APPROVAL SHEET

This dissertation is submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy (Computer Science)

[illegible]

Dee Ann Burgess Weikle

This dissertation has been read and approved by the Examining Committee:

[illegible]

William A. Wulf (Thesis Advisor)

[illegible]

Kevin Skadron (Committee Chairman)

[illegible]

James H. Aylor

[illegible]

Alan P. Bats~~on~~

[illegible]

Sally A. McKee

[illegible]

Jack W. Davidson

Accepted for the School of Engineering and Applied Science:

[illegible]

Dean Richard W. Miksad

School of Engineering and Applied Science

May 2001

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

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

at the

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Dee A. B. Weikle

May 2001

Lovingly Dedicated To

My Parents

Larry Almon Burgess and DeEtta Kay Shoop Burgess

they have inspired and supported me more than can be imagined

and

To Their Mothers

Lucy Ellen Wilson Shoop and Eva Nell Mitchell Burgess

who taught them how

“It is not the critic who counts; not the man who points out how the strong man stumbles, or where the doer of deeds could have done better. The credit belongs to the man who is actually in the arena, whose face is marred by dust and sweat and blood; who strives valiantly; ... who spends himself in a worthy cause; who at the best knows in the end the triumph of high achievement, and who at the worst, if he fails, at least fails while daring greatly, so that his place shall never be with those cold and timid souls who know neither victory nor defeat.”

- Theodore Roosevelt

address at the Sorbonne, 1910

Abstract

This dissertation describes the *Cache Filter Model*, an analytical framework for cache system analysis. This framework provides a language and formal notation that enables researchers to reason and communicate about systems in an insightful new way. There are four major components that form the framework. First, the *TSpec notation* is a 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 artifacts due to chance address bindings 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 system behavior than current measures such as hit rate or average memory access time. This dissertation presents the cache filter framework in detail and demonstrates its use on several example kernels.

Acknowledgements

I realize that these acknowledgements will both exceed the patience of many readers and yet not do full justice to those I am attempting to thank. I ask that those who would rather not be frustrated by emotional testimonials please move on and those who I thank here, both explicitly and implicitly, know that my appreciation extends beyond that which mere words can convey.

First and foremost I must thank my advisor of seven years, Dr. William Wulf. He has believed in my abilities from the beginning and his insight, encouragement, and dedication to my studies have made a tremendous impact on my work. I thank him for his patience and inspiration. I would also like to thank Dr. Sally McKee and Dr. Kevin Skadron who have each served as co-advisors and mentors over the course of the dissertation. They have each provided input and support at crucial junctures and are very appreciated. Dr. Alan Batson has been an advocate within the department and Dr. Jim Aylor has been very helpful in obtaining partial funding for several semesters. I would also like to thank Dr. Paul Reynolds and Dr. Gabe Robins for their encouragement and support in the very early years of the quest for my PhD. In terms of funding is is also important that I thank the Computer Science Department for finding funding several semesters and to the National Science Foundation (NSF), Award Number POWRE-9806043. (Any opinions, findings and conclusions or recomendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).)

I have several friends within the field of Computer Science that I would like to thank for their constant encouragement and support. Foremost among them is Dr. Allison Powell who has been with me from the beginning and performed countless favors as I have had more children, participated in study groups, listened to endless concerns, and has generally been an excellent friend. Along these lines I would also like to thank Kim Gregg, Sherrie Albrecht, Mary Anne Stumbaugh, Chris Milner, Sally McKee, and the members of Bill Wulf's group over the years.

Outside of Computer Science I have several additional friends to thank. Caroline Ramaley, Jody Hesler, Laura Burgess, and Sabine Schattenberg have given me significant help in the form of babysitting, cooked meals, and general cheerleading. The members of the Westminster Peace and Justice Committee and the Tuesday Evening Bible study have also provided constant support.

The most important component of my life, however, is my family. I was fortunate to be born into one of the most miraculous of these entities I have ever experienced. My parents and my brothers gave me the initial impetus and encouragement to succeed as a woman in a technical field, and they have continued to support me to this day. Without the help of my parents, and my mother in particular, this dissertation would most likely not have been possible, and certaintly not have been finished this semester. Their love and support is something I am thankful for every day of my life.

In addition to being fortunate to have been born into a wonderful family, I have been fortunate to have married into another one. My mother and father-in-law, Patricia and Robert Weikle, have logged countless hours with their grandchildren in support of this venture as well as provided love and compassion to their ever grateful daughter-in-law.

Finally, I am fortunate to have my own family with which to share my daily life. They will all be as grateful for this dissertation to be done as I will. I owe a special debt to each of my children, Elizabeth Morgan Weikle, Madison Taylor Weikle, and Richard Almon Mason Weikle for constantly reminding me about what is really important in life. They have all been born while I toiled on this lengthy project and are a constant source of joy.

The most important person of all for me to thank in these pages is my husband, Bobby Weikle. He is my husband, lover, best friend, and constant advocate. In addition to logging many hours with his children alone, listening to my many complaints and fears about the doctoral process, and generally being there when needed, he has always believed in me and my ability to succeed and has always encouraged me to strive for the highest ideals. I am truly grateful to God for blessing my life with his prescence. He honestly is my “favorite” as I often tell him, and I look forward to many more years by his side. In the words of a rather naive poet, “I want to spend my life with him, just walking hand in hand... .”

Table of Contents

List Of Figures

List of Symbols

Symbol	Definition
a_i	*i*th memory reference in a trace (*i*th address)
as	associativity
d(a)	dirty line in cache that contains reference a
δi	in instantaneous hit rate, 0 if *i* is miss, 1 if *i* is hit
ecl	effective cache lines
esn	effective set number, $\frac{sn}{gcf(st,ls) \times gcf(st,sn)}$
fa	fully associative
$\overline{fa}$	"not" fully associative
gcf	greatest common factor
h_i	instantaneous hit rate
λ	placeholder for filtered or decomposed references in a TSpec description

Symbol	Definition
$\overline{\lambda}$	“not” λ, symbol for non-λ elements in trace
$l(a_i)$	instantaneous locality of a_i
$L(T)$	instantaneous locality of whole trace T
l_i	instantaneous locality
lps	lines per set, or line count
LRU	Least Recently Used
ls	linesize
σ	smoothing factor in instantaneous hit rate, 1/2
st	stride of a vector
sz	cache size, in lines
w	window size for instantaneous locality
wb	write-back
wp	write policy
wt	write-through

List of Functions

Function	Description
$d(a)$	the dirty cache line that the word *a* maps to
$D(T)$	deletes all λs from T, leaving only non-λ trace atoms
$D_x(T)$	deletes all λ_xs from T, leaving any other λs or trace atoms
esn	effective lines per set, $\frac{sn}{gcf(st,ls) \times gcf(st,sn)}$
$l(a)$	the cache line that the word *a* maps to
$U(T)$	the unique of T, formed by removing duplicates
$U(\overline{T})$	the unique of the reverse of T, formed by reversing T and removing duplicates
$\|T\|$	the number of trace atoms in T, including λs
$\overline{T}$	the reverse of T, formed by reversing the order of the trace atoms in T
$\Lambda(T)$	replaces all non-λ elements in T with $\overline{\lambda}$, (i.e. “not” λ)

Δ	TSpec "fill", T1 Δ T2 means fill all the non-λ ($\bar{\lambda}$) elements in T1 with consecutive elements of T2
$L(T)$	instantaneous locality of entire trace T
$i(t)$	index of t in cache
$l(a_i)$	analytical instantaneous locality of address a_i

TSpec 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.
$\wedge\mathbf{c}_{\#,+}$	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.
c-	generates the address corresponding to the current value of c; post-decrements the value of the variable c by its first increment.
c(400; 4, 8)	defines the variable c with base address 400, first increment 4, and second increment 8.
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.

Chapter 1

Introduction

This dissertation describes the *Cache Filter Model*, an analytical framework for cache system analysis. This framework provides a language and formal notation that enable researchers to reason and communicate about systems in an insightful new way. There are four major components that form the framework. First, the *TSpec notation* is a 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 artifacts due to chance address bindings 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 system behavior than current measures such as hit rate or average memory access time.

1.1 Cache System Hierarchy

Figure 1 depicts a simplified diagram of the portion of the computer system closest to the CPU. The model, notation, and measures discussed in this dissertation may be used for other levels of the memory hierarchy, but we focus on the *caching system*. A cache is a memory located close to the CPU. Whenever the CPU issues a memory reference, the cache checks to see if it contains the appropriate value. A *cache hit* occurs when the value is found in cache. A *cache miss* occurs when the value is not in cache and must be fetched from a cache farther away from the CPU or from main memory. Caches are faster than main memory, and so more expensive. For that reason they are also smaller than main memory. Caching systems typically consist of a cache hierarchy, with one or more caches arranged in sequence to service the CPU. The smaller, faster caches are closer to the CPU with the first level almost always on the same chip.

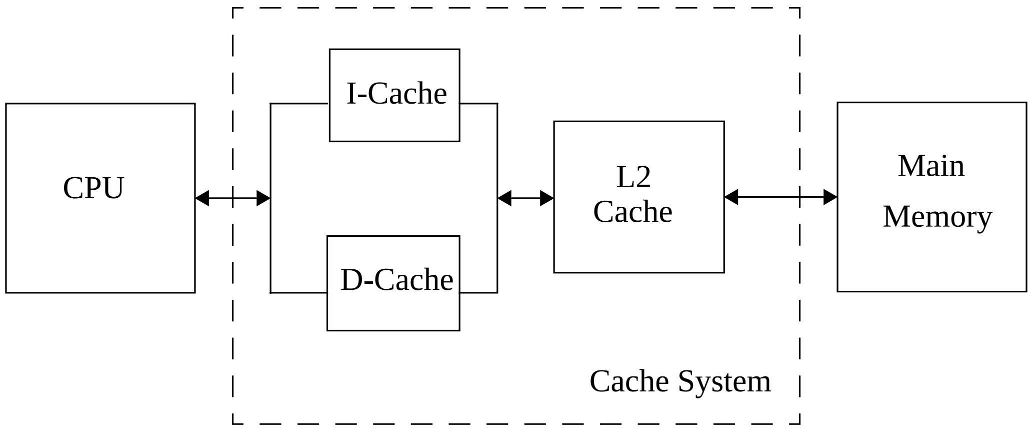


FIGURE 1: Cache System Hierarchy

Caches are arranged as an array of memory locations. Typically, they exploit the principle of *locality of reference* by fetching a fixed amount of data contiguous to the referenced value. The assumption is that whenever a memory location is referenced, it is likely that the referenced location or nearby locations will also be referenced in the near future. Caches can vary widely in their organization, but we concentrate here on direct mapped, set associative, and fully associative caches of varying size with an LRU line replacement policy.

The associativity of a cache determines how the memory locations (lines) are arranged and accessed. In a *direct mapped cache,* an address will map to only one line in the cache. Usually that mapping is *(address)* mod *(number of lines in the cache)*. In a *set associative cache*, an address will map to a set of lines. The number of lines in that set will depend on the associativity of the cache. For example, a two-way set associative cache will have two lines in each set. The mapping for the address in associative caches is usually (*address*) mod (*number of sets in the cache*). In a *fully associative cache* an address can map to any line in the cache. The range of cache associativities is really a continuum of levels of set associativity. Direct mapped caches can be thought of as having an associativity of one. Fully associative caches with *n* lines can be thought of as having an associativity of *n*.

The line replacement policy of a cache determines the line of the data that will be removed from a cache when it becomes completely full. The *Least Recently Used (LRU)* replacement policy replaces the line that was least recently referenced.

1.2 The Cache Design Problem

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 [Bur95, Hen96, Jou97, Wul94]. Processor speeds are increasing much faster than memory speeds. While microprocessor performance has improved steadily at an annual rate of 55% since 1987, DRAM performance has increased at an annual rate of less than 10% [Hen96]. This disparity has caused memory to become the performance bottleneck for many applications. Not only is the current problem serious, but it is growing at an exponential rate.

Every time there is an increase in the speed of a microprocessor, the cache and corresponding memory system must be redesigned to meet the need for faster delivery of instructions and data. There continues to be research and improvement to cache functionality. Smith [Smi82] and Hennessy [Hen96] provide an excellent survey of this research. Typically it 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 propose a modification to the cache hierarchy and then judge that design by running benchmarks through a simulator to determine “hit rates” or average memory access times.

While the above approach has yielded many improvements to the performance of caching systems, it is primarily ad-hoc experimentation with little theory to guide new designs. This hampers the ability of the cache designer to effectively design to specific performance points, or fully understand the impact of research results on actual systems, or even to know if a slight variant of the proposed modification would have better performance. For example, the interaction between specific features such as out-of order execution, branch prediction, pre-fetching, or cache replacement algorithms in the real-time execution of a user application is unclear. Optimizing each one separately may not necessarily lead to a global optimum. In addition, it is difficult to control all the parameters one needs to perform an experiment to isolate the effects of any one of these features.

A final complicating factor is that the current approach to cache design depends on benchmarks and a simulation infrastructure that are non-standard (where different researchers use different compilers, linkers, instructions sets, etc.). These benchmarks and simulators were developed primarily for the purpose of evaluating processor architectures. While being extremely useful and appropriate for analyzing the effects of many CPU optimizations, they do not provide a unified or complete experimental infrastructure that includes memory hierarchy design. An analysis framework, such as proposed here, would allow researchers to abstract away from a particular CPU environment to communicate ideas about the fundamental characteristics of memory systems.

In addition to the lack of unifying theory, cache design is complicated by a lack of measures appropriate for evaluating modern systems. For example, in *Computer Architecture: A Quantitative Approach* by Hennessy and Patterson [Hen96], caching systems are evaluated using metrics such as (1) the aggregate *miss rate*, or fraction of accesses that cannot be serviced by the cache, (2) the execution time of a benchmark, and (3) average memory access time. These metrics are considered by many to be appropriate cache evaluation metrics; this text is a widely used architecture text for both graduate and undergraduate course work.

There are problems with using these measures, however. Evaluating a caching system based only on hit rate ignores the fact that components comprising main memory no longer have a uniform access time for every sequence of requests. Features such as *fast- page mode*[1] allow certain combinations of references to be retrieved faster than others. If some accesses are faster than others, it is possible for a cache with a lower hit rate to provide better performance than a second cache with a higher hit rate. The first cache's misses could be faster memory accesses and require less total time to service than the misses of the second cache. Evaluating based only on average memory access time ignores the fact that memory accesses are generally bursty in nature and difficult to spread out evenly.

Metrics that make use of the characteristics of the references presented to the cache provide greater insight and guidance in the design of cache systems than hit rate, average memory access time, or even execution time. This assertion is motivated by a simple observation: two caches with precisely the same hit rate (or average memory access time or execution time) may achieve that performance in quite different ways. These differences have important implications for certain aspects of modern cache design, multi-level caching systems for example.

1.3 A Motivating Example

A brief example will clarify the shortcomings of hit rate. Consider a loop that accesses memory for two vector elements and two global variables in each iteration. All other code and data references reside in registers. Let 0 and N be the addresses of the global data and

[1] Fast-page mode devices behave as if implemented with a single, on-chip cache line, or page. A memory access falling outside of the address range of the current page forces a new one to be set up, a process that is significantly slower than repeating an access to the current page.

let 1 through N-1 represent the addresses of the vector elements. The expression below represents this loop within a second larger loop.

(0, N, 1, 2, 0, N, 3, 4, 0, N, 5, 6,..., N-2, N-1, 0, N)*.

The first iteration of the inner loop generates the addresses 0, N, 1, 2. The ()* indicates an indefinite number of repetitions of the pattern inside the parentheses, as in Kleene's *. While this is a contrived example reference sequence, it is useful to illustrate a few points.

The references that miss in steady state when this example is presented to a direct-mapped cache of size N and a fully associative, LRU cache of size N are the same in number, but have completely different characteristics. In the direct-mapped cache, 0 and N conflict, but all the references to the vector fit in the cache. As a result, the misses of the direct-mapped cache are (0, N)*. For the fully-associative LRU cache, however, 0 and N stay in the cache, but the vector references always miss. The resulting misses are (1, 2, 3, 4,...)*. In both cases there are exactly two misses per loop iteration after the caches are primed, but the references that miss in the direct-mapped cache are easily captured by a small victim[2] cache. Those that miss in the fully-associative cache need an N-1 size structure to capture.

This does not mean the direct-mapped cache is generally better, but rather that the effectiveness of a caching system is not fully described by hit rate. This is true even for a single-level cache hierarchy. In the above example, if the two reference streams are going directly to a main memory composed of DRAM with fast-page mode, it may be preferable for it to see the reference stream generated when a fully-associative cache is used. Fast-

[2] A victim cache is a small fully associative cache at the output of another larger cache that captures the conflicts (or victims) of the larger cache. See [Jou90] for details.

page mode devices behave *as if* implemented with a single, on-chip cache line, or *page*. A memory access falling outside the address range of the current page forces a new one to be set up, a process that is significantly slower than repeating an access to the current page. Because vector references will often hit in the row- or page-buffer, where they can be accessed more quickly than access sequences without spatial locality, the sequential accesses may be faster than the accesses to N if N is on a different page than 0.

1.4 The Cache Filter Model

The analysis approach described here is inspired by viewing a cache as a filter. As depicted in Figure 2, 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 "shape" the unfiltered references in a way that makes the next level most effective. Then, for example, in the contrived example presented earlier, a cache configuration could be chosen to maximize the effectiveness for this reference sequence of main memory (by choosing a fully-associative cache), or the effectiveness of a second level cache (by choosing the direct mapped cache).

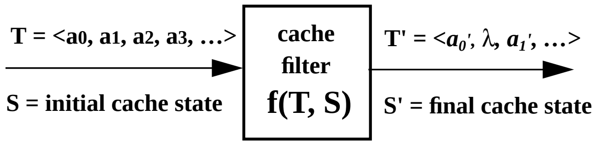


FIGURE 2: The Cache Filter Model

To formalize what is meant by the Cache Filter Model, a few definitions are necessary. 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. 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. 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 $\langle a, a, a\rangle$ generates the output $\langle a, \lambda, \lambda\rangle$ 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 2 illustrates this relationship. The trace-only portion of the output of a filter function is denoted $f^T(T;S)$, and the state-only portion is denoted $f^S(T;S)$.

In analogy to signal processing, the reference string corresponds to a signal and the cache to a filter. Comparing a cache's input and output signals reveals what kind of filtering effect the cache has on that input. The signal processing analogy has appeal, but it only goes so far: tools like Fourier analysis and Laplace transforms don't immediately apply, because caches are not linear or time-invariant. [3]

Because traditional transforms do not give significant insight into cache design, we developed a new transform that is specific to cache design and simplifies modeling of the cache in the new domain. The domain that we transform reference strings to is the TSpec notation described in Chapter 2. Viewing reference strings as combinations of primitive reference patterns allows us to determine the overall effect of a cache on that reference string in a straightforward manner.

1.5 Summary

We claim it is possible to design and evaluate cache memory systems more effectively than is currently done. Specifically, we propose an analytical framework for cache design that provides a common notation for expressing the memory references of a program, a functional cache filter model for comparing traditional cache hierarchy effects, and new *instantaneous* evaluation metrics that give greater insight into the operation of caches. The functional cache filter model is made possible by the concept of an equivalence class of memory references that provides an abstraction for eliminating certain types of random address placement effects. In addition to aiding in the design of improved cache systems,

[3] Linearity requires that an input equal to the sum of two other inputs generates an output equal to the sum of the corresponding two outputs. In terms of a cache, this would require that earlier references *not* affect the output of later references, meaning that the cache has no state.

we hope this approach will lay a foundation for more rigorous analysis of other components of the computer system.

Our intent here is to demonstrate that this type of analysis can be done. The analysis itself can be facilitated in many ways through the use of automation, or software tools designed to assist the researcher. Tools to generate TSpec from source code, perform the filter function table look up, perform the state-trace merge, or compute the alternative measures are all examples of such automation. While these tools are beyond the scope of this dissertation, they will be mentioned when appropriate and are discussed further in the Future Work section of Chapter 6.

1.6 Organization of the Dissertation

This dissertation describes an analytical framework for cache analysis consisting of four components: the TSpec notation, the concept of an equivalence class, a functional filter model of cache behavior, and new cache metrics. The dissertation is organized around these four components. Chapter 2 describes the TSpec notation and the equivalence class concept in detail, including a grammar for the notation. Chapter 3 describes the functional filter approach to cache analysis. Chapter 4 describes two new metrics for cache evaluation, instantaneous locality and instantaneous hit-rate. Chapter 5 demonstrates application of the framework on several kernel examples, and a multi-level cache example. Chapter 6 is the concludes the dissertation. The functional filter dictionary, whose use is described in Chapter 3, is available in Appendix A. Appendix B is a listing of the C code, assembly language and TSpec for all examples used in the dissertation. A glossary of the TSpec notation, a list of Symbols, a list of functions, and a list of figures are available in the front of the dissertation for quick reference. Related work is addressed at the end of each chapter.

1.7 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 of that program in fully-associative caches of varying sizes. Thiebaut and Stone [Thi87] develop an analytical model—“footprints in the cache”—for cache-reload transients 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 [McKK96] 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, but present results as histograms of the locality distributions for the parts of programs in question. In contrast, our approach provides much more than summary information.

Jacob, Silverman, Mudge and Chen [Jac96] develop a mathematical model for determining the optimal size of each level of a cache hierarchy given a certain budget by applying a specific, parameterized model of workload locality. The model is verified against exhaustive simulation of two case studies and in all but one case, the model performs within 5% of optimal. This model is useful for quickly determining the optimal cache sizes for a hierarchy, but relies on a workload characterization model of locality that may not be appropriate for all applications. In addition, no information about other aspects of the caches, such as associativity or line size, is provided. Our approach characterizes the reference string in addition to several aspects of the cache configuration.

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 structures. Harper, Kerbyson, and Nudd [Har99] extend the cache footprints concept [Thi87] and develop a mathematical framework that permits the determination of cache miss ratios as well as conflicts within loops. As in our work, they abstract away chance address bindings using equivalence classes (which they call "translation groups"). Unfortunately, as with CMEs, the analysis is limited to nested loops without internal control-flow constructs. The caches-as-filters model is therefore more advanced in terms of providing a general framework in which any program behavior can be examined.

Chapter 2

TSpec: A Notation for Memory Traces

2.1 Introduction

Interpreting patterns in a processor's output references is complicated by the lack of a succinct notation for human use. Since a reference trace is simply a long list of addresses, it is difficult to see the underlying patterns inherent in it. The source code, while simpler to look at, does not include the effects of compiler optimizations such as loop unrolling, or of linker/loader/compiler/variable address mappings and so can be misleading as to the actual references and order seen by the memory system. To simplify communication of traces between researchers and to understand them more completely, we have developed a notation for representing them that is easy for humans to read, write, and analyze. This notation is called *TSpec*, for trace specification notation.

TSpec has been designed for use in memory hierarchy design with four goals in mind. First, it is intended to assist in communication between people, especially with respect to understanding the patterns inherent in memory references traces. Second, it is the object on which the cache filter model operates. Specifically, the trace and state of the cache are represented in TSpec, these are then the inputs for a function that models the cache, and the result of that function is a modified trace and state that are also represented in TSpec. Third, it supports the future creation of a machine readable version that could be used to generate traces to drive simulators, or for use in tools (such as translators from assembly language to TSpec). Finally, it can be used to represent different levels of abstraction in benchmark analysis.

This work focusses on four different levels of abstraction. Current cache analysis is based on single traces, so TSpec can specify an individual trace. However, there are many accidents of address binding by the compiler or loader in such a trace. It is desirable to be able to analyze all the traces that differ only in those artifacts of binding, or as we call it, the *equivalence class* of traces that differ only because of those binding artifacts. So TSpec has been designed to describe the abstraction of an equivalence class under varying address bindings. Similarly, there are many traces that result from the execution of a single program that, given different input data, follows different execution paths through the program. Here again, TSpec can represent this abstraction. Finally, there exists a set of traces that result from both different address bindings and different input data.

Section 2.2 through Section 2.4 which follow describe the TSpec constructs that are used to describe a single trace, where all the address bindings and the path through the program are known. Descriptions of the other levels of abstraction and the TSpec constructs to support them begin in Section 2.5.

2.2 TSpec Constructs for Single Traces

A *trace specification* is a formal rule that describes a specific trace. It consists of a set of *definitions* followed by a *trace list*. Definitions can be either *variable* or *subtrace* definitions. A trace list is a concatenation of trace atoms, the most basic TSpec construct, surrounded by angle brackets (<>). Trace atoms are *concatenated* by separating each trace atom with a comma. The simplest example of a trace specification has no definitions and only one trace in the trace list. It is a list of address references such as:

<100, 200, 104, 300, 108, 100, 204, 104, 304, 108, 100, 208, 104, 308, 108>

A *trace atom* may be (1) a literal, (2) the symbol λ, (3) a variable, or (4) a subtrace. A literal is an integer with an optional attribute tag. The integer represents an address, and may be specified in decimal or hexadecimal; hexadecimal numbers will be preceded by 0x. Attribute tags other than read/write are not defined by TSpec, but are intended to connote code vs. data, system vs. user, etc. Non-null attribute tags will be represented by an underscore and one or more letters appended to the end of an integer representing the address — thus `100_r` is an integer/tag pair. The read vs. write attribute will be used consistently throughout this document. Read is denoted by _r and write by _w.

λ is used as a placeholder for an integer/tag pair. Its primary role is to maintain the relative order of the integer/tag pairs, which represent the event of a memory request. Since λ is a placeholder for a memory request, and not a clock tick, it does not represent the passage of time. There may be more wall-clock time pass between two consecutive integer/tag pairs than two that are separated by one or more λs. This would simply mean that the memory requests represented by the λs occurred closer together than the consecutive integer/tag pairs.

Variables and subtraces are constructs that allow regular patterns of literals to be described compactly. Variables are described more fully in Section 2.3, and Section 2.4.1 and subtraces in Section 2.3.2.

In the paragraphs that follow we will use two different words to describe what is represented by a TSpec definition. The first is “elaborate”, which we use to refer to the set of all traces that a TSpec specification could be used to represent. The second is “execute”, which we use when we produce a specific trace from the set of all traces. Formally, an instance of a TSpec definition represents a set of traces - all the traces that result from its elaboration. Informally, we speak of “executing” the definition to produce a particular trace in this set. There is a certain amount of useful ambiguity in this duality of terminology. The overall intent should be clear, however, from the context.

2.3 Simple Variables and Operations on Variables

More will be said about variables later, but here we introduce the basic concept and the simple operations on them. A *variable* represents a regular sequence of addresses and is specified by a base address with the appropriate attribute, and an increment or decrement (stride). An example of a variable definition is ***x(400_r, 8)***. ***x*** is the name of the variable, 400 is the value of the base address, _r indicates that it is a read, and 8 is the value of the increment. A variable can be *initialized* (denoted ***!x***) to set its current value to its base address. A variable can also be *post-incremented* (denoted $\boldsymbol{x}_+$) to add the increment to its current value or *post-decremented* (denoted $\boldsymbol{x}_-$) to subtract the increment from its current value. Each time a variable occurs, it generates an address unless it is being initialized. Note that the + and - operators are subscripts. The reason for this will be explained later.

2.3.1 Definite Iteration

A list of trace atoms can be grouped together with parentheses and an optional label. In this way the group is set apart to assist in pattern identification, or to be operated on. The format of a parenthesized group is ***(100_r, 200_r, 300_r)*** or ***$(_{LABEL}$ 100_r, 200_r, 300_r$)_{LABEL}$*** or ***LABEL:(100_r, 200_r, 300_r):LABEL***. Both possibilities are included because there are times when one will be significantly easier to read than the other. The notation that results in the clearest specification is the one that should be used. Note that a LABEL must be unique within a TSpec specification.

A trace atom or a group of trace atoms can be repeated with the *iteration* operator, *. For example, x*4 repeats the value represented by x, 4 times. ***$(_L$x+$)_L$*4*** generates the value of x and post-increments x four times. If the initial value of ***x*** was 100 and the increment 4, the above would generate the trace ***<100_r, 104_r, 108_r, 112_r >*** and the next time x was used, it would generate the address 116_r.

2.3.2 Subtraces

Often it is useful to separate a part of a trace that is reused frequently and use a label to refer to it rather than specifying the whole trace. This makes larger patterns in the reference stream more obvious. It is also useful to have parameters for these subtraces in the instances where subtraces are the same except for one or two positions. The definition of a subtrace has the form:

s(p1, p2) = <100_r, p1, 104_r, p2, 108_r>

The name of the subtrace is s. p1 and p2 are parameters to the subtrace and following the = symbol is simply another trace specification. When using the subtrace in a trace, the

% operator indicates the subtrace should be fully elaborated (i.e., substituted as a whole into the trace). The specification

s(p1, p2) = <100_r, p1, 104_r, p2, 108_r>
< !s, %s(200_r, 300_w), !s, %s(204_r, 304_w)>

would generate the following trace:

<100_r, 200,_r 104_r, 300_w, 108_r, 100_r, 204_r, 104_r, 304_w, 108_r>

The parameters of the subtrace are substituted textually, and then evaluated when that element of the subtrace is executed. This allows variables to be used as parameters; these variables are then evaluated in the context where they are substituted, and thus can evaluate to different addresses. For example, the above trace could also be generated by the following trace specification:

s(p1, p2) = <100_r, p1, 104_r, p2, 108_r>;
f(200_r, 4); t(300_w, 4);
< !f, !t, !s, %s(f_+, t_+), !s, %s(f_+, t_+)>

There are two modes of execution for a subtrace. The first is to run the subtrace from beginning to end without any intervening references. This is the mode demonstrated in both of the previous examples, is called *running* the subtrace, and is denoted with a % symbol preceding the name of the subtrace (***%s(f_+, t_+)***).

The second mode of address generation for a subtrace is called *pulsing* a subtrace and is denoted with an @ symbol before the name of the subtrace (***@s(p1, p2)***). Each subtrace has a control pointer which operates like the control pointer of a program. When a subtrace is “run”, the control pointer moves from the beginning to the end of the subtrace as each element is executed. When a subtrace is pulsed, the control pointer is moved one element and only the address(es) associated with that element are generated. The next time the subtrace is pulsed, the control pointer is moved one more element, and so on. To set the control pointer of a subtrace to its beginning the subtrace is initialized with !, just like a variable.

d = <200_r, 300_r, 204_r, 304_r>;
c(100_r, 4);
< !d, (!c, c_+, @d, c_+, @d, c_+)*2>

The above specification generates the trace below.

<100_r, 200_r, 104_r, 300_r, 108_r, 100_r, 204_r, 104_r, 304_r, 108_r>

2.3.3 Merge

The *merge* (denoted t1 & t2) of multiple traces is formed positionally, one "address" (or variable) at a time. The merge of a single address with any number of λs is defined to be the address. The merge of any number of λs is defined to be λ. The merge of more than one address is undefined and should never occur. For example, < a_1, λ, a_3, λ> & < λ, a_2, λ, a_4> = < a_1, a_2, a_3, a_4>. 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.

```
 <a1,  λ, a3,  λ>
&<λ, a2,  λ, a4>
--------------------
 <a1, a2, a3, a4>
```

FIGURE 3: Merge Example

2.4 Specific Example

So far we have discussed the basic element of TSpec (the trace atom), how a variable can represent a trace atom, and several operations including:

- definite iteration of a variable or a trace (*),
- initialization of a variable or a subtrace (!),
- variable post-increment or decrement (+,-),
- merge of multiple traces (&), and
- concatenation of variables to form a trace (,).

Figure 3 shows a more complete example for a simplified version of the inner loop of a routine to copy a vector from one location to another. The code, denoted by the variable c in this example, has been simplified to allow the pattern to be easily seen in the reference string. One might think of the first c_+ as the address of the instruction to load the element being copied, the second as that of the store to the new location, and the third as that of the branch back for the next element. Notice that this TSpec description represents a very specific trace *T*, as the address bindings and the specific path through the program (number of iterations in this example) are known.

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

TSpec: **c(100_r, 4); f(200_r, 4); t(300_w, 4);**
$\mathbf{<!f, !t, (_L!c, c_+, f_+, c_+, t_+, c_+)_L*3>}$

Reference
String: **100_r, 200_r, 104_r, 300_w, 108_r,**
100_r, 204_r, 104_r, 304_w, 108_r,
100_r, 208_r, 104_r, 308_w, 108_r

FIGURE 4: Copy Example

2.4.1 Trace Variables In More Detail

Now a more complex example can be created by introducing a more general definition and use of *trace variables*. Variables and the increment operator as introduced in Section 2.3 are adequate for describing any trace, but they are most convenient for single-dimensional arrays. We shall therefore extend the definition and increment/decrement syntax to more naturally handle multi-dimensional arrays and multiple code segments.

In the abstract, a variable has a single base address and several increments. Each increment has three components. The *increment value (iv)* determines the size of the increment or stride of a vector access and appears only in the definition. The *increment count (ic)* determines how many of a particular iterator value is added to the base address to get the current value of the variable. The *increment operator (io)* is used to change the value of the increment count. There are four increment operators. ! clears the increment count to zero. + post increments the appropriate increment count by one, - post decrements the approriate increment count by one, and ~ leaves it alone.

The increment values appear only in a variable definition. A complete definition has the form ***variable_name(base, iv1, iv2, …)***. For example, ***x(200_r, 4, 64)***, has the variable name of x, a base address of 200 (is a read), and two increments. Increment 1 has a value of 4 and increment 2 has a value of 64.

The increment counts and their operators appear in variable instances in a trace as a comma delimited subscript to the variable name, called the *control tag string*. The position of the increment count and operator pair in the control tag string indicates which of the increments will be used to increment or decrement the variable. The characters in the first position describe what will happen with the first increment, the ones in the second position what will happen with the second increment, etc. Convention is that the increment that changes fastest is closest to the variable. The general form of a variable instance is $\boldsymbol{variable\_name}_{\boldsymbol{ic1io1,ic2io2,ic3io3,\ldots}}$, where icX stands for increment count X, and ioX stands for increment operator X. For example, $x_{2+,3+}$ would set the first increment count to 2, and the second increment count to 3, then post-increment each of the increment counts by one.

The value generated by a variable instance is its value after the last time it was used. The value of a variable is given by the following formula: $var = base + (iv1 \times ic1) + (iv2 \times ic2) + \ldots$ where *var* is the current value of the variable, *base* is the base address of the variable in the definition, *ic* represents the increment count for a specific increment, and *iv* represents the increment value for a specific increment. Incrementing an increment count increases the value of the variable by the value of the corresponding increment in the variable definition because it increases the corresponding increment count by one. For example, with a definition of x(100_r, 4), x_{3+} would generate the value 112_r, and change the value of the variable to 116_r.

In practice, it is rare to be incrementing more than one increment at a time, or to need to specify the increment count. For convenience then, this general notation, $\boldsymbol{variable\_name}_{\boldsymbol{ic1io1,ic2io2,ic3io3,\ldots}}$, is simplified with the following assumptions:

1. Increment counts and the comma-delimiters may be left out of the control string of a variable instance. If the increment count is left out, it is assumed that it is the same as the last time the variable was used.
2. Operators left off of the end of a control string are assumed to be ~. So x_{+} is equivalent to $x_{+\sim\sim\sim\ldots}$. (Notice that this is different than if there is no control

string at all. See Section 2.7 for details.)

