Chapter 4

Implementation Issues

Chapters 2 and 3 have argued for the collection of detailed, data and code oriented statistics on program behavior. In this chapter, we discuss implementation issues in gathering such detailed statistics and describe the specific solutions used by MemSpy.

In Section 4.1, we discuss MemSpy's simulation-based mechanism for collecting the data required to generate its statistics. Next, in Section 4.2 we discuss issues in implementing the particular data and code oriented statistics provided by MemSpy. This section presents techniques for identifying code and data objects to monitor and heuristics for aggregating statistics on data structures that are likely to be used similarly. In addition, it presents our implementations of statistics on the causes of cache misses and the causes of cache replacements.

Finally, although the discussions in these two sections are grounded in a specific, simulation-based approach, we note that several other data collection methods, such as hardware monitoring and software instrumentation, could also be used to gather the information required by MemSpy. Section 4.3 discusses these other options and explains the rationale behind MemSpy's data gathering approach.

4.1 MemSpy Data Collection

To present the performance information discussed in Chapters 2 and 3, MemSpy must monitor programs at the granularity of individual memory references in the code. It must also maintain statistics separately for different data structures and code segments in the program. Information at this granularity can be difficult to obtain. In the past, program performance monitoring has relied on one of three methods for collecting program data: (i) hardware monitoring support, (ii) software instrumentation, or (iii) software simulation.

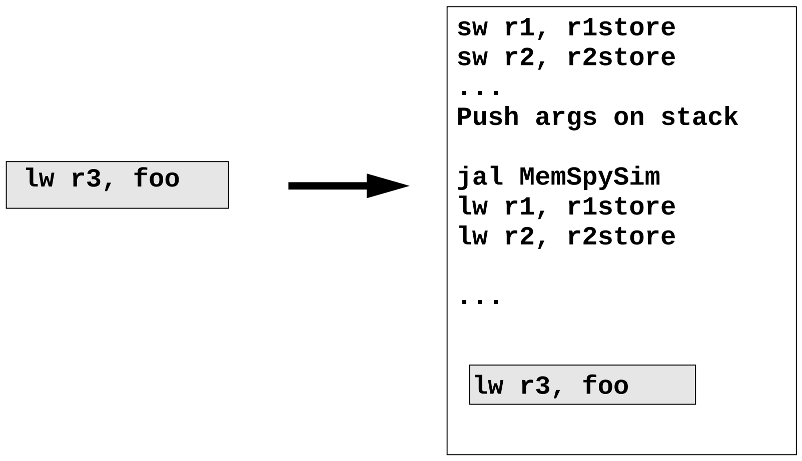
Of these three possibilities, we have implemented MemSpy using a simulation-based method. The main advantage of a simulation-based approach is that it can be very general and portable, since it can be implemented with no specialized hardware support. Unfortunately, it can be difficult to build accurate simulators, and the high execution time overheads of accurate, detailed simulators were previously considered a fundamental limitation to implementing tools in this way. Through the optimizations proposed in Chapters 5 and 6, this dissertation establishes that simulation-based tools, such as MemSpy, can run at speeds that make them competitive with other, less detailed approaches. This section first describes the Tango Lite simulation system on top of which MemSpy is implemented. It then describes the specific program events instrumented for processing by MemSpy's simulator.

4.1.1 Simulator Implementation

MemSpy is built on top of the Tango Lite reference generator [Gol93]. Tango Lite is a direct execution simulator which simulates the execution of both multiprocessor and uniprocessor machines on uniprocessor workstations.[1] In a direct execution simulation, "interesting" events are instrumented at compile-time with additional code to call event simulators. For MemSpy, the four types of events instrumented are: (i) memory references, (ii) procedure calls and returns, (iii) memory allocations, and (iv) synchronizations. Section 4.2 will describe the rationale for instrumenting these events and descriptions of the actions taken on each event. When the code is compiled and run, all uninstrumented events execute directly, at full speed, on the host machine. The progression of time is simulated by clock increments added to basic blocks, as well as clock increments performed by the event simulators.

[1]Tango Lite can also execute on multiprocessors; however MemSpy does not use this feature.

To make this discussion more concrete, Figure 4.1 shows the instrumentation added to a program to simulate a particular event, in this case a memory reference. (Other simulated events are similarly instrumented.) At run time, when the application reaches an instrumented event, it first saves its registers so that the memory simulator will not destroy their values. Next, it calls the MemSpy simulator with arguments indicating the type of event (load, store, procedure call, etc.) and the referenced address. The simulator internally maintains information on the state of the simulated memory hierarchy, as well as the profile information required to report MemSpy's statistics. When the simulator has finished processing the reference, the application registers are restored to their original values, and control is returned to the application.


```
lw r3, foo
```
```
sw r1, r1store
sw r2, r2store
...
Push args on stack

jal MemSpySim
lw r1, r1store
lw r2, r2store

...


lw r3, foo
```

Figure 4.1: MemSpy memory reference instrumentation.

4.2 Data and Code Oriented Statistics

With MemSpy's approach, overall program information can be subdivided into statistics on arbitrary intersections of code and data structures. The different subdivisions and combinations available correspond to different uses of the *hierarchies of focus* discussed in Chapter 2.

One significant issue in implementing data and code oriented statistics is determining a natural granularity at which to present statistics. Choosing the right granularity in both data and code oriented statistics is important because statistics that are too coarse-grained may not localize bottlenecks well enough. On the other hand, statistics that are too fine-grained may not aggregate activity to the point where clear bottlenecks stand out.

Moreover, statistics which are too fine-grained may also be inefficient to implement in terms of (i) storage inefficiency, since more memory will be required to maintain very fine-grained statistics, and (ii) execution time inefficiency, since extra time will be devoted to managing and updating the larger number of statistics variables. Sections 4.2.1 and 4.2.2 discuss issues in collecting statistics at appropriate granularities.

Beyond the initial choice of granularity, the tool needs efficient data structures to organize the statistics. Section 4.2.3 introduces our notion of *statistics bins* and their management. The tool also needs methods for mapping particular code locations or referenced addresses back to the appropriate statistics bin. Sections 4.2.4 and 4.2.5 cover these mapping methods. We also note that significant naming issues arise when implementing data and code oriented statistics. Section 4.2.6 discusses the issues that arise in assigning meaningful names to the statistics bins produced by MemSpy. Finally, Sections 4.2.7 and 4.2.8 discuss implementation issues in maintaining statistics on causes of misses and causes of cache replacements. Our approach in each of these eight sections is to first outline the general issues, and then describe the specific approach taken by MemSpy. Section 4.2.9 is a discussion of other related issues not yet touched on, and Section 4.2.10 is a summary of this section.

4.2.1 Code Oriented Statistics

We first tackle the issue of MemSpy's development of code oriented statistics. Tool implementors can choose from a wide range of natural granularities when implementing code oriented statistics. These include statistics per procedure, per loop nest, per basic block, or even per source code line.

Dividing code oriented statistics at too coarse a granularity can make it difficult for the user to pinpoint bottlenecks. For example per-procedure statistics may be inappropriate in some programs where a single procedure may contain multiple computational phases, each with different memory behavior.

For some scientific programs a natural granularity might be to provide individual statistics for each loop nest. However, in non-scientific code, loops are often smaller and generate fewer references. In these cases, loop oriented statistics may be too fine-grained. In general, tool designers may want to give users a choice between a default, procedure-oriented approach, and a finer-granularity basic block or loop oriented approach.

MemSpy Approach

As shown in the case studies in Chapter 3, MemSpy separates its code oriented statistics by procedures. Since the MemSpy simulator logs procedure entries and exits, it is straightforward to determine which procedure the simulated process is currently executing. From this, it can keep a log of the current state of the procedure call stack. In this way, the current procedure is always known, and can be used to select the appropriate procedure with which to associate statistics. Procedures are typically coarse-grained enough that logging entries and exits is not prohibitively expensive, as finer-grained monitoring might be.

A procedure oriented method was chosen as a moderate tradeoff between the granularity of statistics, and the overhead of gathering them. In most cases, procedure-oriented statistics have been fine-grained enough to localize performance bugs. In one case (Vrender from Appendix A), however, the program was intentionally very non-modular, to improve the performance of the code's inner loop by removing procedure calls. Here, more fine-grained statistics, perhaps on a basic block granularity, may have been more useful. MemSpy could have implemented per-basic-block statistics by logging entries and exits at the basic block, rather than procedure, granularity. However, this more fine-grained logging would increase both (i) the execution time overhead of the tool and (ii) the storage overhead required to keep the more fine-grained statistics.

4.2.2 Data Oriented Statistics

As with code oriented statistics, an important issue in data oriented statistics is collecting and presenting them at appropriate granularities. Once again, information at too fine or coarse a granularity can make it difficult to identify bottlenecks, and from an efficiency standpoint, fine-grained statistics can also lead to higher execution time and storage overheads.

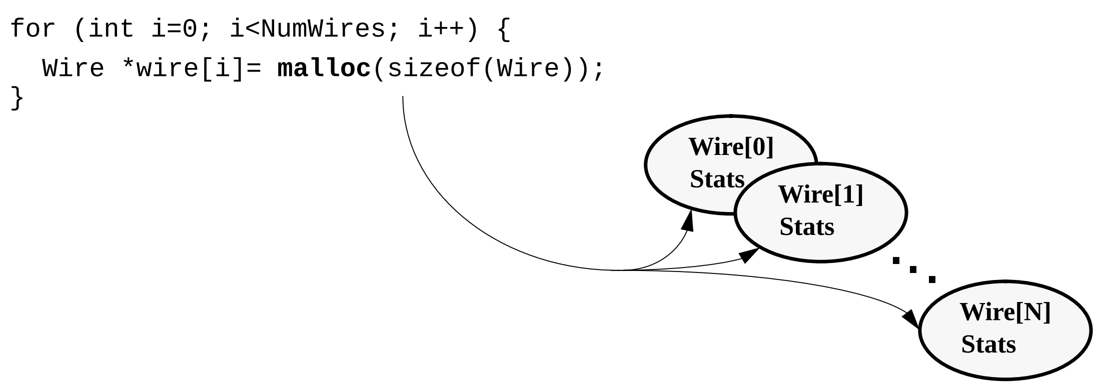
Beyond this, an implementation of data oriented statistics must also consider the different classifications of program data and how to treat each of them. That is, in sequential programs, data is either (i) stack data local to a particular procedure, (ii) static global data, or (iii) data allocated dynamically from the heap. This section will describe granularity issues from the context of heap allocated memory. Section 4.2.9 will discuss extensions to include static and stack data.

An initial attempt at producing data oriented statistics might be to provide separate statistics for *each separately allocated memory range*.

However, this technique of considering each individual memory range to be a separate statistical unit often results in cases where there are many bins with very similar behavior. For example, LocusRoute (a CAD wire routing program from the SPLASH benchmarks) allocates storage for hundreds of wires. Figure 4.2 illustrates this case. Since most of the wires are expected to have similar memory behavior, keeping separate statistics on each wire is not as useful as aggregating statistics for all wires. First, programmers often think of the wires as a group, rather than separately considering individual wires. Second, an individual wire is not likely to be accessed often enough to be a bottleneck by itself. The entire class of wires, however, when viewed together, may have cache miss behavior with characteristics that indicate a memory performance bug.

To automatically aggregate statistics for all wires, one might use an approach which groups into a single data bin *all memory ranges allocated at the same point in the source code*. For example all the allocations in Figure 4.2 would be grouped together, since they all occur on the same source code line, although in different loop iterations.

