Caches as Filters:
A Unifying Model for Memory Hierarchy Analysis

University of Virginia Department of Computer Science
Technical Report CS-2000-16
June 2000

[†]Dee A. B. Weikle, [†]Kevin Skadron, [*]Sally A. McKee, [†]William A. Wulf

[†]Department of Computer Science
University of Virginia,
151 Engineer's Way, PO Box 400740
Charlottesville, VA 22904-4740
{daw4q | skadron@cs.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 outlines the new* caches-as-filters *framework for the analysis of caching systems, describing the functional filter model in detail. This model is more general than those introduced previously, allowing designers and compiler writers to understand why a cache exhibits a particular behavior, and in some cases indicating what compiler or hardware techniques must be employed to improve a cache hierarchy's performance. Three components of the framework, the trace-specification notation, equivalence class concept, and new measures of cache performance, are described in previous publications. This paper extends the framework with a formal definition of the functional filter model and augments the trace-specification notation with additional constructs to describe conditionals and the effects of cache filtering. We then give detailed examples demonstrating the application of the model to a set of examples of a copy kernel, where each example represents a different equivalence class of traces.

1. Introduction

As the processor-memory performance gap grows, the work of today's cache designer becomes increasingly difficult: with each jump in microprocessor speed, the cache hierarchy must be redesigned to keep pace. In addition, some compiler optimizations must be modified to account for the new hierarchy and its latencies. Numerous research efforts investigate improvements in cache performance, yet most take only an *ad hoc* approach, judging a proposed cache modification by running benchmarks through a simulator to determine hit rates or average memory access times. While this has yielded successful new designs, such research has reached a domain of diminishing returns, and the *ad hoc* approach typically yields no insight into *why* a cache behaves as it does or how different components of the hierarchy interact. Formal techniques are therefore becoming more important for the analysis and design of high-performance memory hierarchies.

The *caches-as-filters* framework for memory-hierarchy analysis and design takes a new approach to the analysis of cache behavior. The standard approach is *cache-centric:* behavior is a function of the cache organization, and is typically characterized by hit rate. Our work instead treats caches as *filters,* and takes a *trace-centric* approach: behavior is characterized as a function on the memory-reference sequence, and key aspects of the analysis are invariant across cache organizations.

The chief advantage of this new framework is that it is *general*: it can be used to describe and analyze any sequence or pattern of memory references. It is not limited to loop, array, or other reference patterns with an affine relationship. This generality necessarily involves some complexity in notation: accommodating general reference sequences means that we cannot use simple equations, for example. An important component of the caches-as-filters framework is that the analysis techniques can be automated. The framework is therefore *practical,* even with complex sequences.

The next section describes our approach. Section 3 provides an overview of TSpec notation, and Section 4 describes equivalence classes. Section 5 then uses several examples to demonstrate the application or our approach and to illustrate its benefits. Section 6 describes related work, and Section 7 puts this work in perspective with respect to the larger caches-as-filters research effort.

2. Approach

The caches-as-filters framework includes four components. First, the *TSpec notation* is a formal means by which researchers can communicate with clarity about memory reference sequences and how they behave in different cache-hierarchy organizations. This notation describes not just a trace of references generated by a processor, but also describes traces that have been "filtered" by an arbitrary sequence of caches [new TSpec reference]. Second, the concept of an *equivalence class* of memory reference traces provides an abstraction away from random address-placement effects due to declarations, compiler and linker decisions, and heap allocation. Third, the *functional cache filter model* uses the TSpec notation and equivalence classes to help designers more clearly understand the interactions among the components of a cache hierarchy and the effects of cache systems on traces and memory reference patterns. Fourth, *new metrics* provide more insight into cache design than current measures such as hit rate or average memory access time. Two such metrics—*instantaneous locality* and *instantaneous hit rate*—are introduced elsewhere [Wei98], and have been incorporated them into an interactive tool for rapid visualization of performance traces [Pag99].

This framework takes a trace-centric approach: a cache filters out trace references that hit, changing an input set of references into another, hopefully sparser, output set. 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 performance, the goal of a particular level of cache is not only to filter out the most references, but to filter out those references that the farther levels cannot capture.

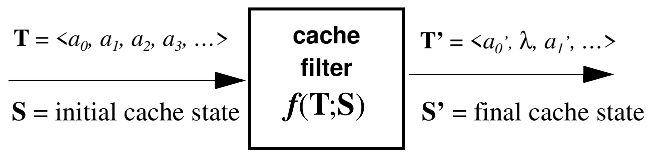

Figure 1: The Cache Filter Model

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, $\langle a_0, a_1, a_2, a_3, a_4, \ldots \rangle$. The subscript indicates the *position* in the reference string, and is only loosely related to wall-clock time. 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 $\boldsymbol{f}^{\mathbf{S}}(\mathbf{T};\mathbf{S})$.

Application of the functional filter model consists of:

1) *Transforming a trace* or segment of source code into TSpec (Section 3),
2) *Determining the equivalence class* (Section 4) applicable for this analysis, and
3) *Applying the filter functions* (Section 5) to the TSpec description in stages, which includes:
   a) *Applying a linesize function*, $\boldsymbol{f}_{\mathbf{LS}}(\mathbf{T})$, that converts all references to the same line to the same address,
   b) *Applying a “number of sets function”*, $\overline{\boldsymbol{P}}(\mathbf{T})$, that partitions trace **T** into separate traces for each set,
   c) *Applying a combination set size and replacement algorithm function*, $\boldsymbol{f}_{\mathbf{SizeRe}}(\mathbf{T};\mathbf{S})$, to each set trace, and
   d) *Recombining set answers* for final results.

3. TSpec Basics

Figure 2 shows an example of the TSpec for a simple for an inner loop that copies one vector to another. The code has been simplified to allow the pattern to be easily seen in the reference string. The following paragraphs explain this example in detail, and a glossary of TSpec notation appears at the end of this paper for the reader’s convenience.

Pseudocode: `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

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 above). 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 variables. *c* represents the code references, and has a base address of 100 and an increment of four. *f* represents the source vector from which data is being copied in the example, and *t* represents the destination vector into which the data is being stored. A variable can be *initialized* (denoted *#x*) to set its current value to its base address. (Note in the example above, 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. The TSpec notation also allows variables to have multiple increments, which is useful for representing arrays. For example, in *x(400; 8, 128)*, *x* has two increments or *iterators*. For convenience, when only the first iterator affects the variable value, we place the manipulation symbol (#, +, -) on the same level as the variable (*e.g., x*+). The examples in later sections demonstrate the utility of this notational shorthand. When multiple iterators are used, the symbols are subscripted in an order corresponding to the iterators they manipulate, and we use ~ to indicate that the corresponding iterator is unchanged. For example, $x_{\#,+}$ clears the changes made to the first iterator and post-increments the base address by the second iterator, which in this case represents the current value of *x* and then sets it to 528. $x_{\sim,+}$ would then represent the value 528 and increment it to 656.

