trace* are used interchangeably, and are denoted by the capital letter "**T**". We use the symbol λ to indicate the position of a reference removed by a cache filter. This allows correlation between the input and output reference strings. For instance, the input <a, a, a> generates the output <a, λ, λ> for most caches. We view the cache as a filter function, ***f***, on the input of the reference string, **T**, and the state of the cache, **S**. The output of a filter function ***f***(**T**;**S**) consists of an output trace, **T'**, and an output state, **S'** (represented as the pair **T'**;**S'**). Figure 1 illustrates this relationship. The trace-only portion of the output of a filter function is denoted $\boldsymbol{f}^{\mathbf{T}}(\mathbf{T};\mathbf{S})$, and the state-only portion is denoted $\boldsymbol{f}^{\mathbf{S}}(\mathbf{T};\mathbf{S})$.

To supplement the cache filter model, we have developed a new transform that is specific to cache design and simplifies modeling of the cache in the new domain. The domain to which we transform reference strings is the TSpec notation outlined below. In addition, viewing reference strings as combinations of *primitive* TSpec reference patterns allows us to determine the overall effect of a cache on that reference string in a straightforward manner.

3. Overview of TSpec Notation

Figure 2 shows an example of the TSpec for an inner loop that copies one vector to another. The code has been simplified to three assembly instructions to allow the pattern to be easily seen in the reference string (one might think of these three instructions as loading an element from the source vector, storing that element in the destination vector, and branching back). The following paragraphs explain this example in detail. The most basic TSpec element is a *trace atom*, a single address or reference in the trace. It can be represented by either a *literal*, by λ, or by a *variable*. A literal is an explicit (constant) numerical address, (e.g., 100 in the reference string below). A variable represents a regular sequence of addresses, and is specified by a base address and an increment (stride). In the copy example, *c*, *f*, and *t* are all

```
C Code:     for i=1 to 3 t[i] = f[i];

TSpec:      c(100, 4); f(200, 4); t(300, 4);
          <!#f, !#t, (!#c, c+, f+, c+, t+, c)*3>

Reference
String:      100, 200, 104, 300, 108,
            100, 204, 104, 304, 108,
            100, 208, 104, 308, 108
```

Figure 2: Copy example

variables. *c* represents the addresses of the code references, and has a base address of 100 and an increment of four. *f* represents the addresses of the source vector from which data is being copied in the example, and *t* represents the addresses of the destination vector into which the data is being stored. For simplicity, references to *i* are assumed to be to a register and not represented here. A variable can be *initialized* (denoted *#x*) to set its current value to its base address. (Note in the example below, all the initializations are preceded by *!*, which simply suppresses the generation of an address. A variable can also be *post-incremented* (e.g., *c+*) so that after its current value is used, the value is updated to be the sum of itself and the increment specified in its definition. In the example above, the first *c+* in the specification represents the address 100 and increments the value of *c* to 104, so that the next occurrence of *c+* represents the address 104.

A trace is represented by a *concatenation* of atoms separated by commas. A variable or a concatenation of variables can then be repeated with the *iteration* operator, *. So in the example above, the **3* after the parentheses causes the trace within the parentheses to be used three times. Notice that since the initialization for *c* is within the parentheses, the address represented by *c* for each iteration are the same, but since the initializations for *f* and *t* are not within the parentheses, the addresses represented by *f+* and *t+* change in each iteration. A * with no explicit iteration count simply means "zero or more repetitions", by analogy to the Kleene star in regular expressions.

The last TSpec operator required here is *merge*, denoted $\mathbf{T_1}$ *&* $\mathbf{T_2}$. It is easiest to visualize this operation by lining the traces up one above the other as if they were going to be "added", and merging each set in the same position in the reference string. The merge of multiple traces is formed one atom at a time. The merge of a single atom with any number of λs is defined to be the atom. The merge of any number of λs is defined to be λ. The merge of multiple non-λ atoms is undefined. For example,

$$< a_1, \lambda, a_3, \lambda > \& < \lambda, a_2, \lambda, a_3 > = < a_1, a_2, a_3, a_4 >.$$

4. Equivalence Classes

When analyzing a memory system, cache designers traditionally work with specific traces for which the address bindings and the input data, and hence the path through each program, are known, much as in the example trace from Figure 2. Sometimes it may be beneficial to abstract away artifacts due to chance address bindings or specific inputs, or to consider the set of all possible traces from a certain piece of source code. To address these issues, we introduce the concept of *equivalence classes*. We divide the set of traces that can be generated by any specific piece of source code into four sets, depending on whether or not the address bindings and input data are known. The relationship among these groups is shown in Figure 3.