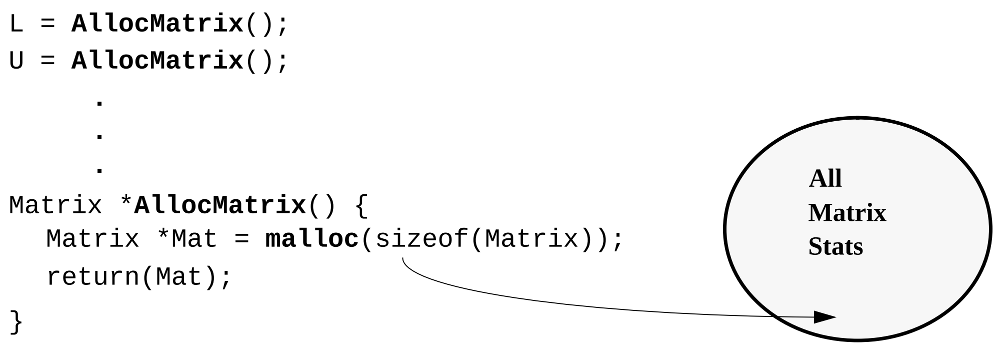
The above strategy can, however, result in too many data objects being aggregated together into a single data bin. This is exemplified in Figure 4.3 by a benchmark LU decomposition program. The program's main data structures are two matrices which are



```
for (int i=0; i<NumWires; i++) {
  Wire *wire[i]= malloc(sizeof(Wire));
}
```

Figure 4.2: Maintaining one bin per memory range in LocusRoute.

allocated at exactly the same point in the source code, within the `AllocMatrix` routine. Here, the programmer would like to view separate statistics for each matrix, since their memory behavior is quite different. However, this technique would merge their statistics together because they use the same static allocation point.



```
L = AllocMatrix();
U = AllocMatrix();
      .
      .
      .
Matrix *AllocMatrix() {
  Matrix *Mat = malloc(sizeof(Matrix));
  return(Mat);
}
```

Figure 4.3: Maintaining one bin per source code line in LU Decomposition.

MemSpy Approach

Because of cases like the ones outlined above, MemSpy chooses a hybrid approach. It aggregates statistics for *all memory ranges allocated at the same point in the source code with identical dynamic procedure call paths*. When a heap allocation occurs, the current source code position and stack are noted. If the current program counter and all the program counters on the stack identically match that for a previously initialized bin, the statistics for this new range of memory are kept in that bin.

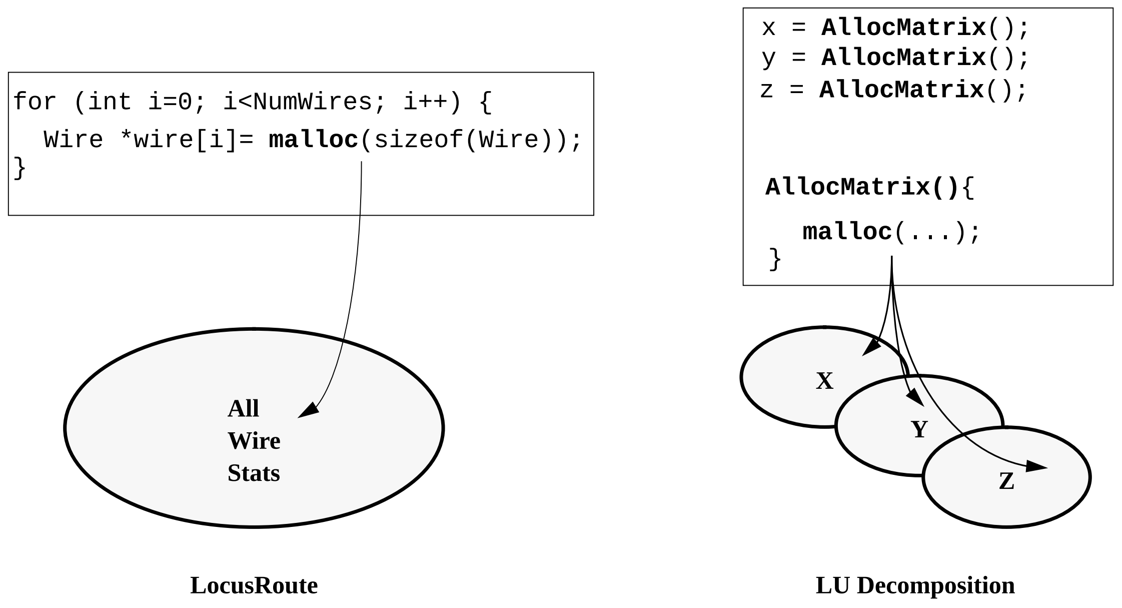
The rationale for this heuristic is that, in our experience, data objects allocated at the same point in the source code via the same call path are usually similar in memory behavior. When memory is allocated in separate calls to a procedure from different call paths, it is monitored in separate bins.[2] This final approach is illustrated in Figure 4.4. Note how LocusRoute's wires, all allocated along a single call path, are aggregated. However the matrices in LU, allocated along two different call paths, are kept separate.

Overall, this approach has worked well in our experience. However, this approach runs into difficulty in cases where the program manages its memory by allocating a large chunk of memory from the heap, and then allocating and freeing portions of that chunk within the program. In these cases, MemSpy's automatic binning will only keep statistics on the initial chunk of memory from the heap, not on the subunits that the program handed out as individual data structures. Section 4.2.9 discusses an extension to this technique that allows MemSpy to differentiate statistics for portions of memory allocated from programmer-managed heaps and free lists.

4.2.3 Organizing Statistics into Bins

The previous subsections discussed how to obtain appropriate granularities for statistics collection and presentation. We now discuss how those statistics are organized and managed in terms of entities called *statistics bins*. A statistics bin is a bucket containing program information (such as memory time, number of cache misses, etc.) collected either for a particular data or code section, or for a pairing of data and code sections in

[2] The exact method used for tracking the call path is similar to that used by Zorn and Hilfinger in their memory allocation profiler, *mprof* [ZH88].



```
for (int i=0; i<NumWires; i++) {
  Wire *wire[i]= malloc(sizeof(Wire));
}
```

```
x = AllocMatrix();
y = AllocMatrix();
z = AllocMatrix();


AllocMatrix(){
   malloc(...);
}
```

Figure 4.4: MemSpy approach to data binning in LocusRoute and LU Decomposition.

the application. That is, a code statistics bin holds information gathered for particular code segments, such as procedures, in the program. A data statistics bin (or data bin for short) is a bucket of statistics about a particular data aggregation. Finally, statistics bins for pairings of code and data allow users to view the behavior of a particular data structure in a particular procedure.

MemSpy Approach

As previously described, MemSpy subdivides the "code axis" of a program by procedures. In addition, it uses the data aggregation heuristic described in the previous subsection to divide the "data axis". Thus, statistics bins are created corresponding to (i) each procedure in the code, (ii) each data aggregation, and (iii) each pairing of a data aggregations with a procedure in which it is referenced.

For each statistics bin, the tool allocates a structure which holds the current counts for the statistics being monitored. These include counts of read latency, write latency, and synchronization latency. In addition, the tool counts the number of misses due to first references, invalidations, and replacements. Information on the causes of replacements is stored in an array with elements for each data bin. Finally, the tool monitors the total time attributed to the bin; this is used for per-procedure statistics that account for computation time.

In addition, the synchronization events in parallel programs, such as `locks` and `barriers`, are also instrumented, and information is collected on the amount of synchronization latency in the program. MemSpy currently offers code oriented, but not data oriented, statistics on synchronization latencies. However, our experiences thus far have indicated that presenting data oriented statistics on individual synchronization variables would be a useful and natural addition to the tool. Such statistics would allow the tool to isolate the effects of individual synchronization variables in the code.

Finally, each statistics bin structure is linked into a two dimensional hash table. The hash table contains a procedure tag, a data tag, a statistics pointer that points to the bin statistics themselves, and pointers to the next procedure bin and the next data bin in the table.

An important implementation issue to consider is storage overhead. In total, each statistics bin incurs an overhead of roughly $52 + (4 * \texttt{NumDataBins})$ bytes. Consider for example the blocked matrix multiplication code from Chapter 3, which has 3 main data structures and only a handful of procedures. The application has a total of 7 data bins, and 22 procedure-data pairs. The total memory required for statistics bins is thus only 1760 bytes. Programs with more different types of fine-grained memory allocation, and allocation in more different procedures have more bins. For example LocusRoute, discussed in Appendix A, has 49 different data bins and 458 procedure-data pairs. This requires 114KB of storage. An uninstrumented run of LocusRoute on the same data set has a total swap usage of roughly 16MB, so even here the statistics bin storage is less than 1% of the total swap memory usage.

4.2.4 Mapping Current Procedure to a Statistics Bin

To implement data and code oriented statistics, the tool must be able to map the current reference and code location back to source level structures. This requires first mapping the current code location back to the procedure it is a part of. Second one must map the referenced address back to the data structure it is contained in. This section discusses the code mapping, while the following section discusses the data mapping.

MemSpy Approach

To support the code mapping, MemSpy maintains a procedure stack. At each instrumented procedure call, MemSpy pushes the new procedure identifier onto its internal stack. It also updates the pointer for the current procedure statistics bin. On procedure returns, the elapsed time statistics for this procedure are updated, the top procedure identifier is popped off the stack and the current procedure statistics pointer is returned to its previous value. In the case of directly recursive procedures, MemSpy does not add a new item to the stack. Rather, it increments a counter indicating the recursion depth. Combining this approach with the data bin mapping described below, MemSpy can generate statistics on pairings of code and data.

4.2.5 Mapping Current Data Address to a Statistics Bin

In comparison to the procedure mapping, implementing the data bin mapping is slightly more complex. This is because maintaining mappings between ranges of memory and their corresponding data bins requires one to know the size and starting positions of all memory allocated by the application.

MemSpy Approach

To support the data mapping, MemSpy logs all memory allocations from the heap, and records which memory ranges correspond to which program variables. We instrument the code to log: (i) the pointer returned by the `malloc` routine, (ii) the size of the allocated block of memory, and (iii) the name of the variable to which the `malloc` return value is assigned.

Recall that statistics bins are maintained for groups of memory ranges allocated at the same static source code location, via the same dynamic call path. Thus, at a memory allocation, MemSpy compares the current source code location and call path to those for all the previously defined data bins. If a match is found, the allocated memory range is considered to be part of that bin. If no match is found, a new bin is created.

A set of arrays of pointers is used to store the information on which memory ranges correspond to each statistics bin. The arrays are managed as a hierarchy, which facilitates fast searches. At the base of this hierarchy is an array with 128 fields corresponding to the different values for the top 7 bits of a data address. For each element, there is a `bin_id` field and a pointer. If the entire memory range corresponds to only a single bin, the `bin_id` is used to indicate which bin that is, and the pointer is NULL. Otherwise, the pointer is non-NULL, indicating that this memory range is further subdivided. In these cases, the pointer is offset with the next 7 bits of the referenced address, to locate a new element which similarly has a `bin_id` and a pointer. Subsequent levels of the hierarchy subdivide using the next 7, 4, 4, and 3 bits respectively. (These subdivisions were chosen empirically to represent a moderate tradeoff between storage overhead and lookup time.) Thus, in six steps or less, this process reaches an element with a NULL pointer, and at this point, the data bin for the memory range is given by the `bin_id` field.

At each memory reference, MemSpy first uses its particular cache model to determine if the reference is a hit or miss. It then searches the bin hierarchy to determine the statistics bin for the currently referenced address. Using the `bin_id` from the structure as an index into a hash table of statistics pointers, it locates the appropriate statistics bin and increments fields depending on whether the reference was a read or write, a hit or miss, and so forth.

A final issue to consider is MemSpy's handling of memory deallocations. Currently, MemSpy does not remove the existing bin mapping for a range of memory when it is deallocated. Rather, it assumes that the next use of the same memory range will be preceded by another memory allocation. On this allocation, all the previous data bin information for that range of memory will be superseded by the newly defined information.