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, is 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*. [Har99].) The sections that follow apply our analysis techniques to expanded versions of the copy example from Section 3, and in the process, they extend the notation to permit descriptions of other members of the equivalence classes $\{\mathbf{T_b}\}$ and $\{\mathbf{T_d}\}$.

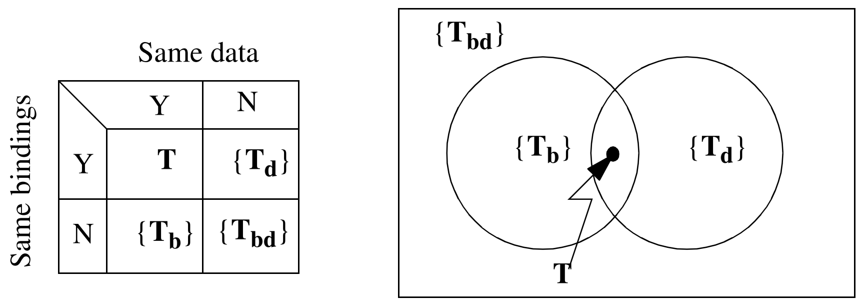

Figure 3: The relationship between traces generated by a specific source program

.

5. Analysis Examples

The following examples illustrate the kinds of analyses that can be performed on individual traces and sets of traces from the equivalence classes. For clarity, all examples are expansions of the copy function shown above. Each starts with C code and the corresponding assembly language generated by gcc.[1] We then show the translation into TSpec and the subsequent analyses for fully associative and direct-mapped caches. Since the examples are small, we choose small cache sizes to highlight situations where unsuspected behavior may occur. Our intent here is to explain the caches-as-filters framework and to demonstrate its application to easily grasped examples, and thus we simplify the explanations by using only virtual addresses. An ideal analysis for multi-level cache systems would use physical addresses for appropriate levels; we reserve incorporating effects of virtual-to-physical translation for future work. Except for pathological cases where small-sized data hit page boundaries, the analyses for the equivalence class under varying address bindings applies to either virtual or physical addresses for the examples in this paper.

5.1. Unconditional Copy C Code, Assembly Language and TSpec

We generated the C code for the more extensive version of the vector copy in Figure 4 with `gcc -O2 -fno-delay-slots`. Figure 5 shows the assembler output, excluding error- and operand-checking code. TSpec for this

1. We use SPARC assembly language, but we abstract away the delay slot and delete extraneous code produced by gcc.

assembly language appears below the figures. For easy verification, the assembly instructions corresponding to the TSpec code references are presented under the TSpec notation. Since we use source code instead of a trace, the number of loop iterations is unknown in this example. For readability, TSpec constructs that do not generate trace atoms (i.e., that are preceded by *!*) are lighter in color.

```
/* The assembly code below is for this copy() function */
void copy(int *f, int *t, int N)
{
  int i;
  for (i=0; i<N; i++)
    t[i] = f[i];
}
/* This main() is simply to illustrate the calling of copy() */
main(int argc, char **argv)
{
  int i;
  int t[3] = {0, 0, 0};
  int f[3] = {10, 11, 12};
  copy(f, t, 3);
  return 0;
}
```

Figure 4: C code for copy example

```
// o0 = f's base address
// o1 = t's base address
// o2 = N
// o3 = i (local)
// The arguments appear in the "o" registers because this is a leaf
// procedure and so the compiler chooses not to allocate a new register
// window
disassembly for a.out

section.text
copy()
    10b64:  96 10 20 00        clr     %o3             // i = 0
    10b68:  87 2a e0 02        sll     %o3, 2, %g3     // compute offset
    10b6c:  96 02 e0 01        add     %o3, 1, %o3     // increment i
    10b70:  c4 02 00 03        ld      [%o0 + %g3], %g2
    10b74:  80 a2 c0 0a        cmp     %o3, %o2
    10b78:  c4 22 40 03        st      %g2, [%o1 + %g3]
    10b7c:  06 bf ff fb        bl      0x10b68
    10b80:  81 c3 e0 08        jmp     %o7 + 8
```

Figure 5: Disassembler output for copy example

c(10b64; 4, 8);
f(%o0; 4);
t(%o1; 4);
<!#c, !#f, !#t, c+, (!$c_{\#,+}$, c+, c+, c+, f+, c+, c+, t+, c+)*, c>
clr sll add ld cmp st bl jmp
// Simplifying this trace yields:
<!#c, !#f, !#t, c+, (!$c_{\#,+}$, c+*3, f+, c+*2, t+, c+)*, c>

5.2. Unconditional Copy Analyses

The first step of the analysis procedure from Section 2 transforms the trace or source code into TSpec, as we did above. The second step determines the equivalence class for which to perform the analysis. The simplest example uses the equivalence class of a specific trace **T**. To this end, we arbitrarily set the number of iterations at three, and assume values for %o0 and %o1. Setting the number of iterations assumes a particular set of input data, and assuming values for %o0 and %o1 assigns base addresses for the arrays *f* and *t*. For simplicity, we focus only on the loop. The modified TSpec description is.

```
c(10b68; 4);
f(FSTART, 4);
t(TSTART, 4);
<!#f, !#t, (!#c, c+*3, f+, c+*2, t+, c+)*3>
```

The next analysis step applies the functional filter model in stages. For this we must represent both the trace and the state for the cache input and output streams. Both may be represented in TSpec, but we must first introduce additional notation for the state. Initially, it might seem that the state could be viewed as an unordered set, yet replacements must occur in order. Having the state description represent that order simplifies the determination of what to replace. 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.