With this notation !c, c_+ can be expressed as c_{0+}. The ! notation for initialization is still required with subroutines and is also useful when initialization is separated from the first address generation as in the first example below. Initializing a variable to its base address with ! also sets all the increment counts to zero.

Generally, multiple increments are used to traverse a data structure such as a matrix with different strides and the number often reflects the number of loop nests used to traverse the structure. As an example of this usage, consider the data portion of matrix multiplication, x * y = z:

x(1000_r, 4, N);
y(4000_r, 4, N);
z(8000_w, 4, N);
<!x, !y, !z, ((($x_{+\sim}$**,** $y_{\sim+}$**, z_**$r_{\sim\sim}$**,** $z_{\sim\sim}$**)*N-1,** $x_{!\sim}$**,** $y_{+!}$**, z_**$r_{\sim\sim}$**,** $z_{+\sim}$**)*N-1,**
$x_{!+}$**, z_**$r_{\sim\sim}$**,** $z_{!+}$**,** $y_{!!}$**) *N>**

Here the innermost loop does a read of a row from x and a read of a column from y, multiplying them and keeping a running sum of the products in z. z is traversed completely once. Each element of z is computed completely before going on to the next one. y is traversed by “column” (the second increment first) while x is traversed by “row” (the first increment first).

The above example does not use increment counts in the control tag. Increment counts are generally used in the control tag to model a program counter for code jumps. These are not necessary until conditionals are introduced. An example of such a program counter model is shown later in Section 2.6.

The attributes of a variable are generally specified as attributes of the base address and are modified in a logical way for each address generated by the variable. In some situations it may make sense to override the attribute for a specific variable in a trace as is done in the matrix multiply example above. In this case, the trace variable may be followed by an underscore and an attribute tag in the trace.

The following sections provide a more complete description of equivalence classes and the more advanced TSpec constructs to support them. At the end is a complete grammar for the TSpec notation.

2.5 Equivalence Classes In Depth and the Larger Picture

As described in the introduction, computer architects typically evaluate cache designs on a few specific traces. These traces are generated by running a particular benchmark suite on a simulation of their new design. This trace is then compared to another single trace resulting from a system that is as close as possible to the same, but without the particular improvement under investigation. While there has been significant work and improvements to benchmarks in the last decade, these results are still point solutions for a whole range of possible results for a specific piece of source code. For example, the results from benchmark runs are for one set of input data, one set of address bindings, and one compiler. Another instance of any of these parameters may conceivably produce drastically different results. To make it clear where a trace has come from and what type of dynamic run it corresponds to, we have developed the concept of equivalence classes of reference traces, or different levels of abstraction as described in the introduction.

If two traces are generated by the same piece of source code, then they will share at least one equivalence class, but which one may vary depending on one or more parameters that are used to describe the equivalence class. For our work we have broken down the set of traces that could be generated by a specific piece of source code into four sets, depending on whether or not the address bindings are known and whether or not the input data is known. The relationship between these groups is shown in Figure 5. The specific trace generated by a benchmark could be thought of as the trace *T*. The set of traces that would be generated with the same source code and the same set of bindings as *T*, but with different sets of input data is denoted $\{T_d\}$, and is referred to as the *equivalence class* of traces *with respect to input data*. $\{T_b\}$ is referred to as the *equivalence class* of traces *with respect to address bindings*, and is the set of traces that has the same source code and input data as *T*, but a different set of address bindings. (Note that this is a generalization of the concept of translation arrays in Harper *et al.* [Har99]) $\{T_{bd}\}$ is the set of traces that has the same source code, but varies the address bindings and the input data. Other equivalence classes exist, such as the equivalence class of traces under varying virtual to physical address bindings, but we do not treat them in our work.[1]

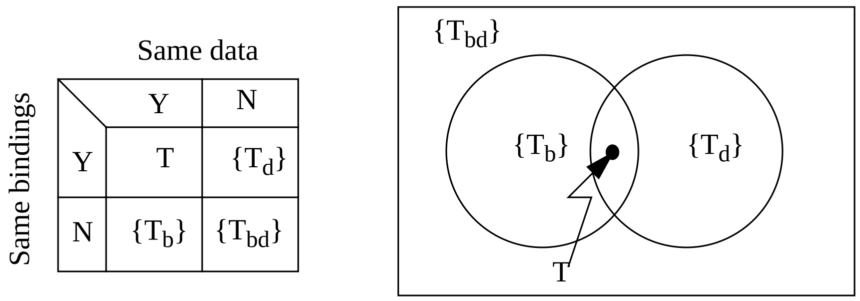


FIGURE 5: Equivalence Classes
The relationship between traces generated by a specific source program by varying bindings only ($\{T_b\}$), input data only ($\{T_d\}$), or both bindings and input data ($\{T_{bd}\}$).

[1] While examining kernels, it is unlikely that a difference would appear between virtual and physical addresses. The actual addresses may differ, but the form of the reference pattern would be the same unless a page boundary was crossed.

Each of the four sets shown in Figure 5 has corresponding constructs in TSpec. A specific trace, T, can be fully described using the constructs shown in Section 2 above. The set of traces $\{T_b\}$, the equivalence class of traces under varying address bindings, can be represented by the constructs described in Section 2, but without the specific base addresses. Consider the copy example in Figure 4. By substituting ? for the specific base address information, one specification can describe a whole class of traces that includes all possible mappings of the code (variable c), and the two data streams (f and t).

c(?_r, 4); f(?_r, 4); t(?_w, 4);
<!f, !t, (c_{0+}, f_+, c_+, t_+, c_+)*3>

Depending on the cache designer's goal, this form of TSpec can be used to do a case analysis of what different mappings would mean to cache performance, or to allow the cache designer to abstract away from those side effects, choose a representative trace for the class as a whole, and understand how a particular cache system would handle the basic reference pattern.

2.6 Expanding TSpec to Describe $\{T_d\}$ and $\{T_{bd}\}$

To describe the equivalence class of traces under varying input data, a way to express conditionals is required. Specifically, support for trace patterns generated by procedural case statements (if-then-else) or indefinite iteration of loops is needed.

2.6.1 Case Statements

An if-then-else clause is a special case of a procedural case statement. To support the description of several different execution paths through a case statement, we use a parenthesized group of trace items separated by |. This denotes a set of smaller traces; only one of which is executed. For example:

c(1000_r, 4);
<c_{0+}, c_+, c_+, {c_+ | c_5 }>

would generate one of the following two trace lists, or the set (equivalence class) containing them both.

<1000_r, 1004_r, 1008_r, 1012_r> or <1000_r, 1004_r, 1008_r, 1016_r>

The above example describes a code segment where the last statement executed is an if-then-else. The then clause is represented by executing the last c_+, and the else clause by c_5. Here, either the c_+, or the c_5 is executed, but not both. For clarity, it is sometimes desirable to label the conditional symbols (|) and the curly brackets that designate the set of possible statements so that their relationship is obvious. Labels are similar to those for parentheses. For example:

c(1000_r, 4);
<c_{0+}, c_+, c_+, $\{_L c_+ |_L c_5 \}_L$>

When conditionals are used, it sometimes becomes necessary to change the increment count of a variable without actually generating that variable. Specifically, there are times when an array pointer may be incremented, but because of a conditional, sometimes accessed and other times not accessed. In those cases, an instance of the variable is preceded by ^ to suppress generation of the address. For example, ^c_+, increments the first increment count of variable c by one, but does not generate an address value.

2.6.2 Indefinite Iteration, Break, and ExceptLast

To represent indefinite iteration, the * symbol is used for loops without a corresponding number to represent the number of iterations. For example, x* means zero or more repetitions of x.

The break construct is analogous to the break in C and continues the specification after the right parenthesis with the same label as the break. For example, consider the above example with a loop and break added.

c(1000_r, 4);
$<(_R c_{0+}, c_+, c_+, \{_L c_+ |_L c_{5+}$, break R $\}_L)_R$*, $c_6>$

Here, if the second case option is taken, the outer loop labeled R is exited and execution continues with the last c_6.

In many cases, it may be unclear how many times a particular loop is executed, but clear that a particular branch case in the loop is executed every time except for the last time through the loop when the exit condition is reached. These cases are so frequent, it is useful to have a specific construct for modeling them. This is done in TSpec by using the key word *exceptlast* and following it with the frequently taken branch code in parentheses.

Consider the following example:

c(1000_r, 4);
$<(c_{0+}, c_+, c_+$, exceptlast(c_+))*, $c_4>$

Here the c_+ in the exceptlast parentheses will generate an address all but the last time through the loop.

These four constructs; case options (|), indefinite iteration (*), break, and exceptlast allow us to model very general paths through a piece of source code. From another perspective, these constructs allow us to describe sets of possible traces that could be generated by a piece of source code. Both are useful models.

2.7 Notational Conventions

The above paragraphs outline the formal notation. Often it is useful to have some shorthand for specific operations. Some of these conveniences have been outlined in earlier sections, but they are all listed here for completeness.

- !x is used for $!x_{!!\ldots}$. This initializes the variable to its base address and all the increment counts to zero.
- x is used for $x_{+\sim\sim\ldots}$ because it is the most frequent operation. x- is used for $x_{-\sim\sim\ldots}$ and x~ for $x_{\sim\sim\sim\ldots}$. Similarly, c_0 is used for $c_{0+\sim\sim\ldots}$.
- The default operation on a subtrace is to pulse it, so if s1 is a subroutine, s1 is equivalent to @s1.
- Commas as a delimiter between trace atoms may be left out if the result is a description that is easier to read.
- Increment counts and commas may be left out of the control string of a variable instance if the meaning is clear.
- If multiple increments are defined for a variable, but only one is explicitly stated, the assumption is that increment counts other than the first one are left alone. So $x_{3\sim}$ is equivalent to $x_{3\sim\sim\sim\sim\ldots}$.
- The label for a parenthesized group of trace atoms may be left out.
- !all initializes all defined variables and subroutines to their base, and all increment counts to zero without generating any addresses

- // indicates all other characters to the end of the line are a comment, and not TSpec
- if no attribute tag is specified the atom is assumed to be a read rather than a write

1. Grammar

In this section, we provide the syntactic definition of the notation. Here, single quotes surround the delimiters of the language being defined. Thus, in the first definition below, a *<trace_specification>* consists of a semicolon-separated sequence of *<definition>*s followed by a *<trace_list>*.

```
<trace_specification> ::= {<definition>‘;’}* {<trace_list>}
<trace_list> ::= {‘<‘ <trace> ‘>&’}*’<‘ <trace> ‘>’
<definition> ::=
            {<variable_def>’;’}*<variable_def> |
            {<subtrace_def>’;’}*<subtrace_def>

<variable_def> ::=
            <trace_variable> ‘(‘ <baseset>‘)’
<baseset> ::=
            <integer> <attr_tag> ‘,’ <increments>
<attr_tag> ::= ‘_r’ | ‘_w’
<subtrace_def> ::=
            <identifier> ‘(‘ <param_list> ‘)’ ‘=’ <trace_list>

<increments> ::= {<increment>’,’}*<increment>
<increment> ::= <integer1>’,’ <integer2>

<param_list> ::=
            {<identifier> ‘,’}*<identifier>

<trace_variable> ::= <identifier><control_tag>
<subtrace> ::= <identifier> ‘(‘ <param_list> ‘)’
<trace> ::= <trace_item>
            | <trace> ‘,’ <trace_item>

<trace_item> ::= ‘!’<action_atom>
            | ‘(‘<identifier> <trace>’)’<identifier>
            | <identifier> ‘:(‘ <trace> ‘):’ <identifier>
            | ‘(‘<identifier> <trace> ‘&’ <trace> ‘)’<identifier>
            | <trace_item>’*’<integer>
            | <trace_atom>
            | ‘break’ <identifier>
            | ‘{‘ <trace> ‘|’ <trace> ‘}’
            | ‘exceptlast (‘ <trace> ‘)’
            | ‘^’<trace_variable>

<trace_atom> ::= <integer><attr_tag> | ‘λ’ | <action_atom>
```

```
<action_atom> ::= <subtrace> | <trace_variable>

<control_tag> ::=  {<integer><control_char> ‘,’}* <integer> <control_char>
           | {<control_char>}* <control_char>
<control_char> ::= ‘+’ | ‘-’ | ‘!’ | ‘~’
```

2.8 Related Work

Some of the first work done characterizing reference patterns was performed by Denning *et al.* on working sets [Den68]. Batson and Madison [Bat76, Mad76, and Bat77] describe reference patterns in terms of their residence in localities of various sizes and lifetimes, and the transitions between these localities. Methods for identifying “major” phases in programs which correspond to intervals of distinctive referencing behavior are outlined. The concept of a Bounded Locality Interval (BLI) is defined and used to discuss the potential use of these concepts in predictive memory management algorithms. Hill [Hil87] and Sugumar and Abraham [Sug92] classify the misses of a cache as one of three types: compulsory, conflict, or capacity.

Chapter 3

The Functional Filter Model

3.1 Introduction

The previous chapter outlined a notation, *TSpec*, that can be used to describe memory reference traces. This chapter uses the notation, extends it to express cache state, and defines a *filter function model* of a cache that operates on the traces depicted by these descriptions. The overall approach is explained and detailed examples of the kernel *copy* are shown with analyses for each equivalence class outlined in the previous chapter.

The filter function model formalizes the definition of a cache's function given a specific state and a specific reference trace. This formalism is its chief advantage, allowing designers to perform an extensive formal analysis for a set of reference traces that represent the execution of a piece of source code. The model is also *general*; it can be used to describe and analyze any sequence or pattern of memory references for any standard type of cache.[1] It is not limited to loop, array, or other reference patterns with an affine relationship, or to fully-associative caches.

3.2 The Functional Cache Filter Model

As explained earlier, this analysis approach is inspired by viewing a cache as a filter. As previously described, a cache filters out the references that hit and transforms an input set of references into another, hopefully sparser, output set. By composing a series of cache filters, as many references as possible are removed from the request string before it is presented to the next level of the memory hierarchy. 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 complement the next level by filtering references that it would not be able to capture.

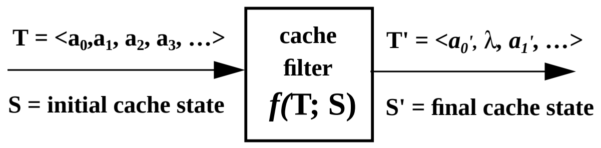


FIGURE 6: The Cache Filter Model

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 6 illustrates this relationship. The trace-only portion of the output of a filter function is denoted $f^T(T;S)$, and the state-only portion is denoted $f^S(T;S)$.

1 Standard cache refers to any sie fully associative, set associative, or direct-mapped cache with LRU replacement whether it is write back or write through.

The representation of a trace, $\boldsymbol{T}$, is the TSpec notation described in the previous chapter. To further clarify definitions, $\boldsymbol{T}$ is used to represent an unfiltered trace and $\boldsymbol{t_x}$ to represent an element in trace $\boldsymbol{T}$ at position (or index) $\boldsymbol{x}$. The position is determined by counting the trace elements from left to right. For example, if $\boldsymbol{T = < 100, 200, 300 >}$, $\boldsymbol{t_1=100}$, $\boldsymbol{t_2=200}$, and $\boldsymbol{t_3=300}$. So, in the abstract, each trace is represented by a set of value-index pairs. The value is the address being read or written and the index is the order or position; elements presented to the cache first have lower indices than those presented later.[2] Also, $\boldsymbol{T_x}$ refers to the entire trace up to and including $\boldsymbol{t_x}$. So $\boldsymbol{T_x = t_1, t_2, t_3, \ldots, t_x}$, and using the definition of $\boldsymbol{T}$ above, $\boldsymbol{T_2= <100, 200>}$**.**

The state can be viewed as a subset of the value-index pairs in the trace. Again, the value refers to the address of the item stored in the cache and the index refers to the order the addresses have been presented to the cache. The most recent address gets the largest index and the element with the smallest index is next to be replaced in a cache that uses the LRU (Least Recently Used) replacement algorithm. Note that the state is a subset of the trace and, in most instances, will have several value-index pairs missing when compared to the trace. It is not necessarily a strict subset, however. When all the references in the trace fit in the cache, and there are no duplicate references in the trace, the state and the trace are exactly the same.

In practice, we use two notations to represent the value-index pairs of a state. First, the state can be explicitly represented as a set of address-index pairs. Generally these are written in reverse order of their indices, so the righmost reference has the smallest index

[2] Recall from the introduction that the order here is only loosely related to wall-clock time. There may be a different, arbitrary amount of wall-clock time between any two TSpec references.

and is the next to be replaced. Writing the state in this order is more straightforward because one starts at the left and writes down the most recent reference and continues back through the trace until the cache size number of references is reached. For example, with indices explicitly stated, the state for a fully-associative, bigger than size three cache after the trace ***<<100, 1>, <200, 2>, <300, 3>>*** has been presented to it would be ***S = {<300, 3>, <200, 2>, <100, 1>}***. Usually, the addresses only are written as with traces, so ***S = {300, 200, 100}***. Note that this trace and state may also be represented in TSpec notation with a variable.

The second notation is to use a trace to denote a state, without explicitly writing the trace (or state) elements. In the simplest case the state is then ***T***. To expand the set of caches for which this type of notation applies, some additional functions on a trace, ***T***, are defined.

The *unique* of ***T***, denoted ***U(T)***[3], is formed by eliminating any duplicates in ***T***.

$$U(T_x) = \begin{cases} t_0 & \text{if } x = 0 \\ U(T_{x-1}),\ \lambda & \text{if } \exists t_y \text{ such that } t_y = t_x \text{ and } y < x \\ U(T_{x-1}),\ t_x & \text{otherwise} \end{cases}$$

The first occurrence of an address is retained, but all subsequent occurrences are replaced with λ. For example, if ***T = <100, 200, 100, 300, 200, 400>, U(T) = <100, 200, λ, 300, λ, 400>***.

[3] U(T) is extended in Section 3.4.2 to account for conflicts in set-associative caches.

The second function is denoted ***D(T)*** and is simply ***T*** with all λs removed. Continuing

$$D(T_x) = \begin{cases} D(T_{x-1}) & \text{if } t_x = \lambda \\ D(T_{x-1}), t_x & \text{otherwise} \end{cases}$$

with the example trace above, ***D(U(T)) = <100, 200, 300, 400>***.

The final functions on a trace defined here are ***|T|***, the *length* of ***T,*** and $\overline{\boldsymbol{T}}$**,** the *reverse* of ***T***. The length of ***T*** is an integer representing the largest index of an element in T, or simply the number of elements in ***T***, including λs. More formally, $|T| = x$ such that $x \geq y \ \forall t_y \in T$. The reverse of ***T***, $\overline{\boldsymbol{T}}$, is formed by reversing the order of the elements in T so that the value that was previously paired with the smallest index, is now paired with the largest index, etc. Formally, $\forall \bar{t}_y \in \overline{T}, \bar{t}_y = t_{|T|-y+1}$

The state for fully associative, LRU caches can now be written as ***S = {<v, i>*** ε ***D(U(*** $\overline{\boldsymbol{T}}$ ***)) | (|D(U(T))| - i) <= sz}*** where *sz* refers to the cache size. This says that the state consists of the last cache-size, unique, elements in ***T***.

In the sections below, the notation will be expanded to include additional cache types and reference strings. The overall analysis method will be presented first with simple unrelated examples, then a complete analysis of the kernel copy will be given.

3.3 Analysis Method

3.3.1 Overview

The method of analyzing caches as filter functions can be broken down into four steps.

1. *Expressing the trace in TSpec*. The original trace may come from static analysis

of source code, the tracing of a specific run of the source code, or simply by creating a TSpec description of an interesting pattern to analyze.

2. *Simplifying the TSpec description* by dividing it into intellectually comprehensible units. This is accomplished with two consecutive techniques. First, *segmentation* breaks the reference string of an entire source code program into smaller concatenated pieces of TSpec. Second, *decomposition* separates these trace segments into even simpler traces, called *primitives*. If remerged, these primitives would form the original trace segment, making decomposition the opposite of merge. The objective of both segmentation and decomposition is to simplify the trace and subsequently the analysis.
3. *Applying the filter functions* of the cache system to the primitives individually using the *filter function dictionary.* (See Appendix A) In most cases, these decomposed traces correspond to primitive that have been previously analyzed and the result of filtering can be looked up in this dictionary.
4. *Recombining the filtered traces.* To recombine the filtered version of the primitives, the definition of merge is extended to include the state. The filtered primitive are re-merged to form a filtered segment, then the filtered segments can be *concatenated* to see the effect of the entire trace. (In the simplest case this trace-state merge becomes the trace-only merge described in Chapter 2. This is the case where there are no conflicts between the decomposed or segmented traces.)

The approach above assumes ***f(T1 & T2; S) = f(T1; S) & f(T2; S)***: the effect of merging two traces and then filtering them is the same as filtering each trace separately and remerging the results**.** By extending the definition of merge to include trace-state pairs this equality can be made to hold. The benefit of this approach is that the effect of many standard caches can be analyzed on a wide spectrum of traces by defining cache-filter functions for only a relatively small number of simple traces called *primitives*. In addition, it is easier to gain insight into *why* and *how* particular traces are affected by particular caches because the effects from the interaction of primitive traces with the cache type are separated from the effects from the interaction of primitive traces with each other.

The sections below describe each of these steps in more detail. After the steps are explained, a complete analysis for the kernel copy is given as an example.

3.3.2 Expressing the Trace in TSpec

A TSpec description may be derived in any of three different methods: static analysis of source code, direct translation of a reference trace from a specific run of a program, or by directly writing an interesing pattern in TSpec.

Static analysis of the source code can be performed by human interpretation of assembly or other source code, or by a machine translator. The majority of the examples in this dissertation were derived by human analysis of assembled C code snippets. Appendix B shows the C code, assembly, and TSpec for these examples. A preliminary investigation of a machine translator built into a compiler based on vpo (Very Portable Optimizer) [Ben] is underway in conjunction with researchers at the University of Utah [Wan01].

Direct translation of large reference traces is tedious enough to require machine assistance. A prototype of a stand-alone machine translator was developed by Nahas *et al.*

[Nah97]. This C program used a simple search algorithm and heuristics to translate traces of the Spec Benchmark suite captured by researchers at Brigham Young University [Fla92, Fla93, Gri92]. The focus of this translator was to determine the length, frequency and effect on caches of streams found in the benchmark suite. A second system [Amy99] to provide bookkeeping mechanisms for a human who performs the pattern recognition, was developed in conjunction with researchers at the University of Utah. Writing a TSpec description of an interesting pattern under investigation allows a cache designer to easily describe important reference behaviors and perform analyses to determine the effects of different cache designs on these patterns.

There are many ways to accomplish the transformation into TSpec and more than one valid TSpec description for a particular reference trace. The description chosen should be the one that makes the analysis most straightforward. The analysis approach outlined below makes only one substantial assumption on the transformation; the variables in the description must be *independent*. Independent means that no address is represented by more than one variable. Guaranteeing independence between variables ensures that an address is accessed by only one variable and no aliasing occurs.

3.3.3 Simplifying the TSpec Description

3.3.3.1 Segmentation

Previous research indicates that memory referencing behavior tends to occur in phases as shown in Figure 7 [Bat76, Bat77, Den68]. First a referencing behavior of one type (pictured as Rx) occurs, then a transition phase (pictured as Tx) occurs, followed by another phase of referencing behavior. The source code of the referencing phase is

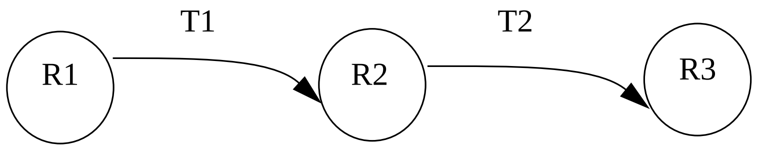


FIGURE 7: Phase Referencing Behavior

referred to as a kernel. Segmentation is the process of identifying these kernels, or the section of trace corresponding to a referencing phase and so the primary referencing behavior of the kernel. There are several ways that segments might be identified, but the crucial point is that this identification simplifies the piece of trace or source code being analyzed.

The method chosen for segmentation depends on the level at which the analysis is to be done. If the general effects of a cache-filter are desired, only kernel analysis may be necessary. If the intent is to fine-tune the operation of a cache already selected, or see what details are causing an effect at a higher level, transition segments may also be analyzed.

In the majority of the examples in this dissertation, segments have been derived by human analysis of the TSpec corresponding to the assembly language as described above. The most common referencing behavior is identified and used as a segment for analysis. Consider the trace portion of the TSpec description for the optimized version of copy shown in Figure 8. The C code, assembly language, and TSpec definitions are given in Section B.2 of Appendix B.

```
< !all, c1*3, (c1 t1_r c1 t2)*3 c1*2                    // copy “to” vector to frame
              (c1 f1 c1 f2_w)*3, c1*3,                  // copy “from” vector to frame
               !c2, c*3, {B1 c2*5 f (Lc26, t, c2*3, {c2, f, | break L})L *3 |B1},     // main
copy
               c213, c2, c15, c120, c1*4>
```

$<$!all, c1*3, (c1 t1_r c1 t2)*3 c1*2 // copy “to” vector to frame
(c1 f1 c1 f2_w)*3, c1*3, // copy “from” vector to frame
!c2, c*3, $\{_{B1}$ c2*5 f $(_{L}c2_{6}$, t, c2*3, {c2, f, | break L}$)_{L}$ *3 $|_{B1}\}$, // main copy
$c2_{13}$, c2, $c1_{5}$, $c1_{20}$, c1*4$>$

FIGURE 8: TSpec, SPARC, Optimized Copy

There are three separate segments in this TSpec description that resemble the pattern ***(!c, c*, f, c*, t, c*)****; that is, some number of code references (represented by 'c'), followed by a read of a “from” element (represented by 'f'), followed by more code references and a write to the “to” element (represented by 't'), followed by some more code references to determine whether another element should be copied. Transitions between this pattern are different forms of ***c****. Since the basic copy is the most common reference pattern for this source, a specific form of it from the TSpec description above is selected as a segment for analysis. Specifically, we choose the main copy loop: ***$(_{L}c2_{6}$, t, c*3, { c2, f, | break L}$)_{L}$ *3*** for the examples in the appendix.

3.3.3.2 Decomposition

Despite the fact that there are an infinite number of possible source programs with an infinite number of inputs, we have found the bulk of the reference patterns that they generate can be described by combinations of a modest set of parameterized primitive patterns (See Chapter 5). This is true because source programs are themselves composed of combinations of similar code constructs. Decomposition has both a mechanism as to how the decomposition is performed, and a goal. The goal is to decompose the segment into primitive traces found in the dictionary described in Appendix A. The details of how the decomposition is performed are described in the following paragraphs.

The two simplest families of primitive patterns are the code loop and the data stream. Consider the general form of the copy kernel from the previous section:

<(!c, c*, f, c*, t, c*)*>

The use of *f* and *t* variables are examples of streams. These consist of a base address and an increment(stride). The pattern created by the variable *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 between the data stream and code loop is that the data stream has no repeated addresses, while the code loop does.

Until now, λ has been used only as a placeholder for filtered references. By extending the use of λ as a placeholder for references from other primitives, the frequency of a primitive within a reference string can be made explicit without describing the particulars of the other primitives. For example, a stream primitive could appear every 4th reference within a reference string representing a code loop. In this case the primitive depicted with f, can be described as **<*(f+, λ, λ, λ)**>**. The λs represent the references to the code.

When decomposing a trace into primitives, there are two considerations. First, all references to a particular variable within a segment should be wholly contained within one primitive. This reduces the chances of conflicts between primitives. Second, if possible, the primitives should be available in the functional filter dictionary described below. This will simplify the analysis process. If necessary, the dictionary can also be extended to include additional primitives.

3.3.4 Applying the Filter Functions

3.3.4.1 Defining Filter Functions

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

There are three possibilities for the output of the functional filter model. In the first situation, where the cache is write back, the line is dirty, and the reference, ***a,*** misses, the output trace consists of the dirty line evicted from the cache, (***d(a)***), and the line that includes the new reference ***a***, ***(l(a)***). The new state, ***S'***, is the same as the initial state except that the dirty line ***d(a)*** has been replaced by the new line ***l(a)***. The write through version of the cache would not have a dirty line to evict, so the output would simply be the reference to fill the line that includes the new reference ***a***, ***(l(a))***, and the new state, ***S'***. If ***a*** hits, the values of the state remain the same, but the new state is denoted ***S''*** to indicate the new indices of the values that were just referenced and hence the updated LRU status. The recursive definition of a cache filter is formalized below. Each of the three situations outlined above are included in the definition.

Recall $F(T; S) = T'; S'$

Where $S' = (S'_{x-1} - d(a)) \cup l(a)$ and $S' = S'_{x-1} - l(a) \cup l(a)$ for a miss with no eviction. This definition requires a formal definition of subtraction and union of state. In

$$F(T_x; S) = \begin{cases} T'_{x-1}, d(a), l(a); S' & \text{if cache wb, dirty line, and } a \notin S \\ T'_{x-1}, l(a); S' & \text{if } a \notin S \text{ and cache wt or wb and clean line} \\ T'_{x-1}, \lambda; S'' & \text{otherwise} \end{cases}$$

subtraction, the elements with the indices of the subtracted line are eliminated. Let ***I*** be the set of indices of the elements of the subtracted line, ***d(a)***, in ***S***, and ***J*** be the set of indices of all elements in S. If $S - d(a) = S'$, then$\forall k \in J$

$$s_k' = \begin{cases} s_k & k \notin I \\ \varnothing & k \in I \end{cases}$$

Union is just the insertion of the new line, ***l(a)***, into the state. Since the last element presented to the cache has the highest index in the state, this amounts to adding all of the new elements to the state with an index greater than the last index. For the purposes of this methodology, there is no “order” within the line; the whole line is given the same index in the state. This makes the assumption that lines are replaced or written all at one time. If $S' = S \cup l(a)$, ***i*** is the index of the element ***a*** in ***T***, and ***J*** is the set of indices of all elements in S, then$\forall k \in J$

$$s_k' = \begin{cases} l(a)_i & k = i \\ s_k & k \in J \end{cases}$$

While the above definition of a filtering function is very general, it provides little insight into what happens on the primitive or kernel levels. To assist the human analyzer, we have developed a dictionary of cache filter functions to be used much like a set of integration tables. One looks up the desired pattern to obtain the corresponding filtering result. See Appendix A for the set of tables.

The primitives are listed with parameters for the number of repetitions and λs. The filtered trace and state, ***f(T; S) = T'; S'***, are listed with parameters for cache associativity, lines per set, linesize, write policy, and any other parameter that is relevant to the primitive. Each primitive can be filtered by looking it up in the dictionary and substituting specific values for the parameters. The whole dictionary would be inappropriate here, but we give the dictionary entry needed for the example at the end of the chapter.

In the dictionary, the state is represented in TSpec and for that, an additional TSpec construct is useful. To write TSpec starting from the end of a trace so that the references appear in with the least recently used (and hence next to be evicted) last on the line, we 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 dictionary entry below, the “n” variables represent parameters for the number of repetitions in the trace. The *effective set number* (esn) refers to the number of sets in a cache that can effectively be used by a particular reference pattern and depends on the line size of the cache, the capacity of the cache, and the stride of the reference pattern. P is used in the dictionary to stand for *primitive* and is simply a piece of decomposed trace. Details on the derivation of the formula for effective set number is in the beginning of Appendix A.

if P= !p, (λ*n1, p, λ*n2)*n3 then
$f_{\mathrm{fa},\,*,1}(\mathrm{P};\ \mathrm{S}^0) = \mathrm{P};\ \{\mathrm{p!},\ (\mathrm{p-})*\min(\mathrm{esn},\ \mathrm{n3})\}$

This entry says that for any read stream, with any number of intervening λs, the filtering effect of an empty, fully associative, LRU cache with a line size equal to the fetch size, will result in an output trace that is the same as the input trace and a cache state that is the part of the reverse stream that will fit in the cache. If the size of the stream is less than the effective set number in the cache then the whole stream remains in the cache. Otherwise, only the last part of the stream remains.

3.3.5 Recombining The Filtered Traces

3.3.5.1 Merge

Recombining filtered primitives requires merging the trace and state of two or more separate filtered primitives. ***f(T1; S1) = T1'; S1'*** and ***f(T2; S2) = T2'; S2'***. We will define ***T'; S' = (T1'; S1') & (T2'; S2')***. Let $\boldsymbol{T_x}$ represent the trace ***T*** up through index ***x*** and $\boldsymbol{S_x}$ indicate the state of the cache after the value with index ***x*** in ***T*** has been presented to the cache. ***d(t)*** represents the dirty cache line that ***t*** maps to and ***l(t)*** represents the cache line that contains ***t***. Without loss of generality, whichever trace has a non-λ element at the ***x*** position is referred to as ***T1***. Then

$$1'_x;S1'_x) \ \& \ (T2'_x;S2'_x) \ = \begin{cases} T'_{x-1},\lambda;\ S'_x & \text{if } t_x \in S'_{x-1} \text{ and wb or } t_x \text{ is read} \\ T'_{x-1},\ l(t1_x);\ S'_x & \text{if } t_x \notin S'_{x-1} \text{ wb not dirty or wt, } t_x\text{writ} \\ T'_{x-1},\ d(t1_x), l(t1_x);\ S'_x & \text{otherwise} \end{cases}$$

where

$$S'_x = \begin{cases} S'_{x-1} - l((t1_x) \cup l(t1_x)) & \text{if } t1_x \in S'_{x-1} \text{ and wb or } t1_x \text{ is read} \\ S'_{x-1} - d((t1_x) \cup l(t1_x)) & \text{otherwise} \end{cases}$$

The first line of the equation for the merge says that when the element in the original, unseparated trace is in the cache, that element is filtered out of the trace and replaced with λ as long as the cache is a *wb* (write back) cache or if the cache is *wt* (write through) and the original element is a read. The filtered λ is concatenated to the filtered trace from the previous element. The state is simply a reordering of the state from the previous element, so that now the line that was accessed is now in the most recently used index. In this way the merged, filtered trace is built up positionally one element at a time.

The second line says that when the original, unseparated trace element is not in the cache, the line for that element, $\boldsymbol{l(t1_x)}$, must be fetched, so it is reflected in the filtered trace. The state is changed by adding the fetched line to the old state, with the LRU index, removing the replaced line from the old state, $\boldsymbol{d(t1_x)}$, and reordering the indices appropriately. This happens whenever the requested element is not in a write back cache where the line being evicted is not dirty, or when the element is being written in a write through cache.

In all other cases, the dirty line that is being evicted from the cache, $\boldsymbol{d(t1_x)}$, must also be written to main memory and so is included in the filtered trace, and removed from the state.