To offer more support for identifying correctness, as well as performance, bugs, MemSpy could monitor programs for accesses to unallocated memory. It could treat memory deallocations as events which redefine the memory range as an “invalid” bin, and references to invalid bins could then be flagged as likely correctness bugs, since references to deallocated memory should not occur.

The storage overhead for this data bin mapping is comprised mainly of the hierarchical array. In this, each array element contains 16 bytes, which holds information on the start and end addresses for the memory range, the bin number (if any) that it points to, and a pointer to a sub-array (if any).

The base array of 128 elements is allocated when the simulator is initialized. This breaks down the memory space according to the top 7 bits of the referenced address. When the second and third levels are used, the sub-arrays also have 128 entries. At the fourth and fifth levels, the sub arrays have 16 entries, and the final level has 8 subdivisions.

With this approach, all simulations have a base overhead of at least 2048 ($= 128 * 16$) bytes for the base array. The subsequent storage overhead required depends heavily on the number of memory ranges allocated, and the size and positioning of each range. In general, for memory allocated in larger chunks and aligned on larger power-of-two boundaries, fewer subarrays need be instantiated.

As a numerical example, consider a program allocating 3 matrices, each of which requires 256KB of storage. For this case, the bin mapping translation will require the following storage. First, a base array of 128 elements will be allocated. In addition, at least three sub-arrays, each also of 128 elements will be allocated since each of the three matrices maps into one second level region. If each matrix maps exactly into a second level region, then no regions need be allocated in the third or subsequent levels. In total, the storage overhead would be 4*128*16 bytes, or 8192 bytes total. This is roughly 1% storage overhead compared to the 3 matrices.

When programs do finer grained allocation, more sublevels will be required to distinguish between smaller ranges of memory, and this will increase the storage required.

However, common `malloc` implementations match well with this hierarchical scheme. In several memory allocation implementations, the finer-grained memory tends to be grouped together, saving larger ranges of memory for large `malloc` requests. These schemes mesh well with our bin mapping scheme; once one subdivision of the mapping table is instantiated down to the lower levels, it is likely to be used for other fine-grained allocations as well.

4.2.6 Naming Issues in Data and Code Oriented Statistics

For a programmer to make use of the tool output, it is important that the tool *automatically* provide intuitive and unique labels, or names, for the statistics bins. Ideally we want to use symbolic variable names from the source program since these are mnemonic for the programmer. In addition, the names must also be unique, so that the programmer can easily identify which data or code object a particular bottleneck pertains to.

In code oriented statistics, naming is fairly straightforward; a tool can generally append a source file name with a procedure name or line number to create a unique name that is easy to locate in the source code. Note that in some cases, however, procedure name aliasing can occur, such that multiple names refer to the same procedure. For example in C language programming, aliasing occurs through the use of function pointers. This could lead to situations where the procedure name selected by the tool was in fact not the most commonly used name for that particular procedure. In general however, procedure aliasing is not common, and the procedure name itself is almost always the natural way to refer to the code region, as opposed to function pointer names that may occasionally be used to call the procedure.

With data oriented statistics, one also would like to associate a unique, intuitive name with each set of memory ranges on which statistics are gathered. This is complicated, however, by two factors. First, it is possible, and sometimes even common, to have multiple names that refer to the same region in memory. For example, aliasing can occur when more than one variable is used as a pointer into a particular data structure. Aliasing can also occur when pointers to data structures are passed as procedure arguments. The difficulty with aliasing is that the tool may choose one name to refer to a statistics bin, when actually one of the other program names used to refer to that memory is more intuitive.

The second complication stems from the converse situation in which memory ranges in multiple, different statistics bins are referred to by the same variable name. This can occur due to the heuristics used for aggregating statistics from several memory ranges, and makes it difficult to guarantee a unique name. Thus, choosing intuitive and unique names becomes quite a significant issue in implementing data oriented statistics.

MemSpy Approach

To provide data and code oriented statistics labels, MemSpy uses the following approaches. First, to refer to code oriented statistics, MemSpy simply uses the static name of the procedure from the source code. More elaborate file label information would be easy to add, but this basic approach has proven fairly satisfactory for the applications we have studied.

The data naming issue is slightly less straightforward, because as mentioned, issues related to both data aggregation and aliasing come into play. For each static appearance of a memory allocation in the code, we name the associated statistics bin with a string that concatenates the *data type* and *variable name* of the pointer which receives the return value from the memory allocation. However, as described in Section 4.2.2, multiple data bins are created for the same allocation point if the allocation is encountered through different procedure call paths. If multiple bins are created based on dynamic call paths, then to be unique their names must be distinguished based on the call path as well. We disambiguate the bin names by prepending a string summarizing the state of the call stack at the allocation point. The final full name is of the form:

```
"ProcName.return_pc.ProcName.return_pc...   .DataType.VarName"
```

By prepending the bin name with call stack information, we guarantee a unique name for each bin. However, in our experience with MemSpy, we have found that the simple short version of the name (`VarName`) is usually unique and sufficiently intuitive for the programmer. For this reason MemSpy displays the short name as a default, and users can request MemSpy bring up the full, unique name when needed.

This approach satisfies the goal of uniqueness, but can occasionally run into difficulties due to aliasing. The weakness arises when the allocated memory is assigned to a temporary variable and then later assigned to a more “significant” variable in the program. For example, a matrix allocation routine may assign the allocated memory to a variable called `tmp` and then pass `tmp` back to a caller procedure where it is assigned to a more intuitively named variable. In these cases, the data bin will receive the name `tmp` rather than the preferred name. Using the call stack information, however, the programmer can generally determine which bin it is. Furthermore in cases like this, users can also interactively rename bins to a new unique name of their choosing.

Overall, this strategy has been effective in providing succinct useful names most of the time, while also guaranteeing uniqueness and flexibility in naming. The case studies in Chapter 3 all show bins and names that were automatically generated by MemSpy. In the Vrender case study discussed Appendix A, however, MemSpy’s automatic naming did run into difficulties. The Vrender code allocates much of its data through its own set of allocation macros. As such, many of the bins in the program are named after the handful of temporary variables appearing in these macros. In addition, the code is highly non-modular, so the call path information appended to the statistics bin is similar for most of the bins, and offers little help in distinguishing them. For this application, the programmers revised their allocation macros to accept an additional argument, the desired bin name. They then manually added a call to MemSpy’s data binning procedure within their allocation routines. These revisions required very few code changes, and allowed MemSpy to be easily used with the application.

4.2.7 Statistics on Causes of Cache Misses

Another important feature discussed in Chapter 2 are statistics on the causes of application misses. To provide this data, the tool needs to track information on whether a memory line has been referenced, and if so, why it most recently left the cache.

Cache misses are caused by one of the following: (i) the line has never been referenced before by this processor, (ii) the line has been *replaced* out of the cache since its last reference, or (iii) the line has been *invalidated* since its last reference. Thus to distinguish between these three cases, at least two bits of state information are required for each memory line in use by each processor. Although a minimum of two bits are required, the implementation can actually choose to use more memory and store them in a larger data type like a `char` or `short`, in order to avoid performing extra calculations to locate the appropriate bit fields.

An interesting issue in logging the cause of a cache miss is in defining which of potentially several cache events is truly the cause of the miss. That is, in the time between when a line leaves the cache and when it is re-referenced, several events may occur which would have caused replacements or invalidations to the line if it were still resident in the cache. If programmers are informed that a line left the cache due to a replacement, they may restructure the program to eliminate that particular replacement, only to find that a different replacement, or perhaps an invalidation, occurs instead.

MemSpy Approach

MemSpy considers the cause of a miss to be the initial event that forced the data out of the cache. Further intervening events that might also have caused replacements or invalidations are not noted. An interesting addition to tools like MemSpy would be support for monitoring the specific reference and caching history, over time, of particular cache lines or data items. By allowing users to request time lines of activity on particular data structures or cache lines, MemSpy could incorporate some of the useful aspects of animation based tools like SHMAP [DBKF90], along with its current approach based primarily on numerical and graphical summaries of activity.

To store the state information for the causes of misses, MemSpy defines a “sparse” array that is indexed by the referenced address. This array is similar in structure to the multilevel array described in Section 4.2.5. That is, to access items from the array, the referenced address is subdivided into 3 fields of bits, and each of these fields is used to index one level of the hierarchical structure. At the leaves of this structure, for each address that has been referenced, the array contains the state bits indicating the cause for the most recent time it left the cache.

4.2.8 Statistics on Causes of Cache Replacements

Finally, understanding problems of cache interference often requires explicit statistics on the causes of cache replacements for each data structure. These statistics are provided so that users, for each data structure, can determine the data structures that contributed to its replacement from the cache.

Statistics on causes of cache replacements are clearly very closely tied to statistics on the causes of misses. As such, they also require that the tool implementor define which of potentially several intervening references caused the cache replacement responsible for a particular cache miss.

MemSpy Approach

To efficiently report information on the causes of replacements for particular statistics bins, we store the bin identifiers of each item currently in the cache. When an item is brought into the cache, a bin lookup must be performed to update the miss statistics. When this item is subsequently pushed out of the cache, the tool needs the bin identifier again to update the replacement statistics. We perform the search when the item is brought into the cache and store that bin identifier along with the cache line. Thus, when the line is replaced out of the cache, we do not need to repeat the bin search. This requires a storage overhead of an extra `short` per cache line, but can significantly accelerate handling replacements, by omitting an extra bin search.

4.2.9 Discussion

Overall, we have found our techniques for generating data and code oriented statistics to be quite effective in practice. This method has been used successfully on a variety of programs, including the SPLASH benchmarks. However, there will still be cases in which the user would like finer divisions of statistics, or even some manual control over the division of data. This section first discusses methods for defining bins at points other than standard memory allocation routines and then describes issues in defining bins for static and stack data.

User Defined Allocation Routines

Currently, the user can choose to use MemSpy's default binning, or to turn off automatic data binning and instead use MemSpy procedure calls to manually compose bins. We are interested in extending beyond these two schemes, however, to give the user more flexibility in directing how data is grouped into bins. One such extension would be to allow programmers to define a list of additional "allocation" and "deallocation" routines, in addition to the default routines currently recognized. (Currently the recognized memory allocation and deallocation routines are `malloc` and `free` calls. In addition, since the SPLASH benchmarks are written using using the Argonne National Laboratory's (ANL) parallel programming macros [LO+87], MemSpy recognizes the ANL shared memory allocation and deallocation macros, `G_MALLOC` and `G_FREE`.) Providing an avenue for defining other allocation routines would allow the tool to consider program-specific allocation routines in addition to the standard routines when performing the data binning.

With this extension MemSpy could keep more useful statistics in instances when programmers maintain their own free lists and heap allocation pools. Currently data allocated from a single `malloc` cannot be automatically subdivided, for example if it is used for elements on a free list. Programmers must explicitly call mapping routines for each element as it is pulled off the free list and initialized. With the proposed extension, data objects allocated from the same pool of `malloc`ed memory, but distributed to the program through several different free lists, could have separate statistics. This would further expand the range of programs for which MemSpy's automatic binning is sufficient and manual intervention is not required.

Automatic Binning for Static and Stack Data

This section has couched its discussion in terms of heap allocated data objects. The current MemSpy implementation does not automatically decompose static and stack data into bins. Instead it presents an overall statistics bin for each of these types of data. In part this is because in the applications we have studied thus far, the main program variables are allocated from the heap. In fact, in our benchmark set, between 63% and

99% of references are to heap data. This section discusses some of the issues that arise when extending MemSpy's binning implementation to static and stack data.