For our first analysis, we use a fully associative, infinite (or any size greater than seven), LRU, write-through cache with a line size of one word. For such a configuration, steps 3a and 3b above (applying the linesize function $\boldsymbol{f_{LS}}(\mathbf{T})$ and the set partitioning function $\overline{\boldsymbol{P}}(\mathbf{T})$) have no effect on **T**. Step 3c, applying a set size and replacement algorithm function, $\boldsymbol{f_{\mathbf{SizeRe}}}(\mathbf{T};\mathbf{S})$, where **T** is the copy example and $\mathbf{S}=\mathbf{S^0}$ (an empty cache), yields:

$$\boldsymbol{f_{\mathbf{FaILru}}}(\mathbf{T};\ \mathbf{S^0}) = \mathbf{T'};\ \mathbf{S'}$$

= <!#f, !#t, (!#c, c+*3, f+, c+*2, t+, c+), (f+, t+)*2>; **S'**

where **S'** = {!c#, !t#, !f#, c-, t-, c-*2, f-, c-*3, (t-, f-)*2}

Representing the state and trace in the same notation makes relationships between them clearer. Here, **S'** is simply **T** in reverse without the repetition of the code references. Experience working with this notation has shown that this notion of the reverse of **T** without duplicates is useful, since this string contains the list of trace items that can be in the state for any LRU cache. We formalize this function as $\boldsymbol{U}(\mathbf{T})$, called the *unique of* ***T***, which is formed by taking the items from **T**, beginning at the end and working backwards, and replacing duplicate items with λ_d. This extends the definition of λ to include a place-holder for duplicate trace atoms. The subscript lets us differentiate between λs that hold places for duplicates (trace items that would be filtered by a cache) and those removed for any other purpose. We define a function $\boldsymbol{D}(\mathbf{T})$ to remove λs of any type from the trace **T**, or $\boldsymbol{D_d}(\mathbf{T})$ to remove only λ_ds. Now we can write the state as:

$$f_{\mathrm{FaILru}}{}^{S}(T;\ S0) = S' = \boldsymbol{D}(\boldsymbol{U}(\mathrm{T}))$$

$$= \boldsymbol{D}_d(\{!c\#,\ !t\#,\ !f\#,\ c-,\ t-,\ c-*2,\ f-,\ c-*3,\ (\lambda_d,\ t-,\ \lambda_d*2,\ f-,\ \lambda_d*3)*2\})$$

$$= \{!c\#,\ !t\#,\ !f\#,\ c-,\ t-,\ c-*2,\ f-,\ c-*3,\ (t-,\ f-)*2\}$$

This notation extends easily to fully-associative caches of finite size. Consider the result of a fully associative LRU cache of size seven[2], again with a line size of one word. Notice that the trace output would be the same for any cache of size seven or smaller. The TSpec for this is:

$f_{\mathrm{Fa7Lru}}(\mathbf{T}; \mathbf{S^0}) = \mathbf{T'}; \mathbf{S'}$
= <!#f, !#t, (!#c, c+*3, f+, c+*2, t+, c+)*3>; **S'**
where **S'** = {!c#, !t#, !f#, c-, t, c-*2, f, c-*2}

Because the cache is too small for this example, every reference misses. As a result, **T'** = **T**, and it is more illuminating to write:

$f_{\mathrm{Fa7Lru}}(\mathbf{T}; \mathbf{S^0}) = \mathbf{T'}; \mathbf{S'}$
= **T**; **S'** where **S'**= {!c#, !t#, !f#, c-, t, c-*2, f, c-*2}

Now we can specify the state in a more general way. Let **S** be an ordered set of *<value, index>* pairs to represent the state of the cache (i.e., **S** = {*<v, i>*} where *v = the address value of the trace atom*, and *i = its index in* ***D***(***U***(**T**)). The state, **S'**, of the cache after the copy example would be **S'** = {*<v,i>* ε ***D***(***U***(copy)) | *i* <= sz}, where *sz* is the size of the cache. Expanding the functions gives:

S' = {<v,i> ε D(!t#, !f#, !c#, c-, t-, c-*2, f-, c-*3, (λ, t-, λ*2, f-, λ*3)*3) | i <= sz}
= {<v,i> ε !t#, !f#, !c#, c-, t-, c-*2, f-, c-*3, (t-, f-)*3) | i <= sz}

This example illustrates that the state of a fully-associative LRU cache of any size is the first *sz* elements in ***D***(***U***(T)).

Now consider the same copy example output from a size-eight, direct-mapped cache. For set associative caches, the mapping of addresses into the cache, and subsequent hits, is more difficult to predict. Each set performs exactly like a fully-associative cache that is the size of the set, and thus we can first partition the trace into a vector of traces (step 3b above), one element of which will correspond to each set, and then we can analyze this vector of traces in the same manner as for the fully-associative cache. By retaining λs for the final step, where we merge the vector of traces back into a single trace, we can see our results as a single trace, too.

Let $\overline{\boldsymbol{P}}(\mathbf{T})$ = the vector of traces formed by splitting **T** into the different traces for each set. To split a TSpec description, we introduce notation to specify a precise reference in such a description. *<fix me>* We use the instance name (which would be the variable name or the constant value of the reference), but if there are multiple uses of the instance name, more information is needed. For example, the copy example has six uses of the *c* variable in each iteration. To specify a particular instance, a dot and number after the variable name are used. The number indicates the position of the instance in the TSpec string. Additional dots may be used to specify the iteration number of that variable (or constant), if it is inside a loop. For example, the last instance of the *c* reference from the last iteration in the copy example above is denoted *c.6.3*. This is the sixth instance of the variable *c* in the third iteration (the last iteration here). The last *t* is specified as *t.1.3*. This notation is extended to multiple loops by subscripting the variable name with a loop label.

For our example above, the code starts in set two, assuming a one-word line (32 bits), because 0x10b68 / four

2. We use a very small cache size here to demonstrate the two different behaviors possible for a fully associative cache on this example. One is that every reference misses for caches of size seven or smaller, and the other is only compulsory misses occur for caches of size eight or larger.

bytes = 0x42da, and 0x42da / eight lines has remainder two. Arrays *f* and *t* can end up at different addresses depending on the particular arrays used to call *copy()*. Here we assume that *f* starts in set four and *t* in set five, in order to show the analysis for a specific **T**.

Below is $\overline{\boldsymbol{P}}^{0\text{-}7}(\mathbf{T})$ for the copy example. Note that the second iteration is underlined to distinguish it from the others. In addition, conflict misses are in bold (but compulsory misses are not). Again, λ is used as a place-holder for trace atoms. In this case, the lambdas are subscripted with *s* to show they result from set separation.