Notice that when the line is bigger than a single element, or an element is dirty, there may be more elements in the filtered output trace than in the original input trace for a given element. This means that the indices in the input and output will not match up exactly on an element-by-element basis. To resolve this problem, we conceptually add a λ to any trace, $\boldsymbol{T}$, that is involved in the analysis. We add one λ to $\boldsymbol{T}$ for each line added for a write back. This keeps the positions comparable for all traces involved in the analysis. If dirty lines are added when filtering primitives, the same positioning adjustment is made to all other primitives and the original trace T.

As an example of merge, let the cache be a fully associative, LRU, write through cache with 5 word-size lines and

(T; S) = (<100, 204, 104, 304_w, 108>;
{<108,5>, <300_w, 4>, <104, 3>, <200, 2>, <100, 1>}),
(T1; S1) = (<100, λ, 104, λ, 108>; {<108,5>, <104, 3>, <100, 1>})
(T2; S2) = (<λ, 204, λ, 304_w, λ>; {<300_w, 4>, <200, 2>})

Then using the above definition of merge yields:

(T1'; S1') = (<λ, λ, λ, λ, λ>; {<108, 10>, <104, 8>, <100, 6>})
(T2'; S2') = (<λ, 204, λ, 304_w, λ>; {<304_w, 9>, <204, 7>})
(T'; S') = (<λ, 204, λ, 304_w, λ>;
{<108, 10>, <304_w, 9>,<104, 8>, <204, 7>, <100, 6>}),

3.3.5.2 Concatenation

The concatenation of two segments, denoted (T1'; S1'), (T2'; S2'), can be defined in terms of the merge from the section above. Specifically,

$$(T1'; S1'), (T2'; S2') = (T1', \lambda^*|T2'|;\ S1')\ \&((\lambda^*|T1'|, T2'); S2')$$

3.4 Example Analysis

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. Each starts with C code and the corresponding assembly language generated by gcc.[4] 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 interesting behavior occurs. 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. Except for pathological cases where small-sized data hit page boundaries, the analyses given below would apply to either virtual or physical addresses.

3.4.1 Unconditional Copy C Code, Assembly Language and TSpec

We generated the C code for the more extensive version of the vector copy in Figure 9 with `gcc -O2 -fno-delay-slots`[5]. Figure 10 shows the assembler output, excluding error- and operand-checking code.

```
/* 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 9: C Code For Copy Example

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

[5] This command runs the gnu C compiler, with most optimizations active, but without filling the delay slots.

TSpec for this 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.

```
      // 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 does not 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 off-
set
         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
```

FIGURE 10: Disassembler Output For Copy Example

```
c(10b64_r, 4);
f(~_r, 4);
t(~_w, 4);
<!c, !f, !t, c, (c1,  c,   c, f, c,    c, t, c)*, c>
                clr  sll add ld   cmp st    bl    jmp
// Simplifying this trace yields:
<!c, !f, !t, c, (c1, c*2, f, c*2, t, c)*, c>
```

FIGURE 11: TSpec, SPARC, GNU, Simplified Copy

3.4.2 Copy Analysis Specific Trace {T}

First, the most straightforward example uses the equivalence class consisting of only a specific trace **T**. To this end, we arbitrarily set the number of iterations at three, and assume values for the base addresses of *f* and *t*. We segment this example, and choose the loop as the segment for analysis. The modified TSpec description is shown in Figure 12.

```
c(10b68_r, 4);
f(FSTART_r, 4);
t(TSTART_w, 4);
<!f, !t, (!c, c*3, f, c*2, t, c)*3>
```

FIGURE 12: Copy TSpec, {T}

The next analysis step, decomposition, breaks the TSpec trace into primitives. Any set of primitives that when merged form the original trace can be used as long as there are no overlapping addresses between primitives, but the analysis is simplified if they are also chosen such that the primitives themselves are already in the filtering dictionary. One possible decomposition of the above TSpec into primitives yields:

```
P1 =   (!c,  c*3,  λ,  c*2,  λ,  c)*3
P2 =   !f, (  λ*3, f,   λ*2,  λ, λ)*3
P3 =   !t, (  λ*3, λ,  λ*2,   t,  λ)*3
```

We are now able to apply the filter functions to the individual primitives. For this example, we will consider two different styles of cache. The first cache, *dm*, will be a direct-mapped write through cache and the second cache, *fa*, will be a fully associative write through cache with LRU replacement.

The most straightforward case is where all references fit in both caches. This is the case where, ***f(T1 & T2; S) = f(T1; S) & f(T2; S)*** and the final merge is simply a TSpec merge of filtered primitives.

If each cache has 12 lines and a linesize of one reference, then all the references can fit. Transforming T into its primitive traces generates the primitive traces P1–P3 above. P1 corresponds to the code references, P2 to the vector being copied from, and P3 to the vector being copied to. A filtering function from the dictionary 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:

$$f_{*,\,\mathrm{wt},\,1}{}^{T}(P1, S) = f_{\mathrm{dm},\,\mathrm{wt},\,1}{}^{T}(P1, S) = P1' \\ = (!c, c*3, \lambda, c*2, \lambda, c), \lambda*16.$$ [6]

This shows that the first iteration of the code misses while subsequent iterations all hit in both caches. The sixteen λs at the end represent the 12 filtered code references from the second and third iterations (6 from each) and the extra 4 from the decomposed data streams (2 per loop). Filtering the two data stream primitives from the dictionary yields:

$$f_{*,\,\mathrm{wt},\,1}{}^{T}(P2, S) = f_{\mathrm{dm},\,\mathrm{wt},\,1}{}^{T}(P2, S) = P2' \\ = !f, (\lambda*3, f, \lambda*2, \lambda, \lambda)*3 = P2 \text{ and}$$

$$f_{*,\,\mathrm{wt},\,1}{}^{T}(P3, S) = f_{\mathrm{dm},\,\mathrm{wt},\,1}{}^{T}(P3, S) = P3' \\ = !t, (\lambda*3, \lambda, \lambda*2, t, \lambda)*3 = P3.$$

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

Assuming the primitives are assigned to cache lines 0, 6, and 9 repectively and merging the three filtered primitives gives the final result:

$$f_{*,\,\mathrm{wt},\,1}{}^{T}(T, S) = f_{*,\,\mathrm{wt},\,1}{}^{T}(P1, S) \,\&\, f_{*,\,\mathrm{wt},\,1}{}^{T}(P2, S) \,\&\, f_{*,\,\mathrm{wt},\,1}{}^{T}(P3, S) = \\ P1' \,\&\, P2' \,\&\, P3' = (!c, c*3, f, c*2, t, c), (\lambda*3, f, \lambda*2, t, \lambda)*2.$$

[6] We use * in subscripts of f() to indicate any parameter can be substituted here.

Here there are the least possible number of misses for the types of caches we are considering - 12 compulsory misses.

Next, consider what happens when the reference string will not fit into either cache. Let the caches be of size six[7] with a line size of one reference. The filtered primitives, P1'-P3' remain the same as in the above example.

When merging the filtered primitives, though, we must use caution. Different types of caches will have different effects during a merge when the capacity is exceeded and conflicts are considered. The state must be included in performing the final merge of the filtered primitives.

For a fully-associative cache, our experiments have shown $f^T(T, S)$ has two possibilities for a loop with no inner repetitions. The first is that a single iteration of the 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 references at the end of the loop always evict the first references in the loop before they can be reused. For this example:

$f^T_{\text{fa, wt, 1}}(T, S) = T' = T = <!f, !t, (!c, c*3, f, c*2, t, c)*3>$
$f^S_{\text{fa, wt, 1}}(T, S) = \{c!, t!, f!, c-, t, c-*2, f, c-\} = \{<v, i> \varepsilon D(U(\overline{T})) \mid |D(U(\overline{T}))| - i <= 6\}$

All of the references miss and there are a total of 24 misses.

For the direct-mapped cache, the results are better (a total of 14 misses) because each address is mapped to a specific location, so the worst-case behavior is avoided. Assuming

[7] We realize that this is an odd cache size. We use it here, because it demonstrates what happens when each individual primitive fits, but together they do not.

that the code references start in cache line 0, and that the vectors *f* and *t* start in cache lines 4 and 5, respectively, we obtain:

$$\begin{aligned} f^{T}_{\text{dm, wt, 1}}(T, S) = T' = \langle !f, !t, (!c, c*3, f, c*2, t, c), \\ !c, \lambda*3, f, \lambda*2, t, c_5, \\ !c, c, \lambda*2, f, \lambda*2, t, \lambda \rangle \end{aligned}$$

$$f^{S}_{\text{dm, wt, 1}}(T, S) = \{c!, t!, f!, c-, t\sim, c-*3, f\sim\} = \{\langle v, i\rangle \varepsilon D(U(\overline{T})) \mid |D(U(\overline{T}))| - i <= 6\}$$

Since there is only one line that each reference can be assigned to, some items are not at resk of eviction before they can be used again. Figure 13 illustrates how this occurs. Each item is written in its line as it occurs in the trace, T. If the item that is currently in that line is referenced again, a λ is written in the line. The set consisting of the last item written in each line is the contents of the cache.

cache line	1st iteration	2nd iteration	3rd iteration
0	c_0	λ t_1	c_0 f_2
1	c_1	λ	λ t_2
2	c_2	λ	λ
3	c_3	λ	λ
4	f_0 c_4	λ	λ
5	t_0 c_5	f_1 c_5	λ

FIGURE 13: Cache Line Assignments for Copy, DM Cache, Six Lines

This analysis provided at least one insight: 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 fully-associative, LRU cache is capable of best case performance, but it is also capable of worst case performance when the “unique” reference string of the loop does not fit in the cache. Though a fully associative cache of large size is too expensive in practice, it is often considered a sought-after standard for maximizing hit rates. This example indicates that while in some circumstances it might have the best case performance, a set associative cache that does not exhibit this worst case behavior may be a more appropriate choice *even without adding the cost and complexity issues*!

Notice that if the cache line assignment were different, the state for the direct-mapped cache may not have been ***{<v, i>*** ε ***D(U($\overline{T}$)) | |D(U($\overline{T}$))| - i <= 6}*** as we have previously defined U(T). In particular, the previous definition of U(T) does not account for conflicts between elements in a set-associative cache, so we expand it as follows:

$$U(T_x) = \begin{cases} t_0 & \text{if } x = 0 \\ U(T_{x-1}), \lambda & \text{if } \exists t_y \text{ such that } t_y = t_x \text{ and } y < x \\ & \text{and there is not } t_k \text{ such that } i(t_k) = i(t_x) \text{ for } y < k < x \\ U(T_{x-1}), t_x & \text{otherwise} \end{cases}$$

where ***i(t)*** is the number of the cache line of address ***t***. Formally, $i(t) = (t \text{ div } ls) \bmod sn$, where ***ls = line size*** and ***sn = set number***.

3.4.3 Copy Analysis - Equivalence Class Under Varying Bindings

{T_b}

The above analysis uses a specific set of address bindings and hence, a specific set of cache line assignments for each cache. In this section we will expand the analysis to include all possible address bindings. We will focus on the worst case and best case performance of two different caches on the same copy example we have been using. Specifying the worst and best case performance will give us an idea of the range of possible outcomes for different caches on this reference pattern.

First, we will strip the base address information from the copy reference string as shown in Figure 14.

```
c(10b68_r, 4);
f(~_r, 4);
t(~_w, 4);
<!f, !t, (!c, c*3, f, c*2, t, c)*3>
```

FIGURE 14: Copy TSpec, {T_b}

Second, the primitive decomposition is done the same way as in the previous section. Specifically,

```
P1 =   (!c,  c*3,  λ,  c*2,  λ,  c)*3
P2 =   !f, (  λ*3, f,   λ*2,  λ, λ)*3
P3 =   !t, (  λ*3, λ,  λ*2,   t,  λ)*3
```

Let us consider a direct-mapped, write through cache with 12 lines the size of a reference. If we consider all possible cache line assignments, we will have three different groups. The first represents the best case performance and consists of all address (cache line) assignments where there is no conflict between the different primitives. This set of possibilities can be represented by the assignment of P1 to begin in cache line 0, P2 to begin in cache line 6, and P3 to begin in cache line 9. The final result after filtering the primitives has 12 compulsory misses:

$$f_{dm, wt, 1}^{T}(T, S) = f_{*, wt, 1}^{T}(P1, S) \,\&\, f_{*, wt, 1}^{T}(P2, S) \,\&\, f_{*, wt, 1}^{T}(P3, S) =$$
$$P1' \,\&\, P2' \,\&\, P3' = <!c, c*3, f, c*2, t, c, (\lambda*3, f, \lambda*2, t, \lambda)*2> = T'$$
$$f_{dm, wt, 1}^{S}(T, S) = \boldsymbol{\{<v, i> \varepsilon\, D(U(\overline{T}))\ |\ |D(U(\overline{T}))| - i <= 12\}} =$$
$$\{c!, t!, f!, c-, t-, c-*2, f-, c-*3, (t-, f-)*2\}$$

The second group of address (cache line) assignments consists of those that conflict somewhat, but not completely. This set will have a variety of miss rates, all that are greater than 12, but less than that of the worst case performance outlined below.

The third group of address (cache line) assignments consists of those that conflict as much as possible. This can be represented by the assignment of P1 to cache line 0, P2 to cache line 3, and P3 to cache line 0. The final results of merging the filtered primitives is:

$$f_{dm, wt, 1}^{T}(T, S) = f_{*, wt, 1}^{T}(P1, S) \,\&\, f_{*, wt, 1}^{T}(P2, S) \,\&\, f_{*, wt, 1}^{T}(P3, S) =$$
$$P1' \,\&\, P2' \,\&\, P3' = <!c, c*3, f, c*2, t, c,$$
$$\lambda, c_1, \lambda, f, c_3, \lambda, t, \lambda,$$
$$\lambda*2, c_2, f, c*2, t, c> = T'$$
$$f_{dm, wt, 1}^{S}(T, S) = \boldsymbol{\{<v, i> \varepsilon\, D(U(\overline{T}))\ |\ |D(U(\overline{T}))| - i <= 12\}} = \{c!, c-*6\}$$

The figure below shows how this result is obtained. The total number of misses is 18; one for every compulsory miss as in the best case (12), and one for every stream reference (6), which in this worst case evicts a code reference before it can be used again.

cache line	1st iteration	2nd iteration	3rd iteration
0	c_0	λ	λ
1	c_1 f_1	c_1	λ
2	c_2	λ f_2	c_2
3	c_3 t_1	c_3	f_3 c_3
4	c4	λ t_2	c_4
5	c5	λ	t_3 c_5
6			
7			
8			
9			
10			
11			

FIGURE 15: Cache Line Assignments, Copy, DM Cache, Twelve Lines

The diagram above also shows that the cache in this example is extremely under-utilized. In fact, this worst case scenario is also the performance that would be obtained from the worst case scenario with a direct mapped, write through cache of only size 6.

Now, let us consider a fully-associative, LRU, write through cache with 12 lines the size of a single reference. Since a fully-associative cache allows a reference to be placed anywhere, and has a replacement policy of LRU, there will be only compulsory misses and the results are the same as in the best case direct mapped example above.

$$f_{\text{fa, wt, 1}}^{T}(T, S) = f_{*,\text{ wt, 1}}^{T}(P1, S) \,\&\, f_{*,\text{ wt, 1}}^{T}(P2, S) \,\&\, f_{*,\text{ wt, 1}}^{T}(P3, S) =$$
$$P1' \,\&\, P2' \,\&\, P3' = <!c, c*3, f, c*2, t, c, (\lambda*3, f, \lambda*2, t, \lambda)*2> = T'$$
$$f_{\text{fa, wt, 1}}^{S}(T, S) = \boldsymbol{\{<v, i>} \,\varepsilon\, \boldsymbol{D(U(\overline{T}))} \mid \boldsymbol{|D(U(\overline{T}))| - i <= 12\}} =$$
$$\{c!, t!, f!, c-, t-, c-*2, f-, c-*3, (t-, f-)*2\}$$

Notice that in the fully associative cache of this size, the best case results are equal to the worst case results, so for this reference string there is no range in performance.

3.4.4 Copy Analysis - Equivalence Class, Varying Data $\{T_d\}$

To expand our analysis methods to include the equivalence class under varying data inputs, we must have a conditional in our code. As an example, the C code in Figure 16 has been modified from Figure 9 to include a conditional. The example is artificial in the interest of clarity. The corresponding assembly code follows in Figure 17.

```
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 16: Copy C Code, 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 17: Disassembler Output For Conditional Copy

The TSpec for this new example appears below. The assembly instructions are shown underneath the code variable for comparison with the source code. Notice that the pointers for f and t are updated outside of the conditional to make sure they are modified even when the copy does not occur.

c(10b64_r; 4);
f(?_r; 4);
t(?_w; 4);
<!c, !f, !t, c, (${}_L$c$_1$, c, (${}_I$ c, c, f~, c, t~ |${}_I$)$_I$, c$_6$, ^f, ^t, c, c)$_L$*3, c$_9$>
clr cmp be sll ld st add cmp bl jmp

We again focus on analyzing only the loop and assume the base addresses are known. This yields the following simplified TSpec:

c(10b68_r; 4);
f(FSTART_r; 4);
t(TSTART_w; 4);
<!f, !t, (${}_L$!c, c*2(${}_I$ c$_2$, c, f~, c, t~ |${}_I$)$_I$, c$_5$, ^f, ^t, c*2)$_L$*3>

Let us start with the simplest cache situation: a fully-associative LRU cache of size 14 or larger so that everything fits in the cache.

$f_{\text{fa,wt, 1}}$(**T**; $\mathbf{S^0}$) = <!f, !t, ($_L$c$_2$, c, ($_I$ c*2, f~, c, t~ |$_I$)$_I$, c$_6$, ^f, ^t, c*2)$_L$*3> ; **S'**
where **S'** = {c!, t!, f!, c-*3, t-, c-, f-, c-*4, (t-, f-)*2}OR- case where if is always taken
{c!, t!, f!, c-*3, t-, c-, f-, c-*4, (t-, f-)} OR - cases where if is taken twice
{c!, t!, f!, c-*3, t-, c-, f-, c-*4, ^t-, ^f-, t-, f- } OR
{c!, t!, f!, c-*3, t-, c-, f-, c-*4} OR - cases where if is taken once
{c!, t!, f!, c-*3, ^t-, c-, ^f-, c-*4, t-, f-} OR
{c!, t!, f!, c-*3, ^t-, c-, ^f-, c-*4, ^t-, ^f-, t-, f-} OR
{c!, t!, f!, c-*3, c$_6$-, c-} - case where if is never taken

By inspecting the source code, we can obtain more information. The TSpec above did not preserve the knowledge that only the second element can be skipped in the copy. Retaining this information in the TSpec lengthens the original TSpec description, but simplifies the final state possibilities. Consider the revised TSpec and resulting cache output.

```
c(10b68_r; 4);
f(FSTART_r; 4);
t(TSTART_w; 4);
<!f, !t, !c, c*4, f, c, t, c*3,
        !c, c*2, (Ic*2, f~, c, t~ |I)I c5, ^f, ^t, c*2,
        !c, c*4, f, c, t, c*3>
```

Notice that the first and third iterations simplify considerably because no jump or conditional needs to be anticipated. The second iteration still has a conditional and we cannot tell from the source code (only from the set-up data itself) whether or not the if will be taken. Now the filtered results from the revised TSpec are:

$f_{\text{fa,wt, 1}}$(**T**; $\mathbf{S^0}$) = <!f, !t, !c, c*4, f, c, t, c*3, ($_I$f~, t~|$_I$)$_I$ ^f, ^t, f, t> ; **S'**
where **S'**={c!, t!, f!, c-*3, t-, c-, f-, c-*4, ^t-, ^f-, t-, f-}OR - case where if is taken twice
{c!, t!, f!, c-*3, c$_6$-, c-} - 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, without narrowing down how many times the if is taken:

$MIN \subseteq S' \subseteq MAX$

where MIN = {c!, t!, f!, c-*3, c_6-, c-} 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.

3.4.5 Copy Analysis - Equivalence Class Under Varying Bindings and Data Inputs {T_{bd}}

In Section 3.4.4 and Section 3.4.5 we show how an analysis can be extended to include entire equivalence classes rather than just one representative. A more complex example is included here, by using the conditional example from Section 3.4.5 and analyzing it for {T_{bd}}. Consider the more general copy example:

c(10b68_r; 4);
f(FSTART_r; 4);
t(TSTART_w; 4);
<!f, !t, ($_L$!c, c*2($_I$ c_2, c, f~, c, t~ |$_I$)$_I$, c_5, ^f, ^t, c*2)$_L$*3>

Expanding this to analyze it for {T_{bd}} by making the number of iterations of the copy arbitrary and the starting positions of the variables unknown yields:

c(?_r; 4);
f(?_r; 4);
t(?_w; 4);
<!f, !t, ($_L$!c, c*2($_I$ c*2, f~, c, t~ |$_I$)$_I$, c_5, ^f, ^t, c*2)$_L$*>

To analyze this TSpec, we must do a worst-case analysis and best case analysis for the address assignments as well as the varying number of loop iterations and paths through the conditional. This can be accomplished by combining the mechanisms described in the two preceding sections.

3.5 Conclusion

The analysis methodology explained above allows a cache designer to gain insight into the filtering effect of different types of caches on different types of reference strings. Even the few analyses above have led to some useful insights. First, fully associative caches become completely ineffective in a loop situation where the unique reference string, $U(\overline{T})$, of one loop iteration does not fit in the cache. This insight does not only apply to reference caches, but may also explain some thrashing phenomena in translation lookaside buffers (TLBs) as well. Second, direct-mapped caches have to be managed well to have best-case behavior, but their worst-case behavior is almost never as bad as that of a fully-associative cache that is too small. This explains in part the effectiveness of the victim cache following a direct-mapped cache. In particular, the direct-mapped cache ensures the worst-case performance of the fully-associative cache does not occur and filters the reference string into the small set of conflicts that will almost certainly fit into the fully associative victim cache.

The next chapter discusses different evaluation metrics for caches, then Chapter 5 illustrates the entire methodology on several kernels.

3.6 Related Work

The majority of the related work for this section was discussed in the related work section of the introduction, Section 1.7. It is useful to note that the work on an analytical model for determining the optimal cache sizes for a hierarchy by Jacob *et al.* [Jac96] develops a series of stack distance curves to describe how many bytes of data are touched between two references to the same item. Measurements are taken directly from address streams, normalized, and plotted to produce a cumulative probability function and a probability density function.

Ladner, Fix, and LaMarca [Lad99] describe a model for the cache performance of algorithms in a direct mapped cache. Several commonly occurring memory access patterns are studied: (i) sequential and random memory traversals, (ii) systems of random accesses, and (iii) combinations of each. Still, characterizations are focused around the number of cache misses per memory access.

Tam, Rivers, Srinivasan, Tyson, and Davidson [Tam99] discuss the effectiveness of *multilateral* caches that exploit reuse pattern information. Multilateral refers to a level of cache that contains two or more data stores that have disjoint contents and operate in parallel. References are split between two caches depending on whether they exhibit more temporal or spatial locality. Another example of a multilateral caching system is the *stack value file* system described in [Lee01]. The stack value file system partitions data references into stack and non-stack references, routes the non-stack references to a conventional cache and then exploits characteristics of the stack references in the stack value file to improve performance. Some of the stack reference characteristics were frist noticed in the development of the CRISP architecture by Ditzel *et al.* [Dit87a, Dit87a].

These systems are evaluated using average memory access time, but their success in the research confirms the intuition from the caches as filters model that caches should be combined in such a way that the references they remove complement, rather than duplicate, one another.

Kin, Gupta, and Mangione-Smith [Kin00] also discusses the concept of a cache as a filter by presenting the *filter cache*. The goal is to have a very small filter cache on chip with the processor to filter out the majority of references, allowing the larger second level cache to be in low-power mode most of the time. Performance is slowed because of the particularly small size of the filter cache, but power is saved.

Chapter 4

New Measures

4.1 Introduction

In *Computer Architecture: A Quantitative Approach* by Hennessy and Patterson [Hen96], caching systems are evaluated using metrics such as (1) the aggregate *miss rate*, or fraction of accesses that cannot be serviced by the cache, (2) the execution time of a benchmark, and (3) average memory access time. These metrics are considered by many to be appropriate cache evaluation metrics - this text is a widely used architecture text for both graduate and undergraduate course work. As first discussed in the introduction to this dissertation, however, there are problems with the use of these measures alone. Evaluating a caching system based only on miss rate ignores the fact that components comprising main memory no longer have a uniform access time for every sequence of requests and that non-blocking caches [Kro81] and out-of-order execution can tolerate some miss latencies. Likewise, evaluating a cache based only on execution time generates little insight into how to improve on a design. Finally, evaluating a cache based only on average memory access time ignores the fact that memory accesses are generally bursty in nature and difficult to spread out evenly over the lifetime of a program.

Metrics that focus on characteristics of the references presented to the cache provide greater insight and guidance in the design of cache systems than do hit rate, average memory access time, or even execution time. This assertion is motivated by a simple observation: two caches with precisely the same hit rate (or average memory access time or execution time) may achieve that performance in quite different ways. These differences have important implications for certain aspects of modern cache design; multi-level caching systems for example.

Consider a permutation of the example presented in the introduction: a loop that accesses memory for a vector element and a global variable in each iteration, and for which all other code and data references reside in registers. Let 0 and N be the addresses of the global data and let 1 through N-1 represent the addresses of the vector elements: (0, 1, N, 2, 0, 3, N,..., 0, N-1, N)*. The first iteration of the inner loop generates the addresses 0, 1. While this is a contrived example, it is useful to illustrate a few points.

The references that miss when this example is presented to a direct-mapped cache of size N and a fully associative, LRU cache of size N will be the same in number, but have different forms. In the direct-mapped cache, 0 and N will conflict but all the references to the vector will fit in the cache. As a result, once the cache is primed, the misses of the direct-mapped cache will be (0, N)*. For the fully-associative LRU cache, however, 0 and N will stay in the cache, but the vector references will always miss. The resulting misses are (1, 2, 3, 4,...)*. In both cases there is exactly one miss per loop iteration, but the references that miss in the direct-mapped cache are easily captured by a small victim cache, while those that miss in the fully-associative cache need an N-1 size structure to capture.

This does not mean the direct-mapped cache is generally better, but that the effectiveness of a caching system is not fully described by hit rate. This is true even for a single-level cache hierarchy because main memory no longer has a uniform access time. In the example above, if the two reference streams are going directly to a main memory composed of a single DRAM, it may be preferable to have the output of the fully-associative cache because vector references will often hit in the row- or page-buffer, where they can be accessed more quickly than access sequences without spatial locality.

4.2 Two New Measures of Effectiveness

To address the issues with performance measures described above, we have developed two alternative measures. The definitions of these measures were chosen for simplicity and correlation to intuition, to expand the concept of caches-as-filters, and to explore the usefulness of non-aggregate measures to memory hierarchy analysis. Others may choose different such definitions.

The first new measure is the *instantaneous hit rate*, h_i. The usual definition of hit rate averages over all references in a string. By contrast, h_i is a function measured at *each point* in the reference string, to emphasize recent behavior. The definition is:

$$h_i = \delta_i + \sigma \cdot h_{i-1}$$

where δ_i is 0 if the i^{th} reference is a miss and 1 if it is a hit, and $0 \le \sigma \le 1$. This definition exhibits the desirable property of decreasing the contribution of hits and misses according to how far they occurred in the past.

In our experiments, we chose σ = 1/2, because this measure may be useful at run-time to indicate phases of reference behavior and dividing by two has a fast implementation. Note that with σ = 1/2, h_i has a maximum value of 2. This happens because a series of cache hits yields the sequence: 1, 3/2, 7/4, 15/8, **….** Each element of this sequence can be represented by:

$$\max\ h_{i-1} = \frac{2^n - 1}{2^{n-1}} = 2 - \frac{1}{2^{n-1}}$$

This expression has a limit of 2 as n approaches infinity. Thus

$$\max\ h_i = 1 + \frac{1}{2}\max\ h_{i-1} = 1 + \frac{1}{2}(2) = 2$$

The second new measure is that of locality in the reference string. To give substance to the intuitive notion, the *instantaneous locality,* $l_{i,}$ of a reference in a string $a_0, a_1, a_2, \ldots$ is defined as:

$$l_i = \sum_{j=0}^{i-1} \frac{1}{|a_i - a_j| + 1} \times \frac{1}{|i-j|}$$

The precise form of this definition is not critical, but this particular form attempts to follow our intuition:

- The first term in the product corresponds to *spatial locality*; by forming the difference between two references, the term is larger when the two addresses are closer together.
- The second term loosely corresponds to a notion of *temporal locality*; weighting the spatial components by the positional difference makes the term larger for references that are closer together in the reference string.

The product of these terms is largest for references that are close both spatially and temporally. By summing over all previous references, we get a measure that is large when there are many prior references that are spatially and temporally local. As a practical matter, since reference strings can be exceedingly long, we generally do not sum from $j = 0$, but rather sum from $j = (i - w)$ for some window size *w*. If *w* is sufficiently large, the terms for smaller *j* are irrelevant to our results, and they can safely be ignored:

$$l_i \approx \sum_{j = i - w}^{i - 1} \frac{1}{|a_i - a_j| + 1} \times \frac{1}{|i - j|}$$

This definition of instantaneous locality has at least two immediate uses. First, the difference in locality between the input and output of a cache provides an alternative and enlightening figure of merit for caches. We believe l_i is a more informative measure than an aggregate miss ratio. The ideal composite of cache filters removes not only the most references possible, but removes all available locality (i.e., any that may be present in the input reference string). This new figure of merit corresponds closely to intuition about the quality of a cache. Caches exploit locality to intercept and remove references. Thus any locality left in the output string signals a failure of the cache (or that the corresponding misses are compulsory). Although, it is the existence of this locality that gives us hope that another level of cache can be effective. Alternatively, if there is no locality in the input to start with, a high miss ratio is not a sign of cache failure.

The second immediate use is that both the instantaneous locality and hit-rate measures give us insight into the underlying patterns of the reference string and the effect of a cache on that string. For example, they reveal that locality is “bursty” — real reference streams tend to have regions of high locality separated by regions of relatively low locality. [Bat76, Bat77, Mad76] Even after a reference string is filtered by a cache, we may still see regions of relatively high locality.

These observations are demonstrated in the example output of Figure 18. The trace is a small portion of the KENBUS workload from the SPEC SDM 1.1 workload suite, captured by the BACH system from Brigham Young University [Gri93]. This workload simulates a multiuser environment by executing a series of common system commands. Figure 18 (a) shows the locality of the input reference string. Figure 18(b) is the instantaneous hit rate as determined by an 8K, direct-mapped, 1-byte line size cache, and Figure 18(c) is the instantaneous locality of the reference string output from this same cache. The reference strings include both instructions and data references. Each dot and hash mark graphed represents one reference in the original string. The window, *w*, of previous references in l_i is 100. The value of σ in h_i is 0.5.

Notice the areas of higher locality in the graph for the input reference trace of Figure 18(a). The peaks indicate there are periods during which a trace may have more locality and a cache can be very effective in eliminating references, but other periods in which it is not effective at all, resulting in bursts of misses as shown in Figure 18(b) and (c). The extent of the burstiness is important because the longer the bursts are, the more difficult it is to minimize the memory latency seen by the processor.

The filtering effect of the cache is also visible in Figure 18. Notice how the spikes in the input locality graph, (a), are filtered out in the output locality graph, (c). Finally, these new measures allow us to identify and analyze regions of higher locality to examine the fine structure of their references and determine what source code constructs generated the higher locality. Examples of this are included in the following section.

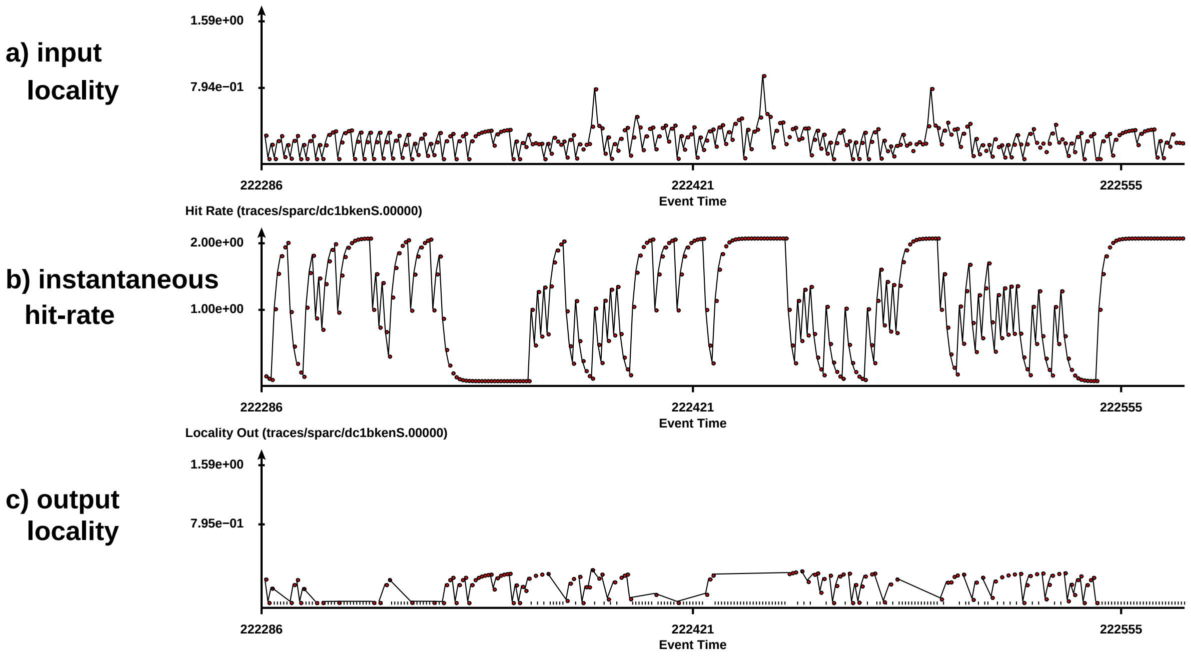


FIGURE 18: Example Measure Output, KENBUS Trace

As mentioned above, we do not claim that the precise definitions we use for h_i and l_i are the best possible measures and we have not run exhaustive experiments on the effect of each parameter on each measure under consideration. Experiments to date have focused on the qualitative aspects of these measures and their ability to provide more insight into the nature of reference patterns.

4.3 Computed Locality Examples

4.3.1 The Motivating Example

Applying these new measures to the example in Section 4.1(repeated below) yields the results in Figure 19 and Figure 20. Here we use k=1 and N=8192 (8K).

$$[0, 1, 8K, 2, 0, 3, 8K, 4, 0, \ldots, (8K-1), 8K]^*$$

Figure 19(a) shows the instantaneous locality after the above expression has been run through once, and the caches are primed. The output locality of the direct-mapped cache shown in Figure 19(b) exhibits a steady-state instantaneous locality close to 2, once the cache is primed. Figure 19(c) and Figure 19(d) display the instantaneous locality of the output of a fully-associative cache. Random replacement yields more hits and a very different output pattern, shown in Figure 19(c). The line from the last locality point visible in Figure 19(c) connects it to the one for the next reference. Since the next miss coming out of the cache happens far in the future, it does not appear on this graph. This separation occurs because references retain their positions from the original input trace. With LRU replacement, the reference string from the primed fully-associative cache would repeat $[1, 2, 3, 4, \ldots, (8K-1)]^*$, and the graph of its instantaneous locality would resemble the picture in Figure 19(d), leveling off at a value close to 1, and dipping slightly when the pattern begins again.