In the figure, **T** represents a trace for which addresses and input values are bound. The set of traces that would be generated with the same source code and the same set of address bindings as **T**, but with different input data, is denoted $\{\mathbf{T_d}\}$, and is referred to as the equivalence class of traces *under varying data input*. Similarly, $\{\mathbf{T_b}\}$ represents the equivalence class of traces *under varying address bindings*. (Note that $\{\mathbf{T_b}\}$ is essentially a generalization of the translation group for arrays described by Harper *et al*. [8].) The sections that follow apply our analysis techniques to the copy example from Section 3, and in the process, extend the notation to permit descriptions of other members of the equivalence class

{$\mathbf{T_b}$}. The notation is expanded by a conditional construct and used to describe members of the equivalence classes {$\mathbf{T_b}$} and {$\mathbf{T_{bd}}$} in [15].

Same bindings \ Same data	Y	N
Y	T	{T_d}
N	{T_b}	{T_{bd}}

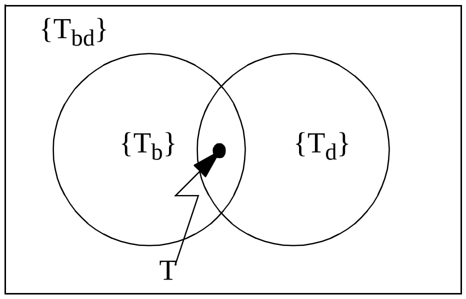

Figure 3: Equivalence Classes
The relationship among traces generated by a specific source program, varying bindings only ({T_b}), input data only ({T_d}), or both bindings and input data ({T_{bd}}).

5. Analysis Method

5.1 Overview

We first set the stage for our analysis by:

1) Characterizing a set of "primitive" traces that, when merged, describe the input trace, and
2) Defining filter functions, f_{cache}(T, S), as syntactic operators that characterize various caches.

Then we can perform the steps of our analysis method on an example by:

3) Transforming the trace to be analyzed into primitive traces, and
4) Applying the filter functions of the cache system to the primitive traces singly, then
5) Recombining the filtered primitive traces via a TSpec merge to see the effect of the cache system on the trace.

For this approach to work, F(T1 & T2, S) = F(T1, S) & F(T2, S) and the definition of merge must be extended to include trace-state pairs. If all references fit in the cache, the trace-state merge simplifies to the trace-only merge. The examples below show the implications of these general ideas. The point here is to understand that because F(T1 & T2, S) = F(T1, S) & F(T2, S), we can analyze most common caches on a wide spectrum of traces by defining cache-filter functions for only a relatively small number of primitives.

Characterizing Primitive Traces

Despite the fact that there are an infinite number of possible source programs with an infinite number of inputs, the reference patterns that they generate can be described by combinations of a small set of parameterized primitive patterns. This is true because source programs are themselves composed of combinations of similar code constructs. The simplest two families of these primitive patterns are the code loop and the stream. The *f* and *t* variables in Figure 3 are examples of streams. They consist of a base address and an increment(stride). *c* is an example of a code loop. These consist of a base address, an increment that defines the length of the code word, and a number of loop iterations. Note that the essential difference is that the stream has no repeated addresses, while the code loop does. There are other families of primitives, but their description is beyond the scope of this paper. Extending the use of λ as a placeholder, we can show the frequency of a primitive within a reference string without describing the particulars of the other primitives. For example, a stream primitive could appear every 4th reference in a code loop and be described as <(f+, λ, λ, λ)*>.

Defining Filter Functions

Once the primitive set is defined, we must define the cache filter function for each primitive and for each type of cache. In general, the filtering function for a cache can be defined one reference at a time as shown below. This function describes the output of a traditional cache. It could be a write-back cache (where dirty/modified lines are written back to the next level only when the line is evicted) or a write-thru cache (where all writes modify main memory by writing the cache line and main memory).

$$f(a,S) = \begin{cases} d(a),\ l(a);S' & \text{if write-back, dirty, } a \text{ not in S} \\ l(a);S' & \text{if write-thru, and } a \text{ not in S} \\ \lambda;S'' & \text{otherwise} \end{cases}$$

In the first situation, where the line is dirty and *a* misses, the output trace consists of the line evicted from the cache, (*d(a)*),and the line that includes the new reference *a, (l(a))*. The new state of the cache is denoted *S'*where *S' = S - d(a)* ∪ *l(a).* The write-through version of the cache would not have a dirty line to evict, so the output would just be the reference to fill the line that includes the new reference *a, (l(a)),* and the new state (*S'*). If *a* hits, the contents of the state remain the same, but the new state is denoted *S"* to indicate the potential for change in the replacement algorithm.