$P^0(\mathbf{T}) = \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \underline{\lambda_s*8,}\qquad \lambda_s*8$
$P^1(\mathbf{T}) = \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \underline{\lambda_s*8,}\qquad \lambda_s*8$
$P^2(\mathbf{T}) = \text{c.1.1}, \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \underline{\text{c.1.2}, \lambda_s*7,}\qquad \text{c.1.3}, \lambda_s*7$
$P^3(\mathbf{T}) = \lambda_s,\ \text{c.2.1}, \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \underline{\lambda_s, \text{c.2.2}, \lambda_s*6,}\qquad \lambda_s, \text{c.2.3}, \lambda_s*6$
$P^4(\mathbf{T}) = \lambda_s,\ \lambda_s,\ \text{c.3.1}, \text{f.1.1}, \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \underline{\lambda_s*2, \mathbf{c.3.2}, \lambda*5,}\qquad \lambda_s*2, \text{c.3.3}, \lambda_s*5$
$P^5(\mathbf{T}) = \lambda_s,\ \lambda_s,\ \lambda_s,\ \text{c.4.1}, \lambda_s,\ \lambda_s,\ \text{t.1.1}, \lambda_s,\ \underline{\lambda_s*3, \text{f.1.2}, \mathbf{c.4.2}, \lambda_s*3,}\ \lambda_s*4, \text{c.4.3}, \lambda_s*3$
$P^6(\mathbf{T}) = \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \text{c.5.1}, \lambda_s,\ \lambda_s,\ \lambda_s,\ \underline{\lambda_s*6, \text{t.1.2}, \mathbf{c.5.2},}\qquad \lambda_s*3, \text{f.1.3}, \lambda_s, \mathbf{c.5.3}, \lambda_s*2$

At this point, the analysis nicely illustrates how the references are split among sets of the cache, and thus we see how effectively space is being utilized, and how set assignments affect performance. In this example, every code reference experiences a conflict miss two out of every eight iterations, as *f* and *t* march through the cache. On the other hand, if variables *f* and *t* could always be assigned to sets zero and one through some compiler technique like padding, the code would never suffer conflict misses. We could also eliminate conflict misses with a two-entry victim cache [Jou90].

At this point, we can easily see where *not* to map variables that have temporal locality, namely sets two through seven. For example, if we repeatedly consulted *f[0]*, or had some scalar in the loop that for some reason did not fit in the registers, mapping *f[0]* or the scalar to any of sets two through seven would cause that code reference to miss on every iteration. Note also that insights of this type are not limited to a unified cache.

Step 3c applies the set size and replacement algorithm function, $\boldsymbol{f_{dm}}(\mathbf{T};\mathbf{S})$, to each set trace. In this example, this replaces repeated addresses with λ_ds. Since this is a direct-mapped cache (i.e., with a set size of one), only addresses that are repeated immediately (without intervening references) are replaced. If this were a cache with set size greater than one, addresses repeated within the number of unique references equal to the size of the sets would be replaced.

$\boldsymbol{f}^{T}_{dm}(\boldsymbol{P}^0(\mathbf{T}; \mathbf{S^0})) = \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\qquad \underline{\lambda_s*8,}\qquad \lambda_s*8$
$\boldsymbol{f}^{T}_{dm}(\boldsymbol{P}^1(\mathbf{T}; \mathbf{S^0})) = \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\qquad \underline{\lambda_s*8,}\qquad \lambda_s*8$
$\boldsymbol{f}^{T}_{dm}(\boldsymbol{P}^2(\mathbf{T}; \mathbf{S^0})) = \text{c.1.1}, \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\qquad \underline{\lambda_d, \lambda_s*7,}\qquad \lambda_d, \lambda_s*7$
$\boldsymbol{f}^{T}_{dm}(\boldsymbol{P}^3(\mathbf{T}; \mathbf{S^0})) = \lambda_s,\ \text{c.2.1}, \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\qquad \underline{\lambda_s, \lambda_d, \lambda_s*6,}\qquad \lambda_s, \lambda_d, \lambda_s*6$
$\boldsymbol{f}^{T}_{dm}(\boldsymbol{P}^4(\mathbf{T}; \mathbf{S^0})) = \lambda_s,\ \lambda_s,\ \text{c.3.1}, \text{f.1.1}, \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\qquad \underline{\lambda_s*2, \mathbf{c.3.2}, \lambda*5,}\qquad \lambda_s*2, \lambda_d, \lambda_s*5$
$\boldsymbol{f}^{T}_{dm}(\boldsymbol{P}^5(\mathbf{T}; \mathbf{S^0})) = \lambda_s,\ \lambda_s,\ \lambda_s,\ \text{c.4.1}, \lambda_s,\ \lambda_s,\ \text{t.1.1}, \lambda_s,\qquad \underline{\lambda_s*3, \text{f.1.2}, \mathbf{c.4.2}, \lambda_s*3,}\qquad \lambda_s*4, \lambda_d, \lambda_s*3$
$\boldsymbol{f}^{T}_{dm}(\boldsymbol{P}^6(\mathbf{T}; \mathbf{S^0})) = \lambda_s, \lambda_s,\ \lambda_s,\ \lambda_s,\ \text{c.5.1}, \lambda_s,\ \lambda_s,\ \lambda_s,\qquad \underline{\lambda_s*6, \text{t.1.2}, \mathbf{c.5.2},}\qquad \lambda_s*3, \text{f.1.3}, \lambda_s, \mathbf{c.5.3}, \lambda_s*2$
$\boldsymbol{f}^{T}_{dm}(\boldsymbol{P}^7(\mathbf{T}; \mathbf{S^0})) = \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \lambda_s,\ \text{c.6.1}, \lambda_s,\ \lambda_s,\qquad \underline{\lambda_d, \lambda_s*7,}\qquad \lambda_s*6, \text{t.1.3}, \mathbf{c.6.3}$

Removing the λs that result from set separation makes it easier to see what happens in each set, and leaving the

λ_ds shows how often we hit in the cache. Here there are a total of seven hits. The ideal performance of this direct-mapped cache is 12 hits (where all of the repeated addresses are removed), but five hits are lost to conflict rather than capacity misses. This TSpec example precisely characterizes the suboptimal cache placement in this example.

$\boldsymbol{D}_s(\boldsymbol{f}^{T}{}_{dm}(\boldsymbol{P}^0(\mathbf{T}), \mathbf{S^0})) =$
$\boldsymbol{D}_s({}^{T}\boldsymbol{f}_{dm}(\boldsymbol{P}^1(\mathbf{T}), \mathbf{S^0})) =$
$\boldsymbol{D}_s(\boldsymbol{f}^{T}{}_{dm}(\boldsymbol{P}^2(\mathbf{T}), \mathbf{S^0})) = \text{c.1.1}, \underline{\lambda_d}, \lambda_d$
$\boldsymbol{D}_s({}^{T}\boldsymbol{f}_{dm}(\boldsymbol{P}^3(\mathbf{T}), \mathbf{S^0})) = \text{c.2.1}, \underline{\lambda_d,} \lambda_d$
$\boldsymbol{D}_s(\boldsymbol{f}^{T}{}_{dm}(\boldsymbol{P}^4(\mathbf{T}), \mathbf{S^0})) = \text{c.3.1, f.1.1}, \underline{\textbf{c.3.2,}} \lambda_d$
$\boldsymbol{D}_s(\boldsymbol{f}^{T}{}_{dm}(\boldsymbol{P}^5(\mathbf{T}), \mathbf{S^0})) = \text{c.4.1, t.1.1}, \underline{\text{f.1.2, }\textbf{c.4.2,}} \lambda_d$
$\boldsymbol{D}_s(\boldsymbol{f}^{T}{}_{dm}(\boldsymbol{P}^6(\mathbf{T}), \mathbf{S^0})) = \text{c.5.1}, \underline{\text{t.1.2, }\textbf{c.5.2,}} \text{ f.1.3}, \textbf{c.5.3}$