Maintaining individual bins for *static* data could be accomplished by prepending the application with code that sets up bins for all static data before the application begins to run. By scanning the symbol table, MemSpy could automatically determine the sizes and names of data structures for which to define bins. Since these bin definitions would occur only once per program run, their execution time overhead is not likely to be significant.

To maintain bins for stack allocated variables, one would need to instrument each procedure entry point with additional code to define bins for the local variables for the duration of the procedure. This would require additional support to note the size and name of local procedure variables. We expect that MemSpy's binning heuristic (aggregating data into bins based on the call path at the memory allocation point) should apply as well to stack allocated data as it does to heap allocated data. (In the case of stack data the data allocation point would be considered to be the procedure invocation at which the local name becomes active.) That is, in general one might expect local procedure variables to have similar behavior for call activations reached through identical call paths.

As in the previous subsections, it is important to gather statistics on static and stack variables at an appropriate granularity, because the statistics granularities have important implications on both the execution time and storage overheads of the implementation. These are discussed in the following paragraphs.

An important issue for binning stack data is the execution time overhead of redefining stack data bins on each procedure call. These redefinitions must occur on each procedure call because in general the local stack variables will have an arbitrary position in memory on each call. Their memory address depends on the state of the stack when the call was made. For some procedures, the overhead of defining these bins could equal or exceed the time spent actually executing the procedure. Thus, tools may need a method for limiting the number of stack bins defined. For very simple procedures, static analysis could calculate the expected runtime of the procedure. This could be used to decide whether to instrument it or not. Most procedures, however, will have loops or conditionals which make it impossible to make accurate static estimates of runtime. In these cases profiling or dynamic checks could also be used to control when the bin definition instrumentation is executed.

Beyond this, since many static and stack variables may be small, make few references, and have little effect on program performance, maintaining separate bins for each of these could also lead to large storage overheads. In addition, it can increase the search time required to locate appropriate bins when processing the references. Thus, future tools might choose to support binning only for static and stack variables larger than some user defined threshold size. Alternatively, the tool could give the user convenient constructs for indicating, at compile time, which variables should be monitored in individual bins.

4.2.10 Summary

To summarize the implementation issues and details discussed in this section, our experiences have indicated that the granularity of the statistics presented is one of the crucial issues in developing a useful and efficient performance monitoring tool. When statistics are kept at too coarse a granularity, one cannot isolate particular bottlenecks and attribute them to particular code or data structures. On the other hand, when statistics are too fine-grained, managing them may require too much storage or processing time.

With this in mind, MemSpy has opted to keep code oriented statistics at a procedure granularity, and in data oriented statistics, we have proposed a method for aggregating statistics for memory ranges allocated at the same source code point via the same call path. Because of the aggregation method used, MemSpy's naming scheme must be able to distinguish data bins even when they are allocated on the same source code line. Thus, to match the data aggregation method, MemSpy prepends a data bin name with a record of the procedure call path at the time of the memory's allocation. These choices for statistics granularity and naming have proven effective on a number of applications tuned.

Internally, MemSpy stores statistics as structures called statistics bins, which collect information either for a particular procedure, or for a particular data aggregation, or for procedure–data pairs. Stack information facilitates the mapping from code locations to bins, while the data bin mapping relies on a hierarchical array which subdivides the mapping information based on bit fields from the referenced address. As the data in Chapter 5 will show, these methods are, overall, quite efficient. In addition, performance optimizations described in Chapters 5 and 6 will further reduce their impact on program performance.

4.3 Other Options in Data Collection

The previous sections have described the collection of program information for MemSpy statistics, based on a simulation approach. At this point we discuss the issues in alternative approaches for collecting this information. The two main alternatives we discuss here are hardware supported monitoring and software instrumented monitoring.

Hardware Monitoring

Hardware tracing support has been used to provide information on machine and program performance, particularly for architectural research and design projects [LLJ$^+$93, BM89, SMDO88, Tha90, Rei90]. Its main advantage is the ability to collect detailed performance information with little perturbation of the application being studied.

This specific hardware support often focuses on providing statistics on cache and bus behavior. These statistics include reference frequency, cache miss rate, and contention delays. In larger parallel machines, monitors also gather statistics on network delays and occupancy. The Stanford DASH Hardware Performance Monitor [LLJ$^+$93] allows users to collect statistics on particular ranges of memory, and to turn monitoring on and off. However, even this fairly specific support is not sufficient to sort information by data bin. Hardware support for data oriented statistics must maintain separate statistics on a large number of memory ranges. Furthermore attempting to maintain MemSpy's statistics on causes of misses in hardware would require unreasonably specialized support for storing previous activity to each cache line. Software post-processing to resimulate collected traces and gather such information is a more feasible technique for gathering statistics on causes of misses, but it increases the overhead of this approach to the point where it is comparable to solutions with no hardware support at all.

Software Instrumentation

Software instrumentation allows for a broader range of statistics to be collected in software with fairly general hardware support, such as high-resolution timers. This technique is different from software simulation because the application code is instrumented with additional code to monitor the program's actual current state as it runs on the target machine. In the performance monitoring tools Mtool [Gol92], Gprof [GKM83], and Pixie [Smi91], this approach is used to count the frequency that each program basic block is executed. One can also instrument code with calls to timer functions to provide information on time durations spent in particular portions of the code. This approach was used in an early version of Mtool [GH90]. Overall, software instrumentation can be faster than full software simulation, but without the specialized hardware requirements of the hardware tracing approaches described in the previous subsection.

The main drawback to this approach is the potential for perturbing the application being studied. The perturbation caused by such instrumentation is a function of the execution time of the added instrumentation, and the frequency that the application needs to execute it. The addition of very fine-grained instrumentation can change the true application reference patterns such that resource queuing on write buffers, memory modules and buses is changed as well. When this contention is important to program performance, this method of monitoring may introduce too much error to be useful. These perturbations can be especially serious in parallel code because changes in application timing can affect the interleaving and control flow of parallel threads.

When fine-grained clocks are provided in user-accessible registers (as in the HP Precision, IBM RS6000, and SPARC Version 9 architectures), some instrumentation can be added with little perturbation. With these registers, latencies of code blocks can be computed using two store operations to note the start and end times of the interval. (More instructions may be required to add this to a running total or compute an address to store the value.) Assuming that a perturbation of 10% additional instrumentation code is acceptable, intervals of code roughly 20 or more instructions long could be instrumented. Since individual memory references occur more often than once every 20 instructions, it is still not feasible to instrument code in this manner to gather accurate detailed statistics like MemSpy's. However, future implementations could consider a combination of (i) *static compile-time analysis* to eliminate dynamic instrumentation for references whose hit/miss outcome is known, with (ii) *software instrumentation* of the remaining references. Methods for correcting statistics to account for monitor perturbation [SM93, MRW92] may also broaden the range of applications in which this approach is feasible.

Software Simulation

Thus, it was with these issues in mind that we chose software simulation techniques for gathering MemSpy statistics. Simulation-based monitoring allows one to collect statistics at arbitrary levels of detail, without danger of perturbing results as would happen when running on a real machine. Furthermore, the simulator offers flexibility in varying machine parameters such as the cache and interconnection network parameters. However, it can be difficult to build simulators which accurately model the behavior of the real hardware. Furthermore, the high overheads of accurate detailed simulators were previously considered a fundamental limitation in simulation based approach. This thesis refutes that belief by demonstrating a simulator and accompanying optimizations which allow detailed statistics to be gathered at low overheads.

4.4 Chapter Summary

This chapter described the design decisions involved in implementing MemSpy. Gathering and presenting performance statistics at appropriate granularities is one of the key issues in generating useful performance profiles. We have found that MemSpy's breakdown of information by data structures and procedures gives useful statistics output at manageable granularities.

Aggregating statistics is often essential in achieving appropriate collection and presentation granularities. To this end, this chapter presented a heuristic for aggregating statistics for similarly used data structures, based on their allocation point and the dynamic call path taken to reach the allocation point. In our experience this has been quite effective in grouping statistics together to better identify bottlenecks.

The aggregation method also leads to interesting naming issues, since MemSpy's naming scheme must be able to distinguish data bins even when they are allocated on the same source code line. Thus, to match the data aggregation method, MemSpy prepends a data bin name with a record of the procedure call path at the time of the memory's allocation. This choice for statistics naming has also been effective on a number of applications tuned. When not suitable for a particular application, users can choose to manually rename particular statistics bins as well.

This chapter also presented details on MemSpy's simulation-based monitoring. We argue that simulation is a much more flexible approach than hardware based monitoring, and with the implementation described here, and the optimizations presented in Chapters 5 and 6, it can generate useful, accurate profiles of application memory performance at very reasonable execution time overheads.

Chapter 5

Performance

In order to be useful to programmers, performance monitoring tools must have acceptably low execution time overheads. This chapter presents performance measurements which demonstrate our claim that MemSpy's detailed simulation-based approach can be implemented quite efficiently.

A major factor in the speed of a simulation-based tool is the degree of accuracy with which the memory system is simulated. The performance results presented in this chapter and Chapter 6 are measured using a simple memory simulator described in Section 5.1.1. The benchmarks used for the performance measurements are described in Section 5.1.2.

Section 5.2 presents the first set of performance results for MemSpy, measured for a "baseline" implementation of the simulator. This implementation has had a number of simple code optimizations performed on it, so while it is a straightforward implementation of the methods presented in Chapter 4, it is already fairly efficient. (For example, the performance of this baseline implementation exceeds the performance of the MemSpy implementation presented in [MGA92].)

Building on this, Section 5.3 proposes new approaches for improving MemSpy performance by specializing the processing of cache hits. Finally, since the performance numbers presented here are based on a fairly simple simulator, Section 5.4 describes additional factors which affect the simulator's accuracy, how they could be incorporated into MemSpy simulators, and at what cost to performance.

5.1 Performance Measurement Setup

Since MemSpy's execution time overhead is dependent on the detail at which the caches and memory system are simulated, this section first discusses characteristics of the memory simulator used. We then describe the application benchmark set chosen.

5.1.1 Simulator Characteristics

The results presented in this dissertation were gathered for a very simple memory simulator that considers a single direct-mapped cache per processor. Cache hits execute in a single processor cycle and cache misses take a fixed, parameterized latency to be serviced. Network contention is not modeled. All heap references and static data references in the code are instrumented for simulation. However, no references to the stack frame are simulated, nor are operating system references simulated. The simulator performs no virtual-to-physical address translation. For multiprocessor simulations, we simulate an invalidation-based protocol. Each write reference to a line causes the simulator to invalidate the line from all other caches.

Even when simulating parallel threads, the simulation is running on a uniprocessor workstation, so we also need a method for determining how to interleave the parallel threads on the sequential machine. Thread execution is interleaved by rescheduling threads at every synchronization point. That is, when the currently executing thread reaches a synchronization point, it performs a "reschedule request". This causes the Tango Lite system to place the current thread in a queue of runnable threads, and dequeue the thread whose clock indicates it is the farthest behind. We have found this rescheduling granularity to be effective much of the time, but when programs synchronize very infrequently, it can lead to inaccuracy. Section 5.4 discusses the efficacy of this approach.

The specific model parameters used in this chapter, as well as in much of Chapter 6, are as follows. The sequential machine has a 128KB direct-mapped data cache with 32 byte lines. The 128KB cache was sized to represent a currently typical size for an off-chip cache. The machine is simulated with a 50 cycle cache miss latency, which is typical for workstation class machines such as current SGI workstations based on

MIPS R4000 processors. In the parallel machine, each processor is simulated with its own 64KB direct-mapped cache with 32 byte lines. These also have a 50 cycle miss penalty. This roughly models a uniform memory access time (UMA) multiprocessor in which all cache misses are serviced in approximately constant time. While this model is simple, we have found that it captures most of the important memory system behavior in applications used with MemSpy. Since the majority of production multiprocessors are still fairly small-scale UMA machines with less than 32 processors, this simulator matches them reasonably well.