While the above definition is very general, it provides little insight into what happens on the primitive or kernel levels. We are developing a catalog of cache filter functions that operates much like a set of integration tables. The primitives are listed with parameters for the number of repetitions, λs, cache size, and cache associativity. Each primitive can be filtered as a whole instead of one reference at a time by looking it up in the catalog and substituting specific values for the parameters. The whole catalog would be inappropriate here, but we give one example and then use results from the catalog in the examples below.

To understand the single catalog entry, an explanation of the state representation is needed. The state is an ordered set of index-value pairs <i, v> and, with the addition of one more construct, can be represented in TSpec. The caches in the examples below use LRU replacement. To write TSpec with LRU order, it is useful to start from the *end* of a TSpec construct, rather than the beginning. By analogy with *!#c*, we define *!c#* to initialize a variable to its *last* value in the trace. In the catalog entry below, the *x* variables represent parameters.

f_{FAILRU}(<!#a, (λ*x1, a+, λ*x2)*x3>; S) = *T'*; *S'*
where *T'* = <!#a, (λ*x1, a+, λ*x2)*x3> and
S' = !a#, a-*x3, *S*

This catalog entry says that for any stream, with any number of intervening λs, the filtering effect of a fully associative infinite LRU cache with a line size of one word, will result in an output trace that is the same as the input trace and a new cache state that is the reverse stream concatenated with the old state.

5.2 Example Analysis

Transforming Traces

For the purpose of our analysis methods we choose the representative of the trace from its equivalence class {T_b} without any chance address bindings. This representative is depicted by describing the shape of the trace's primitives without the specific address bindings. It can be described by the TSpec notation *without the base address information* and allows us to see the underlying form of the references. As an example of the transformation that must take place, consider the copy example of Figure 3. This segment consists of three primitive traces, one code loop and two streams. They are described as follows:

T1 = (!#c, c+, λ, c+, λ, c)*3
T2 = !#f, (λ, f+, λ, λ, λ)*3
T3 = !#t, (λ, λ, λ, t+, λ)*3

Applying Filter Functions

We are now able to apply the filter functions to the individual primitives in the previous section. For our complete example, we will use the reference string in Figure 3 and consider its output from two different styles of cache. The first cache, DM, will be a direct-mapped cache and the second cache, FA, will be a fully associative cache with LRU replacement. The rule-of-thumb in the cache community has been that in most situations, FA would be the more effective cache.

Consider what happens when the reference string will not fit into either cache. Let the caches be of size 4 with a line size of one reference. Transforming T into its primitive traces generates the traces T1–T3 above. T1 corresponds to the code references, T2 to the vector being copied from, and T3 to the vector being copied to. The filtering function is now applied to each of the primitives singly. The trace portion of the output prior to merging is the same for each cache in this example. The code output is:

$f_{DM}{}^{T}$(T1, S) = $f_{FA}{}^{T}$(T1, S) = T1'
= (!#c, c+, λ, c+, λ, c+), λ*10.

This shows that the first iteration of the code misses while subsequent iterations all hit in both caches. The ten λs at the end represent the 10 filtered references from the second and third iterations (5 from each). Filtering the two data stream primitives yields

$f_{DM}{}^{T}$(T2, S) = $f_{FA}{}^{T}$(T2, S) = T2'
= !#f, (λ, f+, λ, λ, λ)*3 = !#f, (f+, λ*4)*3 = T2 and

$f_{DM}{}^{T}$(T3, S) = $f_{FA}{}^{T}$(T3, S) = T3'
= !#t, (λ, λ, λ, t+, λ)*3 = !#t, (t+, λ*4)*3 = T3.

This demonstrates that each of the streams comes through untouched because they contain no repeating addresses.

When merging the filtered primitives, we must use caution. Different types of caches will have different effects during a merge when the capacity is limited. For direct-mapped caches the merged miss rate can be estimated based on the effects of the conflict misses of one iteration. If we evaluate using the metric *expected misses*, or Em[T], this yields the following expected performance: T1 expects 3 misses; T2 expects 3 misses; T3 expects 3 misses. Approximately 1 additional miss per iteration (other than in the first iteration) will occur due to conflicts arising from capacity issues, *i.e.*,1*2 = 2. Overall, then:

Em[T] = Em[T1] + Em[T2] + Em[T3] + Em[Capacity]
= 3 + 3 + 3 + 2
= 11 misses

The approximate trace output of the direct-mapped cache is $f^{T}{}_{DM}$(T, S) = (!#c, c+, f+, c+, t+, c+), (!#c, c+, f+, t+)*2 where the precise c+ in the second set of parentheses depends on the particular code reference evicted each iteration.

For a fully-associative cache experiments have shown f^{T}(T, S) has two possibilities for a loop of this type. The first is that the whole loop fits in the cache, and the output is only the compulsory misses. The second possibility is if the loop does not fit, in which case $f^{T}{}_{FA}$(T, S) = T because the latter references in the loop always evict the first references in the loop before they can be reused. For our example $f^{T}{}_{FA}$(T, S) = T = !#f, !#t, (!#c, c+, f+, c+, t+, c+)*3. All of the references miss and there are a total of 15 misses—poorer performance than the direct-mapped case.