Remerging the set results gives us the final trace answer:

$$\boldsymbol{f}^{T}{}_{dm}(\mathbf{T}; \mathbf{S^0}) = \boldsymbol{f}^{T}{}_{dm}(\boldsymbol{P}^0(\mathbf{T}; \mathbf{S^0})) \;\&\; \boldsymbol{f}^{T}{}_{dm}(\boldsymbol{P}^1(\mathbf{T}; \mathbf{S^0})) \;\&\; \boldsymbol{f}^{T}{}_{dm}(\boldsymbol{P}^2(\mathbf{T}; \mathbf{S^0})) \;\&\; \ldots \;\&\; \boldsymbol{f}^{T}{}_{dm}(\boldsymbol{P}^7(\mathbf{T}; \mathbf{S^0}))$$
$$= \text{!\#f, !\#t, !\#c, c+*3, f+, c+*2, t+, c, !\#c, !c+*2, } \lambda_d\text{*2, c+, f+, c+*2, t+, c+, !\#c, !c+*4, } \lambda_d\text{*3, f+, c+, t, c+}$$

The state analysis is performed separately. We again apply the set separation, $\overline{\boldsymbol{P}}(\mathbf{T})$, and then we find $\overline{\boldsymbol{U}}(\overline{\boldsymbol{P}}(\mathbf{T}))$.

$\boldsymbol{U}(\boldsymbol{P}^0(\mathbf{T})) = \lambda_s*24$
$\boldsymbol{U}(\boldsymbol{P}^1(\mathbf{T})) = \lambda_s*24$
$\boldsymbol{U}(\boldsymbol{P}^2(\mathbf{T})) = \lambda_s*7, \text{c.1}, \lambda_s*7, \lambda_d, \lambda_s*7, \lambda_d$
$\boldsymbol{U}(\boldsymbol{P}^3(\mathbf{T})) = \lambda_s*6, \text{c.2}, \lambda_s*7, \lambda_d, \lambda_s*7, \lambda_d, \lambda_s$
$\boldsymbol{U}(\boldsymbol{P}^4(\mathbf{T})) = \lambda_s*5, \text{c.3}, \lambda_s*7, \lambda_d, \lambda_s*6, \text{f.1.1}, \lambda_d$
$\boldsymbol{U}(\boldsymbol{P}^5(\mathbf{T})) = \lambda_s*3, \text{c.4}, \lambda_s*7, \lambda_d, \text{f.1.2}, \lambda_s*4, \text{t.1.1}, \lambda_s*2, \lambda_d, \lambda_s*3$
$\boldsymbol{U}(\boldsymbol{P}^6(\mathbf{T})) = \lambda_s*2, \text{c.5}, \lambda_s, \text{f.1.3}, \lambda_s*3, \lambda_d, \text{t.1.2}, \lambda_s*9, \lambda_d, \lambda_s*4$
$\boldsymbol{U}(\boldsymbol{P}^7(\mathbf{T})) = \text{c.6, t.1.3}, \lambda_s*13, \lambda_d, \lambda_s*2, \lambda_d, \lambda_s*5$

Since we have a direct-mapped cache, our final state for each set will be the first non-λ element in each:

$$\boldsymbol{U}(\boldsymbol{P}^j(\mathbf{T})).\,\mathbf{S'} = \{\langle v, i\rangle\ \varepsilon\ \overline{\boldsymbol{D}}(\overline{\boldsymbol{U}}(\overline{\boldsymbol{P}}(\mathbf{T}))) \mid i \le 1\}$$
$$= \{\text{c.1, c.2, c.3, c.4, c.5, c.6}\} = \{\text{!\#c, c+*6}\}$$

5.3. Unconditional Copy Analyses with Varying Base Addresses

Now consider not just the specific trace generated by the copy example above, but rather the equivalence class of traces under address binding. This class can be described by the TSpec description above without specific base addresses for the variables.

```
c(CSTART; 4);
f(FSTART, 4);
t(TSTART, 4);
<!#f, !#t, (!#c, c+*3, f+, c+*2, t+, c)*3>
```

When analyzing this set of traces, we must first perform a case analysis of potential conflicts between variables. It is not necessary to perform a systematic analysis of every possible combination of set assignments for each variable, since several of these cases are equivalent. For example, the end result of every variable's being assigned to set zero is the same as the end result of every variable's being assigned to any other set. The general set of cases for this problem can be thought of as a range of cases, spanning a spectrum as the number of conflict misses goes from zero to six. The “minimum” case happens when none of the stream references evict a code reference before it is repeated. The “maximum” case happens when every stream reference evicts some code reference before it can be repeated.

Consider the minimum case more closely. Zero conflicts may seem impossible in a direct-mapped cache, as there are only eight sets and 12 individual references. Yet consider the situation where variable *c* is assigned to set two, and variables *t* and *f* are both assigned to set zero. (Remember, this is one representative of a group of assignments that achieve the same result. The important point is that *t* and *f* are both assigned to the same starting set, which is two sets “before” the starting set for *c*.) The second iteration is underlined.

$\boldsymbol{P}^0(\mathbf{T}) = \lambda_s, \lambda_s, \lambda_s,$ f.1.1, $\lambda_s, \lambda_s,$ t.1.1, $\lambda_s,$ $\underline{\lambda_s{*}8,}$ $\lambda_s{*}8$
$\boldsymbol{P}^1(\mathbf{T}) = \lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s,$ $\underline{\lambda_s{*}3,\ \text{f.1.2},\ \lambda_s{*}2,\ \text{t.1.2},\ \lambda_s.}$ $\lambda_s{*}8$
$\boldsymbol{P}^2(\mathbf{T}) =$ c.1.1,$\lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s,$ $\underline{\text{c.1.2},\ \lambda_s{*}7,}$ c.1.3,$\lambda_s{*}2$, f.1.3, $\lambda_s{*}2$, t.1.3, λ_s
$\boldsymbol{P}^3(\mathbf{T}) = \lambda_s,$ c.2.1,$\lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s,$ $\underline{\lambda_s,\ \text{c.2.2},\ \lambda_s{*}6,}$ λ_s, c.2.3, $\lambda_s{*}6$
$\boldsymbol{P}^4(\mathbf{T}) = \lambda_s, \lambda_s,$ c.3.1,$\lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s,$ $\underline{\lambda_s{*}2,\ \text{c.3.2},\ \lambda_s{*}5,}$ $\lambda_s{*}2$, c.3.3, $\lambda_s{*}5$
$\boldsymbol{P}^5(\mathbf{T}) = \lambda_s, \lambda_s, \lambda_s, \lambda_s,$ c.4.1,$\lambda_s, \lambda_s, \lambda_s,$ $\underline{\lambda_s{*}4,\ \text{c.4.2},\ \lambda_s{*}3,}$ $\lambda_s{*}4$, c.4.3, $\lambda_s{*}3$
$\boldsymbol{P}^6(\mathbf{T}) = \lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s,$ c.5.1,$\lambda_s, \lambda_s,$ $\underline{\lambda_s{*}6,\ \text{c.5.2},\ \lambda_s.}$ $\lambda_s{*}6$, c.5.3, λ_s
$\boldsymbol{P}^7(\mathbf{T}) = \lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s,$ c.6.1, $\underline{\lambda_s{*}7,\ \text{c.6.2},}$ $\lambda_s{*}7$, c.6.3