The fact that the direct-mapped cache produces higher l_i values indicates that there is more locality left in the output string. Hence another level of cache would be more effective here than it would be backing the fully-associative cache. This also means that the direct-mapped cache is not as effective at exploiting the locality in its input string, and perhaps a stand-alone direct-mapped organization is not the best design choice for this level of the hierarchy with this reference string. A direct-mapped cache backed by a small victim[1] cache, however, might be more effective than one or more levels of associative cache.

[1] The victim cache (developed by Jouppi [Jou90]) refers to a small fully associative cache placed after the first level cache to capture conflict misses.

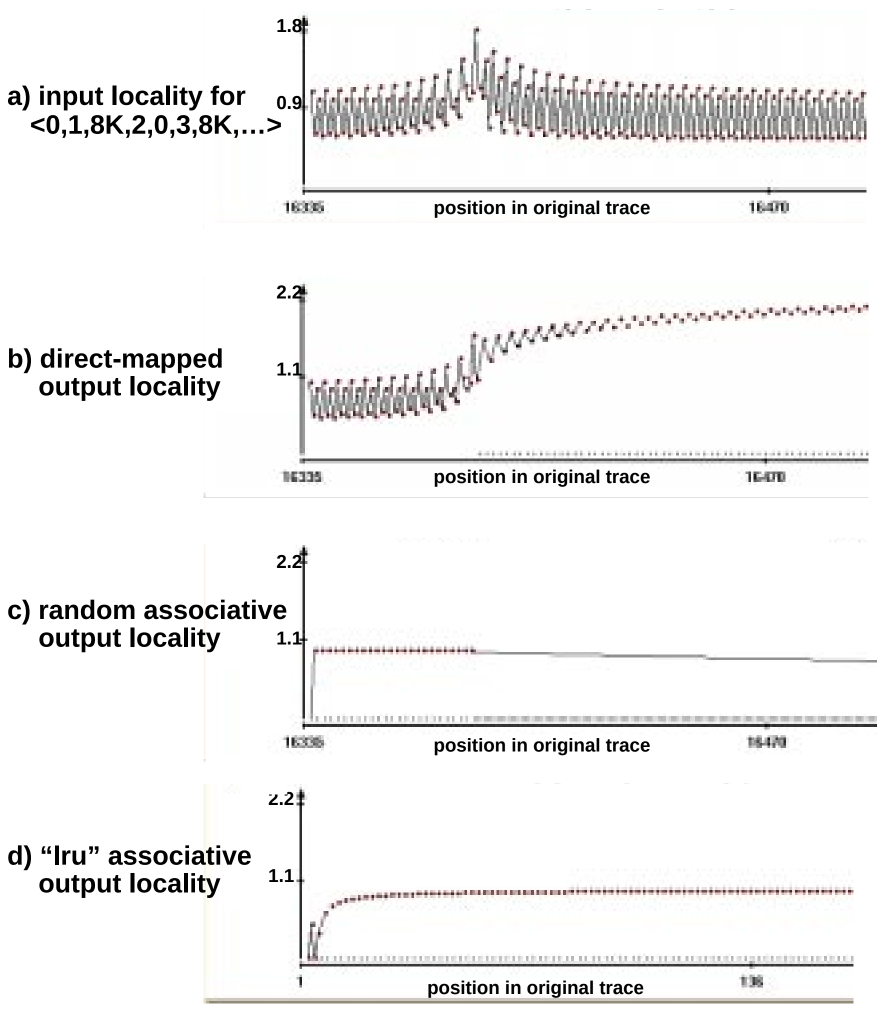


FIGURE 19: Motivating Example: Instantaneous Locality Transition to Steady-State

Figure 20 shows the instantaneous hit rate for the motivating example reference string after the caches are primed, with the input from Figure 19(a). Unlike an aggregate measure, the instantaneous hit rate shows for what part of the trace the cache is performing well, and for what part it is not. Further attention can then be given to the problem regions. For example, looking at the instantaneous locality for a region of poor performance can determine if the input string has any significant amount of locality for the cache to use. Looking at the trace itself in that region will identify the reference pattern that exhibits the problem.

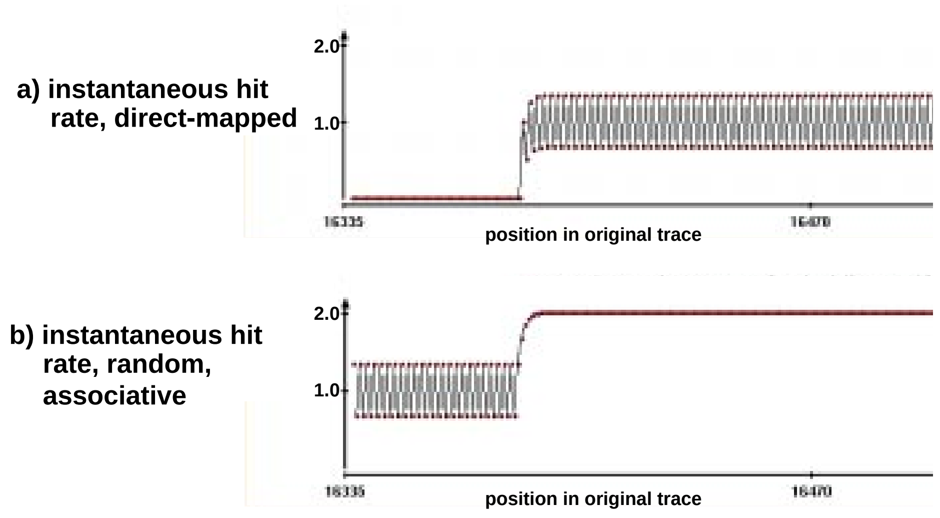


FIGURE 20: Motivating Example: Instantaneous Hit Rate Transition to Steady-State

4.3.2 Recognizing Program Constructs

To get a better feel for what the instantaneous locality measures can tell us, we used TSpec to describe a simple loop accessing multiple data streams (*daxpy*) and a loop nest accessing two-dimensional arrays (*matmul*). Figure 21 shows pseudocode for these program fragments. Figure 22 through Figure 27 show the locality measures for 8K caches. In the discussion that follows, three categories of program constructs are identified. One is a stream, or vector-like access. The other two are code loops; one single loop and one doubly-nested loop.

daxpy	`for i = 1 to 10000`
	`    y[i] = y[i] + a * x[i]`

matmul	`for i = 1 to 10`
	`    for k = 1 to 10`
	`        reg = x[i][k]`
	`        for j = 1 to 10`
	`            z[i][j] += reg * y[k][j]`

FIGURE 21: Pseudocode for Program Constructs

4.3.2.1 Daxpy Example

Figure 22 shows input and output locality from start-up for the code portion of a 10,000-element *daxpy* computation. Each loop iteration is easily discerned in the input locality pattern in Figure 22(a) because each single loop forms a hump in the locality graph. Once the loop is loaded, all references hit in the instruction cache, thus all of the locality from the input in Figure 22(a) is filtered out in the output in Figure 22(b). This output was generated with a direct-mapped cache, but because this loop fits in cache, results for both replacement policies are nearly identical (e.g., see the combined reference string output of both caches in Figure 24).

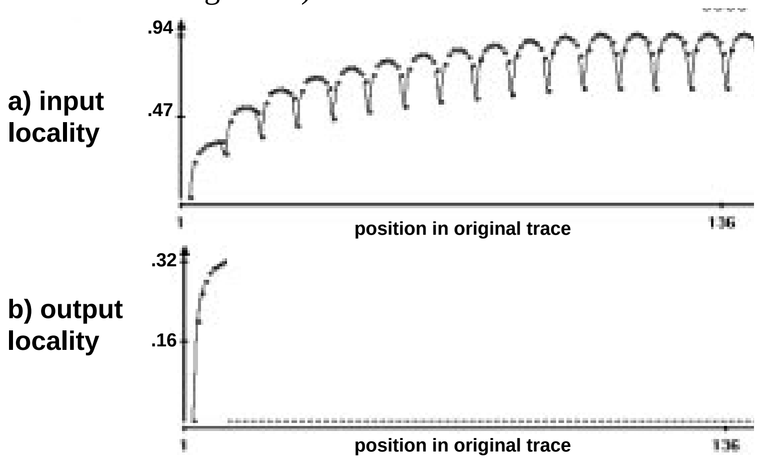


FIGURE 22: Instantaneous Locality for *daxpy* Instructions, DM or FA cache

Figure 23 shows the locality graphs at start-up for the data portion of the *daxpy* trace. It's easy to see the pattern in the input of Figure 23(a). The first three references, to *a*, *x[1]*, and *y[1]*, have relatively little locality, but the instantaneous locality value rises dramatically at the second reference to *y[1]*. The rising curve defined by every fourth dot represents the repeated references to scalar *a*. In the direct-mapped cache output of Figure 23(b), the locality from the repeated references to *a* and *y[i]* are filtered out (the repeated sets of two ticks just above the *x* axis indicate where these cache hits occurred). Since the data set does not fit in an 8K cache, the patterns at the right end of the locality graphs in Figure 23 will repeat throughout the computation for both the direct-mapped and associative data caches. Notice how in both parts of Figure 23 the locality values representing sequential references to a vector (whether it be to vector a with stride zero, or x and y, each with stride one) ramp up to a point and then become a straight line. This ramping up to a straight line is the basic pattern behind every stream. (For more details see Section 4.4.1) The actual value the line approaches is dependent on the stride of the stream and the separation between individual references to that stream

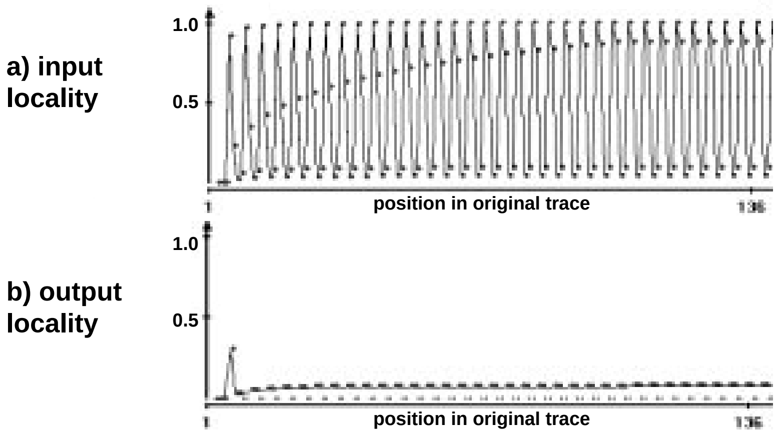


FIGURE 23: Instantaneous Locality for *daxpy* Data

Figure 24 shows what happens with *daxpy* for a unified instruction and data cache, either direct-mapped or 4-way set associative. The loop iterations are still evident in the repeated patterns in the input of Figure 24(a). It is also possible to see the stream patterns from Figure 23(a). The output locality in Figure 24(c) and Figure 24(d) resembles the output locality of Figure 23(b) spread out in time. In all these traces, each new reference to *x* and *y* misses the cache, but all other instruction and data references hit. The instantaneous hit rate in Figure 24(b) demonstrates this graphically. The value of h_i drops at precisely two points during each loop iteration corresponding to each reference to *x* or *y*. The replacement policy never comes into play.

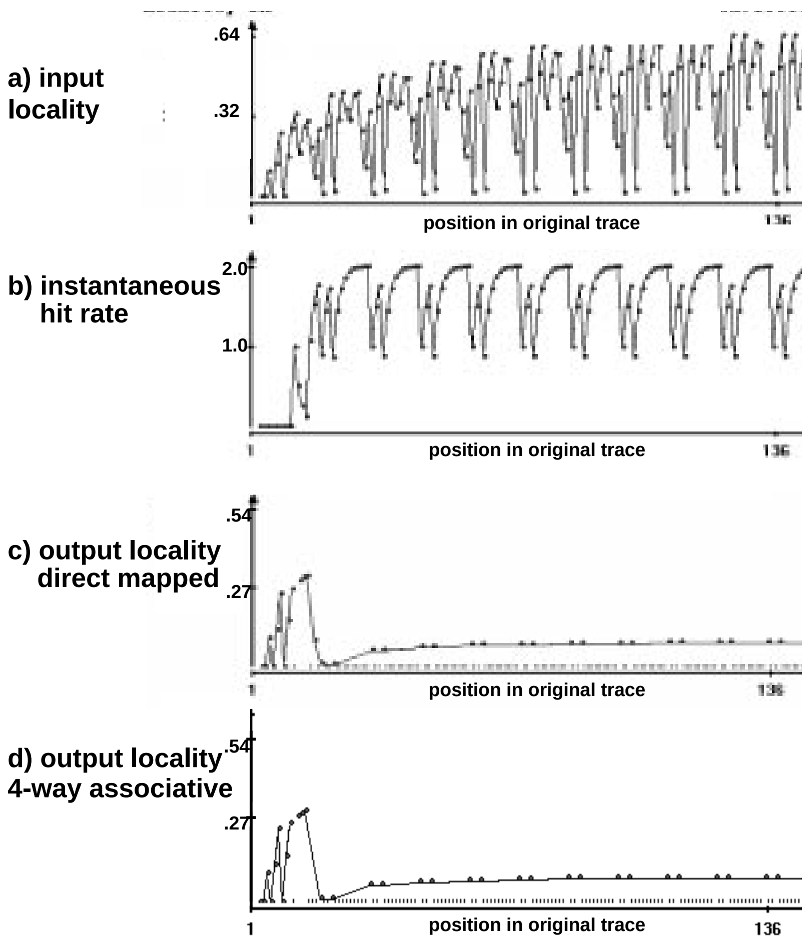


FIGURE 24: Results For *daxpy* Combined Trace

4.3.2.2 Matmul Example

Figure 25 through Figure 27 illustrate results for matrix multiplication on 10x10 matrices. We choose the small problem size to make patterns in the resulting graphs more readily apparent. The code description for this computation is modeled after the assembly language output of gcc on an HP PA-RISC. There are 36 static instructions in the *j*-loop, 27 more in the *k*-loop, 5 more in the surrounding *i*-loop, and 5 in the prologue. The nested loops give rise to a more complex locality pattern than the simple *daxpy* loop, as evidenced by the graph in Figure 25(a).

The prologue and the first iteration of each loop generate roughly the first inch of the input locality graph in Figure 25(a); the instruction addresses for this segment are sequential, so input locality rises steadily as for a stream. The numerical value of any locality point is much less important than the patterns the points create. The larger dips signal the backward branch at the end of the *j*-loop. The entire segment depicts 3+ iterations of the innermost loop, or roughly 140 total instructions.

Figure 25(b) shows the instantaneous hit rate, which is the same for both the direct-mapped and the 4-way set associative cache. Once the loops are loaded, the computation runs entirely from cache. Since there are no cache misses until all 10 iterations of the *j*-loop have finished, there is no locality left in the cache output shown in Figure 25(c). (Recall that the small hash marks at the bottom of the graph represent hits in the cache.)

Figure 25(d)-Figure 25(f) show the locality and hit rate graphs at the end of the first iteration of the *k*-loop. The dip in input locality in Figure 25(d) at the same place that the instantaneous hit rate drops in Figure 25(e) marks the execution of the remaining instructions in the intermediate loop (and the corresponding cache misses), then the pattern for the inner loop picks up again in the right half of Figure 25(d). Notice the pattern of the doubly nested loop apparent in the combination of Figure 25(a) and (d). Smaller dips indicate the backward branch in the inner loop and the less frequent large dip in (d) corresponds to the backward branch of the outer loop. By this point, all instructions for all three loops are resident in cache, and there are no more instruction misses in the entire trace. This can be seen in the low output locality in Figure 25(f).

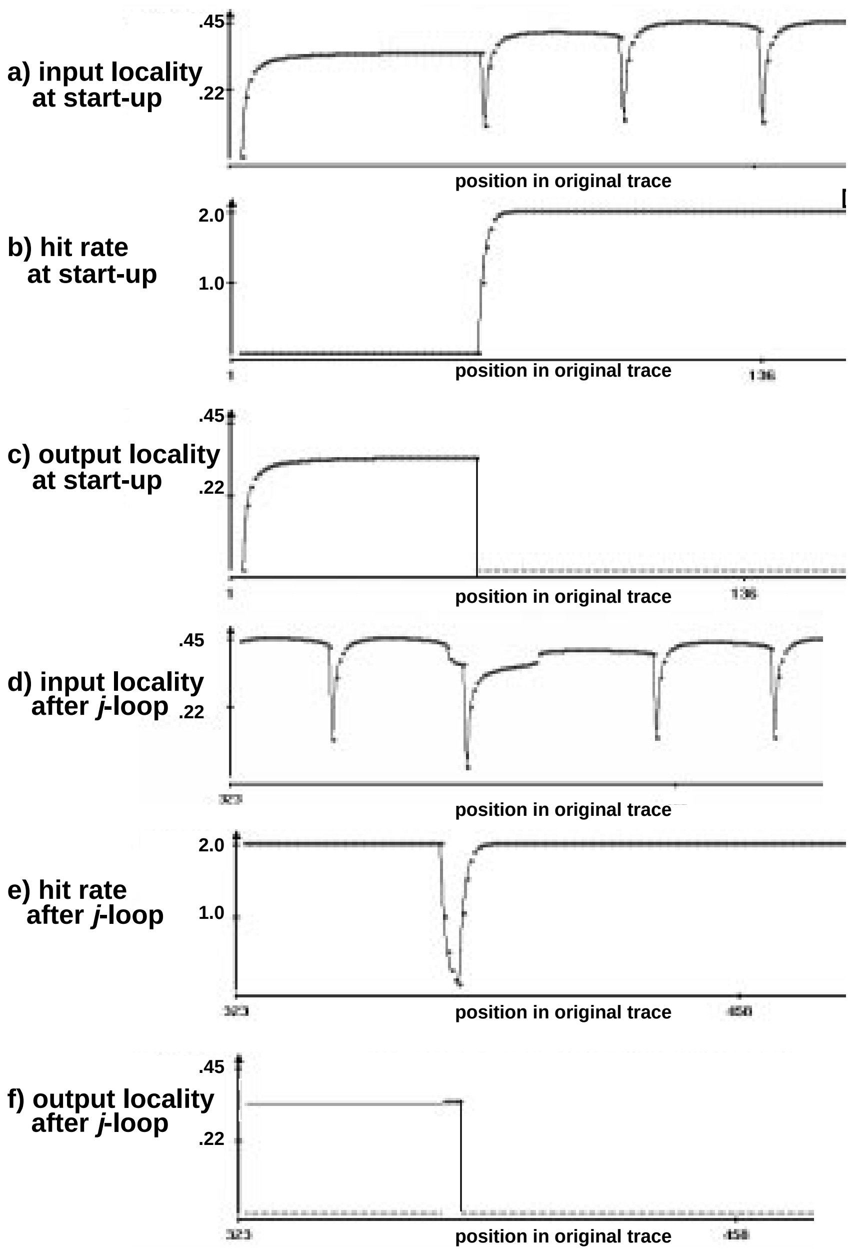


FIGURE 25: Results For *matmul* Instruction Trace

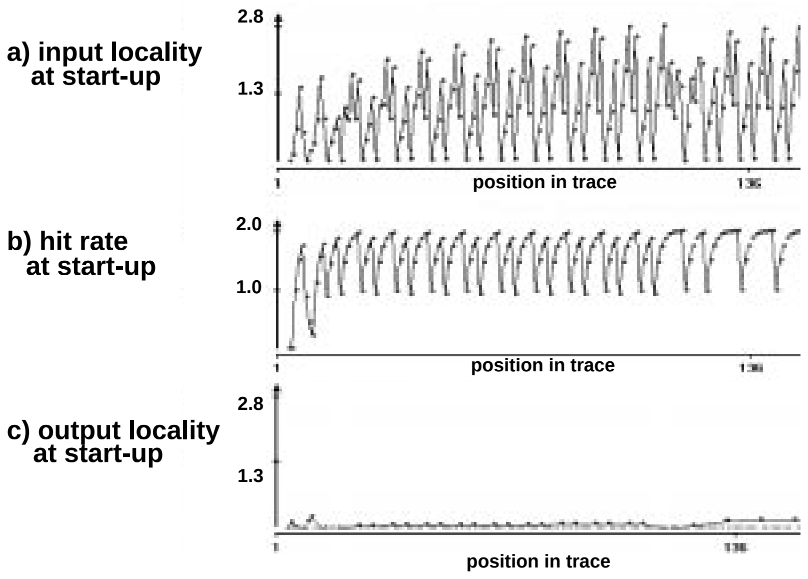


FIGURE 26: Results For *matmul* Data Trace

Figure 26 illustrates data locality at the beginning of the *matmul* trace. The peaks in the input locality graph represent references to automatic or temporary variables on the stack in between references to arrays *y* and *z*. We can see one iteration of the *k*-loop ending where the locality pattern in Figure 26(a) dips slightly when we load the next element of *x*. The locality output in Figure 26(c) shows that stack references and repeated accesses to *z* are filtered out by the cache. Recall that we modeled a cache with very short lines; one with longer lines could take more advantage of spatial locality among the array references

Figure 27 shows the combined code and data string as the *k*-loop completes its first iteration. The hit rate in Figure 27(b) drops as the cache services the compulsory instruction misses we saw in Figure 25(e). The locality output in Figure 27(c) reflects the drop in hit rate during this transition from inner loop to intermediate loop.

.

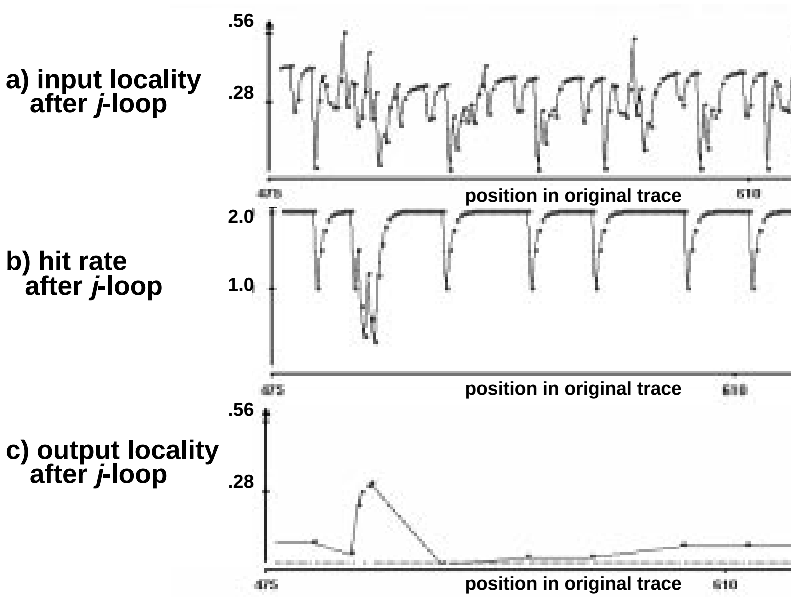


FIGURE 27: Results For *matmul* Combined Trace

In this section, we have taken two program fragments, and have decomposed the full reference traces coming from the CPU, separating the strings into individual primitives representing code and data. In applying our locality measures to the individual and merged strings, we have observed that one can discern the contributions of individual scalars or data strings in the locality graphs, as for the scalar variable *a* in *daxpy*, the stack references in *matmul*, or the vector accesses in both benchmark fragments. In addition, loops are also clearly visible as “hump” patterns.

4.4 Analytical Locality Measures

Now that these new measures are defined, we show their relationship to the analysis methodology presented in the last chapter by demonstrating the instantaneous locality applied to two primitives: streams and code loops. Recall that in the analysis methodology developed earlier, we decompose each reference string into primitives. If the locality measure can be defined analytically for these primitives and if L(F(T1 & T2)) = L(F(T1) & L(F(T2)), we will be able to see the effect of the filter functions on the locality measure analytically.

Recall the copy example:

<!f, !t, (!c, c*3, f, c*2, t, c)*3>

Consider the three primitives:

P1 =   (!c, c*3, λ, c*2, λ, c)*3
P2 =   !f, (λ*3, f, λ*2, λ, λ)*3
P3 =   !t, (λ*3, λ, λ*2, t, λ)*3

To compute the locality measure exactly as described above, we need to have the address values represented by λ in these primitives. If we make the simplifying assumption that all variables have a large distance from each other so that the locality contribution of any address with respect to λ or an address from a different primitive is effectively 0, the instantaneous locality can be computed analytically. The revised instantaneous locality, l_i, is defined as follows:

$$l_i = \begin{cases} 0 & \text{if } a_i = \lambda \\ \sum_{j = i-w}^{i-1} l_{ij} & \text{otherwise} \end{cases}$$

$$\text{where } l_{ij} = \begin{cases} 0 & \text{if } a_j = \lambda \\ \dfrac{1}{|a_i - a_j| + 1} \times \dfrac{1}{|j - i|} & \text{otherwise} \end{cases}$$

With this extended definition, L(F(T1 & T2)) = L(F(T1) & L(F(T2)) holds because only the locality contributed by addresses within the same primitive contribute to the locality of a particular reference, *even in the merged representation of the trace*.

4.4.1 Analytical Instantaneous Locality for Streams

Working through some complicated algebra, a closed form for the analytic locality of a stream can be found. Let ***T = <!f, (λ*, f, λ*)*>***. Then,

$$l_i(t_i) = \begin{cases} \displaystyle\sum_{j=1}^{(i-i_b)\ \mathrm{div}\ i_e} \frac{1}{j \times st + 1} \times \frac{1}{j \times i_e} & \text{otherwise} \\ 0 & \text{if } t_i = \lambda \end{cases}$$

where i_b = index of the base address of the stream, *st* refers to the stride of the stream, and i_e is the index difference between the stream references in the overall string (i.e., the number of λs in the primitive plus 1). This makes j range from 1 to the number of previous non-λ references in the stream and $j \times i_e$ represents the distance (in indices) between $\boldsymbol{a_i}$ and the previous non-λ reference. Note that the stride from the original TSpec description is necessary to perform this computation. As with the computed version, this measure corresponds closely to intuition:

- The first term in the product corresponds to *spatial locality*; by forming the difference between two references, the term is larger when the two addresses are closer together.
- The second term loosely corresponds to a notion of *temporal locality*; weighting the spatial components by the positional difference makes the term larger for references that are closer together in the reference string.

As an example, consider the specific stream described by:

f(100_r, 4);
<!f, (λ*2, f, λ)*4>

When executed, this description yields:

T = < <λ, 1>, <λ, 2>, <100, 3>, <λ, 4>, <λ, 5>, <λ, 6>, <104, 7>, <λ, 8>, <λ, 9>, <λ, 10>, <108, 11>, <λ, 12>, <λ, 13>, <λ, 14>, <112, 15>, <λ, 16>>

If t_i ***= <112, 15>***, then

$$l_i(t_i) = \sum_{j=1}^{(15-3)\text{ div }4} \frac{1}{j\times 4+1}\times\frac{1}{j\times 4} = (1/5)\times(1/4)+(1/9)\times(1/8)+(1/13)\times(1/12)$$

$$= (1/20)+(1/72)+(1/156) = 0.70299$$

which, as we expect, corresponds to the “computed” version from Section 4.2:

$$l_i = \sum_{j=i-w}^{i-1} \frac{1}{|a_i-a_j|+1}\times\frac{1}{|j-i|} = 0+0+\frac{1}{112-100+1}\times\frac{1}{|3-15|}+0+0+0$$

$$+\frac{1}{112-104+1}\times\frac{1}{|7-15|}+0+0+0+\frac{1}{112-108+1}\times\frac{1}{|11-15|}+0+0+0=$$

$$(1/13)\times(1/12)+(1/9)\times(1/8)+(1/5)\times(1/4) = 0.70299$$

4.4.2 Analytical Instantaneous Locality for Code Loops

In a similar fashion, a closed form of the analytic instantaneous locality of a code loop can be derived. Let ***T = <(!c, c*)*>*** and i_b and i_e are the index of the first instance of the base address and the number of elements in the loop respectively. Then,

$$l_i(t_i) = \sum_{j=1}^{i-i_b} \frac{1}{|((i-j-i_b)\bmod i_e)-((i-i_b)\bmod i_e)|\times st+1}\times\frac{1}{j}$$

Here ***j*** moves backwards in the trace from ***i-1***, summing the relative localities. The difference between the addresses corresponding to the element t_i and the element referred to by ***j*** is determined by their relative positions in the inner loop. $((i-j-i_b)\bmod i_e)$ determines the position in the inner loop of the ***j*** element, and $((i-i_b)\bmod i_e)$ determines the position of the ***i*** element. The difference between these two positions multiplied by the stride gives the distance between the two addresses.

As an example, consider the specific code loop as specified by:

c(100_r, 4);
<(!c, c*3)*4>

Executing this description yields:

T* = <** <100, 1>,*** <104***, 2>, <108, 3>,*** <100***, 4>,*** <104***, 5>,*** <108***, 6>,***
<100***, 7>,*** <104***, 8>, <108, 9>,*** <100***, 10>,*** <104***, 11>,*** <108***, 12>>***

If ***t_i = <104, 5>***, then

$$l_i(t_i) = \sum_{j=1}^{5-1} \frac{1}{|((5-j-1) \bmod 3) - ((5-1) \bmod 3)| \times 4 + 1} \times \frac{1}{j} =$$

$$\frac{1}{5} \times \frac{1}{1} + \frac{1}{5} \times \frac{1}{2} + \frac{1}{1} \times \frac{1}{3} + \frac{1}{5} \times \frac{1}{4} =$$

It is left to the reader to see that this corresponds to the computed version.

Closed-form analytical versions can be derived for other basic patterns found in the filter function dictionary. Other patterns generate much more complicated formulae and little insight is gained by their additional derivation. In practice, the locality measure of the specific primitives under evaluation is straightforward to compute. The main point behind these analytical versions is not the precise form. Rather it is the idea that they can be derived without using the actual addresses present in the basic pattern by focusing on the relationships between the addresses in the primitive. This allows us to gain insight into a large number of specific trace patterns without computing the locality for each individual trace. The next sections shows graphs of the analytic locality of several different streams and code loops to demonstrate their general form.

4.4.3 Analytical Locality Graphs for Streams

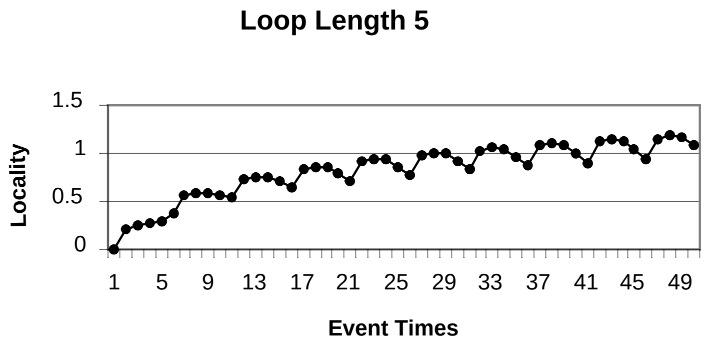
Figure 28 illustrates the instantaneous locality for a variety of streams. These values

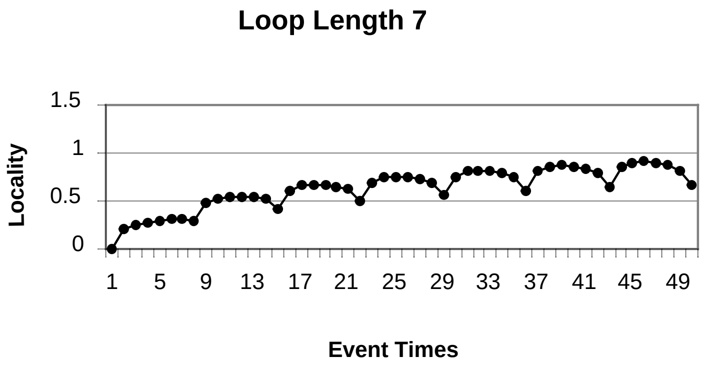
FIGURE 28: Stream Analytical Locality

were obtained using ***f(?, stride); <!f, f*>***, as the TSpec description of the stream. The different lines are the result of varying the size of the stride. Notice that the Stride 0 stream is a maximum value for the locality measure and does not have a limit. A Stride 0 stream represents repeated references to the same location, generating a locality value that increases steadily. Using the measure in this form has the feature that the longer a trace is, the greater its locality for capacity or the greater its locality can be. All other stride values approach a limit, which decreases as the stride increases.

4.4.4 Analytical Locality Graphs for Code Loops

Figure 29 shows the basic shape of the analytic locality of code loops of differing lengths. The results were obtained using the TSpec description, ***c(?, 4); <(!c, c*length)*>***





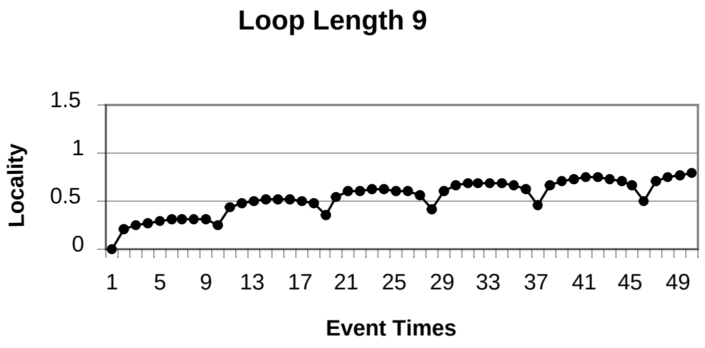


FIGURE 29: Analytic Locality For Code Loops (!c, c+*)*

where the length is varied between 5 and 9. As the loop gets longer, the locality value decreases and the humps representing the loops get wider.

FIGURE 30: Analytical Locality for Nested Code Loop
(!c, c*3, (c4, c*3)*3)*

Figure 30 shows the analytic locality for a nested code loop. The TSpec description used here is:

c(?, 4);
<(!c, c*3, (c4, c*3)*3, c*2)*>

Each major hump in the graph represents the locality of an iteration of the outer loop. The lowest points at the beginning of each major hump represent the start of the loop where the locality is lowest because of a jump back to the beginning of the loop. Each major hump consists of first three increasing locality points for the initial three references at the beginning of the outer loop, then three sets of three points generating smaller humps that represent iterations of the inner loop, then a final two decreasing locality points representing the final two references in the outer loop.