In the arena of larger-scale multiprocessors, however, most follow a NUMA, or non-uniform memory access time, model. In these machines, memories are distributed across the processors such that cache miss penalties are larger when a remote memory services the cache miss, rather than local memory. MemSpy's simple simulator can capture the general trend of the memory behavior on such machines, which can often be sufficient to identify the prime sources of poor application memory behavior. It does not, however, indicate the potential benefits of localizing data by explicitly placing it on a memory near the processors that access it more frequently. When such information is important, users can trade speed for additional accuracy, by using simulators that account for a broader range of machine effects. Section 5.4 discusses some of the major abstractions in the simulator and how they affect its accuracy.

5.1.2 Benchmark Applications

In this section, we give descriptions and basic application characteristics for the sequential and parallel benchmarks used in performance measurements in the dissertation. The main factors that affect MemSpy execution time overhead are the application's cache miss rate and its reference frequency. These two factors are important because together they determine the amount of additional monitoring code in the MemSpy instrumented version of the application.

These benchmark descriptions also establish the scope of MemSpy's applicability to programs with a wide range of source code sizes. By evaluating MemSpy's performance on substantial sequential and parallel applications from the engineering and scientific community, we show that its simulation-based implementation is useful on “real”, not just “toy” benchmarks.

Sequential Benchmarks

For the studies on sequential benchmarks, we use the four C language benchmarks listed in Table 5.1. All are compiled using the `-O2` optimization level on the MIPS Compiler, version 2.1. The characteristics shown were measured for a 128KB direct-mapped cache with 32 byte lines.

Table 5.1: Sequential application characteristics. Measured for a 128KB direct-mapped cache with 32 byte lines.

Application	Source Code Lines	Data Set Used	Total Mon. Refs. (Heap Refs.) (M)	Cache Miss Rate (Heap Refs.)(%)
MATMUL	492	N=512, b=64	139.5	18.2
ESPRESSO	13706	ti.in	150.4	0.23
TRI	2311	tk32.O	109.8	6.1
MP3D	1771	400K particles	240.5	3.8

The first application, blocked matrix multiply (denoted as MatMul), was illustrated in the first case study of Chapter 3. With a cache miss rate of 18.2%, this application is clearly a representative of the class of applications with poor memory performance expected to be tuned with MemSpy. This cache miss rate will be an important factor in determining MemSpy’s execution time overhead, as well as in determining the amenability of this application to optimizations like hit bypassing and reference trace sampling.

The second application, Espresso, is one of the SPEC89 benchmark applications [SPE89]. It is a logic minimization program used for Computer Aided Design (CAD) of digital circuits. Espresso’s exceptionally small cache miss rate (0.23%) makes it especially interesting to study in Chapter 6, where we show that the cache miss rate of the application is a factor in determining the accuracy of reference trace sampling applied to MemSpy runs.

The third application, Tri, is the code illustrated in the second case study of Chapter 3. Tri is a scientific kernel used in several sparse matrix applications. The code solves for the vector x in $Mx = b$, where $M$is a sparse, lower triangular matrix. We run it as a uniprocessor program here, and it has a moderate cache miss rate of 6.1%.

The fourth application, Mp3d, is taken from the SPLASH benchmark suite [SWG92]. Since it is also used as one of the parallel benchmarks, it is described in more detail in the following subsection.

Parallel Benchmarks

For our parallel studies we use applications taken from the SPLASH [SWG92] benchmark suite. They are coarse to medium grained parallel programs from a variety of engineering and science disciplines. The benchmarks used are written in C, with parallel constructs from the Argonne National Laboratories parallel macro package [LO$^+$87]. As with the sequential benchmarks, all have been compiled using the `-O2` optimization level on the MIPS Compiler, version 2.1. The benchmarks range in size from 1300 source code lines up to nearly 11,000, indicating MemSpy's ability to instrument and analyze applications of substantial size. Table 5.2 show some basic application characteristics for the benchmark set, running with 16 processors, with 64KB direct-mapped cache per processor and 32 byte lines.

Table 5.2: Parallel application characteristics. Measured using 16 processors, each with a 64KB direct-mapped cache with 32 byte lines.

Application	Source Code Lines	Data Set Used	Total Mon. Refs. (Heap Refs.) (M)	Cache Miss Rate (Heap Refs.)(%)
MP3D	1771	400K particles	240.5	8.5
CHOLESKY	2274	tk15.O	164.2	3.2
WATER	1340	343 particles	157.5	0.39
LOCUS	10951	Primary2.grin	120.9	1.1

Mp3d is an application taken from the aerospace domain. It simulates the behavior of particles in rarefied flow as they pass through the upper atmosphere at hypersonic speeds. Mp3d has the highest cache miss rate of the parallel applications studied in this dissertation: 8.5% for a 16 processor run, where each processor has a 64KB cache.

Cholesky is a benchmark which performs a parallel Cholesky factorization of a sparse, positive definite matrix. This computation arises in many numerical applications in the science and engineering, including structural analysis and device and process simulation. The application has a moderate miss rate of 3.2%.

The third parallel benchmark is Water. It performs an N-body molecular dynamics simulation to evaluate forces in a collection of water molecules in the liquid state. Of the parallel applications studied in this dissertation, Water has the lowest cache miss rate: 0.39% for the 16 processor run. This low miss rate has implication on MemSpy's execution time overhead, as well as on the accuracy of trace sampling evaluated in Chapter 6.

The final parallel application studied in this dissertation is LocusRoute. LocusRoute performs automatic routing of VLSI standard cell circuits, attempting to minimize the overall area of the circuit. LocusRoute has a miss rate of 1.1% for the 16 processor simulation.

These eight applications were chosen to span a variety of application domains including numerical computations, scientific and engineering applications. In addition, they span a wide range of application behaviors including cache miss rate and source code size. As such they give an indication of the variety of applications which are amenable to MemSpy's simulation-based style of performance monitoring. The wide range of cache miss rates also offers interesting contrasts when we evaluate the efficacy of optimizations like hit bypassing and reference trace sampling, whose performance depends on the cache miss rate of the application being monitored.

5.2 Performance of Baseline MemSpy Implementation

We begin our analysis of MemSpy's performance by presenting the execution time overheads for sequential and parallel applications on a "baseline" MemSpy implementation. These are baseline measurements in the sense that the MemSpy code being evaluated is an efficient, but straightforward, implementation of the statistics described in Chapter 4.

In Section 5.3, we will build on this presentation by proposing techniques that specialize the actions taken on cache hits, in order to optimize performance.

5.2.1 MemSpy Performance on Sequential Benchmarks

For each of the sequential benchmark applications, Table 5.3 shows the performance overhead of a MemSpy run. The first column of data shows the execution time for the application running directly on the uniprocessor workstation with no MemSpy instrumentation. The timing measurements are made on a DECstation 5000/240 workstation, which uses a 40MHz MIPS R3000 processor. The second column shows the execution time for the same application running on the same workstation, with MemSpy instrumentation. Finally the third column shows the overhead factor calculated by dividing the second column by the first. The overheads range from roughly a factor of 18 to slightly under a factor of 60.

Table 5.3: Sequential Applications: MemSpy performance when simulating a 128KB direct-mapped cache.

Application	Uninstrumented Uniprocessor Exec. Time (sec.)	MemSpy Exec. Time (sec.)	Overhead Relative to Uninstrumented
MATMUL	65.5	1404	21.4
ESPRESSO	24.8	1428	57.6
TRI	59.3	1076	18.1
MP3D	42.5	1483	34.9

To understand the causes of this overhead, we note that the amount of overhead incurred in processing an application is a function of (i) the number of instrumented events the application executes and (ii) the work that is performed on each of these instrumented events. We divide our discussion of MemSpy overhead according to those two categories.

Since MemSpy adds additional instrumentation to (i) memory references, (ii) procedure calls and returns, and (iii) memory allocations, applications which have a large number of these events tend to incur larger overheads when using MemSpy. For example, in the Tri application, only 8.3% of the instructions executed get instrumented, compared to values of 16% to 21% for the other applications. Partly due to this fact, Tri has the lowest overhead of the four sequential applications, a factor of 18.1.

Table 5.4: Sequential applications: Overhead incurred by each type of instrumented MemSpy event, as a percentage of total MemSpy overhead.

Application	Memory References (%)	Procedure Calls and Returns (%)	Memory Allocations (%)
MATMUL	99.9+	0.0	0.0
ESPRESSO	98.8	1.2	0.0
TRI	98.3	1.7	0.0
MP3D	97.8	2.2	0.0

To further understand the overhead data, Figure 5.4 breaks down the MemSpy execution time overhead according to the three categories of events processed: memory references, procedure calls and returns, and memory allocations. From this table, we see that 97% or more of the overhead is due to processing memory references. (Note that these overhead breakdowns are gathered using pixie [Smi91] profiles, and thus do not account for the memory stall time of either the application or the additional MemSpy instrumentation.) Thus, at this point we focus our discussion specifically on the processing overhead required for memory references in MemSpy.

Figure 5.1 gives a schematic representation of the actions MemSpy performs on each simulated memory reference. These actions are categorized into three types of overhead: (i) memory simulation itself, (ii) statistics bin searches, and (iii) context switches into and out of the simulator (register saves and restores). The figure indicates these different categories with different types of cross-hatching.

Using these same three categories of overhead, Table 5.5 shows breakdowns of the time MemSpy spends processing memory references. Roughly 40% of MemSpy's reference processing time is spent in memory simulation itself. This consists of (i) checking the cache data structures to see if the reference is a hit or a miss, (ii) updating the data

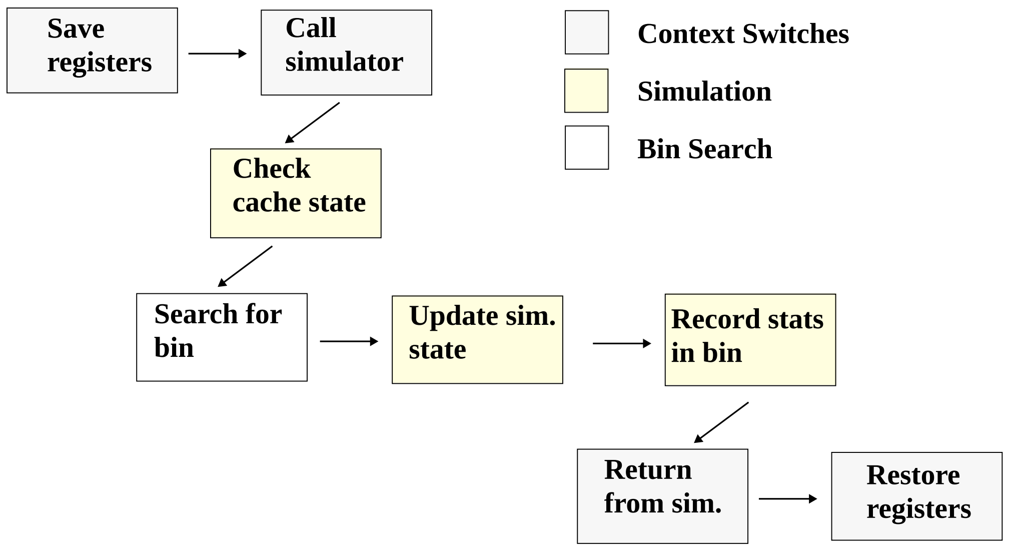


Figure 5.1: Sources of MemSpy instrumentation overhead.

Table 5.5: Sequential applications: Overhead incurred by each category of memory reference processing overhead, as a percentage of total MemSpy overhead due to memory reference processing.