Replacing immediate repeats with λ_d (which amounts to performing $\boldsymbol{f_{dm}}$ here) gives:

$\boldsymbol{f}^{\boldsymbol{T}}_{\text{dm}}(\boldsymbol{P}^0(\mathbf{T}); \mathbf{S^0}) = \lambda_s, \lambda_s, \lambda_s,$ f.1.1, $\lambda_s, \lambda_s,$ t.1.1, $\lambda_s,$ $\underline{\lambda_s{*}8,}$ $\lambda_s{*}8$
$\boldsymbol{f}^{\boldsymbol{T}}_{\text{dm}}(\boldsymbol{P}^1(\mathbf{T}); \mathbf{S^0}) = \lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s,$ $\underline{\lambda_s{*}3,\ \text{f.1.2},\ \lambda_s{*}2,\ \text{t.1.2},\ \lambda_s.}$ $\lambda_s{*}8$
$\boldsymbol{f}^{\boldsymbol{T}}_{\text{dm}}(\boldsymbol{P}^2(\mathbf{T}); \mathbf{S^0}) =$ c.1.1,$\lambda_s, \lambda_s, \lambda_s, \lambda_s\ \lambda_s, \lambda_s, \lambda_s,$ $\underline{\lambda_d,\ \lambda_s{*}7,}$ $\lambda_d, \lambda_s{*}2$, f.1.3, $\lambda_s{*}2$,t.1.3, λ_s
$\boldsymbol{f}^{\boldsymbol{T}}_{\text{dm}}(\boldsymbol{P}^3(\mathbf{T}); \mathbf{S^0}) = \lambda_s,$ c.2.1,$\lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s,$ $\underline{\lambda_s,\ \lambda_d,\ \lambda_s{*}6,}$ $\lambda_s, \lambda_d, \lambda_s{*}6$
$\boldsymbol{f}^{\boldsymbol{T}}_{\text{dm}}(\boldsymbol{P}^4(\mathbf{T}); \mathbf{S^0}) = \lambda_s, \lambda_s,$ c.3.1,$\lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s,$ $\underline{\lambda_s{*}2,\ \lambda_d,\ \lambda_s{*}5,}$ $\lambda_s{*}2, \lambda_d, \lambda_s{*}5$
$\boldsymbol{f}^{\boldsymbol{T}}_{\text{dm}}(\boldsymbol{P}^5(\mathbf{T}); \mathbf{S^0}) = \lambda_s, \lambda_s, \lambda_s, \lambda_s,$ c.4.1,$\lambda_s, \lambda_s, \lambda_s,$ $\underline{\lambda_s{*}4,\ \lambda_d,\ \lambda_s{*}3,}$ $\lambda_s{*}4, \lambda_d, \lambda_s{*}3$
$\boldsymbol{f}^{\boldsymbol{T}}_{\text{dm}}(\boldsymbol{P}^6(\mathbf{T}); \mathbf{S^0}) = \lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s,$ c.5.1,$\lambda_s, \lambda_s,$ $\underline{\lambda_s{*}6,\ \lambda_d,\ \lambda_s.}$ $\lambda_s{*}6, \lambda_d, \lambda_s$
$\boldsymbol{f}^{\boldsymbol{T}}_{\text{dm}}(\boldsymbol{P}^7(\mathbf{T}); \mathbf{S^0}) = \lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s, \lambda_s,$ c.6.1, $\underline{\lambda_s{*}7,\ \lambda_d.}$ $\lambda_s{*}7, \lambda_d$

Remerging the set traces produces:

T’ = <!#f, !#t, (!#c, c+*3, f+, c+*2, t+, c), (λ*3, f+, λ*2, t+, λ)*2>
and **S’** = { t.1.1, t.1.2, t.1.3, c.2, c.3, c.4, c.5, c.6} = {!#t, t+*3, !#c, !c+*2, c+*5}

This simple reassignment of addresses results in different starting-set assignments, bringing the performance of the direct-mapped cache up to the performance of a fully-associative cache of the same size. In the case where the number of unique addresses in the loop exceeds the number of lines available in the cache (here cache size seven or smaller), a direct-mapped cache will actually outperform a fully associative cache. In every situation, the fully associative cache evicts code before it can be used again, and thus there are no hits. The direct-mapped cache, on the other hand, hits for some code references because the two intervening stream references cannot evict all code references before they are reused. Finally, note that these examples point to the usefulness of stream buffers and victim caches [Jou90], methods for writing around caches for references that are not reused [Jou93], and techniques such as data padding by the compiler to achieve the minimal number of conflict misses [Leb94]. This analysis also shows how cache-conscious data placement [Cal98] can improve cache performance, and how our framework might work in conjunction with such techniques.

5.4. Conditional Copy

With the introduction of conditionals, it is useful to be able to increment a variable without generating an address. This allows the same variable to represent the code or a stream throughout a loop, regardless of whether internal conditional code is executed. For example, "!" before an atom is used on the "not taken" part of the TSpec to represent skipping over the body of an if statement.

A conditional is represented by placing the taken and not-taken paths inside parentheses and separating them by the symbol "|". The parentheses and the "|" are subscripted with the same label to prevent confusion in nested conditionals. For example, an if statement with four instructions within a piece of sequential code might look like $<!\#c, c{+}{*}3, ({}_I c{+}{*}4 \,|_I\, (!c{+}){*}4)_I, c{+}{*}5>$. Loop branches do not use this construct, but rather the "*" after parentheses.