4.5 Example - Computed Locality With Real Traces

Next we will look at an example of the computed locality on a real trace with a realistic cache system. Consider a cache hierarchy similar to one of the Intel Pentium® Pro chip sets [9], with L1 instruction and data caches that are both of size 8K with 32-byte lines. The L1 data cache is 2-way set associative, and the instruction cache is 4-way set associative. The L2 cache is unified, 256K, 4-way set associative with 32-byte lines. We compared this L2 with two other unified, 256K caches, both direct-mapped, but the first has 32-byte lines, and the second has 128-byte lines. Our experiments used random replacement, which differs from the LRU and pseudo-LRU policies of the Pentium® Pro, but this difference in policies does not materially affect our conclusions

When we analyzed the 061.kenbus1 [Gri93] trace for these hierarchies, we found that the direct-mapped L2 with 128-byte lines performs much better than either of the L2 caches with 32-byte lines. The two 32-byte line L2 caches performed very similarly. Figure 31 shows input and output localities for a section of the trace. It is evident that the associative L2 cache with 32-byte lines fails to capture much of the available locality in the reference string: the locality output graph in Figure 31(c) is almost identical to the locality input graph in Figure 31(a). In contrast, there is relatively little locality left in Figure 31(e) for the direct-mapped L2 with 128-byte lines.

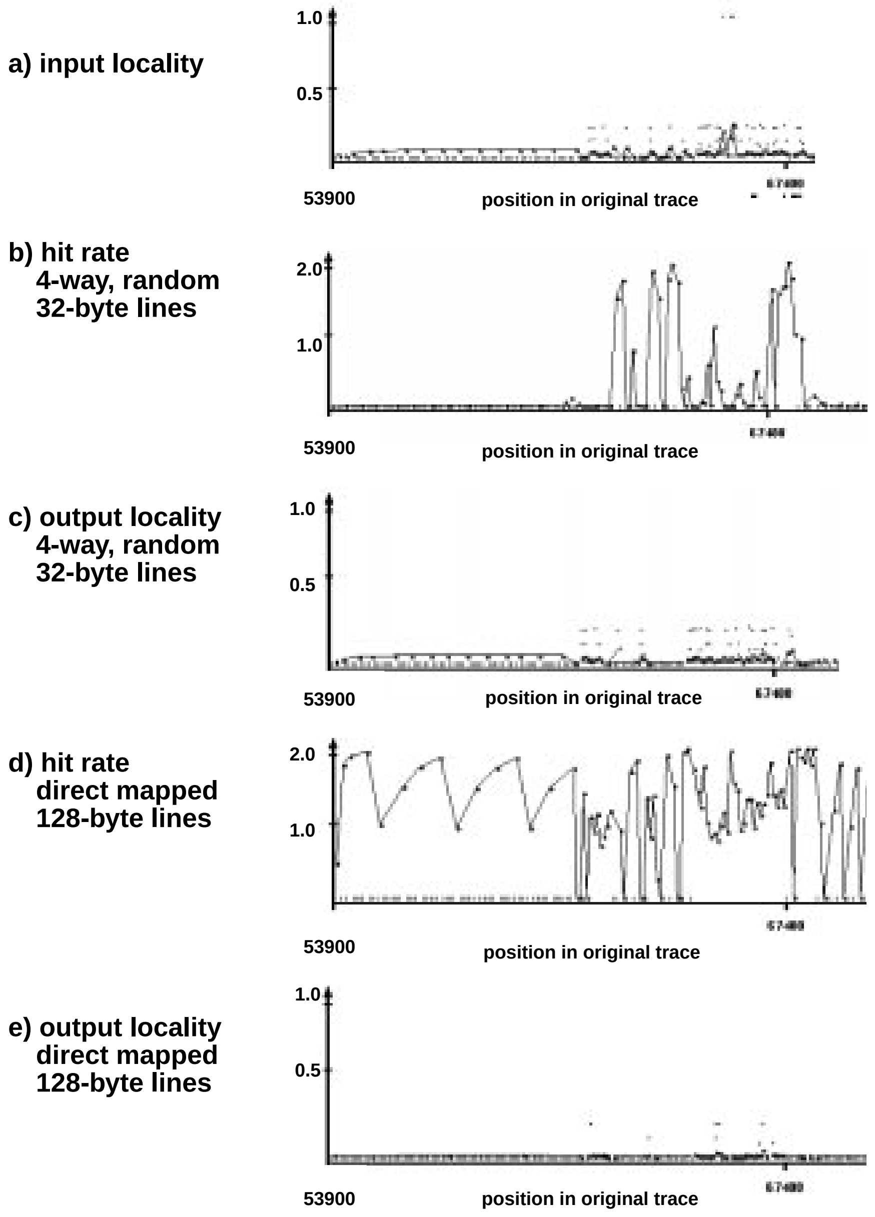


FIGURE 31: Comparison of Two L2 Configurations

Based on our analyses, our intuition is that the cache at level *N+1* should be designed with a different organization and/or replacement policy from the cache at level *N*. Otherwise both caches are likely to miss on the same kinds of inputs. For instance, our results suggest that longer lines are preferable in an L2 cache backing L1 caches similar to the Pentium® Pro’s.

We cannot draw conclusions about the L2 replacement policies, since 450,000 references is not enough to observe performance differences between direct-mapped and set-associative L2 caches of this size. Nonetheless, we expect that making the L1 and L2 replacement algorithms different will also yield better performance. Comparing aggregate hit-rates should indicate that other L2 configurations may perform better than the implementation with 32-byte lines, but would not be as useful in explaining why. Future work will test this hypothesis for a range of replacement policies.

4.6 Conclusions

We have introduced the concept of viewing caches as filters, and have presented the results of and observations on some initial experiments with this new approach to memory hierarchy performance analysis. We have demonstrated that the instantaneous hit rate and the instantaneous locality measures can give us more insight into memory referencing behavior than the traditional aggregate hit rate, related these measures back to our analysis methodology with analytical forms of the measures, and shown an example of these measures in analyzing the effectiveness of a particular memory hierarchy. The next chapter will use aspects of the new framework on specific examples to demonstrate its effectiveness.

4.7 Related Work

The traditional measures of the quality of a caching strategy have been aggregates such as the miss rate. Other measures break down summary performance data spatially or according to bandwidth requirements. For instance, Tyson *et al.* perform a detailed characterization of cache behavior for individual load instructions [Tys95], and Abraham *et al.* study the memory referencing behavior of individual machine-level instructions [Abr93]. Both studies confirm that a very small number of load and store instructions account for a majority of data cache misses. Evidence that misses are bursty in both time and space are available in Thiebaut's work regarding the fractal dimensions of computer programs and the work of Voldman and Hoevel regarding software-cache interactions. [Thi89, Vol81] Johnson *et al.* measure spatial reuse fractions for cache lines, finding that over half the time data fetched in a cache with a uniform, large line size wastes bus bandwidth and cache space [Joh97]. Huang and Shen measure the average bandwidth requirements of a program as a function of available local memory [Hua96], and Burger *et al*. calculate traffic ratios, traffic inefficiencies, and effective pin bandwidths for different levels of the memory hierarchy, arguing that pin bandwidth will be a severe performance bottleneck for future microprocessors [Bur96].

New approaches to characterizing program locality make it possible to represent and discuss locality and caching properties in concrete terms. Brehob and Enbody propose a mathematical model of locality that uses the distance between references in a trace to capture temporal locality, and a correspondence to cache lines to capture spatial locality [Bre96]. Grimsrud *et al.* introduce a method of quantifying the locality in a trace and visually representing it as a three dimensional surface [Gri96]. They explore some of the properties of this formulation, and show the correlation between graphical features and specific reference patterns, demonstrating the utility of their locality measure through two applications as a visualization tool: characterizing and summarizing workload locality, and evaluating the effectiveness of benchmarks in exercising memory hierarchies.

Although these kinds of summary data provide some insight into characteristics of the benchmark being analyzed, they do not provide details about cache behavior during specific phases of the program's execution or how cache behavior changes over time. McKinley and Temam take a step towards more detailed analysis by quantifying the locality characteristics of numerical loop nests [McKK96]; their locality measurements reveal important differences between loop nests and whole programs, and refute some popular assertions, but like Brehob and Enbody's, their approach presents results as histograms of the locality distributions for the parts of programs in question. In contrast, our approach aims to provide much more than summary information.

Finally, Rivers *et al.* [Riv98] catogorize references into four groups, non-temporal-non-spatial, non-temporal-spatial, temporal-non-spatial, and temporal-spatial. They track the movement of references between these categories and use this information to determine which cache should be used for each particular reference.

Chapter 5

Example Analyses

5.1 Introduction

The preceding chapters elaborated on the four components of a new framework for memory hierarchy analysis: the TSpec notation for expressing memory reference traces, the equivalence class concept for identifying a set of traces to analyze, the filter function model of a cache for gaining insight into the effects of a cache on a reference string, and new metrics for evaluating caching systems. This chapter uses the framework in several examples to show it can work for different reference behavior categories, different assembly languages, and multi-level caches. Conditional constructs were covered in Chapter 3. Due to space limitations, we do not give exhaustive analyses for each of these situations; rather, we sample the space with a few indicative examples. In particular, we focus on the TSpec constructs and filter functions necessary to evaluate the examples. Some readers have found it useful to read several sections of Appendix A prior to continuing with the examples in the rest of this Chapter.

5.2 Reference Pattern Applicability

To demonstrate the framework applies to source programs with different types of referencing behavior, we target here three general categories: scientific computation, recursion and pointer chasing (heap use), and conditionals.

5.2.1 Scientific Computation Examples

We use the term scientific computation to refer to code that has a predictable reference pattern consisting of loops that operate on large data structures where the basic access pattern is known from the source code. Examples are daxpy, matmul, and a sampling of the Livermore Fortran Kernels. The C code, assembly language and TSpec translations for all examples are in Appendix B. The TSpec translation for the data accesses of the Livermore Fortran Kernels is in Table 1. These examples were compiled using a vpo (Very Portable Optimizer) [Ben88] backend combined with an lcc front end. The assembly language was generated with one or more of three possible optimization options: completely unoptimized, unoptimized with register allocation, and fully optimized. (For more specific details on the exact optimizations and flag options used, please see Appendix B, Section B.1) In this section, we focus on the unoptimized versions. See Section 5.3.2 for the effect of optimizations.

5.2.1.1 Daxpy Example

Daxpy multiplies a vector, ***x***, by a scalar, ***A***, and stores the result in a vector, ***y***. It consists of one main loop that performs the computation and has three main data structures (***x***, ***y***, and ***A***) that are not put in registers. Here we focus on the SPARC unoptimized assembly with register allocation. The C code and assembly for the kernel are in Figure 32.

C Code:
```
for (i=0; i<N; i++) {
    y[i] += A * x[i];
}
```
Assembly Code:
```
.L2:sll     %o4,3,%g1
    add     %g1,%o1,%o3
    ldd     [%o3],%f0
    sethi   %hi(L7_78297),%o5
    ldd     [%o5 + %lo(L7_78297)],%f2
    ldd     [%g1 + %o0],%f4
    fmuld   %f2,%f4,%f2
    faddd   %f0,%f2,%f0
    std     %f0,[%o3]
    add     %o4,1,%o4
    cmp     %o4,%o2
    bl      .L2
    nop
```

FIGURE 32: Daxpy Kernel, C and SPARC Code

The TSpec for the unoptimized assembly code with register allocation is:

c(?_r, 4);
y(?_r, 4, 8);
x(?_r, 4, 8);
A(?_r, 4);

< !all, c*4, $(_{L2}c_4$, c*2, $y_{+\sim}$, $y_{-\sim}$, c*2, A+, A-, c, $x_{+\sim}$, x_{-+},
c*3, $y\_w_{+\sim}$, y_w_{-+}, $c*4)_{L2}$*5, c*2>

The c variable represents the references to the code. The ***y***, ***x***, and ***A*** variables represent the same variables in the source code. Each load double in the assembly code is represented by two references to the variable in the TSpec representation. The first increment value in each definition is for the intra-word access. The second increment value is for the element access. If appropriate, a load double could be modeled as one word, or fetch size in the TSpec. The double fetch is used here to provide a different example than those presented in Chapter 3. The four initial code references and two final code references represent code outside the loop, L2, and not shown in the assembly code.

To model the double fetch of the scalar, A, only an increment with the value of the word size is needed. Consider the double fetch, ***<A+, A->***. By incrementing on the first fetch, the value of A for the second fetch will be the second part of A. By decrementing on the second fetch, the value of A for the next loop iteration will return to the first part of A. For each of the vectors, two increment values are needed. The first handles the double fetch as described above for A. The second provides the increment for the next element of the vector for the next loop iteration. In **<$y_{+\sim}$, y_{-+}>**, the first access increments only the first increment to implement the double fetch. The second access decrements by the first increment value (a word size) to reset to the beginning of a word, and increments by the second increment value to get to the next element.

Since the second increment is exactly twice the first increment in the above example, the vector double fetches could also have been implemented as ***<y+, y+>***. The above example was used to demonstrate a more general solution.

The daxpy TSpec can be decomposed into the following primitives as described in Section 3.3.3.2:

P1 = < !c, c*4, $(_{L2}c_4$, c*2, λ*2, c*2, λ*2, c, λ*2, c*2, λ*2, c*4$)_{L2}$*5, c*2>
P2 = < !y, λ*4, $(_{L2}$λ*3, $y_{+\sim}$, $y_{-\sim}$, λ*9, y_$w_{+\sim}$, y_w_{-+}, λ*4$)_{L2}$*5, λ*2>
P3 = < !A, λ*4, $(_{L2}$λ*7, A+, A-, λ*11$)_{L2}$*5, λ*2>
P4 = < !x, λ*4, $(_{L2}$λ*10, $x_{+\sim}$, x_{-+}, λ*8$)_{L2}$*5, λ*2>

Note that functional filters for each of these primitives are included in Appendix A. We could now continue with the analysis of this kernel by filtering the primitives above and remerging the results. For such an analysis see Section 5.4. The point to be made by this decompostion is that the kernel daxpy can be analyzed by the methods that we have outlined in Chapter 3. It can be described in TSpec and it has functional filters available in the dictionary for a possible primitive decomposition.

5.2.1.2 Matmul Example

Matmul (Matrix multiply) multiplies an NxN matrix, ***x***, by an NxN matrix, ***y***, and stores the result in an NxN matrix, ***z***. It consists of three code loops and has three major data structures (the three NxN matrices) that do not fit in registers. For this kernel, all three possible optimization options were used to generate the assembly language and are included in Appendix B. In this section we focus on the unoptimized version with register allocation. The C code for the kernel is shown below.

```
void matmul(double x[SIZE][SIZE], double y[SIZE][SIZE],
            double z[SIZE][SIZE], int N)
{
  int i;
  int j;
  int k;
  double r;

  for (i=0; i<N; i++) {
    for (j=0; j<N; j++){
      r = x[i][j];
      for (k=0; k<N; k++) {
        z[i][k] += r*y[j][k];
      }
    }
  }
```

FIGURE 33: C Code Matmul

The unoptimized assembly language with register allocation is shown in Appendix B. Recall the initial increment of four enables intra-word accesses. The corresponding TSpec description is:

```
z(BASEz_r, 4, 8, 40);
y(BASEy_r, 4, 8, 40);
x(BASEx_r, 4, 8, 40);
c(BASEc_r, 4);
```

<!all, c*4, ($_{L2}c_{5}$, c*3,
($_{L6}c_{9}$, c*3, $x_{+\sim}$, $x_{-\sim+}$, c*4,
($_{L10}c_{17}$, c*5, $z_{+\sim\sim}$, $z_{-\sim\sim}$, c*2, $y_{+\sim\sim}$, $y_{-\sim\sim}$, c*3, $z\_w_{+\sim\sim}$, $z\_w_{-+\sim}$,
c*4$)_{L10}$*5,
c*4, ^$y_{!!+}$$)_{L6}$*5,
c*4, ^$z_{!!+}$, ^$x_{!!+}$$)_{L2}$*5, c*2>

Again, the c variable represents the code accesses and the other TSpec variables represent accesses to the variables in the source with the same name. In this example, the L2-loop in the TSpec corresponds to the i-loop in the source code, L6 in the TSpec corresponds to the j-loop in the source, and L10 in the TSpec corresponds to the k-loop in the source. Inside the L10 loop there are first several code references. This code updates the addresses to be fetched. The last of this set of 6 code references represents the load of the z-value. Then there is an update of another address and the load of the y-value. The y-value is then multiplied by the x-value already fetched and currently in a register. That result is then added to the z-value, and the result stored with the write to z. The final code references are to test the counter and branch back to the beginning of the loop if appropriate. For detailed assembly code, see Appendix B.

Each of the matrices have increment values corresponding to a word size, an element size (double word), and a row size. Recall that the ^ suppresses the generation of an address while incrementing or initializing the increment counters for a variable.

This TSpec description can be broken down into the following primitives:

$P1 = <!c, c*4, (_{L2}c_5, c*3,$
$(_{L6}c_9, c*3, \lambda*2, c*4,$
$(_{L10}c_{17}, c*5, \lambda*2, c*2, \lambda*2, c*3, \lambda*2, c*4)_{L10}*5,$
$c*4, \lambda)_{L6}*5,$
$c*4, \lambda*2)_{L2}*5, c*2>$

$P2 = <!x, \lambda*4, (_{L2}\lambda*4, (_{L6}\lambda*4, x_{+\sim}, x_{-\sim+}, \lambda*4, (_{L10}\lambda*19)_{L10}*5, \lambda*5)_{L6}*5,$
$\lambda*5, \hat{}x_{!!+})_{L2}*5, \lambda*2>$

$P3 = <!z, \lambda*4, (_{L2}\lambda*4, (_{L6}\lambda*10, (_{L10}\lambda*6, z_{+\sim\sim}, z_{-\sim\sim}, \lambda*9, z\_w_{+\sim\sim}, z\_w_{-+\sim}, \lambda*4)_{L10}*5,$
$\lambda*5)_{L6}*5,$
$\lambda*4, \hat{}z_{!!+}, \lambda)_{L2}*5, \lambda*2>$

$P4 = <!y, \lambda*4, (_{L2}\lambda*4, (_{L6}\lambda*8, (_{L10}\lambda*10, y_{+\sim\sim}, y_{-\sim\sim}, \lambda*9)_{L10}\ \lambda*4, \hat{}y_{!!+})_{L6}*5,$
$\lambda*6)_{L2}*5, \lambda*2>$

Since the x variable in P2 never uses the second increment, it can be seen as a double access to a repeated stream of stride 40. This primitive is available in the functional filter dictionary. Primitive P1, a triple nested code loop, is a straightforward extension of the filter function entry for a doubly nested code loop. P3, a series of repeated read-write streams, can be filtered by using the dictionary entry for a single read-write stream and concatenating the results, or by extending the dictionary to include a doubly nested read-write stream. P4 is a doubly nested read stream and is a straightforward extension to the dictionary's repeated read stream.

5.2.1.3 Livermore Fortran Kernel Examples

The above examples highlight most of the characteristics of the code patterns in the scientific examples. To investigate a larger number of data access patterns, we considered a sampling of the Livermore Fortran Kernels. The Livermore Fortran Kernels are a set of scientific benchmark codes [McM86]. We include them here because they include examples of sparse matrix computations. Table 1shows the C code and corresponding

TSpec of the general form of the data accesses for each of six kernel loops. More details are available in Appendix B.

The data patterns from these loops are either similar to those seen in daxpy or matmul (k1, k3, k5, k12) as described above, or are rather unique (k2, k4). Those that are unique have not been added to the filter function dictionary largely because their general form is unweildy combined with the fact that their inclusion would not solve a large number of existing problems. However, writing down the filtering function for a specific example of a primitive can be done from first principles as described in Chapter 3.

TABLE 1: Livermore Fortran Kernel Data Access Patterns

k#	Kernel Code	TSpec Data Reference Form

k1.c	for (k = 0; k < n; k++)
	x[k] = q + y[k] * (r * zx[k + 10]
	+ t *zx[k + 11]);
		<!all, ^$z_{!10}$, ($_{L19}zx_{+\sim}$. $zx_{+\sim}$, $zx_{+\sim}$, $zx_{!+}$,$y_{+\sim}$,
		y_{-+}, $x\_w_{+\sim}$, x_w_{-+})*>

k2.c	L2: ipnt = ipntp;
	ipntp = ipntp+ii;
	ii = ii/2;
	i = ipntp;
	for (k = ipnt+1; k < ipntp;k+=2)
	{ i++;
	x[i] = x[k] - v[k]*x[k-1] -
	v[k+1]*x[k+1];}
	if (ii > 1) goto L2;
		<!all, (($x_{\sim-}$, v-, x+, $v_{\sim+}$, $x_{\sim\sim+}$, $x\_w_{\sim\sim-}$)*)*>

k3.c	for (k = 0; k < n; k++)
	q = q + z[k] * x[k] + z[k + 1] *
	x[k + 1];
		<!all, (x-, z-, $x_{\sim+}$, $z_{\sim+}$)*>

k4.c	for (i = 0; i < n; i++) {
	for (k = 7; k<1001; k +=m) {
	lw = k-6;
	temp = x[k-1];
	for (j = 5; j < n; j +=5) {
	temp = temp - x[lw]*y[j];
	lw++; }
	x[k-1] = y[5] * temp; } }
		<((!x, $x_{\sim+}$, !y, ($y_{+\sim}$,y_{-+}, x+, x-)*, y5,
		^$x_{!+}$, x)*)*>

k5.c	for (i = 0; i < n; i++)
	x[i] = z[i] * (y[i] - x[i - 1]);
		<!all, (x+, x_{-+}, y+, y_{-+}, z+, z_{-+}, x_w+,
		x_w_{-+})*>

k12.
c
	for (k = 0; k < n; k++)
	x[k] = y[k + 1] - y[k];
		<!all, (y+~, y-,-, y+~, y!+, x_w+~,
		x_w-+)*>

5.2.1.4 Discussion of Scientific Examples

While the examples above are not exhaustive, they show that reference behaviors similar to these scientific kernels can be described by TSpec and analyzed with the filter function model. Specifically, it is interesting to note that the TSpec construct of a variable is well suited to describe the access patterns in all of these kernels. Multiple increments allow data structures (in particular matrices and vectors) to be traversed with different strides or element sizes. For example, this allowed convenient notation for striding through matrices of double words either by element, or by row size. Which stride or size is being used can be changed depending on the loop index or fetch size desired. TSpec also allows for setting the “code” variables to different starting values. This enables a straightforward expression of an arbitrary number of nested code loops.

The primitives in the function filter dictionary of Appendix A include many of those necessary to analyze scientific codes. It could be supplemented in a straight forward manner to include more primitives for kernels of this type of behavior. In those situations where a filter function is unwieldy, the function for the primitive can be written using the basic principles outlined in Chapter 3. Even in these cases, some of the primitives in the kernel will be in the filter dictionary, which simplifies the analysis for the whole kernel.

5.2.2 Recursion Example

For a discussion of the effect of recursion and heap use, we use pseudocode from the Quicksort algorithm[1] as shown in Figure 34. From the scientific examples above we learned that basic code loops and leaf subroutine calls can be modeled with TSpec. Also, we learned that these compilers fetched the operands in the order that the expressions are evaluated (from right to left) in the examples. Using this information and the pseudocode

[1] This pseudocode is from Cormen *et al.*[Cor89].

for quicksort, we can write the general form of the TSpec that corresponds to this source code.

```
initial call to sort entire array Quicksort(A, 1, length[A])

Quicksort (A, p, r)
if p < r
   then q = Partition(A, p, r)
      Quicksort(A, p, q)
      Quicksort(A, q+1, r)

Partition (A, p, r)
   x = A[p]
   i = p-1
   j = r+1
   while TRUE
   do repeat j = j-1
         until A[j] <= x
      repeat i = i+1
         until A[i] >= x
      if i < j
         then exchange A[i], A[j]
         else return j
```

FIGURE 34: Quicksort Pseudocode

Let us assume that the loop indices and local variables will be put in registers at the beginning of subroutine calls, and a “callee saves” protocol where the called subroutine saves and restores registers. This means that the array A is the only data structure that is in memory as the subroutines run. The TSpec corresponding to the pseudocode for Partition in Figure 34 is:

```
c(?_r, 4);          // variable representing code accesses
A(?_r, 4);          // variable representing accesses to array A
sp(?_r, 4);         // variable representing accesses to stack

<!all, (c, sp_w)*,               // save registers
  c, A?, c*,                     // first three assignments and reference to A
  (L1^A?, (c*, A)*,              // beginning of while, first repeat
     ^A?, (c*, A)*,                        // second repeat
    c*, exceptlast(c*, A-, A?))L1*, c* // end of Partition
    (c, sp-)*>                             // restore registers
```

Quicksort can now be modeled as multiple concatenated calls to Partition. When one call to Partition is broken down into primitives, we get a nested code loop, one “write stream” and one “read stream” each for the save and restore of registers, and a “random within a range” primitive for the accesses to A. All of these primitives are included in the filter function dictionary. For some caches, the “random within a range” primitive requires the specific instance of the primitive to be analyzed from the basic principles outlined in Chapter 3 because the behavior is not regular enough to be characterized by any of the current dictionary entries.

5.2.2.1 Heap Use (Linked Lists)

The obvious data structure that is not included in the above examples is that of a linked list or other linked data structures that make use of the heap. In the model developed here, the general case of these data structures could be modeled as the “random within a range” primitive or decomposed further into “repeated single read/write” or “read-write stream” primitives. This approach has the benefit of simplifying the analysis of the rest of the code and separating the effects of the heap accesses from the effects of the other data accesses on the cache.

Another approach to analyzing the linked data structures using the heap would be to investigate the actual access patterns generated by these accesses. There has been research done that indicates these type of accesses have much more regularity than indicated by their structure. This concentrated analysis is beyond the scope of this dissertation, but the framework proposed here provides a mechanism for approaching such an analysis and presenting the results.

5.3 Assembly Language Applicability

5.3.1 Different Assembly Languages

The examples throughout the dissertation at this point have been generated using assembly language for a SPARC target machine. Several of the examples presented in Section 5.2 have also been assembled using a MIPS target machine. Those results are available in Appendix B. The TSpec descriptions corresponding to the different assembly languages differ little for the kernels studied. An example of the daxpy kernel in MIPS assembly language and corresponding TSpec is shown in Figure 35.

The differences between these two languages are primarily small differences in the number of code references to accomplish a task and in branches. The SPARC has a delayed branch. Instructions in the delayed branch can be annulled if the branch is taken. This results in more use of the *exceptlast()* feature in TSpec modeling SPARC code than in TSpec modeling MIPS code.

Modeling assembly language for a target machine without a RISC instruction set would differ slightly, but could still be represented in TSpec. For example, modeling a

Pentium instruction set would require modeling instructions of varying lengths. This could be accomplished with different increment values for a single code variable.

C Code:
```
for (i=0; i<N; i++) {
     y[i] += A * x[i];}
```

MIPS Code:
```
.L2:     sll     $7,$8,3
         addu    $9,$7,$5
         l.d     $f0,($9)
         l.d     $f2,L7
         addu    $2,$7,$4
         l.d     $f4,0($2)
         mul.d   $f2,$f2,$f4
         add.d   $f0,$f0,$f2
         s.d     $f0,($9)
         addu    $8,$8,1
         blt     $8,$6,.L2
.L000:   j       $31
```

TSpec Description:

c(?_r, 4);
y(?_r, 4, 8);
x(?_r, 4, 8);
A(?_r, 4);

MIPs:

< !all, c*3, ($_{L2}$c$_3$, c*2, y$_{+\sim}$, y$_{-\sim}$, c, A+, A-, c*2, x$_{+\sim}$, x$_{-+}$, c*2, y_w$_{+\sim}$, y_w$_{-+}$, c*2)$_{L2}$*5, c>

SPARC:

< !all, c*4, ($_{L2}$c$_4$, c*2, y$_{+\sim}$, y$_{-\sim}$, c*2, A+, A-, c, x$_{+\sim}$, x$_{-+}$, c*3, y_w$_{+\sim}$, y_w$_{-+}$, c*4)$_{L2}$*5, c*2>

FIGURE 35: Daxpy, MIPS Assembly and TSpec Description

5.3.2 Optimizations

Many of the examples in the appendix were also assembled with different optimization levels. Three different categories of optimizations were used: completely unoptimized,

unoptimized with register allocation, and fully optimized. The examples in the sections above are primarily unoptimized with register allocation.

Without register allocation, all locals and parameters require memory fetches so the TSpec descriptions are longer and more tedious. The primitives introduced by these additional fetches can be modeled with “read-write”, “single read/write”, or “uneven read/write within a range” primitives. All of these primitives are available in the filter function dictionary of Appendix A.

Fully optimized assembly language for these small kernels differs little from unoptimized assembly code with register allocation. Occasionally the memory fetches are in a slightly different order or a memory fetch may be eliminated or moved out of a loop. The most common difference is that the fully optimized version has fewer code references because code is moved outside a loop, or other, more efficient code sequences have been substituted. These more efficient sequences are ususally shorter. An example of the TSpec description for the kernel portion of matrix multiply assembled with each optimization category is in Figure 36. The variables for the TSpec descriptions and the assembly language corresponding to each are included in detail in Appendix B.

C Code:
```
for (i=0; i<N; i++) {
         for (j=0; j<N; j++){
             r = x[i][j];
             for (k=0; k<N; k++) {
                z[i][k] += r*y[j][k];}}}
```

Unoptimized:

< !all, c*2, px_w~, c, py_w~, c, pz_w~, c, pN_w~, c, li_w~, c,
c_{67}, c, li~, c, pN~, c*3, // c67 is where L5 starts
$(_{L2}$c, lj_w~, c,
c_{59}, lj~, c, pN~, c*3,
$(_{L6}$ c_9, li~, c*2, l1_w~, c, lj~, c*2, l1~, c, px~, c*2, $x_{+\sim}$, x_{-+}, c, lr_w+, lr_w-, c, lk_w~, c,
c51, c, lk~, c, pN~, c*3,
$(_{L10}$, c_{21}, li~, c*2, l3_w~, c, lj~, c*2, l4_w~, c, lk~, c*2, l5_w~, c, l5~, c, l3~, c, pz~,
c*3, l6_w~, c, l6~, c, $z_{+\sim}$, $z_{-\sim}$, c, lr+, lr-, c, l5~, c, l4~, c, py~, c*2, $y_{+\sim\sim}$, $y_{-+\sim}$, c*3,
l6~, c, $z_{+\sim}$, $z_{-\sim}$, c, lk~, c*2, lk_w~, c, pN~, c*3$)_{L10}$*5,
c, lj~, c*2, lj_w~, c, lj~, c, pN~, c*3, ^$y_{!!+})_{L6}$*5,
c, li~, c*2, li_w~, c, li~, c, pN~, c*3, ^$z_{!!+}$, ^$x_{!!+})_{L2}$*5,
c*2 >

Unoptimized with Register Allocation:

<!all, c*4, $(_{L2}c_5$, c*3,
$(_{L6}c_9$, c*3, $x_{+\sim}$, $x_{-\sim+}$, c*4,
$(_{L10}c_{17}$, c*5, $z_{+\sim\sim}$, $z_{-\sim\sim}$, c*2, $y_{+\sim\sim}$, $y_{-\sim\sim}$, c*3, $z\_w_{+\sim\sim}$, $z\_w_{-+\sim}$, c*4$)_{L10}$*5,
c*4, c*3, ^$y_{!!+})_{L6}$*5,
c*4, ^$z_{!!+}$, ^$x_{!!+})_{L2}$*5, c*2>

Fully optimized:

< !all, c*6, $(_{L2}c_6$, c*6,
$(_{L6}$c*3, $x_{+\sim}$, x_{-+}, c*10,
$(_{L10}$c, $y_{+\sim}$, y_{-+}, c, $z_{+\sim}$, $z_{-\sim}$, c*3, $z\_w_{+\sim}$, z_w_{-+}, c*4$)_{L10}$*5,
c*4, ^$y_{!!+})_{L6}$*5,
c*4, ^$z_{!!+}$, ^$x_{!!+})_{L2}$*5,
c*2>

FIGURE 36: Matmul TSpec Descriptions, Different Optimization Levels

5.4 Multi-level Cache System Example

Up to this point we concentrated on generating the TSpec and primitives for several kernels, but stopped short of a complete analysis. This section takes one of the kernels, daxpy, and analyzes it for the multi-level cache hierarchy shown in Figure 37. This hierarchy is similar to the basic cache configuration of many modern microprocessors. It consists of separate instruction and data caches at the L1 level, directly connected to the CPU, and a single, unified cache at the L2 level.

To perform the analysis we will look at the daxpy trace at each level in the hierarchy. The traces are labeled at each point of interest in Figure 37. T is the initial combined instruction and data trace for the daxpy kernel. TI is the instruction-only portion of T. TD is the data-only portion of T. TI' and TD' are the filtered versions of TI and TD respectively. TID' is the merged version of the traces TI' and TD'. TID' is then filtered by L2 to produce TID2'.

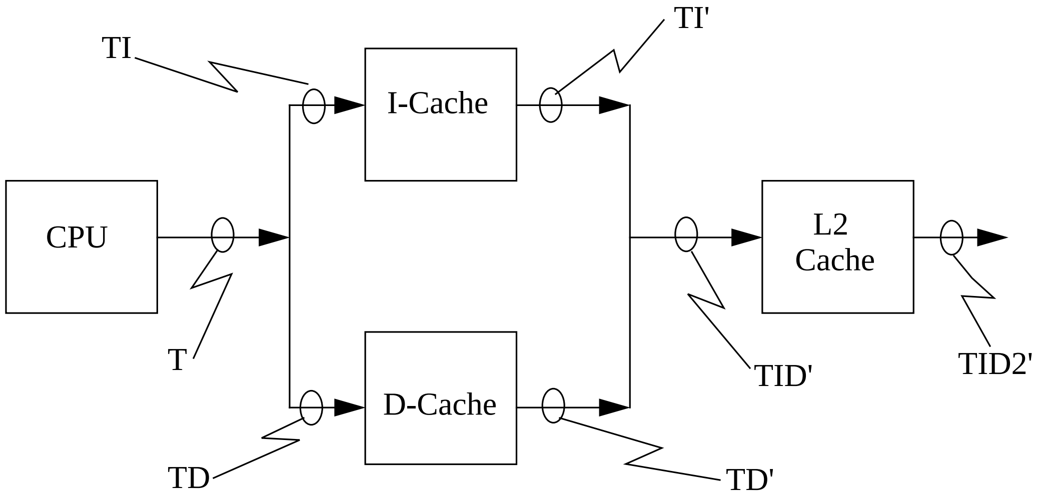


FIGURE 37: Multi-level Cache Hierarchy Example

The initial daxpy trace from the unoptimized SPARC assembly with register allocation as described in Section 5.2.1.1[2] is:

[2] The code of the internal loop is different here by one code reference.

c(?_r, 4);
y(?_r, 4, 8);
x(?_r, 4, 8);
A(?_r, 4);

$T = < !all, c*4, ({}_{L2}c_4, c*2, y_{+\sim}, y_{-\sim}, c*2, A+, A-, c, x_{+\sim}, x_{-+},$
$c*2, y\_w_{+\sim}, y\_w_{-+}, c*4)_{L2}*5, c*2>$

By focusing on the kernel, we have performed segmentation, the first step of the analysis approach discussed in Section 3.3.3.1. Since we have chosen a hierarchy that uses separate instruction and data caches, the next step is to separate T into TI and TD.