Application	Memory Simulation (%)	Bin Lookup (%)	Context Switches (%)
MATMUL	39.1	34.9	25.9
ESPRESSO	37.5	27.8	34.8
TRI	45.1	32.1	22.7
MP3D	29.6	37.2	33.2

structures on cache misses, and (iii) recording the causes of cache misses. For the benchmarks studied here, cache hit simulation takes roughly 85 cycles. Simulating a cache miss takes between 180 and 320 cycles.

Another third of the memory reference processing time is taken doing the register saves and restores that are required when entering and leaving the simulator. This overhead comes to between 80 and 85 cycles per reference.

Finally, roughly a third of the time is spent in finding the appropriate statistics bins to track statistics for each reference. This consists of accessing the mapping table described in Section 4.2.2, to find the appropriate statistics bin for each reference. This overhead is quite variable, since it depends on the number of bins in the application, and the layout of memory ranges to bins. For the applications considered here, this overhead varies from roughly 65 to roughly 106 cycles per reference. Overall MemSpy overheads for processing cache hits are roughly 230 to 275 cycles per reference, and for cache misses are about 320 to 510 cycles. To summarize, memory reference processing overhead is the main factor in MemSpy performance, and that processing time is roughly evenly split between simulating references, performing bin searches, and performing application-simulator context switches.

A final factor in analyzing MemSpy overheads figures is determining how the execution time dilates between an uninstrumented and MemSpy instrumented application. For example, in the uninstrumented code, cache hits execute much more quickly than cache misses, typically in one cycle. In contrast, cache misses may take roughly 50 cycles on current high-performance workstations. Thus if we assume that MemSpy hit processing takes 250 cycles and MemSpy miss processing takes 500 cycles, then cache hits will be dilated by a factor of 250, while cache misses are only dilated by a factor of 10. In other words, despite the additional processing required on cache misses compared to cache hits, the overall time dilation experienced for low cache miss rate applications (like Espresso) is greater than that for poorly behaved applications (like MatMul). This observation explains why Espresso's overhead is significantly larger than that for MatMul, despite the fact that MatMul has a slightly higher percentage of instrumented instructions.

The next subsection will give similar breakdowns for parallel applications running with MemSpy. Following that, we will discuss each of these overheads, and examine optimizations designed to reduce each of them.

5.2.2 MemSpy Performance on Parallel Benchmarks

Table 5.6 gives performance results for running MemSpy on parallel applications. The execution time overhead is shown relative to a uniprocessor execution time of the program. To compare this to an actual multiprocessor execution time of the program, one would multiply these overheads by the expected program speedup. We see that the overhead, now ranging from factors of 64 up to factors of 115, is significantly higher than for the sequential benchmarks.

Table 5.6: Parallel Applications: MemSpy performance when simulating 16 processors, each with a 64KB direct-mapped cache.

Application	Uninstrumented Uniprocessor Exec. Time (sec.)	MemSpy Exec. Time (sec.)	Overhead Relative to Uninstrumented
MP3D	42.5	2698.4	63.5
CHOLESKY	67.8	2007.4	67.8
WATER	79.3	2456.8	79.3
LOCUS	114.4	2506.1	114.4

Since we once again find that memory reference processing forms the bulk of the overhead, Table 5.7 gives a breakdown of memory processing overhead into the categories previously discussed. This breakdown shows that most of the overhead increase in moving from sequential to parallel applications is due to an increase in the complexity of the memory simulator. In the parallel case, memory simulation accounts for 42% to 54% of the overhead. By contrast, in the sequential case, it accounted for only 30% to 45%. There are two primary sources for this overhead increase. First, simulation of write references takes longer in the parallel case than in the sequential case. Second, there is additional context switching overhead in the parallel case.

Table 5.7: Parallel applications: Overhead incurred by each category of memory reference processing overhead, as a percentage of total MemSpy overhead due to memory reference processing.

Application	Memory Simulation (%)	Bin Lookup (%)	Context Switches (%)
MP3D	53.5	26.6	19.9
CHOLESKY	45.1	32.1	22.7
WATER	49.4	22.7	28.0
LOCUS	41.7	35.7	23.6

The increase in write simulation time occurs mainly because in the simulations of parallel benchmarks, one must issue invalidations on all write references in order to maintain coherence in the multiple caches. To verify this, we used pixie to profile the simulation time for reads and writes in parallel applications. The read processing time in the parallel case is similar to that for the sequential benchmarks, ranging from 227 to 261 cycles. However the write processing times range from 550 to 634 cycles, a factor of 1.8 larger. Thus, invalidation processing, the primary difference between read and write simulation, represents significant additional overhead in processing references for the parallel benchmarks.

The second increase compared to sequential benchmarks, context switch overhead, stems from the fact that we are simulating multiple threads on a sequential machine by interleaving their execution. Thus, there is additional overhead required to context switch between the simulated threads. For parallel applications, the column in Figure 5.7 marked “Context Switches” includes both (i) register saves/restores to switch from application to simulator and back and (ii) register saves/restores to switch from one application thread to another. However, since we are context switching only at synchronization points (and these benchmarks synchronize relatively infrequently) this additional overhead is not substantial. Context switches *from application to simulator* incur 19.4% to 24.1% of the pixie reported overhead. By contrast, context switches *between threads* account for only 0.04% to 0.12% of the pixie overhead.

Overall, the primary performance difference between simulations of the parallel and sequential benchmarks stems from the invalidations processing. With the exception of this, their behavior is similar, and subsequent sections will treat the sequential and parallel benchmarks as a single group.

5.3 Performance Optimizations

Section 5.2 outlined the basic performance of the MemSpy system and showed that the main sources of overhead are: (i) memory simulation itself, (ii) statistics bin searching, and (iii) context switching. In this section we discuss optimizations to improve performance. The first optimization is targeted mainly at reducing the time spent doing bin searches. The second optimization is targeted at reducing time spent on context switching. The trace sampling optimization discussed in Chapter 6 will target memory simulation time itself.

5.3.1 Statistics on Cache Misses Only

Tables 5.5 and 5.7 indicate that 23% to 37% of MemSpy's execution time overhead is spent searching for the appropriate statistics bin for a particular referenced address in a particular procedure. The hierarchical search method described in Section 4.2.5 in Chapter 4 is already quite efficient, so at this point we expect that the most significant performance improvements will come from locating instances when the bin search can be omitted, rather than from straightforward code optimizations.

We note that the "interesting" memory events, from a performance tuning point of view, are events which incur memory stalls. For most architectures, only cache misses incur stalls. Thus, we can capture most of the interesting behavior of applications without keeping any statistics on cache hits at all. Even in poorly behaving programs, most references are typically cache hits. By keeping no statistics on hits, we can avoid the bin search overhead for a majority of the references.

Figure 5.2 shows the new overhead for both sequential and parallel programs when statistics are kept only for cache misses. This technique reduces overheads by 20% to 30%

in most cases. One application, LocusRoute, now shows a very large improvement (nearly a 50% reduction) as a result of this change. Note that in some cases the performance improvement actually exceeded the fraction of time spent in bin finding. While mainly targeted at bin search overhead, this optimization also reduces time spent on statistics updates within the simulator itself. Thus, the benefits from it come from both the bin search overhead as well as the simulation overhead categories.

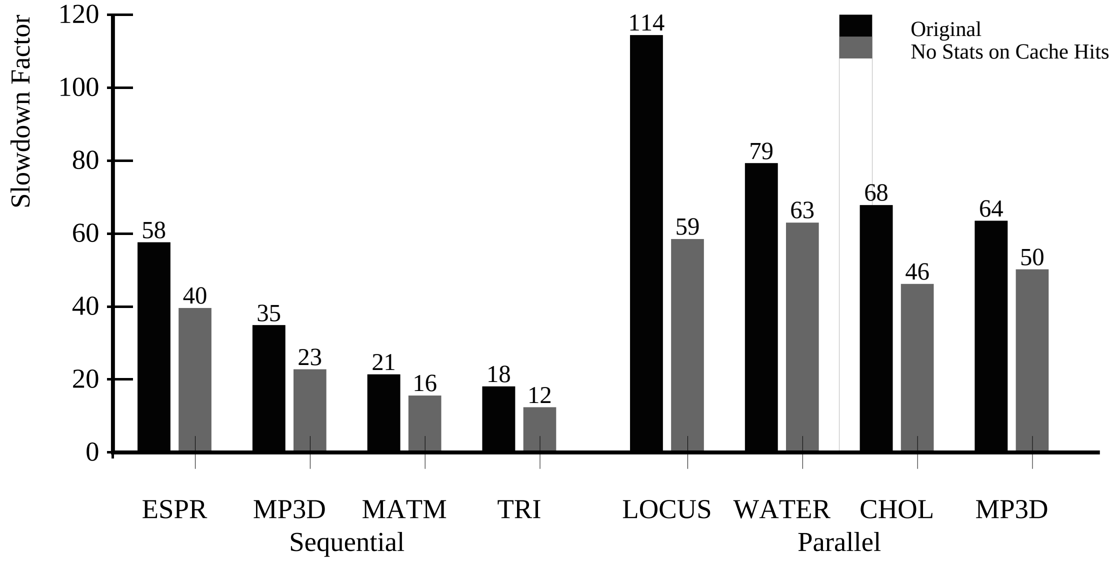


Figure 5.2: MemSpy performance when not gathering statistics on cache hits.

Finally, note that with this change to monitoring only misses, statistics on cache miss *rate* are not directly available. The output will report miss counts or stall times incurred, rather than miss rates as the primary metric. In our experience these miss counts or stall time totals are more useful than miss rates in focusing the user's attention on code bottlenecks. In particular, frequently referenced data structures with moderate miss rates may incur more stall time than infrequently referenced data structures with large miss rates. The case studies in Chapter 3 illustrate our use of MemSpy with this focusing mechanism. If desired, relatively inexpensive profile analysis (through pixie or program counter sampling for example) could augment these miss counts with total reference counts. This would allow miss *rates* to be computed for code units like basic blocks or procedures.

5.3.2 Hit Bypassing

The previous subsection showed how the bin lookup and simulation components of MemSpy's execution time overhead could be dramatically decreased by eliminating statistics monitoring on cache hits. We now focus on optimizing the context switching overhead required to go to and from the memory simulator on all references. To accomplish this, we modify the processing of cache hits to bypass calls to MemSpy entirely. As shown in Tables 5.5 and 5.7, 20% to 35% of MemSpy's memory processing overhead is in register saves and restores associated with context switching to the simulator. Avoiding these register operations on cache hits should significantly reduce overhead.

Recall that in the original implementation, we save a full set of registers, and then call the simulator to check if the reference is a cache hit or miss. The cache check itself uses very few registers (roughly 4-7 depending on the simulator and the implementation), and if the reference is a hit, we return almost immediately, restoring the full set of registers. Note that although the simulator only uses a handful of the saved application registers on a cache hit, it still stores and reloads the full set of registers. Eliminating these unnecessary register saves and restores is the goal of hit bypassing.

Figure 5.3 illustrates our approach. With hit bypassing, we embed the cache hit check into the register save phase. This is implemented such that we initially save only the registers required to check if the reference is a cache hit or a cache miss. If the reference is a cache miss, we complete the rest of the register saves and continue simulating. If the reference is a cache hit, we restore the minimal subset of registers, and return to the application. Using 4 registers for the hit check, this code sequence requires about 25 instructions on a cache hit. Of these, the simulator hit check takes about 14 instructions, and the rest are for saving and restoring registers, and other control instructions.