As an example, the C code in Figure 6 has been modified from Figure 4 to include a conditional. The example is artificial in the interest of clarity. The corresponding assembly code follows in Figure 7.

```
void copy(int *f, int *t, int N)
{
  int i;
  for (i=0; i<N; i++)
    if (i != 1)
      t[i] = f[i];
}

main(int argc, char **argv)
{
  int i;
  int t[3] = {0, 0, 0};
  int f[3] = {10, 11, 12};

  copy(f, t, 3);
  return 0;
}
```

Figure 6: C code for copy with embedded conditional

```
// o0 = f's base address
// o1 = t's base address
// o2 = N
// o3 = i (local)
// The arguments appear in the "o" registers because this is a leaf
// procedure and so the compiler chooses not to allocate a new register
// window
section .text
copy()
        10b64:  96 10 20 00        clr          %o3            // i = 0
        10b68:  80 a2 e0 01        cmp          %o3, 1
        10b6c:  02 80 00 05        be           0x10b7c
        10b70:  87 2a e0 02        sll          %o3, 2, %g3    // compute offset
        10b74:  c4 02 00 03        ld           [%o0 + %g3], %g2
        10b78:  c4 22 40 03        st           %g2, [%o1 + %g3]
        10b7c:  96 02 e0 01        add          %o3, 1, %o3    // increment i
        10b80:  80 a2 c0 0a        cmp          %o3, %o2
        10b84:  06 bf ff fb        bl           0x10b68
        10b88:  81 c3 e0 08        jmp          %o7 + 8        // return
```

Figure 7: Disassembler output for conditional copy

The TSpec for this new example appears below:

c(10b64; 4);
f(o0; 4);
t(o1; 4);
<!#c, !#f, !#t, c+, ($_L$!#c+, c+, c+, ($_I$ c+, c+, f+, c+, t+ |$_I$ (!c+)*3)$_I$, c+, c+, c+)$_L$*, c+>
clr cmp be sll ld st add cmp bl jmp

We again focus on analyzing the loop, and initially on the effects of the conditional. Assume that we know the base addresses of the vectors and the number of loop iterations. This yields the following simplified TSpec:

c(10b68; 4);
f(o0; 4);
t(o1; 4);
<!#f, !#t, ($_L$!c$_{\#,+}$, c+*2, ($_I$ c+*2, f+, c+, t+ |$_I$ (!c+)*3)$_I$, c+*3)$_L$*3>

Let us start with the simplest cache situation: a fully-associative LRU cache of size 13 or larger.

f_{FALRU}(**T**; **S**$^{\mathbf{0}}$) = <!#f, !#t, ($_L$!c$_{\#,+}$, c+*2, ($_I$ c+*2, f+, c+, t+ |$_I$ (!c+)*3)$_I$, c+*3)$_L$*3> ; **S'**

where **S'** = !c#, !t#, !f#, c-*3, t-, c-, f-, c-*4, (t-, f-)*2 OR - case where the if is always taken
!c#, !t#, !f#, c-*3, t-, c-, f-, c-*4, (t-, f-) OR - case where the if is taken twice
!c#, !t#, !f#, c-*3, t-, c-, f-, c-*4 OR - case where the if is taken once
!c#, !t#, !f#, c-*3, (!c-)*3, c-*2 - case where if is never taken

This notation is useful for analyzing all possibilities, but it is cumbersome: the number of cases grows with the number of loop iterations. A simpler approach expresses the state in terms of the MIN and MAX possible cache states after execution of the trace. For the above example:

$$MIN \subseteq S' \subseteq MAX$$

where MIN = !c#, !t#, !f#, c-*3, (!c-)*3, c-*2 and
MAX = !c#, !t#, !f#, c-*3, t-, c-, f-, c-*4, (t-, f-)*2

This second method would be useful for a compiler: the MIN indicates which items will *always* be available in the cache, and the MAX, which additional items *may* be in the cache. Software prefetching and other techniques should be applied first to items that are known not be in the cache.

6. Related Work

Other researchers have also explored better ways to design and analyze caches through new models or measures. Voldman and Hoevel [Vol81] describe an adaptation of standard Fourier analysis techniques to the study of cache systems. The cache is viewed as a “black box” boolean signal generator, where “ones” correspond to cache misses and “zeroes” to cache hits. The spectrum of this time sequence is used to identify tight loops accessing regular data structures and the general structure of instruction localities. Thiebaut [Thi89] models programs as one-dimensional fractal random-walks and uses the model to predict the behavior of the miss ratio curve for fully-associative caches of varying sizes. Thiebaut and Stone [Thi87] develop an analytic model for cache-reload transients—*footprints in the cache*—to describe the effects of context switches. Lebeck and Wood [Leb94] describe a cache profiling system and show how it can guide code modifications that reduce cache misses.

McKinley and Temam [McK96] take a step towards more detailed analysis by quantifying the locality characteristics of numerical loop nests. Their locality measurements reveal important differences between loop nests and whole programs and refute some popular assertions. They present results as histograms of the locality distributions for the parts of programs in question, whereas our approach provides much more than summary information.

Two recent frameworks share many of the same goals as ours. Ghosh and Martonosi's Cache Miss Equations (CMEs) [Gho98] perform compile-time analysis of loops to generate a system of linear Diophantine equations describing the program's memory behavior such that solutions to these equations represent potential misses in the code. CMEs allow for precise, mathematical, compile-time analysis of cache misses in cache memories of arbitrary associativity, but are currently limited to analysis of loops without interior control-flow like if-then-else structures. Harper, Kerbyson, and Nudd [Har99] extend the cache footprints concept [Thi89] and develop a mathematical framework that permits the determination of cache miss ratios as well as conflicts within loops. They abstract away chance address bindings using equivalence classes (which they call "translation groups"), as do we. Unfortunately, as with CMEs, the analysis is limited to nested loops without internal control-flow constructs. In this respect, our caches-as-filters model is more general: we provide a general framework in which any program behavior can be examined.

7. Conclusions and Future Work

We have described a new analytical framework for studying the behavior of different cache organizations. The framework consists of four components: the TSpec notation, equivalence classes of memory references, the functional cache filter model, and new metrics. This paper demonstrates the use of the first three components on variations of a simple kernel. The examples demonstrate how code sequences (including those with conditional constructs) can be described with TSpec, and they introduce several functions that permit the description of cache effects for arbitrary code sequences and arbitrary cache associativity. More generally, the examples demonstrate how this framework can be used to *understand* cache behavior—for instance, why direct-mapped caches can outperform caches of higher associativity—and how the framework can be used to guide cache-conscious data placement and other optimizations.