$TI = < \lambda*3, !c, c*4, ({}_{L2}c_4, c*2, \lambda*2, c*2, \lambda*2, c, \lambda*2,$
$c*2, \lambda*2, c*4)_{L2}*5, c*2>$

$TD = < !y, !A, !x, \lambda*5, ({}_{L2}\lambda*3, y_{+\sim}, y_{-\sim}, \lambda*2, A+, A-, \lambda, x_{+\sim}, x_{-+},$
$\lambda*2, y\_w_{+\sim}, y\_w_{-+}, \lambda*4)_{L2}*5, \lambda*2>$

One advantage to the TSpec descriptions is worth mentioning at this point; the TSpec description contains most of the information needed to design a cache that can maximize performance on that particular reference stream. In this example, we can tell from the number of code references in the loop L2 that the I-cache needs a minimum of 12 lines to hold the loop and so obtain only compulsory misses in the output. Any smaller cache would have worse performance and a larger instruction cache is unnecessary for this particular kernel. We can also tell that the data cache would need to be big enough to hold two arrays of five double elements (x and y), plus a double scalar (A). If there were write-around capabilities, we could identify x as a candidate for write-around because it is never reused.

5.4.1 Locality of Daxpy, Daxpy Code, and Daxpy Data

In addition to looking at cache requirements, it is worthwhile to note at this point the locality graphs of the three traces T, TI, and TD. These graphs are shown in Figure 38.

FIGURE 38: Daxpy Instantaneous Locality

Each trace assumes the same starting address for a particular variable. c starts at 100, A at 1000, y at 1008, and x at 2000. Note that the overall locality of each TI and TD on their own is generally higher than that of T. This helps explain why separating the reference string into two separate strings improves performance. Not only are there two paths to the cache, but the locality of each reference string is greater, giving each cache more potential for success. It also gives the cache system designer an opportunity to tune each cache to the different types of reference strings for which it will be used.

5.4.2 Decomposition Into Primitives

Continuing with our analysis, we focus on the loop and decompose TI and TD into primitives. TI is already in the form of a repeated code loop concatenated with a couple short code segments on each side. For this example we will just focus on the loop.

$$TI = TI1 = < \lambda*3, !c, ({}_{L2}c_4, c*2, \lambda*2, c*2, \lambda*2, c, \lambda*2, c*2, \lambda*2, c*4)_{L2}*5, >$$

TD can be decomposed into three primitives:

$$TD1 = < !y, \lambda*4, ({}_{L2}\lambda*3, y_{+\sim}, y_{-\sim}, \lambda*9, y\_w_{+\sim}, y\_w_{-+}, \lambda*4)_{L2}*5>$$

$$TD2 = < \lambda, !A, \lambda*3, ({}_{L2}\lambda*7, A+, A-, \lambda*11)_{L2}*5>$$

$$TD3 = < \lambda*2\ !x, \lambda*2, ({}_{L2}\lambda*10, x_{+\sim}, x_{-+}, \lambda*8,)_{L2}*5>$$

The next step is to filter the primitives either with the filter function dictionary in Appendix A, or by reverting to the recursive definition in Chapter 3. The rest of the analysis proceeds assuming that the I-Cache is a direct-mapped, read-only, cache of size 16 lines with a line size of one, and the D-Cache is a two-way set associative cache with a LRU line replacement policy and cache size of 16 lines (8 sets of two) of size one.

5.4.3 Filtering TI

TI1 can be filtered using the primitive entry in Section A.2.4 of the filter function dictionary for a “repeated read stream” or “single code loop”. The dictionary entry reads as follows:

p(?, st); // *st > 0*
*P = <P1*n2> such that D(P1) = <!p, p*n1>*

$f_{*,*,1}(P; S^0) = <P1,\ (\lambda*|P1|)*(n2\text{-}1)>; \{p!, (p\text{-})*n1\}$ if $n1 \leq ecl$ *(i)*

The top two lines describe the primitive that this entry can filter. Recall that D(P) is equal to P with all of the λs removed. Specifying P as the repetition of another trace P1, and describing P1 in terms of D(P1) allows this entry to work for a code loop with any number of intervening λs. n2 represents the number of repetitions of the stream, and n1 the number of elements in the stream.

The third line (labeled (i)) is the entry that is applicable if the loop fits into the cache (n1 < ecl). The term ecl stands for effective cache lines and refers to the number of lines that this stream can make use of in the cache. This is dictated by whether the stride and the number of sets are relatively prime. For details on how to obtain this number for streams of varying strides, see Appendix A, Section A.1.1. In this case, ecl = 16 and n1 = 12. This filtering function is subscripted with *,*,1. The first * indicates it applies for a cache with any kind of associativity: direct mapped, set associative, or fully associative. The second * indicates the function applies for either a write through cache or a write back cache. The 1 indicates it applies for caches with a linesize of 1 (or where the linesize equals the fetch size).

Applying (i) to TI1 generates TI1' and $f^{S}_{1,wb,1}(TI1; S^0)$as shown below.

TI1' = < λ*3, !c, ($_{L2}c_4$, c*2, λ*2, c*2, λ*2, c, λ*2,
c*2, λ*2, c*4)$_{L2}$, λ*80, >
$f^S_{1,wb,1}$(TI1; S^0) = {c!, (c-)*12}

The first iteration of the code loop is reflected at the output. The code loop fits in the cache, so the subsequent iterations are just represented as λs. Since there are 20 references in each iteration of the loop, the last four iterations are represented by 80 λs.

5.4.4 Fill and Bind

To understand the filter function entries for TD, two additional functions on TSpec descriptions are required. The first function is denoted Λ(T) and removes all of the non-λ elements in T and replaces them with the placeholder $\bar{\lambda}$, ("not λ"). For example, if **T = < λ, λ, f, λ, t, λ >** then **Λ*(T)* = < λ, λ, $\bar{\lambda}$, λ, $\bar{\lambda}$, λ >.** More formally, $\forall i$ such that $(1 \le i \le |T|)$ if T = Λ(T1) then

$$t_i = \begin{cases} \lambda \text{ if } (t1_i = \lambda) \\ \bar{\lambda} \text{ otherwise} \end{cases}$$

The second function, called *fill* and denoted by (Δ), operates on two traces by filling the $\bar{\lambda}$ elements of the first trace with the second trace. An element in the second trace may be λ, in which case $\bar{\lambda}$ is replaced by λ. For example if ***T1* = < λ, λ, $\bar{\lambda}$, λ, $\bar{\lambda}$, λ >** and ***T2* = < *f*, λ >**, then ***T* = *T1* Δ *T2* = < λ, λ, *f*, λ, λ, λ >.**

The combination of Λ and Δ are used to allow us to remove all the non-λ elements from a trace and then "refill" those non-λ slots with the filtered version of the original non-λ elements. This process is necessary to maintain the correct placement of all λs in the filtered trace.

5.4.5 Filtering TD

Recall that the D-Cache is a two-way associative, write back cache consisting of 16 lines (8 sets of 2 lines) with LRU replacement on the lines. The primitives being filtered are:

TD1 = $< !y, \lambda*4, (_{L2}\lambda*3, y_{+\sim}, y_{-\sim}, \lambda*9, y\_w_{+\sim}, y\_w_{-+}, \lambda*4)_{L2}*5>$

TD2 = $< \lambda, !A, \lambda*3, (_{L2}\lambda*7, A+, A-, \lambda*11)_{L2}*5>$

TD3 = $< \lambda*2\ !x, \lambda*2, (_{L2}\lambda*10, x_{+\sim}, x_{-+}, \lambda*8,)_{L2}*5>$

TD1 is a “double read-write” stream. It can be filtered by extending the function in Appendix A, Section A.2.9 for a “read-write” stream. Since we have 10 actual writes in TD1 and 16 lines available, the appropriate entry is (ii):

p(?, st) // *st > 0*
*P such that D(P) = <!p, (p~, p_w)*n1>*

$f_{*,wb,1}(P; S^0) = <\Lambda(P)\ \Delta\ (!p, (p, \lambda)*n1)>; \{p!, (p\_w-)*n1\} \quad n1 \le ecl \quad (ii)$

Since the primitive in this entry is for a single read-write, the function must be extended to handle double read-writes. The effect of doubling the size of the read-write is that only half as many elements will fit. In this case the definition can be extended simply by representing the ps as double reads and double writes, and replacing n1 by 2 x n1. Filtering TD1 with this extended definition generates:

TD1' = $< !y, \lambda*4, (_{L2}\lambda*3, y_{+\sim}, y_{-\sim}, \lambda*15)_{L2}*5>$

$f^{S}_{2,wb,1}(\mathrm{TD1}; S^0) = \{ y!, (y\_w-, y\_w_{+-})*5\}$

Note that the state captures the idea that the elements have been written. This is important for a write back cache, where if these elements are evicted, their values will need to be written back to the next level of cache.

Filtering TD2 and TD3 is a straightforward application of the filter function dictionary entry for a “repeated read”, Appendix A, Section A.2.1, and a “double read stream”, Appendix A, Section A.2.10. The results of that filtering are shown below.

$TD2' = < \lambda, !A, \lambda*3, (_{L2}\lambda*7, A+, A-, \lambda*11)_{L2}, \lambda*80>$

$f^{S}_{2,wb,1}(TD2; S^{0}) = \{ A!, A-, A\}$

$TD3' = < \lambda*2\ !x, \lambda*2, (_{L2}\lambda*10, x_{+\sim}, x_{-+}, \lambda*8,)_{L2}*5>$

$f^{S}_{2,wb,1}(TD3; S^{0}) = \{ x!, (x-, x_{+-})*5\}$

The last step is to perform a trace-state merge on the TD primitives to obtain TD'.

5.4.6 Merging TD1', TD2', and TD3'

So far it has not been necessary to know the base addresses or the particular cache set to which the elements map. We will perform this part of the analysis on $\{TD_b\}$, and determine a best-case and a worst-case performance for the daxpy data through the D-Cache. When this approach is used, we write one TSpec description for the worst-case result and one for the best-case-case result. Note that in these examples the TSpec descriptions each represent a set of traces that have the same result, namely all those whose base address assignments result in the assumed cache set assignment.

When performing the trace-state merge, it may be necessary to return to the original trace and replace λs with elements that were originally filtered, but that are evicted by another primitive. In the worst case, a trace-state merge can approximate the complexity of a cache simulation. Therefore, this merge is one of the areas of the filter function approach that would benefit from automation. To perform the merge on the primitives of TD, we will use diagrams similar to those introduced in Chapter 3.

Recall,

TD^{3} = < !y, !A, !x, λ, ($_{L2}$λ*3, $y_{+\sim}$, $y_{-\sim}$, λ*2, A+, A-, λ, $x_{+\sim}$, x_{-+},
λ*2, y_$w_{+\sim}$, y_w_{-+}, λ*4)$_{L2}$*5, λ*2>

TD1' = < !y, λ*4, ($_{L2}$λ*3, $y_{+\sim}$, $y_{-\sim}$, λ*15)$_{L2}$*5>

TD2' = < λ, !A, λ*3, ($_{L2}$λ*7, A+, A-, λ*11)$_{L2}$, λ*80>

TD3' = < λ*2 !x, λ*2, ($_{L2}$λ*10, $x_{+\sim}$, x_{-+}, λ*8,)$_{L2}$*5>

The worst D-cache performance will occur when the most references possible are evicted. Since there are no repeated references to x, these references will always miss and do not need to be evicted to generate a worst-case performance scenario. The second set of references to y are filtered in TD1', but if they were to conflict with two other primitives then they would be evicted and need to be referenced again. It would require a conflict with two other primitives because the cache is two-way set associative. y will conflict with two other primitives the most when all primitives map to the same cache set.

This worst-case scenario is reflected in Figure 39. This figure displays the cache set assignments for the D-Cache. The cache set number is on the left and each reference is placed in the appropriate cache set for each iteration. If a reference hits, a λ is placed in the appropriate cache set instead. Without loss of generality, we assume the single cache set that all primitives map to is 0.

[3] This TD is slightly different than the original TD because we focus here on the loop only, removing the few code references at either end of the loop.

Set	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5
0	y_0 A_0 x_0 y_w_0	A_0	λ	λ	y_8 λ x_8 y_w_8
1	y_1 A_1 x_1 y_w_1	A_1	λ	λ	y_9 λ x_9 y_w_9
2		y_2 x_2 λ			
3		y_3 x_3 λ			
4			y_4 x_4 λ		
5			y_5 x_5 λ		
6				y_6 x_6 λ	
7				y_7 x_7 λ	

FIGURE 39: Daxpy Data, Worst-case Cache Set Assignments, 2-way, 16 lines

In the first iteration, all references are to the same two cache sets. The initial references to y and A each take one line in set 0, when x is referenced it evicts y because it is the least recently used line in set 0. Then when y is referenced for the second time in the first iteration, it evicts A, so when A is referenced in the second iteration it must be referenced again. In subsequent iterations, x and y map to the same cache set, but since there are two lines in each set, y is still present in the cache when the write occurs. This happens until the final iteration, when all primitives reference the same cache line again. The reference to A still hits because it has not been evicted, but the y write reference in this last iteration misses again.

The result of the trace-state merge for the worst-case cache set assignment is:

$$TD' = <!y,!A,!x,\lambda, (\lambda*3, y_{+\sim}, y_{-\sim}, \lambda*2, A+, A-, \lambda, x_{+\sim}, x_{-+}, \lambda*2, y\_w_{+\sim}, y\_w_{-+}, \lambda*4)*2, (\lambda*3, y_{+\sim}, y_{-\sim}, \lambda*5, x_{+\sim}, x_{-+}, \lambda*8)*2, \lambda*3, y_{+\sim}, y_{-\sim}, \lambda*5, x_{+\sim}, x_{-+}, \lambda*2, y\_w_{+\sim}, y\_w_{-+}, \lambda*4>$$

$$f^S_{2,wb,1}(TD; S^0) = \{!y, !x, (y\_w-, y\_w-, x-, x-)*4\}$$

In the first two iterations every reference will miss. In the third and fourth iterations references to A and the second references to y will hit. In the final iteration, the references to y miss again. Recall that this result is for an entire set of traces whose base address assignments result in all three primitives beginning in the same cache set. The state is just the last 4 elements of each stream. With two streams, each a double size, 4 elements apiece fill the 16 available lines in the cache. Note that the state includes the information that the y values have been written. In this way, the notion of a dirty line is captured in case these values are evicted and a write back required later.

The best-case cache set assignment for the daxpy data would be an assignment where no more than two primitives access the same set in a single iteration, resulting in only compulsory misses. One such possible assignment is depicted in Figure 40. The critical issues for finding such a best-case cache set assignment are to make sure that as x and y wrap around to the cache set to which A is assigned, they touch that cache set over an iteration apart so that A is never the least recently used item in the set.

Set	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5
0	y_0 A_0 y_w_0	λ	λ x_4	λ	y_8 λ y_w_8
1	y_1 A_1 y_w_1	λ_1	λ x_5	λ	y_9 λ y_w_9
2		y_2 λ		x_6	
3		y_3 λ		x_7	
4	x_0		y_4 λ		x_8
5	x_1		y_5 λ		x_9
6		x_2		y_6 λ	
7		x_3		y_7 λ	

FIGURE 40: Daxpy Data, Best-case Cache Set Assignments,
2-way, 16 lines

With the best-case scenario of only compulsory misses, the output of the D-cache becomes:

$$TD' = <!y,!A,!x,\lambda, (\lambda*3, y_{+\sim}, y_{-\sim}, \lambda*2, A+, A-, \lambda, x_{+\sim}, x_{-+}, \lambda*8), (\lambda*3, y_{+\sim}, y_{-\sim}, \lambda*5, x_{+\sim}, x_{-+}, \lambda*8)*4>$$

$$f^S_{2,wb,1}(TD; S^0) = \{y!, x!, A!, y\_w-*8, x-*4, \wedge x-*2, x-*2, A-*2\}$$

Since this case of only compulsory misses is uninteresting for the analysis of the L2 cache because the L2 cache output would be just the compulsory misses no matter what the cache looked like, we continue the analysis using the worst-case TD'.

5.4.7 Merging TD' and TI'

The merging of TD' and TI' is a straightforward trace-only merge as defined in Chapter 2. It is performed on an element-by-element basis. λ merged with any element produces that element. Two λs merged together produces λ. Any two elements other than λ merged together is undefined. The resulting TSpec description of TID' is shown below.

$$\begin{aligned}
TD' = \langle !y, !A, !x, \lambda, &(\lambda*3, y_{+\sim}, y_{-\sim}, \lambda*2, A+, A-, \lambda, x_{+\sim}, x_{-+}, \lambda*2, y\_w_{+\sim}, y\_w_{-+}, \lambda*4)*2,\\
&(\lambda*3, y_{+\sim}, y_{-\sim}, \lambda*5, x_{+\sim}, x_{-+}, \lambda*8)*2,\\
&\lambda*3, y_{+\sim}, y_{-\sim}, \lambda*5, x_{+\sim}, x_{-+}, \lambda*2, y\_w_{+\sim}, y\_w_{-+}, \lambda*4\rangle
\end{aligned}$$

$$\begin{aligned}
TI' = \langle \lambda*3, !c, ({}_{L2}c_4, &c*2, \lambda*2, c*2, \lambda*2, c, \lambda*2,\\
&c*2, \lambda*2, c*4)_{L2}, \lambda*80, \rangle
\end{aligned}$$

$$\begin{aligned}
TID' = \langle !y, !A, !x, !c, c_4, &c*2, y_{+\sim}, y_{-\sim}, c*2, A+, A-, c, x_{+\sim}, x_{-+}, c*2,\\
&y\_w_{+\sim}, y\_w_{-+}, c*4,\\
&\lambda*3, y_{+\sim}, y_{-\sim}, \lambda*2, A+, A-, \lambda, x_{+\sim}, x_{-+}, \lambda*2, y\_w_{+\sim}, y\_w_{-+}, \lambda*4,\\
&(\lambda*3, y_{+\sim}, y_{-\sim}, \lambda*5, x_{+\sim}, x_{-+}, \lambda*8)*2,\\
&\lambda*3, y_{+\sim}, y_{-\sim}, \lambda*5, x_{+\sim}, x_{-+}, \lambda*2, y\_w_{+\sim}, y\_w_{-+}, \lambda*4\rangle
\end{aligned}$$

5.4.8 Filtering TID' with L2

Several recent microprocessor designs have L2 caches that are significantly larger than the L1 data cache and are direct mapped. In keeping with this ratio, let the L2 cache used for this analysis be a 64 line direct mapped cache. We will perform the analysis assuming that the base address assignments result in the cache set assignments chosen for the D-Cache. We will still do best and worst-case analysis.

In the worst-case L2 scenario, the primitives that mapped to the same cache set in the D-Cache would still map to the same cache set in L2. This could happen because the size of the L2 cache is a multiple of the size of the D-Cache. In addition to the conflicts this generated in the D-Cache, there is now a code segment that could conflict. Because only one set of code references comes through to this cache, it does not cause any extra misses

Set	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5
0	c4 y_0 A_0 x_0 y_w_0	A_0			
1	c5 y_1 A_1 x_1 y_w_1	A_1			
2	c6	y_2 x_2 y_w_2			
3	c7	y_3 x_3 y_w_3			
4	c8		y_4 x_4		
5	c9		y_5 x_5		
6	c10			y_6 x_6	
7	c11			y_7 x_7	
8	c12				y_8 x_8 y_w_8
9	c13				y_9 x_9 y_w_9
	c14 c15				
63					

FIGURE 41: Daxpy L2, Worst-case Cache Set Assignments, DM, 64 lines

though. With this worst-case set assignment, all of the data primitives still conflict and none of the references in TID' are filtered out. A diagram of the cache set assignments are shown in Figure 41.

The resulting TSpec description is:

TID2' = TID' = <!y,!A,!x,!c, c_4, c*2, $y_{+\sim}$, $y_{-\sim}$, c*2, A+, A-, c, $x_{+\sim}$, x_{-+}, c*2,
$y\_w_{+\sim}$, y_w_{-+}, c*4,
λ*3, $y_{+\sim}$, $y_{-\sim}$, λ*2, A+, A-, λ, $x_{+\sim}$, x_{-+}, λ*2, $y\_w_{+\sim}$, y_w_{-+}, λ*4,
(λ*3, $y_{+\sim}$, $y_{-\sim}$, λ*5, $x_{+\sim}$, x_{-+}, λ*8)*2,
λ*3, $y_{+\sim}$, $y_{-\sim}$, λ*5, $x_{+\sim}$, x_{-+}, λ*2, $y\_w_{+\sim}$, y_w_{-+}, λ*4>

$f^{S}_{2,wb,1}$(TID; S^0) = {c!, y!, A!, c-*2, y-*2, ^y-*4, y-*2, x_7, x-*3}

In the best-case scenario, all of the primitives map to completely different areas of the direct mapped cache and everything but compulsory misses are eliminated. In this case, the result is:

TID2' = <!y,!A,!x,!c, c_4, c*2, $y_{+\sim}$, $y_{-\sim}$, c*2, A+, A-, c, $x_{+\sim}$, x_{-+}, c*2,
λ*2, c*4,
(λ*3, $y_{+\sim}$, $y_{-\sim}$, λ*5, $x_{+\sim}$, x_{-+}, λ*8)*4>

$f^{S}_{2,wb,1}$(TID; S^0) = {c!, A!, x!, y!, c-*12, A-*2, x-*8, y_w*8}

5.4.9 Locality of T, TID', and TID2'

It is now possible to evaluate the multi-level cache system in Figure 37 with the locality measure. The locality graph below shows the locality graphs of the first parts of the input trace, T, both best and worst-case output of the combined instruction and data caches, and both best and worst-case output of the L2 cache. Since the best-case output of the L1 level is the same as the best-case output of L2, and the worst-case output of L2 is the same as the worst-case output of L1, there are only a total of three traces in the graph. The traces to correspond to the TSpec descriptions were generated using the following base address assignments: c- 0, y - 1024, x - 2048, A - 3072. These numbers reflect the assumptions in the cache set assignments and place each primitive relatively far from each other. Notice that the locality measures for all three traces are the same for the majority of the first iteration. The only difference is the TID2' trace has hits for the second two references to y. Then the curves begin to differ. TID2' has misses only for the first references to y and x. The TID' trace has misses for all those of the TID2' trace, and some additional misses from conflicts, including the very last two misses from y at the end of the graph.

FIGURE 42: Locality of T*, Worst-case TID', and Best-case TID2'

Recall the TSpec descriptions.

$T^* = <\ !all,\ ({}_{L2}c_4,\ c^*2,\ y_{+\sim},\ y_{-\sim},\ c^*2,\ A+,\ A-,\ c,\ x_{+\sim},\ x_{-+},$
$c^*2,\ y\_w_{+\sim},\ y\_w_{-+},\ c^*4)_{L2}{}^*5>$

$TID' = <!y,!A,!x,!c,\ c_4,\ c^*2,\ y_{+\sim},\ y_{-\sim},\ c^*2,\ A+,\ A-,\ c,\ x_{+\sim},\ x_{-+},\ c^*2,$
$y\_w_{+\sim},\ y\_w_{-+},\ c^*4,$
$\lambda^*3,\ y_{+\sim},\ y_{-\sim},\ \lambda^*2,\ A+,\ A-,\ \lambda,\ x_{+\sim},\ x_{-+},\ \lambda^*2,\ y\_w_{+\sim},\ y\_w_{-+},\ \lambda^*4,$
$(\lambda^*3,\ y_{+\sim},\ y_{-\sim},\ \lambda^*5,\ x_{+\sim},\ x_{-+},\ \lambda^*8)^*2,$
$\lambda^*3,\ y_{+\sim},\ y_{-\sim},\ \lambda^*5,\ x_{+\sim},\ x_{-+},\ \lambda^*2,\ y\_w_{+\sim},\ y\_w_{-+},\ \lambda^*4>$

$TID2' = <!y,!A,!x,!c,\ c_4,\ c^*2,\ y_{+\sim},\ y_{-\sim},\ c^*2,\ A+,\ A-,\ c,\ x_{+\sim},\ x_{-+},\ c^*2,$
$\lambda^*2,\ c^*4,$
$(\lambda^*3,\ y_{+\sim},\ y_{-\sim},\ \lambda^*5,\ x_{+\sim},\ x_{-+},\ \lambda^*8)^*4>$

When analyzing the relationship between the input and output of the different levels in this cache hierarchy, it can be seen that the first level cache does a fairly good job removing most of the locality. Indeed it performs perfectly if the cache set assignments in the D-Cache are done well. If the cache set assignments are not done well, the L2 cache can capture the other non-compulsory references from T, as long as the cache set assignments are not the worst-case scenario. It is possible, though, with certain cache set assignments, that the L2 cache will not capture any more references than the L1 combination caches. This is because the direct mapped aspect of the L2 cache makes it vulnerable to poor cache set assignments. Increasing the size of the L2 cache by making it more associative rather than simply a big direct mapped cache would allow these potential worst-case conflicts to be captured. This last observation is consistent with the approach taken by the filter function model; each cache acts as a filter and those further down the hierarchy should be designed to capture references that cannot be captured by the preceding caches.

5.5 Summary

This chapter has shown that analysis with the functional filter framework as defined in Chapters 2-3 can be done. First we examined several kernel examples to demonstrate they could be translated into TSpec and filtered using the functional filter model. Then we performed an analysis for a multi-level cache hierarchy on the kernel daxpy. The next chapter summarizes the dissertation, making specific conclusions and outlining future work.

Chapter 6

Conclusion

We have shown it is possible to design and evaluate cache memory systems more formally than is currently done. For this formalization, we have proposed an analytical framework for cache design that provides a common notation, TSpec, for expressing the memory references of a program, a functional cache filter model for comparing traditional cache hierarchy effects, and new instantaneous evaluation metrics that give greater insight into the operation of caches. The functional cache filter model is made possible by the concept of an equivalence class of memory references that provides an abstraction for eliminating certain types of random address placement effects. In addition to aiding in the design of improved cache systems, we believe this approach can provide a foundation for more rigorous analysis of other components of the computer system.

6.1 Contributions

The development of the caches-as-filters framework has enabled several observations. The first and most important of these is that, taken as a whole, the framework provides a formalism through the language that enables a researcher to focus on the caching system and reason about it in new and insightful ways. We have found the TSpec language, aspects of the filter function model such as U(T), and the locality measure to have reshaped our conversations as researchers, leading us to further investigation and inquiry. Several other observations are listed below, but this is the most important contribution.

6.1.1 Additional Observations

Observation 1: Primitives provide comprehensible and reusable units for analysis.

Observation 2: Decomposition identifies independent sources of locality.

Observation 3: Decomposition separates internal construct conflicts from external.

Observation 4: Equivalence classes differentiate between point results and class results for experiments or analyses.

Observation 5: Equivalence classes differentiate between what can be expected from a cache and what is dictated by other aspects of the system.

Observation 6: Fully associative caches may not be the high standard for removing references that they are often expected to be and this framework allows a researcher to identify when they are likely not to meet this expectation.

Observation 7: The instantaneous locality provides an enlightening figure of merit for caches, enabling discussion about how a cache “shapes” references to the next level.

Observation 8: The instantaneous locality measure helps explain some known design benefits, such as the benefit of separate instruction and data caches, or the use of multi-lateral caches [Tam99].

Observation 9: Caching systems should be designed so that individual caches complement each other, removing references others in the system cannot.

Observation 10: TSpec provides a formal language for researchers to discuss reference patterns and exchange information about them.

6.2 Future Work

6.2.1 Memory Access Patterns

One of the more interesting pieces of future work would be to perform more experiments on real codes to determine exactly how well the primitives in the filter function dictionary span the space of reference traces. Frequently in our discussions we have analyzed a trace for a particular kernel, just to realize it looks very similar to some other kernel. This effort would need to be supported by some of the automation discussed in Section 6.2.4 below.

6.2.2 Filter Function Model

This dissertation showed that the filter function model can be used to analyze kernels and gave some insight into why some traditional cache designs provide performance improvements. Developing new caching solutions was beyond the scope of this work, but we feel sure that this kind of analysis methodology will allow us to do just that. Future work could expand the filter function dictionary for different application spaces and expand the model to include larger line sizes. In addition, the model could be used to investigate the behavior of translation look aside buffers (TLBs), write buffers, and possibly branch predictors. These structures are relatively small, lending themselves well to analysis with this model. They are also critical to performance and even small improvements might make an impact.

6.2.3 Locality Measures

Our initial locality measures have proven useful in preliminary investigations, but there are several ways in which they might be improved (or changed to illustrate other properties of memory system behavior). We describe some potential differences here.

To reduce the amount of computation time and state required for each memory reference, we have developed a simpler measure that has similar analytical characteristics to the instantaneous locality measure described in Section 4.2. This measure introduces the concept of a *historical address* that attempts to summarize information about all previous addresses in the string. It does this by applying an exponentially smoothed weighting factor, β, to each address. Assuming a reference string $\langle a_0, a_1, a_2, a_3, a_4, \ldots \rangle$, this "quick" instantaneous locality measure, q_i, is defined as:

$$q_0 = 0$$

$$q_i = \frac{\alpha}{|A_{i-1} - a_i| + 1} + (1 - \alpha) \cdot q_{i-1} \text{ where } i > 0,\ \alpha \leq 1$$

Where the historical address, A_i, is:

$$A_0 = a_0$$

$$A_i = \beta a_i + (1 - \beta) \cdot A_{i-1} \text{ where } i > 0,\ \beta \leq 1$$

This measure is sensitive to the values of weighting factors α and β, so care must be taken in choosing them, but preliminary investigations show that this formula produces similar curves to those presented in Chapter 4.

Other formulas for the spatial locality component may be more useful than the difference in bytes between two addresses. Measures that use a step function to incorporate the notion of cache line sizes or bus fetch sizes should be investigated, so that items that are equally “close” in terms of the memory system organization will have the same spatial locality value. Similarly, for memory components that perform automatic prefetching, the spatial locality component could reflect the prefetch distance, since items that lie within the prefetch distance are “closer” than those beyond this distance.

In the temporal component of a locality measure for cache hierarchies, it may make more sense to use the number of unique addresses between two references (the LRU distance), instead of the total number of addresses. Another potentially interesting variant entails incorporating optimal replacement (the OPT distance, or the number of unique addresses between the current reference and the next reference to it in the future) into the temporal locality component.

Another measure we are considering in conjunction with locality measures, is an entropy measure to determine the predictability of a reference string.

Future work could expand these results by:

- running more experiments on longer traces;
- developing new measures of locality and cache efficiency;
- using our measures to characterize workloads and to evaluate their effectiveness with respect to exercising memory hierarchies;
- defining the mathematical properties required in a locality measure.

In addition to developing these new measures, the current measures could be refined. In particular an investigation into the appropriate method for normalizing them should be undertaken. The measures as they stand were meant to be exercised with different window values and smoothing factors. In addition, we found it useful for the instantaneous locality value to have the properties that a longer trace has more capacity for locality, and that two traces, one simply a first part of the second, would have the same locality graph for the parts of the trace that are the same. Care would need to be taken that any normalization maintain these properties and facilitate comparisons between different traces.

6.2.4 Automation

Several aspects of this framework can be automated now that some of the basic ideas have been worked out. First, a translator packaged with a compiler that could generate a form of TSpec for a given source program would enable quick and accurate examination of more kernels. Second, tools that assist in determining patterns in memory reference traces and categorizing them would enable further investigation into potential primitives. Third, assistance in the merge operation would enable larger segments to be concatenated and analyzed in a timely manner. Finally, the new measures could be incorporated into current simulation infrastructures such as SimpleScalar [Bur97] so that more researchers could make use of them. Some of these tools have been developed. While not appropriate for public distribution, they demonstrate that much of this automation is feasible.

6.3 Closing Remarks

It is worthwhile to emphasize that this framework formalizes the operation of a caching system, providing a means for researchers to reason about such systems in insightful ways. It is our sincere hope that others will adopt the philosophy of this framework and assist in improving and adding to it to develop a shared body of knowledge about fundamental aspects of caching systems. In this way, the community can reap the full benefits of the talents of every research team.

Appendix A

Filter Dictionary

A.1 General Information

This appendix contains the functional filter dictionary referred to in the previous chapters. This section contains information that is applicable to all of the individual filter functions including an explanation of the formula for the effective set number (esn) in a cache, the formal definitions of *fill* and *bind*, and an explanation of the format for each entry in the dictionary. Section A.2 describes the filter function for each primitive in detail. Section A.3 contains a table listing of the filter functions without any explanations.

A.1.1 Explanation of the Effective Set Number (esn) Formula

Throughout this dissertation we have used the term *effective set number* (esn) to refer to the number of sets in a cache that can be used by a particular reference string. This leads to the term *effective cache lines* (ecl) to refer to the total number of cache lines that can be used by a particular reference string for the entire cache. The number of effective cache lines is simply the number of *lines per set* (lps) multiplied by the effective set number (esn), $ecl = lps \times esn$. For a fully associative cache, ***lps*** is equal to the number of lines in the cache and the effective set number is always one because there is only one set. For a direct mapped cache *lps = 1* and ***esn* =** number of lines in the cache.

Effective set number is used primarily to generalize stream accesses. [Since code can be seen as a stream access with a stride equal to the word size of an instruction, this includes many common primitives.] The formal definition of effective set number is given below. Here *sn* represents the number of sets in the cache, *st* represents stride of the

$$esn = \frac{sn}{gcf(st,ls) \times gcf(st,sn)}$$

stream, and *ls* represents the line size. *gcf* refers to the greatest common factor of the numbers in parentheses. For fully associative caches, ***sn = 1***. For direct mapped or set associative caches, ***sn*** is equal to the number of lines in the cache set. Linesize and stride are discussed in terms of the memory fetch size. If one whole line is fetched from the next level of memory at once, we consider the line size to be one. If the stride is one line size, then we consider the stride to be one. A two-way set associative cache with a line size of one memory fetch, a memory fetch size of 4 bytes, and total cache size of 4K bytes would have an effective set number of 512. A stream with a stride of 4 lines (16 bytes) would be able to use 128 sets of this cache, thus it would have an ***esn = 128***.

The intuition behind this formula is that the number of sets a stream can make use of in a cache is dependent on how many sets the stream can map to before it begins to conflict with itself. If a stream’s stride is relatively prime to the number of sets and ls, then every set in the cache can be used before the stream begins to evict itself. The evictions due to the conflicting stream elements effectively reduce the cache sets available to the stream in a particular cache, hence the name effective set number.

Consider the case where the line size is one and the formula simplifies to: $esn = \frac{sn}{gcf(st,sn)}$. In this case the derivation of the formula is fairly straightforward. We want to find the element, *n*, in the stream where the conflicts begin. Without loss of generality we can assume that the first element in the stream maps to cache set zero. We are then looking for the smallest *n* greater than zero such that $(n \times st) \bmod sn = 0$. Hence $n \times st = k \times sn$ for some $(k = 1,2,3,\ldots)$. Certainly this is true when ***st* = *k* = *1*** and ***n* = *sn***. In some cases, however, where the stride has factors in common with *sn*, *n* could be even smaller. Specifically, the *n* can be smaller by the number of factors that *st* and *lps* have in common, or ***gcf(st, lps)***, and $n = esn = \frac{sn}{gcf(st,sn)}$. A similar argument can be made for the effect of the line size on *sn*.