In the case of a parallel application, writes that cause invalidations must always be simulated, even if they are cache hits. Thus, on all parallel writes, we enter the simulator,

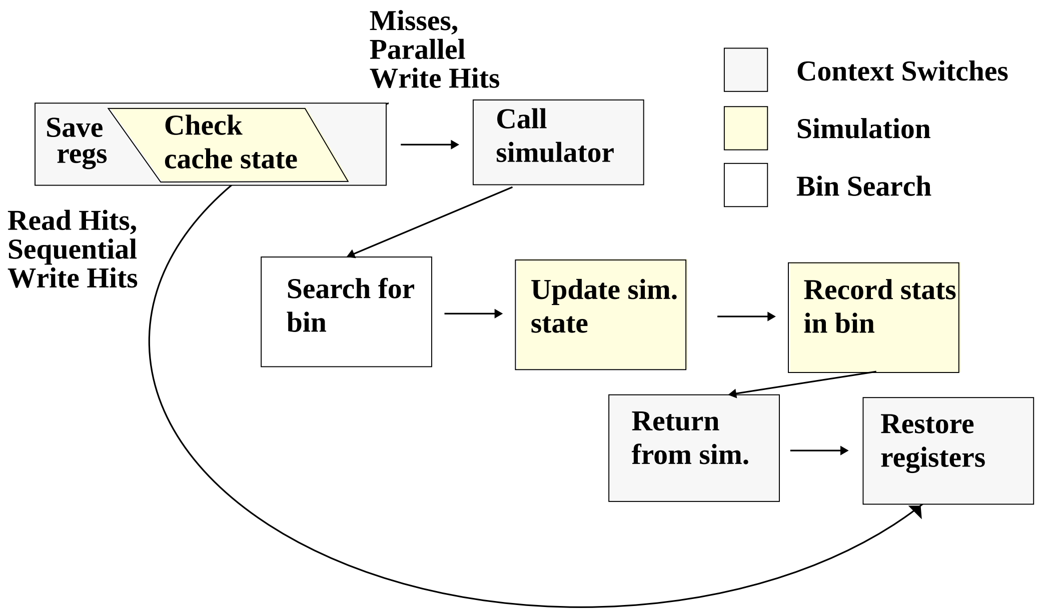


Figure 5.3: Bypassing registers saves and restores on cache hits.

but we do not keep statistics on write hits. (A further possible optimization here would be to have the simulator record whether each cache line is currently actively shared by multiple processors or not. When a memory line is only cached by one processor, both read and write references could be bypassed.) The parallel hit check code requires 5 registers and uses a total of 32 instructions to process a read hit. Of these, 19 are to perform the simulation, and the remaining 13 are for register operations and control instructions.

Figure 5.4 shows the performance benefits of using this hit bypassing method. When compared to MemSpy performance after the “no hit statistics” optimization, hit bypassing offers further additional performance improvements ranging from 6% to over 50%. In general, the sequential applications benefit more from hit bypassing – all show improvements of 35% or more. This is because the sequential memory simulator is simpler, so applications tend to spend a larger proportion of time in application-simulator context switches. Thus proportionally, optimizing these context switches has greater benefit in sequential applications.

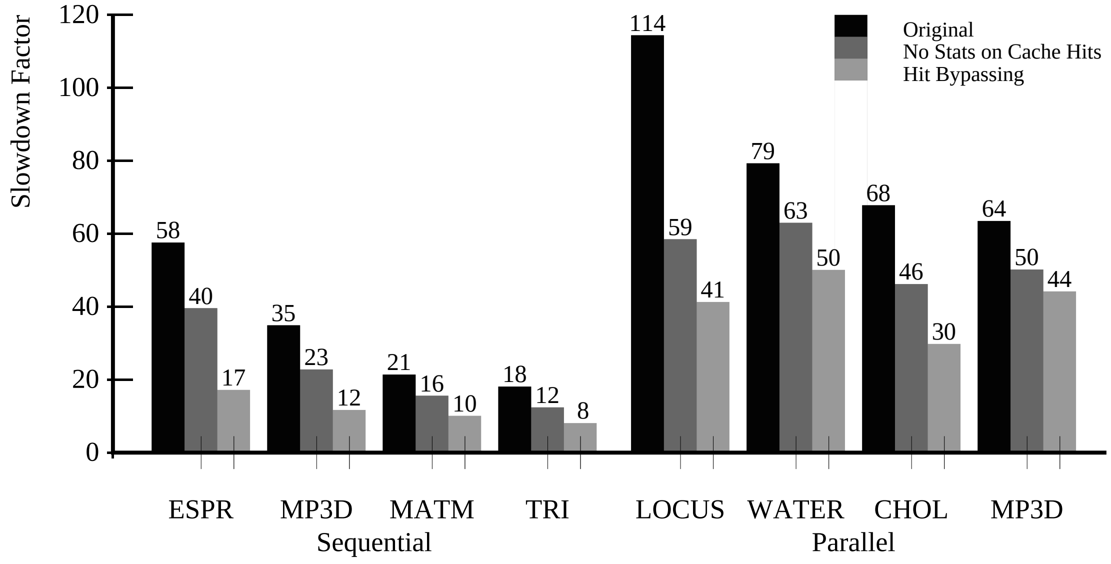


Figure 5.4: MemSpy performance with hit bypassing.

Of the parallel applications, those with a high fraction of read hits benefit the most from this optimization. This stems from the fact that parallel read hits can be bypassed while parallel writes cannot. For example LocusRoute and Cholesky have the highest percentages of read hits of the parallel applications (89.8% and 80.2%, respectively). Among these applications they also show the greatest relative improvements in performance: LocusRoute exhibits an overhead decrease of 29.4% due to this optimization while Cholesky shows a drop of 35.5%. Combining the effects of the two cache hit optimizations, LocusRoute's and Cholesky's overheads are cut by more than a factor of two. The other two parallel applications see performance improvements of roughly 30%.

Overall the hit bypassing optimization improves performance by factors of 1.5 to 3.4 compared to the original MemSpy implementation. This brings MemSpy's overheads down to factors of 8 to 17 for sequential code, and 30 to 50 for parallel code.

Sequential overheads in this range make MemSpy quite attractive already. The opportunity to run a 1 minute sequential program in 10 minutes and get detailed memory behavior profiles makes MemSpy a very reasonable addition to a collection of tuning tools. When first tuning an application, programmers may be happy to use a low-overhead tool like Gprof to get high-level insights on application behavior, but they often reach a point where they require more specific information on memory performance. Without tools like MemSpy, programmers might be forced, in these cases, to manually add specific monitoring code to their application source code, in order to identify the performance bottlenecks. In comparison to the tedious and error-prone task of hand-instrumenting code, a 10 minute wait for detailed statistics on a normally 1 minute program run is often a welcome alternative.

MemSpy overheads for parallel programs are slightly higher than for sequential programs, but still often attractive. As in the sequential case, MemSpy's overheads are very reasonable when weighed against the daunting alternatives of either (i) attempting to tune a program with no tool guidance at all or (ii) manually instrumenting code to discern bottlenecks. While overheads in the parallel benchmarks scale with the number of processors simulated, the majority of multiprocessors built in the next decade are expected to be small-scale machines of roughly 32 processors or less. At these levels of parallelism, MemSpy's overhead factors for parallel benchmarks are quite satisfactory. Moreover, Chapter 6 discusses further MemSpy performance optimizations making use of the technique of reference trace sampling that further reduce these overheads.

5.4 Simulator Accuracy and Completeness

The monitoring approach we have described thus far concentrates on collecting detailed application statistics with as low an overhead as possible. For the particular cache simulator studied, the proposed optimizations bring the application overheads down to factors of 8 to 17 for sequential code and factors of 30 to 50 for parallel code. This simulator, while reasonably accurate, does not simulate several "real world" effects. This section briefly discusses some of the elided effects and their likely impact on the accuracy and "completeness" of the simulation.

5.4.1 Stack Reference Filtering

The performance results presented here assume that only static and heap references will be simulated. That is, references through the program stack pointer are *not* instrumented for simulation. This approach has been termed *stack reference filtering*; it is based on the assumption that stack references typically have as good or better hit rates than other types of references. This assumption is supported by recent work by Goldschmidt [Gol93]. His studies on the SPLASH benchmark suite indicated that for cache metrics such as the read miss rate, the maximum error due to stack reference filtering is less than 10%, and the average error was less than 2%. These results were gathered for applications compiled using the `-O2` level of the MIPS compiler; Goldschmidt also points out that such code optimization tends to drastically reduce the number of stack references in the code, making their behavior less significant.

In our benchmarks, we also find that good accuracy can be obtained with this approach. In the sequential applications, from 63% to over 99% of the total program references are to heap or static data, and thus are still simulated even with stack filtering. In general we measure only a small relative deviation in cache miss rate between the full trace and the stack filtered trace. Tri and Mp3d are the sequential benchmarks that are affected the most. In Tri, the filtered trace has a cache miss rate of 6.1%, an 18% increase from the true cache miss rate of 5.2%. In Mp3d, there is a 15% deviation between the filtered miss rate of 3.8% and the unfiltered miss rate of 3.3%. For the other sequential applications, the deviation between the full trace miss rate and the filtered trace miss rate is quite small– less than 0.1% of absolute deviation.

The event filtering approach matches well with the parallel programming model we use. In this model, all references to shared data are either in the heap or in statically defined data. Stack references are defined to be private. Thus since it is typically the shared referencing behavior of parallel programs that requires tuning, filtering out stack events has little affect on accuracy. In the parallel benchmarks, from 75% to 97% of the total program references are to heap or static data. As with the sequential benchmarks, there is generally good agreement between the statistics with and without filtering. Mp3d has the largest absolute deviation between the cache miss rate with and without filtering. When the stack references are not simulated, the cache miss rate is overestimated by

1.25%, or a 17.2% increase. Water also has a significant *relative* error in the estimate, a 24% deviation between the true value of 0.31% and the event filtered value of 0.39%. However, the absolute deviation in this case is quite low anyway. For the most part, we have found that such deviations are of little consequence in the context of application performance tuning.

5.4.2 Virtual vs. Physical Addresses

We perform our simulations using the virtual addresses generated by the application. No simulation of a Translation Lookaside Buffer (TLB), or virtual to physical address translation is performed. This offers a significant performance optimization. In particular, it greatly reduces the processing required for cache hits, which allows the hit bypassing optimization presented in Section 5.3.

For this abstraction to be valid, the cache behavior of the virtually addressed references and the physically addressed references should be similar. The behaviors will differ when the virtual to physical mapping is such that virtual pages map to different cache lines than physical pages. This would lead to different cache replacement patterns, and possibly different cache interference behavior. However several current operating systems (including for example the Silicon Graphics IRIX Operating System) use heuristics for memory allocation that (when possible) place consecutive pages of virtual memory onto physical pages that map consecutively into the cache. When this is the case, data structures will map similarly into the cache, whether virtually or physically addressed. Thus the cache behavior of the virtual and physical addresses will be similar and the optimization of no TLB simulation will have little effect.

5.4.3 Relaxed Event Ordering

As previously stated, we simulate multiprocessor execution by interleaving the execution of parallel application threads on a uniprocessor. By default, MemSpy interleaves threads at synchronization operations in the code. That is, at each synchronization event, the simulator enqueues the currently executing thread and dequeues the thread whose clock indicates it is “farthest behind” in the simulation. There will generally be shared memory references between synchronization operations that may affect the caching behavior of other processors. Thus, different (but all strictly correct) interleavings of these references may lead to slightly different application behavior. This potential inaccuracy affects only multiprocessor simulations, since clearly uniprocessor simulations are single-threaded.