The framework is *precise* in that it can exactly describe the cache behavior of a particular program or memory-reference trace, as opposed to approximating it with equations. In addition, the framework can be made *practical*, for the required tasks can largely be automated (e.g., the functional filter model steps can each be automated, as these steps are similar to aspects of cache simulation). The benefit of this approach over simulation is that the designer and compiler writer can see the intermediate steps that point to why a particular cache behavior occurs. In addition, we have a preliminary tool that computes the new measures, displays them in a way that contributes to the user's understanding of the reference string and cache behavior, and allows the user to navigate through the trace [Wei98], and we have experimental tools to automate pattern matching to translate a trace into TSpec. We will soon have a tool to translate a compiler intermediate language to TSpec. Our framework is *general* in that it is not limited to specific code structures such as loops, for it can accommodate control-flow structures, even conditionals. Most importantly, it is *useful* in that it can point to potential solutions and clearly illustrate why a particular cache behavior occurs.

This work suggests a number of avenues for future exploration. Prior work has described two new cache-locality measures [Wei98], but the complementary analytical measures on TSpec examples have yet to be fully developed. We intend to use the framework to completely analyze a range of reference kernels, including transaction processing, mpeg and jpeg kernels, and we will use these analyses to study the impact of compiler optimizations on these kernels. TSpec includes facilities for arbitrary annotation of trace elements [new TSpec reference], and these can be used to track producer-consumer delays, thereby identifying the latency-tolerance of memory references and—in conjunction with the caches-as-filters analysis—guiding the decision of which data to promote into the nearest caches. TSpec is also an excellent environment in which to analyze the behavior of writes and the impact of structures like coalescing write buffers, new write-buffer retirement policies, and write caches, extending the work in [Jou93] and [Ska97]. Finally, many of these analyses will be simplified by future software tools.

References

[Bat76] A.P. Batson and A.W. Madison, “Measurements of Major Locality Phases In Symbolic Reference Strings”, *Proc. International Symposium on Computer Performance, Modeling, Measurement and Evaluation*, Mar.1976.

[Bat77] A.P. Batson, D.W.E. Blatt, and J.P. Kearns, “Structure Within Locality Intervals”, *Proc. Third International Symposium on Modeling and Performance Evaluation of Computer Systems*, 1977.

[Bre96] 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, Nov. 1996.

[Cal98] B. Calder, C. Krintz, S. John and T. Austin. "Cache-conscious Data Placement." *Proc. ASPLOS-VIII*, Oct. 1998.

[Den68] P. Denning, “The working set model for program behavior”, *Communications of the ACM* vol. 11 no. 5, May, 1968.

[Gho98] S. Ghosh, M. Martonosi, and S. Malik, “Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity”, *Proc. ASPLOS-VIII*, Oct., 1998.

[Gri96] K. Grimsrud, J. Archibald, R. Frost, and B. Nelson, “Locality as a Visualization Tool”, *IEEE Transactions on Computers*, vol. 45 no. 11, Nov. 1996.

[Har99] J. Harper, D. Kerbyson, and G. Nudd, “Analytical Modeling of Set-Associative Cache Behavior”, *IEEE Transactions on Computers*, vol. 48 no. 10, Oct. 1999.

[Hil87] M.D. Hill, “Aspects of Cache Memory and Instruction Buffer Performance”, Ph.D. dissertation, Univ. of California at Berkeley, Nov. 1987.

[Jou90] N. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers", *Proc. ISCA-17*, May 1990.

[Jou93] N. Jouppi, "Cache Write Policies and Performance", *Proc. ISCA-20*, May 1993.

[Leb94] A.R. Lebeck and D.A. Wood, “Cache Profiling and the SPEC Benchmarks: A Case Study”, *IEEE Computer*, Oct. 1994.

[McK96] K. McKinley and O. Temam, “A Quantitative Analysis of Loop Nest Locality”, *Proc. ASPLOS-VII*, Oct. 1996.

[Pag99] D. Page, J. Irwin, H. Muller, and D. May, GraphNavigator tool, http://www.cs.bris.ac.uk/Research/LanguagesArchitecture/Predictable/performance/demo.graphNavigator.html.

[Ska97] K. Skadron and D.W. Clark. “Design Issues and Tradeoffs for Write Buffers." In Proc. HPCA-3, pp. 144-55, February 1997.

[Sug92] R.A. Sugumar and S.G. Abraham, “Efficient Simulation of Caches under Optimal Replacement with Applications to Miss Characterization”, *Proc. ACM SIGMETRICS*, pp. 24-35, May 1993.

[Thi89] D. Thiebaut, “On the Fractal Dimension of Computer Programs and its Application to the Prediction of the Cache Miss Ratio”, *IEEE Transactions on Computers*, vol. 38, no. 7, July 1989.

[Thi87] D. Thiebaut and H.S. Stone, “Footprints in the Cache”, *ACM Transactions on Computer Systems*, vol. 5 no. 4, Nov 1987.

[Vol81] J. Voldman and L. Hoevel, “The Software-Cache Connection”, *IBM Journal of Research and Development*, vol. 25 no. 6, Nov. 1981.

[Wei98] Dee A. B. Weikle, Sally A. McKee, Wm. A. Wulf, "Caches As Filters: A NewApproach to Cache Analysis", *Sixth International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (MASCOTS'98), July19-24, 1998, Montreal Canada.

[Wei00] Dee A. B. Weikle, Sally A. McKee, Kevin Skadron, Wm. A. Wulf, “Caches As Filters: A Framework for the Analysis of Caching Systems”, *to appear in Grace Murray Hopper Conference 2000*, Sept. 14-16, 2000.

Appendix A: Glossary

!#c sets the variable *c* to its base address as in its definition; does not generate an address.

!c# sets the variable *c* to the last address used in the TSpec description; does not generate an address.

!#c$_{\#,+}$ sets the variable *c* to its base address plus its second increment value from its definition; does not generate an address.

c+ generates the address corresponding to the current value of *c*; post-increments the value of the variable *c* by its first increment or iterator.

c- generates the address corresponding to the current value of *c*; post-decrements the value of the variable *c* by its first increment or iterator.

c(400; 4, 8) defines the variable *c* with base address 400, first increment four, and second increment eight.

c*n generates an address corresponding to the current value of *c n* times, *e.g., c*4* is the same as *c*, *c*, *c*, *c*.

!c+*n increments the *c* variable by its first increment *n* times; does not generate any addresses in the trace.

(c+*n | (!c+*n)) represents an if statement: only one of the two clauses separated by "|" is executed. For clauses of any visual complexity, subscripts on the parentheses and the "|" separator should be used for clarity.

T_1&T_2 merges two traces, *T1* and *T2*, one element at a time. λ merged with a trace atom is the trace atom, the merge of two λs is λ, the merge of two trace atoms is undefined.

λ functions as a placeholder for removed trace atoms in a trace.

λ_d functions as a placeholder for duplicate items removed from a state representation.