This method of analysis has already provided two insights. The first is that when a loop reference pattern does not fit in a fully-associative LRU cache, an MRU replacement algorithm or a direct mapped cache provides better performance. The second insight is that a non-fully associative LRU cache can be viewed as several parallel fully-associative caches, operating on separate sections of the reference stream. By separating our initial reference string into separate reference strings for each set in an x-way LRU associative cache, we can perform our analysis as outlined above. The output of each set after priming will either be all λs (if capacity is not a problem), or exactly the input reference stream (if there is not sufficient capacity). Different functional rules will need to be developed for caches (or sets) with different replacement algorithms, but this rule covers a large number of current cache designs.

6. Summary

In this paper we have outlined a new analytical framework to increase the effectiveness of cache design and research. The feasibility of this analysis has been shown by demonstrating its use on a simple example kernel, *copy*. Developing this example yielded two insights regarding fully-associative LRU caches. In addition, it sheds light on why random replacement caches may yield more consistent performance than LRU replacement caches. By separating traces into primitives, we can utilize a catalog of cache filter functions to formally determine the effects of a cache on a general class of kernels.

Acknowledgments

This work was supported in part by the National Science Foundation (NSF) under Award Number POWRE-9806043. The views, findings, and conclusions or recommendations contained herein are those of the authors, and should not be interpreted as necessarily representing the official views, policies or endorsements, either express or implied, of the NSF or the U.S. Government.

References

[1] A.P. Batson, D.W.E. Blatt, and J.P. Kearns. Structure within locality intervals. In *Proc. of the Third International Symposium on Modeling and Performance Evaluation of Computer Systems*, 1977.

[2] A.P. Batson and A.W. Madison. Measurements of major locality phases in symbolic reference strings. In *Proc. of the International Symposium on Computer Performance, Modeling, Measurement and Evaluation*, March 1976.

[3] M. Brehob and R. Enbody. A mathematical model of locality and caching. Michigan State Univ. Computer Science Dept. Technical Report, TR-MSU-CPS-96-42, November 1996.

[4] D. Burger, J.R. Goodman, A. Kagi. Quantifying memory bandwidth limitations of current and future microprocessors. In *Proc.of the 23rd International Symposium on Computer Architecture*, May 1996.

[5] P. Denning. The working set model for program behavior. *Communications of the ACM* 11(5), May, 1968.

[6] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proc. of the Eighth International Symposium on Architectural Support for Programming Languages and Operating System*s, October, 1998.

[7] K. Grimsrud, J. Archibald, R. Frost, and B. Nelson. Locality as a visualization tool. *IEEE Transactions on Computers*, 45(11), November 1996.

[8] J. Harper, D. Kerbyson, and G. Nudd. Analytical modeling of set-associative cache behavior. *IEEE Transactions on Computers*, 48(10):1009, October 1999.

[9] B. Jacob, P. Chen, S. Silverman, T. Mudge. An analytical model for designing memory hierarchies. *IEEE Transactions on Computers*, 45(10):1180-94, October 1996.

[10] N. P. Jouppi and P. Ranganathan. The relative importance of memory latency, bandwidth, and branch limits to performance. In *Proc. of the ISCA 97 Workshop on Mixing Logic and DRAM: Chips that Compute and Remember*, June 1997.

[11] S.A. McKee, Wm.A. Wulf, D.A.B. Weikle. TSpec: A specification language for reference traces. Univ. of Virginia Dept. of Computer Science Technical Report CS-97-19, August 1997.

[12] K. McKinley and O. Temam. A quantitative analysis of loop nest locality. In *Proc. of the Seventh International Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[13] D. Thiebaut. On the fractal dimension of computer programs and its application to the prediction of the cache miss ratio. *IEEE Transactions on Computers*, 38(7), July 1989.

[14] J. Voldman and L. Hoevel. The software-cache connection. *IBM Journal of Research and Development*, 25(6), November 1981.

[15] D.A.B. Weikle, K. Skadron, S.A. McKee, Wm. A. Wulf. Caches as filters: A unifying model for memory hierarchy analysis. Univ. of Virginia Dept. of Computer Science Technical Report CS-2000-16, June 2000.

[16] D.A.B. Weikle, S.A. McKee, Wm. A. Wulf. Caches as filters: A new approach to cache analysis. In *Proc. of the Sixth International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, July, 1998.

[17] W.A. Wulf, and S.A. McKee. Hitting the memory wall: Implications of the obvious. ACM *Computer Architecture News*, 23(4), September 1995.