A.1.2 Λ(T), Fill (Δ) and Bind

To simplify the description of the output of several of the functional filters, some additional functions on a trace T, and one function on TSpec elements are useful.

The first function is denoted Λ(T) and removes all of the non-λ elements in T and replaces them with the placeholder $\overline{\lambda}$ ("not" λ). For example, if ***T* = < λ, λ, *f*, λ, *t*, λ >** then **Λ *(T)* = < λ, λ, $\overline{\lambda}$, λ, $\overline{\lambda}$, λ >.** More formally, $\forall i$ such that $(1 \le i \le |T|)$ if T = Λ(T1) then

$$t_i = \begin{cases} \lambda \text{ if } (t1_i = \lambda) \\ \overline{\lambda} \text{ otherwise} \end{cases}$$

The second function, called *fill* and denoted by (Δ), operates on two traces by filling the $\overline{\lambda}$ elements of the first trace with the second trace. An element in the second trace may be λ, in which case $\overline{\lambda}$ is replaced by λ. For example if $\boldsymbol{T1 = < \lambda, \lambda, \overline{\lambda}, \lambda, \overline{\lambda}, \lambda >}$ and $\boldsymbol{T2 = < f, \lambda >}$, then $\boldsymbol{T = T1 \,\Delta\, T2 = < \lambda, \lambda, f, \lambda, \lambda, \lambda >}$**.** Note that fill is not commutative.

Finally, the last function we define here is *bind*, denoted by (:). Bind "binds" together two TSpec constructs so that they may be considered as having the same index in a trace. For example, ***T = <f: f_w >*** is considered to be ***T = < <f: f_w, 1> >*** rather than ***T = < <f, 1>, <f_w, 2>>***. Bind is useful when using a fill to describe the output of a write back cache. When a write back occurs, it can be bound to the access that caused the write back and so "filled" in to a single $\overline{\lambda}$.

A.1.3 Dictionary Format Details

The filtering function for a number of primitives are detailed in the section below. Each primitive and its filtering function(s) for a variety of caches are described in a separate subsection. First the general form of the primitive is given using parameters. P is used to denote the primitive, and specific elements in P are denoted with p. Then the filtering function formula is presented in terms of the primitive (P) and its parameters. Each function has three cache characterization modifiers in the subscript that specify its applicability as far as associativity (as), write policy (wp) and line size (ls). Fully associative caches are denoted *fa*. If a function applies to every cache associativity other than a fully associative, it is denoted $\overline{fa}$ ("not" *fa*). Direct mapped caches are denoted by an associativity of 1. All other set associativities are denoted by the number of lines per set. The write policy can either be write thru (*wt*) or write back (*wb*). The line size (as discussed above) is represented in terms of the number of memory fetches required to fill the line. Each function is then represented as $f_{\mathrm{as,wp,ls}}(P; S^0) = P', S'$. It is assumed that all cache line replacement algorithms are (Least Recently Used) LRU. If any cache parameter is listed as *, the function applies to caches with any value for that parameter.

A.2 Detailed Descriptions of Filters for Primitives

A.2.1 Repeated Single Read

p(?, ?);
$P = \langle !p, \lambda*n1, p\sim, P1\rangle$ where $D(P1) = \langle (p\sim)*n2\rangle$

$f_{*,*,*}(P; S^0) = \langle !p, \lambda*n1, p\sim, \lambda*|P1|\rangle; \{p!, p\sim\}$

This function describes the filtering of any single repeated read for a cache of any associativity, whether it is write thru or write back, and of any line size. Note that the primitive may contain an arbitrary number of preceding λs and that the repeated reads may be separated by any number of interleaving λs. These λs are retained in the output by adding |P1| λs instead of n2 or |D(P1)| λs.

A.2.2 Read Stream

p(?, st);   // st > 0
$P = \langle !p, (\lambda*n1, p, \lambda*n2)*n3\rangle$

$f_{*,*,1}(P; S^0) = \langle P\rangle; \{p!, (p-)*min(ecl, n3)\}$

This result is a sub case of the primitive presented in Section A.2.3. Since there are no repeated addresses in the read stream, the cache passes every read from the input to the output and the output trace equals the input trace. After the read, the cache state consists of as much of the stream as will fit in the cache.

A.2.3 Read Stream, Straight Line Code

p(?, st); // st > 0
*P such that D(P) = <!p, p*n1>*

$f_{*,*,1}(P;\, S^0) = \langle P\rangle;\ \{p!,\ (p-)*\min(ecl,\ n1)\}$

Since there are no repeated addresses in the read stream, the cache passes every read from the input to the output and the output trace equals the input trace. After the read, the cache state consists of as much of the stream as will fit in the cache. (Note this is a generalization of Section A.2.2 for irregular interleaving λs.)

A.2.4 Repeated Read Stream, Single Nest Code Loop

p(?, st); // st > 0
*P = <P1*n2> such that D(P1) = <!p, p*n1>*

$f_{*,*,1}(P;\, S^0) = \langle P1,\ (\lambda*|P1|)*(n2-1)\rangle;\ \{p!,\ (p-)*n1\}$ if $n1 \le ecl$ *(i)*

$f_{fa,*,1}(P; S^0) = \langle P\rangle;\ \{p!,\ (p-)*ecl\}$ if $n1 \ge ecl$ *(ii)*

$f_{\overline{fa},*,1}(P;\, S^0) = \langle \Lambda(P)\ \Delta\ [P1,\ ({}_{L}P1_{n1\text{-}ecl},\ \lambda*(2ecl\text{-}n1),\ p_{ecl+1},\ p*(n1\text{-}ecl\text{-}1))_{L}*(n2\text{-}1)]\rangle;$
$\{p!,\ (p-)*ecl\}$ if $ecl \le n1 \le 2 \times ecl$ *(iii)*

$f_{\overline{fa},*,1}(P;\, S^0) = \langle P\rangle;\ \{p!,\ (p-)*ecl\}$ if $(2 \times ecl \le n1)$ *(iv)*

Some general comments are useful to set the stage for the more complicated results. First, recall that Λ(T) removes all the non-λ elements from T and T1 Δ T2 “fills” the $\overline{\lambda}$ elements of T1 with consecutive elements of T2. These definitions allow us to use

Λ(P) Δ P' to remove the non-λ elements of P and refill it with the filtered version of P and so maintain the correct λ relationships. Second, there are three categories of behavior for streams in set associative caches. These three categories correspond to the stream fitting completely in the cache, case (i) above, the stream partially fitting in the cache, case (iii) above, or the stream completely evicting itself before it can be reused, case (iv) above. The values for when these boundary points arise are depicted in Figure 43. The parameters are

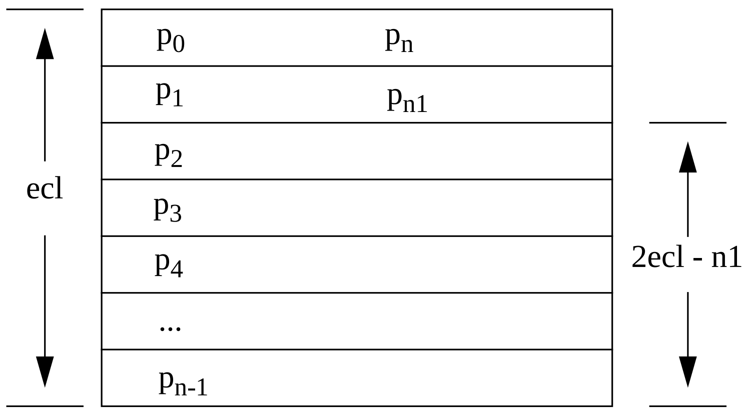


FIGURE 43: Values for Behavioral Boundaries As Stream Size Increases

those applicable for (iii) and the primitive above.

(i) states that if the loop or stream (P1) is small enough to fit in the cache, then the trace output consists of just one iteration of P1 followed by λs for the rest of the iterations. The state is also one iteration of P1, but the last one. This is true regardless of the associativity or write policy of the cache. If P1 does not fit in the cache, the results differ depending on the associativity of the cache and the length of P1.

(ii) describes the case where the cache is fully associative and P1 does not fit in the cache. Then the cache does not filter any references from the input and the output trace is equal to the input trace. The state contains as much of the latter part of P1 as will fit in the cache.

(iii) and (iv) cover two separate cases for direct mapped and set associative caches. The first (iii) is the situation where P1 does not all fit in the cache, but partially fits in the cache. Some parts of the middle of P1 (specifically, 2ecl - n1 elements) will stay in the cache and be available for future iterations, but the beginning and end of P1 will evict one another, and so be passed on to the output trace.

(iv) shows the case where the length of P1 is greater than twice the effective number of cache lines (ecl), so all of P1 is evicted from the cache before it can be used again. Here, as in the fully associative case, all of the input trace is passed on to the output trace and they are equal.

A.2.5 Repeated Single Write

p(?, ?);
$P = \langle !p, \lambda * n1, p\_w\sim, P1\rangle$ where $D(P1) = \langle (p\_w\sim) * n2\rangle$

$f_{*,wt,1}(P; S^0) = \langle P\rangle; \{p!, p\_w\sim\}$ *(i)*

$f_{*,wb,1}(P; S^0) = \langle !p, \lambda * n1, p\_w\sim, \lambda * |P1|\rangle; \{p!, p\_w\sim\}$ *(ii)*

(i) shows the case of a write thru cache, of any associativity. In this case, all writes are passed on to the output, so the output trace is equal to the input trace. (ii) shows the case of a write-back cache. Regardless of the associativity of a write back cache, a single write will never conflict with itself, so only the first write will be passed from the input to the output trace. Note that the repetitions of the write in the input may be interleaved with any number of λs. To retain these λs in the output trace, |P1| is used instead of n2 to indicate the number of λs in the output trace.

A.2.6 Write Stream

p(?, st) ; // st > 0
*P such that D(P) = <!p, p_w*n1>*

$f_{*,wt,1}(P; S^0)$ = *<P>; {p!, (p_w-)*min(ecl, n1)}* (i)

$f_{*,wb,1}(P; S^0)$ = *<P>; {p!,(p_w-)*n1}* $n1 \leq ecl$ (ii)

$f_{*,wb,1}(P; S^0)$ = *<Λ(P) Δ [D(P_{ecl}), (p_w_y:p_w_{y+ecl})*(n1-ecl)]>;* (iii)
where y=1,2,3,...(n1-ecl)
*{p!, (p_w-)*ecl}* $ecl < n1$

(i) shows that for a write through cache, all writes are passed through to the output and the output is equal to the input. The state contains as much of the end of the stream as will fit in the cache. (ii) describes what happens when the whole stream fits into a write back cache. Since there are no repeated accesses, every access passes through to the output and again the output equals the input. Since the whole stream fits in the cache, the state is equal to the stream. (iii) describes the situation when the stream is longer than the number of effective cache lines and once the stream begins to wrap around, the write back of the dirty line from earlier in the stream must be added for each access. The state is as much of the end of the stream as will fit in the cache. Note that the subscript ecl on P refers to the trace index where the subscripts of p represented by the variable y are a TSpec iterator count.

A.2.7 Uneven Reads Within Range, R x st

p(?, 1);
*P such that D(P) = <!p, (p_x)*n1> where x = random value 1-R*

$f_{*,*,1}(P; S^0)$ = *<U(P)>; {last ecl of D(U($\overline{P}$))}*

For this primitive, the function is reverting back to the recursive definition to determine the output by computing U(P) and U($\overline{P}$). (Recall $\overline{P}$ is the reverse of P.) For the purpose of computing ecl, this primitive has a stride of one so esn is equal to sn. Note that it is possible that there are enough conflicts or so few references in the primitive that the cache may not be filled with the pattern. When dealing with this kind of primitive, it may be preferable to break it down to even smaller primitives where there is a clear relationship between the elements.

A.2.8 Uneven Writes Within Range, R x st

p(?, 1);
P such that $D(P) = <!p, (p\_w_y)*n1>$ *where y = random value 1 ... R*

$f_{*,wt,1}(P; S^0) = P; \{\text{last ecl of } D(U(\overline{P}))\}$ *(i)*

$f_{fa,wb,1}(P; S^0) = <U(P)>; \{D(U(\overline{P}))\}$ $\quad |D(U(\overline{P}))| \le ecl$ *(ii)*

$f_{fa,wb,1}(P; S^0) = <\Lambda(P)\ \Delta\ [U(P)_{ecl}, (***)>; \{\text{last ecl of } D(U(\overline{P}))\}$*(iii)*

(i) shows the case of a write through cache where every write passes from the input to the output and the output is equal to the input. The state is the last effective cache lines of D(U($\overline{P}$)). Again the stride is equal to one so the esn equals the actual set number. Also, there may be so few references in D(U($\overline{P}$)) that the cache is not filled with the pattern. (ii) covers the case of a fully associative cache where the pattern fits in the cache. Here the duplicates only are removed from the input. There are no conflicts or write backs because it is a fully associative cache and everything fits. (iii) shows the write backs that would be needed once the fully associative cache is filled. Here the subscript ecl on U(P) refers to the trace index. Note that there is no entry for the set associative and direct-mapped cases in a write back cache. This is because it is impossible to tell when the conflicts will occur and cause write backs without making use of the recursive definition. In this situation the primitive should either be broken into smaller primitive whose elements have a more clear relationship to one another, or the recursive definition given in Chapter 3 (***) should be used.

A.2.9 Read, Write Stream

p(?, st)   // st > 0
*P such that D(P) = <!p, (p~, p_w)*n1>*

$f_{*,wt,1}(P; S^0) = \langle P\rangle; \{p!, (p\_w-)*\min(ecl, n1)\}$ (i)

$f_{*,wb,1}(P; S^0) = \langle\Lambda(P)\ \Delta\ (!p, (p, \lambda)*n1)\rangle; \{p!, (p\_w-)*n1\}$ $n1 \le ecl$ (ii)

$f*,wb,1(P; S0) = \langle\Lambda(P)\ \Delta\ [D(P_{ecl}), (p\_w_y{:}p\_w_{y+ecl})*(n1-ecl)]\rangle;$ (iii)
where $y=1,2,3,...(n1-ecl)$ $ecl < n1$
$\{p!, (p\_w-)*ecl\}$

Recall that p~ does not increment the variable, but p_w or p_w+ do. (i) shows that for a write through cache, all writes are passed through to the output and since all the reads are the first access to that address they are also passed through. Therefore, the output is equal to the input. The state contains as much of the end of the stream as will fit in the cache. Note that the write is kept as part of the state to indicate this line is dirty in case a future write back is needed.

(ii) describes what happens when the whole stream fits into a write back cache. Since the writes are repeated accesses, they do not pass through to the output. Since the whole stream fits in the cache and the last access to each element is a write, the state contains the write accesses.

(iii) describes the situation when the stream is longer than the number of effective cache lines and once the stream begins to wrap around, the write back of the dirty line from earlier in the stream must be added for each access. The state is as much of the end of the stream as will fit in the cache. Note that the subscript ecl on P refers to the trace index where the subscripts of p represented by the variable y are a TSpec iterator count.

A.2.10 Double Read Stream

$p(?, ws, st)$ // $st > 0$
P such that $D(P) = <!p, (p_{+\sim}, p_{-+})*n1>$

$f_{*,*,1}(P; S^0) = <P>; \{p!, (p\text{-}\sim, p\text{+-})*min(ecl, n1)\}$

This shows the form of a double read stream and its filtered output. A double read is one to a vector element that is twice the word size of the machine, and twice the line size. Since there are no repeated accesses, the output trace is equal to the input trace.

A.2.11 Double Write Stream

$p(?, ws, st)$ // $st > 0$
P such that $D(P) = <!p, (p\_w_{+\sim}, p\_w_{-+})*n1>$

$f_{*,wt,1}(P; S^0) = <P>; \{p!, (p\_w\text{-}\sim, p\_w\text{+-})*min(ecl, n1)\}$ (i)

$f_{*,wb,1}(P; S^0) = <P>; \{p!, (p\_w\text{-}\sim, p\_w\text{+-})* n1\}$ $n1 \le ecl$ (ii)

$f_{*,wb,1}(P; S^0) = <\Lambda(P)\ \Delta\ [D(P_{ecl}), (p\_w_y{:}p\_w_{y+ecl}, p\_w_{y+1}{:}p\_w_{y+1+ecl})*(n1\text{-}ecl)]>;$
where $y=1,2,3,...(n1\text{-}ecl)$
$\{p!, (p\_w\text{-}\sim, p\_w\text{+-})*ecl\}$ $ecl < n1$ (iii)

(i) shows that for a write through cache, all writes are passed through to the output and the output is equal to the input. The state contains as much of the end of the stream as will fit in the cache. (ii) describes what happens when the whole stream fits into a write back cache. Since there are no repeated accesses, every access passes through to the output and again the output equals the input. Since the whole stream fits in the cache, the state is equal to the stream. (iii) describes the situation when the stream is longer than the number of effective cache lines and once the stream begins to wrap around, the write back of the dirty linse from earlier in the stream must be added for each access. The state is as much of the end of the stream as will fit in the cache. Note that the subscript ecl on P refers to the trace index where the subscripts of p represented by the variable y are a TSpec iterator count.

A.2.12 Conditional

p(?, ws);
*P1 = <!p, p*n1>*
$P2 = <p_{n1+1}, p*n2>$
$P3 = <p_{n1+n2+2}, p*n3>$
$P4 = <p_{n1+n2+n3+3}, p*n4>$

P such that D(P) = <P1, {P2 | P3}, P4>

$$f_{*,*,1}(P; S^0) = <P>; \{last\ ecl\ of\ D(P)\} = \\ <P>; \{\{last\ ecl\ of\ \overline{P4}, \overline{P2}, \overline{P1}\}\ OR\ \{last\ ecl\ of\ \overline{P4}, \overline{P3}, \overline{P1}\}\}$$

This entry shows two different ways of representing the state when encountering a conditional. The actual output will depend on which part of the input is actually used. Since there are no repeating references the output trace is equal to the input trace.

A.2.13 Single Nest Code Loop With Conditional

p(?, ws);
*P1 = <!p, p*n1>*
$P2 = <p_{n1+1}, p*n2>$
$P3 = <p_{n1+n2+2}, p*n3>$
$P4 = <p_{n1+n2+n3+3}, p*n4>$

*P such that D(P) = <(P1, {P2 | P3}, P4)*n5>*

$$f_{*,*,1}(P; S^0) = <\Lambda(P)\ \Delta\ [P1, \{P2 \mid P3\}, P4), (\ \{\lambda*n1, P2, \lambda*(n4+1) \mid \lambda*n1, P3, \lambda*(n4+1) \mid \lambda*(n1+n2+n4+2) \mid \lambda*(n1+n3+n4+2)\})*(n5-1)]>; \qquad (i)$$

$$\{D(\overline{P})\}\ \text{where}\ D(\overline{P}) = \{\{\ \overline{P4}, \overline{P2}, \overline{P1}\}\ OR\ \{\overline{P4}, \overline{P3}, \overline{P1}\}\ OR\ \{\overline{P4}, \overline{P3}, \overline{P2}, \overline{P1}\}\}$$

$$n1 + n2 + n3 + n4 + 3 \le ecl$$

(i) shows the filtering function in the case where the loop fits in the cache. The output passes through the whole loop in the first iteration, with either P2 or P3, but not both. In the subsequent iterations one of four possibilities can occur each time. First the loop could be run with either P2 or P3 again, whichever was not executed in the first loop. Second, the loop could rerun P2 or P3 again. The state for this case is just the part of the loop outside the conditional, and whatever conditional piece of code is actually run. The conditional part could be either P2 or P3 alone, or both P2 and P3.

The filtering function when part of the loop does not fit in the cache depends on how much of the loop does not fit. The output for subsequent iterations will be changed in that the beginning and ends of the loop will overlap and kick each other out of the cache much like the nested loop code in Section A.2.4. There will be more dependence on exactly which section of the code is executed and the relative sizes, but once the loop is at least twice as long as ecl, every piece of code will be evicted before it is used again and the output trace will equal the input trace, and the state will equal the last part of the loop that is executed.

A.2.14 Doubly Nested Code Loop

$p(?, ws);$
$P1 = <!p, p*n1>$
$P2 = <p_{n1+1}, p*n2>$
$P3 = <p_{n1+n2+2}, p*n3>$

P such that $D(P) = <(P1, (P2)*n4, P3)*n5>$

$$f_{fa,*,1}(P; S^0) = < \Lambda(P)\ \Delta <P1, P2, \lambda*((n2+1)(n4-1)), P3, \lambda*(n1+(n2+1)(n4)+n3+1)(n5-1)>>; \{<\overline{P3}, \overline{P2}, \overline{P1}> \} \quad n1 + n2 + n3 + 2 \le ecl \qquad (i)$$

$$f_{fa,*,1}(P; S^0) = < \Lambda(P)\ \Delta <(P1, P2, \lambda*((n2+1)(n4-1)), P3)*n5>>; \{\text{last } ecl \text{ of } <\overline{P3}, \overline{P2}, \overline{P1}>\} \quad n2 \le ecl < n1 + n2 + n3 + 2 \qquad (ii)$$

$$f_{fa,*,1}(P; S^0) = <P>; \{\text{last } ecl \text{ of } <\overline{P3}, \overline{P2}, \overline{P1}>\} \quad ecl \le n2 \qquad (iii)$$

(i) demonstrates the effect of a fully associative cache on the doubly nested loop when everything fits in the cache. One iteration of each access is passed on to the output trace. The state contains one of each reference. (ii) demonstrates the effect of the fully associative cache when the inner loop only fits into the cache. (iii) demonstrates the effect when even the inner loop does not fit in the cache. In this every reference is evicted before it can be used again and the output trace equals the input trace. The state is as much of the last part of the loop as will fit in the cache.

Appendix B

Code and TSpec Listings

B.1 General Settings

All assembly code included in this appendix was generated with a vpo (very portable optimizer) back end and an lcc front end. Two versions of the backend were used: one for the SPARC instruction set and one for the MIPS instruction set. Both backends are being developed at the University of Virginia by Davidson *et al.* [Ben88]. This compiler was chosen because its method of implementing optimizations is similar for different target machines and it was believed to generate relatively accurate comparisions in addition to being readily available. Three different sets of compiler options were used for most of the kernels. The first, referred to as completely unoptimized uses the settings -VGLA. The second, referred to as unoptimized with register allocation uses the settings -VGLAO. The third, referred to as fully optimized uses the settings -VGLAOFICMSV. The details of what each setting means are included in Table 2.

TABLE 2: VPO Compiler Option Descriptions

Option	Description
V	evalutation order determination
G	sets global links
L	does control flow fixups
A	indicates assembly output
F	delay slot filling (SPARC only)
O	register assignment
I	instruction scheduling (SPARC only)
C	common subexpression elimination
M	code motion
S	strength reduction and induction variable elimination

B.2 Copy

B.2.1 Copy C Code Listing

```
more copy.c
void copy(int *f, int *t, int N)
{
  int i;
  for (i=0; i<N; i++)
    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;
}
```

B.2.2 SPARC, Optimized, Assembly Code

```
more copy_opt.s
.section ".data"
.align 8
.K_D0:
.word 0
.word 0
.K_F0:
.word 0
.common __va_first_parm,4,4
.common __builtin_alloca,4,4
.global copy
.global main
.section ".text"
.align 8
copy:
.type copy,#function
! File = "../ccode/copy.c", Line = 3
! File = "../ccode/copy.c", Line = 5
        cmp     %g0,%o2
        bge     .L000
        nop
        sll     %o2,2,%g4
        mov     %o1,%g2
sub     %o0,%o1,%g3
        add     %g4,%o1,%g4
        ld      [%g2 + %g3],%o4
.L2:
! File = "../ccode/copy.c", Line = 6
        st      %o4,[%g2]
! File = "../ccode/copy.c", Line = 5
        add     %g2,4,%g2
        cmp     %g2,%g4
        bl,a    .L2
        ld      [%g2 + %g3],%o4
.L000:
! File = "../ccode/copy.c", Line = 7
        retl
        nop
.section ".rodata"
.align 4
.local L8_59018
L8_59018:
.word 0
.word 0
.word 0
.align 4
.local L10_59018
L10_59018:
.word 10
.word 11
.word 12
.section ".text"
.align 8
main:
.type main,#function
.l1.1_f = -16
.l1.0_t = -32
        save    %sp,(-128),%sp
! File = "../ccode/copy.c", Line = 10
! File = "../ccode/copy.c", Line = 12
        sethi   %hi(L8_59018),%o0
        add     %o0,%lo(L8_59018),%o0
        ld      [%o0 + 0],%o1
        st      %o1,[%fp + .l1.0_t]
        ld      [%o0 + 4],%o1
        st      %o1,[%fp + (.l1.0_t + 4)]
        ld      [%o0 + 8],%o1
st      %o1,[%fp + .l1.0_t]
        ld      [%o0 + 4],%o1
        st      %o1,[%fp + (.l1.0_t + 4)]
        ld      [%o0 + 8],%o1
        st      %o1,[%fp + (.l1.0_t + 8)]
! File = "../ccode/copy.c", Line = 13
        sethi   %hi(L10_59018),%o0
        add     %o0,%lo(L10_59018),%o0
        ld      [%o0 + 0],%o1
        st      %o1,[%fp + .l1.1_f]
        ld      [%o0 + 4],%o1
        st      %o1,[%fp + (.l1.1_f + 4)]
        ld      [%o0 + 8],%o1
        st      %o1,[%fp + (.l1.1_f + 8)]
! File = "../ccode/copy.c", Line = 14
        add     %fp,.l1.1_f,%o0
        add     %fp,.l1.0_t,%o1
        call    copy
        mov     3,%o2
! File = "../ccode/copy.c", Line = 19
        mov     %g0,%i0
        ret
        restore
```

B.2.3 SPARC, Optimized, TSpec

```
TSpec (copy_opt.s)
c1(MAIN_r, 4)
c2( COPY_r, 4);                // code variable
t (T_w, 4);                    // “to” vector
f (F_r, 4);                    // “from” vector
t1 (L8_59018_w, 4);            // main “to” vector
t2 (l1.0_t_w, 4);              // frame “to” vector
f1 (L10_59018_r, 4);           // main “from” vector
f2 (l1.1_f_r, 4);              // frame “from” vector
```

< !all, c1*3, (c1 t1_r c1 t2)*3 c1*2 // copy “to” vector to frame
(c1 f1 c1 f2_w)*3, c1*3, // copy “from” vector to frame
!c2, c*3, {$_{B1}$ c2*5 f ($_{L}$c2$_{6}$, t, c2*3, {c2, f, | break L})$_{L}$ *3 |$_{B1}$},// main copy routine
c2$_{13}$, c2, c1$_{5}$, c1$_{20}$, c1*4>

B.2.4 SPARC, Unoptimized Assembly Code

more copy_nopt.s

```
.section ".data"
.align 8
.K_D0:
.word 0
.word 0
.K_F0:
.word 0
.common __va_first_parm,4,4
.common __builtin_alloca,4,4
.global copy
.global main
.section ".text"
```

```
.align 8
copy:
.type copy,#function
.l0.0_i = -4
.l0.1_1 = -8
        save    %sp,(-72),%sp
.p0.0_f = 68
        st      %i0,[%fp + .p0.0_f]
.p0.1_t = 72
        st      %i1,[%fp + .p0.1_t]
.p0.2_N = 76
        st      %i2,[%fp + .p0.2_N]
! File = "../ccode/copy.c", Line = 3
! File = "../ccode/copy.c", Line = 5
        st      %g0,[%fp + .l0.0_i]
        ba,a    .L5
.L2:
! File = "../ccode/copy.c", Line = 6
        ld      [%fp + .l0.0_i],%o0
        sll     %o0,2,%o0
        st      %o0,[%fp + .l0.1_1]
        ld      [%fp + .l0.1_1],%o0
        ld      [%fp + .p0.0_f],%o1
        ld      [%o0 + %o1],%o0
        ld      [%fp + .l0.1_1],%o1
        ld      [%fp + .p0.1_t],%o2
        st      %o0,[%o1 + %o2]
! File = "../ccode/copy.c", Line = 5
        ld      [%fp + .l0.0_i],%o0
        add     %o0,1,%o0
        st      %o0,[%fp + .l0.0_i]
.L5:
        ld      [%fp + .l0.0_i],%o0
        ld      [%fp + .p0.2_N],%o1
        cmp     %o0,%o1
        bl      .L2
        nop
! File = "../ccode/copy.c", Line = 7
        ret
        restore
.section ".rodata"
.align 4
.local L8_59018
```

```
L8_59018:
.word 0
.word 0
.word 0
.align 4
.local L10_59018
L10_59018:
.word 10
.word 11
.word 12
.section ".text"
.align 8
main:
.type main,#function
.l1.0_t = -16
.l1.1_f = -32
.l1.2_i = -36
        save    %sp,(-128),%sp
.p1.0_argc = 68
        st      %i0,[%fp + .p1.0_argc]
.p1.1_argv = 72
        st      %i1,[%fp + .p1.1_argv]
! File = "../ccode/copy.c", Line = 10
! File = "../ccode/copy.c", Line = 12
        sethi   %hi(L8_59018),%o0
        add     %o0,%lo(L8_59018),%o0
        add     %fp,.l1.0_t,%o1
        ld      [%o0 + 0],%o2
        st      %o2,[%o1 + 0]
        ld      [%o0 + 4],%o2
        st      %o2,[%o1 + 4]
        ld      [%o0 + 8],%o2
        st      %o2,[%o1 + 8]
        ld      [%o0 + 8],%o2
        st      %o2,[%o1 + 8]
! File = "../ccode/copy.c", Line = 13
        sethi   %hi(L10_59018),%o0
        add     %o0,%lo(L10_59018),%o0
        add     %fp,.l1.1_f,%o1
        ld      [%o0 + 0],%o2
        st      %o2,[%o1 + 0]
        ld      [%o0 + 4],%o2
        st      %o2,[%o1 + 4]
        ld      [%o0 + 8],%o2
        st      %o2,[%o1 + 8]
! File = "../ccode/copy.c", Line = 14
        add     %fp,.l1.1_f,%o0
        add     %fp,.l1.0_t,%o1
        mov     3,%o2
        call    copy
        nop
! File = "../ccode/copy.c", Line = 19
        mov     %g0,%i0
        ret
        restore
```

B.2.5 SPARC, Unoptimized, TSpec

```
argc(ARGC_r, 0);
argv(ARGV_r, 0);
d1(.l1.0_t_w, 4);
d2(.l1.1_f_r, 4);
t(TO_w, 4);
f(FROM_r, 4);
c(MAIN_r, 4; COPY_r, 4; COPY+24_r, 4; RET_r, 4);
F(.p0.0_f_r, 0);
T(.p0.1_t_r, 0);
N(.p0.2_N_r, 0);
i(.l0.0_i_r, 0);
l(.l0.1_1_r, 0);

<!all, c*2, arc, c, argv, c*3, (c, d1, c, t)*3,                    // set up TO vector
   c*3, (c, d2, c, f_w)*3,                                          // set up FROM vector
   c*5, !c2, c*2, F_w, c, T, c, N_w, c, i_w, c,                     // set up locals for copy
   (!c3, c, i, c*2, l_w, c, l, c, F, c, f, c, l, c, T_r, c, t, c, i, c*2, i_w,
     c, i, c, N, c*3)*3, c*2, !c4, c*3>
```

B.3 Matmul

B.3.1 Matmul C Code Listing

```
(mamba) /af4/daw4q/vpostuff/ccode $ more matmul.c

#define SIZE 5

void matmul(double x[SIZE][SIZE], double y[SIZE][SIZE],
            double z[SIZE][SIZE], int N)
{
  int i;
  int j;
  int k;
  double r;

  for (i=0; i<N; i++) {
    for (j=0; j<N; j++){
      r = x[i][j];
      for (k=0; k<N; k++) {
        z[i][k] += r*y[j][k];
      }
    }
  }

}
/* This main() is simply to illustrate the calling of matmul() */
main(int argc, char **argv)
{
  int i;
  int j;
  double x[SIZE][SIZE];
  double y[SIZE][SIZE];
  double z[SIZE][SIZE];

  for (i=0; i < SIZE; i++)
    for (j=0; j < SIZE; j++)
      x[i][j] = y[i][j] = 2.0;

  matmul(x, y, z, SIZE);

#ifdef DEBUG
  printf("t = {%d, %d, %d}\n", t[0], t[1], t[2]);
#endif
  return 0;
}
```

B.3.2 SPARC, Unoptimized Assembly with Register Allocation

```
(mamba) /af4/daw4q/vpostuff/sparc $ more matmul_noptr.s
.section ".data"
.align 8
```

```
.K_D0:
.word 0
.word 0
.K_F0:
.word 0
.common __va_first_parm,4,4
.common __builtin_alloca,4,4
.global matmul
.global main
.section ".text"
.align 8
matmul:
.type matmul,#function
! File = "../ccode/matmul.c", Line = 6
! File = "../ccode/matmul.c", Line = 12
        mov     %g0,%g6
        cmp     %g6,%o3
        bge     .L000
        nop
.L2:
! File = "../ccode/matmul.c", Line = 13
        mov     %g0,%g2
        cmp     %g2,%o3
        bge     .L001
        nop
.L6:
! File = "../ccode/matmul.c", Line = 14
        smul    %g6,40,%g3
        sll     %g2,3,%g7
        add     %g3,%o0,%g3
        ldd     [%g7 + %g3],%f6
! File = "../ccode/matmul.c", Line = 15
        mov     %g0,%o5
        cmp     %o5,%o3
        bge     .L002
        nop
.L10:
! File = "../ccode/matmul.c", Line = 16
        smul    %g6,40,%g3
        smul    %g2,40,%g1
        sll     %o5,3,%g5
        add     %g3,%o2,%g3
        add     %g5,%g3,%o4
        ldd     [%o4],%f0
        add     %g1,%o1,%g3
        ldd     [%g5 + %g3],%f4
        fmuld   %f6,%f4,%f2
        faddd   %f0,%f2,%f0
        std     %f0,[%o4]
! File = "../ccode/matmul.c", Line = 17
! File = "../ccode/matmul.c", Line = 15
        add     %o5,1,%o5
        cmp     %o5,%o3
        bl      .L10
        nop
.L002:
! File = "../ccode/matmul.c", Line = 18
! File = "../ccode/matmul.c", Line = 13
        add     %g2,1,%g2
        cmp     %g2,%o3
        bl      .L6
        nop
.L001:
! File = "../ccode/matmul.c", Line = 19
! File = "../ccode/matmul.c", Line = 12
        add     %g6,1,%g6
        cmp     %g6,%o3
        bl      .L2
        nop
.L000:
! File = "../ccode/matmul.c", Line = 21
        retl
        nop
.align 8
main:
.type main,#function
.l1.2_x = -200
.l1.3_y = -400
.l1.4_z = -600
        save    %sp,(-696),%sp
! File = "../ccode/matmul.c", Line = 24
! File = "../ccode/matmul.c", Line = 31
        mov     %g0,%g1
.L15:
! File = "../ccode/matmul.c", Line = 32
        mov     %g0,%o5
```

```
.L19:
! File = "../ccode/matmul.c", Line = 33
        smul    %g1,40,%o3
        sll     %o5,3,%o4
        sethi   %hi(L24_47095),%o0
        ldd     [%o0 + %lo(L24_47095)],%f2
        add     %fp,.l1.3_y,%o2
        add     %o3,%o2,%o1
        std     %f2,[%o4 + %o1]
        add     %fp,.l1.2_x,%o2
        add     %o3,%o2,%o1
        std     %f2,[%o4 + %o1]
! File = "../ccode/matmul.c", Line = 32
        add     %o5,1,%o5
        cmp     %o5,5
        bl      .L19
        nop
! File = "../ccode/matmul.c", Line = 31
        add     %g1,1,%g1
        cmp     %g1,5
        bl      .L15
        nop
        add     %g1,1,%g1
        cmp     %g1,5
        bl      .L15
        nop
! File = "../ccode/matmul.c", Line = 35
        add     %fp,.l1.2_x,%o0
        add     %fp,.l1.3_y,%o1
        add     %fp,.l1.4_z,%o2
        mov     5,%o3
        call    matmul
        nop
! File = "../ccode/matmul.c", Line = 40
        mov     %g0,%i0
        ret
        restore
.section ".rodata"
.align 8
.local L24_47095
L24_47095:
.word 0x40000000
.word 0
```

B.3.3 Matmul TSpec, Unopt. SPARC with Register Allocation

```
z(BASEz_r, 4, 8, 40);
y(BASEy_r, 4, 8, 40);
x(BASEx_r, 4, 8, 40);
c(BASEc_r, 4);
```

<!c, !x, !y, !z, c*4, ($_{L2}c_5$, c*3,
($_{L6}c_9$, c*3, $x_{+\sim}$, $x_{-\sim+}$, c*4,
($_{L10}c_{17}$, c*5, $z_{+\sim\sim}$, $z_{-\sim\sim}$, c*2, $y_{+\sim\sim}$, $y_{-\sim\sim}$, c*3, z_$w_{+\sim\sim}$, z_$w_{-+\sim}$, c*4)$_{L10}$*5,
c*4, c*3, ^$y_{!!+}$)$_{L6}$*5,
c*4, ^$z_{!!+}$, ^$x_{!!+}$)$_{L2}$*5, c*2>

B.3.4 TSpec, Unopt. SPARC, No Register Allocation

```
px(?_r, 0);
py(?_r, 0);
pz(?_r, 0);
pN(?_r, 0);
li(?_r, 0);
lj(?_r, 0);
l1(?_r, 0);
lj(?_r, 0);
lk(?_r, 0);
l3(?_r, 0);
l4(?_r, 0);
l5(?_r, 0);
l6(?_r, 0);
lr(?_r, 4, 8, 40);
x(?_r, 4, 8. 40);
y(?_r, 4, 8, 40);
z(?_r, 4, 8, 40);
```

< !all, c*2, px_w~, c, py_w~, c, pz_w~, c, pN_w~, c, li_w~, c,
c_{67}, c, li~, c, pN~, c*3, // c67 is where L5 starts
($_{L2}$c, lj_w~, c,
c_{59}, lj~, c, pN~, c*3,
($_{L6}$ c_9, li~, c*2, l1_w~, c, lj~, c*2, l1~, c, px~, c*2, $x_{+\sim}$, x_{-+}, c, lr_w+, lr_w-, c, lk_w~, c,
c51, c, lk~, c, pN~, c*3,
($_{L10}$, c_{21}, li~, c*2, l3_w~, c, lj~, c*2, l4_w~, c, lk~, c*2, l5_w~, c, l5~, c, l3~, c, pz~,
c*3, l6_w~, c, l6~, c, $z_{+\sim}$, $z_{-\sim}$, c, lr+, lr-, c, l5~, c, l4~, c, py~, c*2, $y_{+\sim\sim}$, $y_{-+\sim}$, c*3,
l6~, c, $z_{+\sim}$, $z_{-\sim}$, c, lk~, c*2, lk_w~, c, pN~, c*3)$_{L10}$*5,
c, lj~, c*2, lj_w~, c, lj~, c, pN~, c*3, ^$y_{!!+}$)$_{L6}$*5,
c, li~, c*2, li_w~, c, li~, c, pN~, c*3, ^$z_{!!+}$, ^$x_{!!+}$)$_{L2}$*5,
c*2 >

B.3.5 TSpec, Optimized SPARC Matmul

This TSpec represents the inner kernel code only.

x(?_r, 4, 8, 40);
y(?_r, 4, 8, 40);
z(?_r, 4, 8, 40);
c(?_r, 4);

< !all, c*6, ($_{L2}$c$_{6}$, c*6,
($_{L6}$c*3, x$_{+\sim}$, x$_{-+}$, c*10,
($_{L10}$c, y$_{+\sim}$, y$_{-+}$, c, z$_{+\sim}$, z$_{-\sim}$, c*3, z_w$_{+\sim}$, z_w$_{-+}$, c*4)$_{L10}$*5,
c*4, ^y$_{!!+}$)$_{L6}$*5,
c*4, ^z$_{!!+}$, ^x$_{!!+}$)$_{L2}$*5,
c*2>

B.4 Daxpy

B.4.1 Daxpy C Code Listing

```
(mamba) /af4/daw4q/vpostuff/ccode $ more daxpy.c

#define A 3.0
#define SIZE 5

void daxpy(double x[SIZE], double y[SIZE], int N)
{
  int i;

  for (i=0; i<N; i++) {
    y[i] += A * x[i];
  }

}
/* This main() is simply to illustrate the calling of daxpy() */
main(int argc, char **argv)
{
  int i;
  int j;
  double x[SIZE];
  double y[SIZE];
  for (i=0; i < SIZE; i++)
      x[i] = 2.0;

  daxpy(x, y, SIZE);

}
```

B.4.2 SPARC, Unopt. Assembly with Register Allocation

```
(mamba) /af4/daw4q/vpostuff/sparc $ more daxpy_noptr.s
.section ".data"
.align 8
.K_D0:
.word 0
.word 0
.K_F0:
.word 0
.common __va_first_parm,4,4
.common __builtin_alloca,4,4
.global daxpy
.global main
.section ".text"
.align 8
daxpy:
.type daxpy,#function
! File = "../ccode/daxpy.c", Line = 6
! File = "../ccode/daxpy.c", Line = 9
        mov     %g0,%o4
        cmp     %o4,%o2
        bge     .L000
        nop
.L2:
! File = "../ccode/daxpy.c", Line = 10
        sll     %o4,3,%g1
        add     %g1,%o1,%o3
        ldd     [%o3],%f0
        sethi   %hi(L7_78297),%o5
        ldd     [%o5 + %lo(L7_78297)],%f2
        ldd     [%g1 + %o0],%f4
        fmuld   %f2,%f4,%f2
        faddd   %f0,%f2,%f0
        std     %f0,[%o3]
! File = "../ccode/daxpy.c", Line = 11
! File = "../ccode/daxpy.c", Line = 9
        add     %o4,1,%o4
        cmp     %o4,%o2
        bl      .L2
        nop
.L000:
! File = "../ccode/daxpy.c", Line = 13
        retl
        nop
.align 8
main:
.type main,#function
.l1.1_x = -40
.l1.2_y = -80
        save    %sp,(-176),%sp
! File = "../ccode/daxpy.c", Line = 16
! File = "../ccode/daxpy.c", Line = 22
        mov     %g0,%o2
.L9:
! File = "../ccode/daxpy.c", Line = 23
        sethi   %hi(L14_78297),%o0
        ldd     [%o0 + %lo(L14_78297)],%f0
        sll     %o2,3,%o0
        add     %fp,.l1.1_x,%o1
        std     %f0,[%o0 + %o1]
! File = "../ccode/daxpy.c", Line = 22
        add     %o2,1,%o2
        cmp     %o2,5
        bl      .L9
        nop
! File = "../ccode/daxpy.c", Line = 25
        add     %fp,.l1.1_x,%o0
        add     %fp,.l1.2_y,%o1
        mov     5,%o2
        call    daxpy
        nop
        mov     %g0,%i0
! File = "../ccode/daxpy.c", Line = 27
        ret
        restore
.section ".rodata"
.align 8
.local L14_78297
```

```
L14_78297:
.word 0x40000000
.word 0
.align 8
.local L7_78297
L7_78297:
.word 0x40080000
.word 0
```

B.4.3 TSpec, Daxpy, Unopt. SPARC with Register Allocation

This is the TSpec translation for the daxpy function only.

c(?_r, 4);
y(?_r, 4, 8);
x(?_r, 4, 8);
A(?_r, 4);

< !all, c*4, $(_{L2}c_4$, c*2, $y_{+\sim}$, $y_{-\sim}$, c*2, A+, A-, c, $x_{+\sim}$, x_{-+}, c*2, y_$w_{+\sim}$, y_w_{-+}, c*4$)_{L2}$*5, c*2>

B.4.4 SPARC, Optimized Assembly

```
        mov     5,%o2
        call    daxpy
        nop
        mov     %g0,%i0
! File = "../ccode/daxpy.c", Line = 27
        ret
        restore
.section ".rodata"
.align 8
.local L14_78297
L14_78297:
.word 0x40000000
.word 0
.align 8
.local L7_78297
L7_78297:
.word 0x40080000
.word 0
       sub     %o0,%o1,%g4
        mov     %g0,%g5
        sll     %o2,3,%g6
.L2:
```

```
! File = "../ccode/daxpy.c", Line = 10
        ldd     [%o5 + %lo(L7_78297)],%f2
        ldd     [%g3 + %g4],%f0
        fmuld   %f2,%f0,%f0
        ldd     [%g3],%f2
        faddd   %f2,%f0,%f0
        std     %f0,[%g3]
! File = "../ccode/daxpy.c", Line = 11
! File = "../ccode/daxpy.c", Line = 9
        add     %g5,8,%g5
        cmp     %g5,%g6
        bl      .L2
        add     %g3,8,%g3
.L000:
! File = "../ccode/daxpy.c", Line = 13
        retl
        nop
.align 8
main:
.type main,#function
.l1.1_x = -40
.l1.2_y = -80
        save    %sp,(-176),%sp
! File = "../ccode/daxpy.c", Line = 16
! File = "../ccode/daxpy.c", Line = 22
        add     %fp,.l1.1_x,%o4
        sethi   %hi(L14_78297),%o3
        mov     %o4,%g1
        add     %o4,40,%g2
        ldd     [%o3 + %lo(L14_78297)],%f0
.L9:
! File = "../ccode/daxpy.c", Line = 23
        std     %f0,[%g1]
! File = "../ccode/daxpy.c", Line = 22
        add     %g1,8,%g1
        cmp     %g1,%g2
        bl,a    .L9
        ldd     [%o3 + %lo(L14_78297)],%f0
! File = "../ccode/daxpy.c", Line = 25
        add     %fp,.l1.1_x,%o0
        add     %fp,.l1.2_y,%o1
        mov     5,%o2
        call    daxpy
        mov     %g0,%i0
! File = "../ccode/daxpy.c", Line = 27
        ret
        restore
.section ".rodata"
.align 8
.local L14_78297
L14_78297:
.word 0x40000000
.word 0
.align 8
.local L7_78297
L7_78297:
.word 0x40080000
.word 0
```

B.4.5 TSpec, Daxpy, SPARC Optimized

c(?_r, 4);
y(?_r, 4, 8);
x(?_r, 4, 8);
A(?_r, 4);

< !all, c*8, $(_{L2}c_{8}$, c, A+, A-, c, $x_{+\sim}$, x_{-+}, c*2, $y_{+\sim}$, $y_{-\sim}$, c*2, $y\_w_{+\sim}$, y_w_{-+}, c*4$)_{L2}$*5, c*2>

B.4.6 MIPS, Unoptimized Assembly with Register Allocation

```
^C(mamba) /af4/daw4q/vpostuff/mips $ more daxpy_noptr.s
.globl daxpy
.globl main
.text
.align 8
.ent daxpy
daxpy:
        .frame  $sp,0,$31
 # File = "../ccode/daxpy.c", Line = 6
 # File = "../ccode/daxpy.c", Line = 9
        move    $8,$0
        bge     $8,$6,.L000
.L2:
 # File = "../ccode/daxpy.c", Line = 10
        sll     $7,$8,3
        addu    $9,$7,$5
        l.d     $f0,($9)
        l.d     $f2,L7
        addu    $2,$7,$4
        l.d     $f4,0($2)
        mul.d   $f2,$f2,$f4
        add.d   $f0,$f0,$f2
        s.d     $f0,($9)
 # File = "../ccode/daxpy.c", Line = 11
 # File = "../ccode/daxpy.c", Line = 9
        addu    $8,$8,1
        blt     $8,$6,.L2
.L000:
 # File = "../ccode/daxpy.c", Line = 13
        j       $31
.end daxpy
.align 8
.ent main
main:
        .set    noreorder
        .cpload $25
        .set    reorder
        subu    $sp,$sp,112
        .frame  $sp,112,$31
.l1.1.84_x = 16
.l1.2.84_y = 56
        sw      $31,108($sp)
        .cprestore      104
        .mask   0x90000000,-4
 # File = "../ccode/daxpy.c", Line = 16
 # File = "../ccode/daxpy.c", Line = 22
        move    $4,$0
.L9:
 # File = "../ccode/daxpy.c", Line = 23
        l.d     $f0,L14
        sll     $2,$4,3
        addu    $3,$sp,.l1.1.84_x
        addu    $2,$2,$3
        s.d     $f0,0($2)
 # File = "../ccode/daxpy.c", Line = 22
        addu    $4,$4,1
        li      $3,5
        blt     $4,$3,.L9
 # File = "../ccode/daxpy.c", Line = 25
```

```
        addu    $4,$sp,.l1.1.84_x
        addu    $5,$sp,.l1.2.84_y
        li      $6,5
        la      $25,daxpy
        jal     $25
        move    $2,$0
 # File = "../ccode/daxpy.c", Line = 27
        lw      $31,108($sp)
        addu    $sp,$sp,112
        j       $31
.end main
.data
.align 8
L14:
.word 1073741824
.word 0
.align 8
L7:
.word 1074266112
.word 0
```

B.4.7 TSpec, MIPS Unopt. Daxpy with Register Allocation

c(?_r, 4);
y(?_r, 4, 8);
x(?_r, 4, 8);
A(?_r, 4);

< !all, c*3, $(_{L2}c_3$, c*2, $y_{+\sim}$, $y_{-\sim}$, c, A+, A-, c*2, $x_{+\sim}$, x_{-+}, c*2, $y\_w_{+\sim}$, y_w_{-+}, c*2$)_{L2}$*5, c>

B.5 Livermore Loops

B.5.1 Hydro (k1.c)

B.5.1.1 C Code Listing

```
(mamba) /af4/daw4q/vpostuff/ccode/livermore $ more k1.c
/*
 * Kernel 1 Hydro
 *
 * Translated from Fortran by Sanjay Jinturkar
 * 6/28/92
 */
#define LOOP 50000

#ifndef TIMES
#define TIMES 100
#endif

main()
{
   double x[LOOP], y[LOOP], zx[LOOP+23];
   int i;

   for (i = 0; i < LOOP; i++)
      x[i] = y[i] = zx[i] = 3.0;

   for (i = 0; i < TIMES; i++)
      loop (x, y, zx, LOOP);
}

loop(x, y, zx, n)
double x[], y[], zx[];
int n;
{
  int k;
  double q = 12.0;
  double r = 5.2;
  double t = 13.4;

  for (k = 0; k < n; k++)
     x[k] = q + y[k] * (r * zx[k + 10] + t * zx[k + 11]);
}
```

B.5.1.2 TSpec, k1 kernel, SPARC Optimized Data Accesses Only

zx(?_r, 4, 8);
x(?_r, 4, 8);
y(?_r, 4, 8);

<!all, ^$z_{!10}$, ($_{L19}zx_{+\sim}$. $zx_{+\sim}$, $zx_{+\sim}$, $zx_{!+}$, $y_{+\sim}$, y_{-+}, x_$w_{+\sim}$, x_w_{-+})*>

B.5.2 First Differential (k12.c)

```
(mamba) /af4/daw4q/vpostuff/ccode/livermore $ more k12.c
/*
 * Kernel 12 First Diff.
 *
 * Translated from Fortran by Sanjay Jinturkar
 * 6/28/92
 */
#define LOOP 50000

#ifndef TIMES
#define TIMES 100
#endif

main()
{
   double x[LOOP], y[LOOP];
   int i;

   for (i = 0; i < LOOP; i++)
      x[i] = y[i] = 3.0;

   for (i = 0; i < TIMES; i++)
      loop (x, y, LOOP);
}

loop(x, y, n)
double x[], y[];
int n;
{
    int k;

    for (k = 0; k < n; k++)
       x[k] = y[k + 1] - y[k];
}
```

B.5.2.1 TSpec, k12 kernel, SPARC, Unopt. with Register Allocation

The TSpec below is for the data accesses only.

x(?_w, 4, 8);

y(?_r, 4, 8);

<!all, (y+~, y-,-, y+~, y!+, x_w+~, x_w-+)*>

B.5.3 ICCG, Incomplete Cholesky (k2.c)

```
(mamba) /af4/daw4q/vpostuff/ccode/livermore $ more k2.c
/*
 * Kernel 2 ICCG (incomplete cholesky)
 *
 * Translated from Fortran by Sanjay Jinturkar
 * 6/28/92
 */
#define LOOP 50000

#ifndef TIMES
#define TIMES 4000
#endif

main()
{
   double x[LOOP], v[LOOP];
   int i;

   for (i = 0; i < LOOP; i++)
      x[i] = v[i] = 3.0;

   for (i = 0; i < TIMES; i++)
      loop (x, v, LOOP);
}

/*
 * KERNEL 2
 *
 * Translated from Fortran by Sanjay Jinturkar
 * 6/28/92
 */


loop(x, v, n)
double x[], v[];
int n;
{
   int i, k;
   int ii, ipntp, ipnt;

     ii = 1001;
     ipntp = 0;
L2:  ipnt = ipntp;
     ipntp = ipntp+ii;
     ii = ii/2;
     i = ipntp;
     for (k = ipnt+1; k < ipntp; k+=2) {
       i++;
       x[i] = x[k] - v[k]*x[k-1] - v[k+1]*x[k+1];
       }
     if (ii > 1) goto L2;
}
```

B.5.3.1 TSpec, k2 kernel, Optimized SPARC

The TSpec below is for data accesses only.

v(?_r, 4, 8);
x(?_r, 4, 8, “ii”);

<!all, (($x_{\sim-}$, v-, x+, $v_{\sim+}$, $x_{\sim\sim+}$, $\text{x\_w}_{\sim\sim-}$)*)*>

B.5.4 Inner Product (k3.c)

B.5.4.1 C Code Listing

```
(mamba) /af4/daw4q/vpostuff/ccode/livermore $ more k3.c
/*
 * Kernel 3 Inner Product
 *
 * Translated from Fortran by Sanjay Jinturkar
 * 6/28/92
 */
#define LOOP 50000

#ifndef TIMES
#define TIMES 100
#endif
```

```
main()
{
   double x[LOOP], z[LOOP];
   int i;

   for (i = 0; i < LOOP; i++)
      x[i] = z[i] = 3.0;

   for (i = 0; i < TIMES; i++)
      loop (x, z, LOOP);
}

/*
 * KERNEL 3 INNER PRODUCT
 *
 * Translated from Fortran by Sanjay Jinturkar
 * 6/28/92
 */


loop(x, z, n)
double x[], z[];
int n;
{
   int k;
   double q = 52.3;

   for (k = 0; k < n; k++)
      q  = q + z[k] * x[k] + z[k + 1] * x[k + 1];
}
```

B.5.4.2 TSpec, k3 kernel, Optimized SPARC

x(?_r, 4, 8);
z(?_r, 4, 8);

<!all, (x-, z-, $x_{\sim+}$, $z_{\sim+}$)*>

B.5.5 Tri-Diagonal Elimination (k4.c)

B.5.5.1 C Code Listing

```
(mamba) /af4/daw4q/vpostuff/ccode/livermore $ more k4.c
/*
 * Kernel 4 Tri-Diagonal Elimination
 *
 * Translated from Fortran by Sanjay Jinturkar
 * 6/28/92
 */
#define LOOP 1001

#ifndef TIMES
#define TIMES 100
#endif

main()
{
   double x[LOOP], y[LOOP];
   int i;

   for (i = 0; i < LOOP; i++)
      x[i] = y[i] = 3.0;

   for (i = 0; i < TIMES; i++)
      loop (x, y, LOOP);
}

/*
 * KERNEL 4 BANDED LINEAR EQUATIONS
 *
 * Translated from Fortran by Sanjay Jinturkar
 * 6/28/92
 */


loop(x, y, n)
double x[], y[];
int n;
{
   int i, k, lw, j;
   int m = (1001-7)/2;
   double fw = 1.00000E-25;
   double temp;

   for (i = 0; i < n; i++) {
      for (k = 7; k<1001; k +=m) {
         lw = k-6;
         temp = x[k-1];
         for (j = 5; j < n; j +=5) {
            temp = temp - x[lw]*y[j];
            lw++;
            }
         x[k-1] = y[5] * temp;
         }
      }
}
```

B.5.5.2 TSpec, k4 kernel, Optimized SPARC

x(?_r, 4, 20, m);
y(?_r, 4, 20);

<((!x, $x_{\sim+}$, !y, ($y_{+\sim}$,y_{-+}, x+, x-)*, y5, ^$x_{!+}$, x)*)*>

B.5.6 Tri-diagonal Elimination (k5.c)

B.5.6.1 C Code Listing

```
(mamba) /af4/daw4q/vpostuff/ccode/livermore $ more k5.c
/*
 * Kernel 5 Tri-Diagonal Elimination
 *
 * Translated from Fortran by Sanjay Jinturkar
 * 6/28/92
 */
#define LOOP 50000

#ifndef TIMES
#define TIMES 100
#endif
```

```
main()
{
   double x[LOOP], y[LOOP], z[LOOP];
   int i;

   for (i = 0; i < LOOP; i++)
      x[i] = z[i] = 3.0;

   for (i = 0; i < TIMES; i++)
      loop (x, y, z, LOOP);
}

/*
 * KERNEL 5 TRI-DIAG
 *
 * Translated from Fortran by Sanjay Jinturkar
 * 6/28/92
 */


loop(x, y, z, n)
double x[], y[], z[];
int n;
{
   int i;
   double q = 52.3;

   for (i = 0; i < n; i++)
      x[i] = z[i] * (y[i] - x[i - 1]);
}
```

B.5.6.2 TSpec, k5 kernel, Optimized SPARC

z(?_r, 4, 8);
y(?_r, 4, 8);
x(?_r, 4, 8);

<!all, (x+, x$_{-+}$, y+, y$_{-+}$, z+, z$_{-+}$, x_w+, x_w$_{-+}$)*>

Bibliography

ABR93 S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta, "Predictability of Load/Store Instruction Latencies", In *Proceedings of the Twenty-sixth International Symposium on Microarchitecture (MICRO-26)*, December 1993.

AGA93 A. Agarwal, and S. Pudar. "Column-associative caches: A technique for reducing the miss rate of direct-mapped caches," In *20th Annual International Symposium on Computer Architecture (ISCA'20)*, San Diego California, May 16-19.

AMY99 L. R. Amy, and S. A. McKee, Personal Communications, *CRA Distributed Mentor Project*, August 1999.

BAT76 A. P. Batson and A.W. Madison, "Measurements of Major Locality Phases In Symbolic Reference Strings", In *Proceedings of the International Symposium on Computer Performance, Modeling, Measurement and Evaluation*, March 1976.

BAT77 A. P. Batson, D. W. E. Blatt, and J. P. Kearns, "Structure Within Locality Intervals", In *Proceedings of the Third International Symposium on Modeling and Performance Evaluation of Computer Systems*, 1977.

BEL66 L. A. Belady, “A Study of Replacement Algorithms for a Virtual-Storage computer”, *IBM Systems Journal*, 5(2):78-101, 1966.

BEN88 M. E. Benitez, and J. W. Davidson. “A Portable Global Optimizer and Linker”, In *Proceedings of the SIGPLAN Notices 1988 Symposium on Programming Language Design and Implementation (PLDI)*, Atlanta, GA, June 1988.

BHA97 D. Bhandarkar, and J. Ding, “Performance Characterization of the Pentium® Pro Processor”, In *Proceedings of the Third International Symposium on High Performance Computer Architecture (HPCA-3)*, San Antonio, TX, February 1997.

BRE96 M. Brehob and R. Enbody, “A Mathematical Model of Locality and Caching”, *Michigan State University Computer Science Department Technical Report*, TR-MSU-CPS-96-42, November 1996.

BUR95 D. C. Burger, J. R. Goodman, and A. Kagi, “The Declining Effectiveness of Dynamic Caching for General-Purpose Microprocessors”, *University of Wisconsin-Madison Computer Science Department Technical Report,* 1261, January 1995.

BUR96 D. Burger, J. R. Goodman, A. Kagi, “Quantifying Memory Bandwidth Limitations of Current and Future Microprocessors”, In *Proceedings of the Twenty-third International Symposium on Computer Architecture, (ISCA-23)*, May 1996.

BUR97 D. C. Burger and T. M. Austin. “The SimpleScalar Tool Set, Version 2.0," In *Computer Architecture News*, 25 (3), pp. 13-25, June, 1997.

CAL98 B. Calder, C. Krintz, S. John and T. M. Austin. “Cache-conscious Data Placement.” In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, Oct. 1998.

COR89	T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusets, 1989.
DEN68	P. Denning. "The working set model for program behavior.", *Communications of the ACM* 11:5, May, 1968.
DIT87A	D. R. Ditzel, H. R. McLellan, and A. D. Berenbaum, "Design Tradeoffs to Support the C Programming Language in the CRISP Microprocessor", In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOSII)*, October, 1987.
DIT87A	D. R. Ditzel, H. R. McLellan, and A. D. Berenbaum, "The Hardware Architecture of the CRISP Microprocessor", In *Proceedings of the 14th Annual Symposium on Computer Architecture (ISCA)*, 1987.
FAR94	K. Farkas, and N. Jouppi. "Complexity/performance tradeoffs with non-blocking loads," In *Proceedings of the Twenty-first Annual International Symposium on Computer Architecture (ISCA-21)*, Chicago. April 1994.
FLA92	J. Flanagan, B. Nelson, J. Archibald, K. Grimsrud, "BACH: BYU Address Collection Hardware; The Collection of Complete Traces", In *Proceedings of the Sixth International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, September 1992.
FLA93	J. Flanagan, B. Nelson, J. Archibald, K. Grimsrud, "Incomplete Trace Data and Trace-Driven Simulation: A Case Study", In *Proceedings of the International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, January 1993.
FRA95	C. W. Fraser, D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*, Benjamin Cummings, Redwood City, California, 1995.

GHO98 S. Ghosh, M. Martonosi, and S. Malik, “Precise Miss Analysis for Program Transformations with Caches of Arbitrary Associativity”, In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, October, 1998.

GRI92 K. Grimsrud, J. Archibald, M. Ripley, K. Flanagan, and B. Nelson, “BACH: A Hardware Monitor for Tracing Microprocessor-Based Systems”, In *Technical Report Department of Electrical and Computer Engineering, Brigham Young University*, TR-G150-92.1, October 1992.

GRI93 K. Grimsrud. “Quantifying Locality”, *Brigham Young University Dissertation*, 1993.

GRI96 K. Grimsrud, J. Archibald, R. Frost, and B. Nelson., “Locality as a Visualization Tool”, *IEEE Transactions on Computers*, Vol. 45, No. 11, November 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.

HEN96 J. L. Hennessy, and D. A. Patterson. *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, second edition, San Francisco, California, 1996.

HIL87 M. Hill. “Aspects of Cache Memory and Instruction Buffer Performance”, *Ph.D. Thesis, University of California at Berkeley, Computer Science Division*, Technical Report UCB/CSD 87/381, November 1987.

HUA96 A. Huang and J. Shen.”The Intrinsic Bandwidth Requirements of Ordinary Programs”, In *Proceedings of the Seventh International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOSVII)*, October, 1996.

JAC96 B. Jacob, P. Chen, S. Silverman, T. Mudge. “An Analytical Model for Designing Memory Hierarchies,” *IEEE Transactions on Computers,* vol. 45, no 10, October, 1996.

JOH97 T. L. Johnson, M. C. Merten, and W. Hwu, “Run-time Spatial Locality Detection and Optimization”, In *Proceedings of the Thirtieth International Symposium on Microarchitecture, (MICRO-30)*, December 1997.

JOU90 N. Jouppi. “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers”, In *Proceedings of the Seventeenth Annual International Symposium on Computer Architecture (ISCA-17)*, Seattle, May, 1990.

JOU93 N. Jouppi, “Cache Write Policies and Performance,” In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture (ISCA-20),* May 1993.

JOU97 N. P. Jouppi, and P. Ranganathan, “The Relative Importance of Memory Latency, Bandwidth, and Branch Limits to Performance” In *Twenty-fourth International Symposium on Computer Architecture (ISCA-24) Workshop on Mixing Logic and DRAM: Chips that Compute and Remember,* June, 1997.

KIN00 J. Kin, M. Gupta, and W. H. Mangione-Smith, “Filtering Memory References to Increase Energy Efficiency”, *IEEE Transactions on Computers,* Jan. 2000.

KRO81 D. Kroft, “Lockup-free Instruction Fetch/Prefetch Cache Organization”, *Proceedings of the 8th Annual Symposium on Computer Architecture*, May, 1981.

LAD99 R. E. Ladner, J. D. Fix, and A. LaMarca. “Cache Performance Analysis of Traversals and Random Accesses”, *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1999.

LEB94	A. Lebeck, D. Wood. “Cache profiling and the SPEC benchmarks: A case study”, *IEEE Computer* 27:10, October 1994.
LEE01	H. S. Lee, M. Smelyanskiy, C. J. Newburn, and G. S. Tyson, “Stack Value File: Custom Microarchitecture for the Stack”, In *Proceedings of the Seventh International Symposium on High Performance Computer Architecture (HPCA-7)*, 2001.
LIP68	J. Liptay. “Structural aspects of the System/360 Model 85, Part II: The cache,” *IBM Systems Journal* 7:1, 15-26. 1968.
MAD76	A. W. Madison, and A. P. Batson, “Characteristics of Program Localities”, *CACM*, May 1976, 19:5.
MCKK96	K. McKinley and O. Temam, “A Quantitative Analysis of Loop Nest Locality”, In *Proceedings of the Seventh International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, October 1996.
MCKS97	S. A. McKee, W. A. Wulf, and D. A. B. Weikle, “TSpec: A Specification Language for Reference Traces”, *University of Virginia Department of Computer Science Technical Report* CS-97-19, August 1997.
MCM86	F. H. McMohan, “The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range”, Lawrence Livermore National Laboratory, Livermore, CA, 1986.
MOW92	T. Mowry, S. Lan, and A. Gupta. “Design and evaluation of a compiler algorithm for prefetching,” In *Proceedings of the Fifth Internationa Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, Boston, October 1992.
NAH97	M. D. Nahas, and W. A. Wulf, CS-97-16 “Data Cache Performance When Vector-Like Accesses Bypass the Cache”, *University of Virginia Department of Computer Science Technical Report*, CS-97-16, July, 1997.

PAL94	S. Placharla, and R. Kessler. “Evaluating stream buffers as a secondary cache replacement,” In *Proceedings of the Twenty-first International Symposium on Computer Architecture(ISCA-21),* April, 1994.
PRZ88	S. Przybylsk, and J. Hennessy. “Performance tradeoffs in cache design,” In *Proceedings of the Fifteenth International Symposium on Computer Architecture,* May-June, 1988.
QUI91	R. Quinnell, “High-speed DRAMs”, *Electronic Design News*, May 23, 1991.
RIV98	J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, “Utilizing Reuse Information in Data Cache Management”, In *Proceedings of the International Conference on Supercomputing,* July 1998.
ROT96	E. Rotenberg, S. Bennett, and J. Smith. “Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching”, *IEEE* 1072-4451 1996.
SMI82	A. Smith. “Cache memories,” *Computing Surveys,* 14:3, September, 1982.
SMI83	J. Smith, and J. Goodman. “A Study of Instruction Cache Organizations and Replacement Policies,” In *Proceedings of the Tenth International Symposium on Computer Architecture (ISCA-10),* Stockholm, 1983.
SRI98	S. Srinivasan, and A. Lebeck. “Load Latency Tolerance in Dynamically Scheduled Processors”, *International Symposium on Microarchitecture (MICRO)*, November 1998.
SUG92	R. Sugumar, and S. G. Abraham. “Efficient Simulation of Caches Under Optimal Replacement With Applications to Miss Characterization,” *University of Michigan Department of Computer Science Technical Report*, CSE-TR-143-92, 1992.

TAM99 E. S. Tam, J.A. Rivers, V. Srinivasan, G. S. Tyson, and E. D. Davidson. “Active Management of Data Caches by Exploiting Reuse Information”, *IEEE Transactions on Computers*, vol. 48, no. 11, p. 1244, November 1999.

THI87 D. Thiebaut and H. S. Stone, “Footprints in the Cache,” *ACM Transactions on Computer Systems*, vol. 5 no. 4, Nov. 1987.

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.

TYS95 G. Tyson, M. Farrens, J. Farrens, A. Pleszkun. “A Modified Approach to Data Cache Management,” In *Proceedings of the Twenty-eighth International Symposium on Microarchitecture (MICRO)*, Nov.-Dec., 1995.

VOL81 J. Voldman, and L. Hoevel. “The Software-Cache Connection”, *IBM Journal of Research and Development*, Vol. 25, No. 6, November 1981.

WAN01 Hua Wang, and S. A. McKee, “Personal Communications”, January 2001.

WEI00A D. A. B. Weikle, K. Skadron, S. McKee, W. A. Wulf, “TSpec: A Notation for Describing Memory Reference Traces”, *University of Virginia Department of Computer Science Technical Report*, CS-2000-23, August 2000.

WEI00B D. A. B. Weikle, S. A. McKee, K. Skadron, and W. A. Wulf, “Caches As Filters: A Framework for the Analysis of Caching Systems”, In *Proceedings of the Grace Murray Hopper Conference 2000*, Sept. 14-16, 2000.

WEI98 D. A. B. Weikle, S. A. McKee, W. A. Wulf, “Caches As Filters: A New Approach to Cache Analysis”, In *Sixth International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunciation Systems (MASCOTS98)*, July 19-24, 1998.

WIL65 M. Wilkes. “Slave Memories and Dynamic Storage Allocation,” *IEEE Transactions on Electronic Computers* EC-14:2, April, 1965.

WUL94 W. A. Wulf, and S. A. McKee, “Hitting the Memory Wall: Implications of the Obvious”, *ACM Computer Architecture News*, Vol. 23, No. 4, September 1995.