Goldschmidt studies thread interleaving for multiprocessor simulations in his thesis [Gol93]. He determines that the relaxed event ordering employed by default in MemSpy offers good accuracy in caching metrics, with significant performance benefits. Simulation speed is increased by factors of two to three. His data indicate that on average, estimates of *read miss counts* are within 4.3% of the true value obtained with thread rescheduling at every reference. Average estimates of total elapsed time are even more accurate, with less than 1% error. One application from his benchmark set, LocusRoute, experiences larger errors: the read miss error is 21.5% and the error in elapsed time is 3%. This is because LocusRoute makes a large number of unsynchronized reads and writes to its main shared data structure. As with the other optimizations discussed, when extremely good accuracy is required, the user can request more fine-grained thread interleaving, with additional overhead.

5.4.4 Operating System References

Operating system references can also affect the cache behavior of applications. Operating system references during library and system calls can displace application cache lines, inducing additional misses when they are re-referenced. Torrellas [Tor92] reports that for his benchmarks, roughly 20% of application misses stemmed from operating system effects.

One could include the effect of operating system behavior at a variety of levels of detail. Simple models might involve, for example, stochastically simulating operating system cache interference. For example, one could replace a randomly chosen collection of cache lines on system calls.

However, some work is also in progress to include operating system effects more accurately and completely. For example, the work of Chen and Bershad [Che93b, CB93] examines a system in which system software and the operating system kernel were instrumented for tracing. They used a modified version of epoxie [Wal92] (an assembly level instrumentor) to instrument applications and system code to generate reference traces. They report overheads factors of roughly 115 for simple simulations of sequential programs with this approach [Che93a]. Thus, these more comprehensive monitors seem to be a promising extension of the current MemSpy tool. For the engineering and scientific workloads we have studied thus far, however, operating system references do not significantly affect memory system performance.

5.4.5 Multiprogramming References

Multiprogramming can further impact application memory behavior. Multiplexing the execution of different applications can cause them to interfere in the cache. The memory behavior of the application being tuned will be affected differently by different multiprogramming workloads.

It may be possible to include some multiprogramming effects without the overhead of simulating multiple applications. For example, one may be able to stochastically simulate the cache-purging effects of multiprogramming, without actually doing a detailed simulation of the entire workload. By having the simulator clear some user-selectable fraction of the cache lines during periodic context switches, the target application could see many of the effects of a multiprogrammed workload, without the large simulation overhead.

To get more precise assessments of the effect of multiprogramming, one could predict a likely multiprogramming workload, and build Tango Lite simulators to run multiple application threads at the same time. To do this, one would need to simulate the behavior of several applications, while typically only tuning the behavior of one application. Thus, when computing "runtime" overhead for a tool being used in this way, one would have to compare the total simulation time of *all* the interleaved applications in the workload, divided by the true runtime of only the application being tuned. Assuming the N simulated applications have similar run times, the overhead is at least N-1 times larger than the overhead of simulating the target application alone. (If one is tuning *all* the workload applications, one could argue that this effect is not significant.)

5.5 Chapter Summary

This chapter has examined the execution time performance of MemSpy on both sequential and parallel benchmarks. We first examined the baseline performance for sequential benchmarks and found overheads of roughly 18 to 58 times. Baseline parallel overheads are slightly higher: factors of 63 to 114.

We then examined optimizations to improve the performance of MemSpy. These optimizations are targeted at reducing the overhead at cache hits, the most common case in memory simulation. They particularly aim to streamline the context switching overhead and bin search overhead for cache hits. With *hit bypassing*, MemSpy performance is improved by factors of two or more in most cases. Following these optimizations, MemSpy overheads range from 8 to 17 times for sequential programs. For parallel programs, they range from roughly 30 to 50 times. These overheads are already quite attractive, given the usefulness of the detailed statistics presented.

Having examined these overheads and optimizations, we also discussed the effects on simulator performance of several simulation abstractions made in MemSpy. In general, they have small effects on accuracy for the applications studied here. Users are often willing to accept small losses in accuracy in exchange for large performance benefits. In fact, one could imagine a spectrum of possible tool tradeoffs between simulator accuracy and performance. Initial performance evaluations would be run with simple, but very fast simulators. As programmers fine-tune their code, they could then choose more detailed simulations, with slightly higher overheads. With such a spectrum in mind, Chapter 6 next introduces another performance optimization, reference trace sampling, that also makes small sacrifices in accuracy for significant further gains in speed.

Chapter 6

Optimizing MemSpy Performance Using Sampling

The previous chapter discussed MemSpy's performance and introduced optimizations which significantly reduced its execution time by streamlining the processing required on cache hits. At this point we look to the technique of *reference trace sampling* for further performance improvements.

Reference trace sampling is the technique of estimating cache behavior while simulating only portions of a reference trace, rather than simulating the full trace. Intuitively, this promises significant speedup, since one incurs the full simulation overhead only on a fraction of the full reference stream. If one samples such that only one tenth of the references are simulated, one can hope for a speedup of up to a factor of ten. On the other hand, there is an inherent tradeoff between the fraction of references simulated and the accuracy of the simulation results.

The main goal of this chapter is to show that within the context of a performance debugging tool, reference trace sampling can be used effectively to improve the tool's performance while retaining acceptable accuracy. Section 6.1 presents breakdowns of MemSpy overhead which indicate why this approach is promising from a performance standpoint, and Section 6.2 gives background information on the performance and accuracy issues of sampling.

In Sections 6.3 and 6.4, we present results showing how the accuracy of the sampled results varies with parameters such as the number and size of samples taken, the cache size, and the number of processors. Although these sections discuss accuracy in terms of a single metric, the cache miss rate, Section 6.5 discusses how sampling accuracy extends to other MemSpy metrics as well.

In Section 6.6, we present MemSpy execution time measurements which show the success of reference trace sampling at improving tool performance. In general, we find that the parameter settings required for accurate sampled output allow for significant performance improvements. Section 6.7 then contrasts the method of sampling discussed here, time sampling, with other sampling methods. It also discusses several avenues for improving sampling accuracy.

6.1 Motivation

The motivation for implementing sampling stems from the MemSpy overhead breakdowns following the hit bypassing optimization of Chapter 5. These breakdowns, shown in Tables 6.1 and 6.2, subdivide MemSpy overhead into four components. The first three columns of data are the three categories of overhead incurred when MemSpy processes memory references. The fourth column indicates the proportion of time MemSpy spends processing procedure calls and returns. The data indicate that memory simulation and context switches comprise the bulk of MemSpy's overhead when hit bypassing is used. (Recall that in the sequential case these overheads are computed for a 128KB direct-mapped cache with 32 byte lines. In the parallel case, these overheads are computed for 16 processor runs, where each processor has a 64KB direct-mapped cache with 32 byte lines.)

To reduce the impact of these overheads, one could take two courses. First, one could optimize the memory simulator itself. At this point, however, the memory simulator is already quite simple and optimized, so further optimizations are not likely to yield significant improvements. (For cache hits, the simulation is coded in assembly language, and comprises only 14 to 19 instructions embedded into the context switching code.)

Table 6.1: Sequential applications: Overhead incurred by memory reference processing and procedure logging, as percentages of total MemSpy overhead using hit bypasssing.

Application	Miss Simulation (%)	Bin Lookup (%)	Context Switches Plus Hit Check (%)	Procedure Logs (%)
MATMUL	38.0	16.3	45.7	—
ESPRESSO	1.1	0.3	88.7	9.7
TRI	19.6	8.4	61.8	9.9
MP3D	5.9	13.8	71.7	8.6

Table 6.2: Parallel applications: Overhead incurred by memory reference processing and procedure logging, as percentages of total MemSpy overhead using hit bypasssing.

Application	Miss + Write Simulation (%)	Bin Lookup (%)	Context Switches Plus Read Hit Check (%)	Procedure Logs (%)
MP3D	62.9	8.5	26.8	1.9
CHOLESKY	54.7	4.8	39.6	0.8
WATER	43.4	0.3	32.8	23.5
LOCUS	29.9	6.6	39.8	23.7

A second, more promising course is to reduce the frequency that the simulator is called. Section 5.4 discussed some initial tradeoffs along these lines, which included omitting simulation of stack frame references and operating system references. In this chapter, we turn to reference trace sampling as another approach to reducing the frequency of simulation.

6.2 Background

Although reference trace sampling can take several forms, this chapter focuses on the use of time sampling. Time sampling is implemented by intermittently turning reference simulation on and off as a reference trace is processed. Figure 6.1 illustrates this by highlighting the references sampled for simulation. Section 6.7.2 discusses other approaches, such as (i) set sampling [KHW91], in which particular cache lines or sets are selected for simulation and (ii) processor sampling [CD93], in which, for parallel applications, particular processors are selected for simulation.

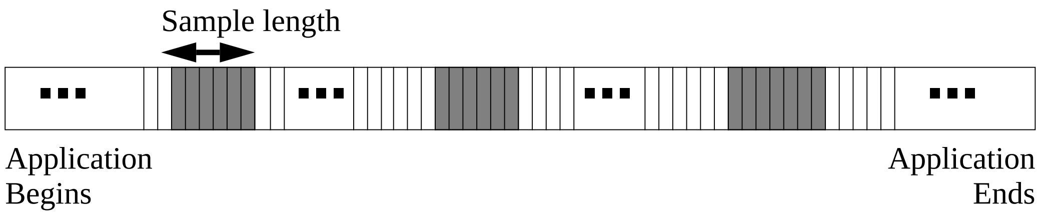


Figure 6.1: Time samples in an application reference trace.

Our implementation of time sampling has two main parameters. The first of these parameters is the *number of samples taken*. Intuitively, increasing the number of samples taken will make our sampled trace more representative of the full trace. The second parameter is the *sample length*, or number of references contained in each sample. Increasing the length of samples decreases the significance of the *unknown references* occurring due to a cold-start effect at the beginning of each sample. A third dependent parameter is the *sampling ratio*, which is computed by taking the ratio of the total number of references within the samples, divided by the total number of references in the run. Thus, it can be calculated as:

$$SamplingRatio = \frac{SamplesTaken \times SampleLength}{TotalReferences}$$

The sampling ratio can be used as an intuitive estimator for both the expected accuracy and the expected performance improvement from a particular sampling set-up. With respect to accuracy, one would expect that a sampling ratio closer to 1 would be preferable, since more references are being simulated. On the other hand, with respect to performance, the benefit one can expect from time sampling is limited by the reciprocal of the sampling ratio. For example, when simulating one tenth of the full trace, one would not expect more than a factor of ten performance improvement over full simulation. The following subsections discuss accuracy and performance issues for sampled simulations in more detail.

6.2.1 Performance Issues

Our primary purpose in using time sampling is to improve MemSpy's performance. An initial requirement for good performance is that the sampling ratio be kept small enough that significant performance improvements are possible. To this end, later subsections will evaluate the accuracy constraints on the sampling ratio. Beyond this, the sampling implementation must also be structured to benefit as much as possible when simulation is turned off. We have seen that a large fraction of MemSpy's time is spent in the register saves and restores required when switching from application to the memory simulator and vice versa. So, an implementation which simply turns off simulation *within* the MemSpy simulator will not improve performance significantly. Rather, to be worthwhile the implementation must circumvent the overheads of register saves and restores whenever the simulation is turned off. Our sampling implementation and resulting performance are described in Section 6.6.

6.2.2 Accuracy Issues

Simulation results from time sampling are subject to two orthogonal forms of inaccuracy: (i) error due to non-representative samples and (ii) error due to unknown references. Both are described below.

Error due to non-representative samples: This error corresponds to the deviation between the application's true miss rate when all references are simulated, and the application's true miss rate measured during sampled regions only.

The error due to non-representative samples is common to all forms of sampling, including the program counter sampling already common in several performance profiling tools including Gprof [GKM83] and Quartz [AL90]. In general, this error can be controlled by increasing the number of samples taken. Figure 6.2 illustrates how one can vary the number of samples taken, while fixing the total number of references simulated.

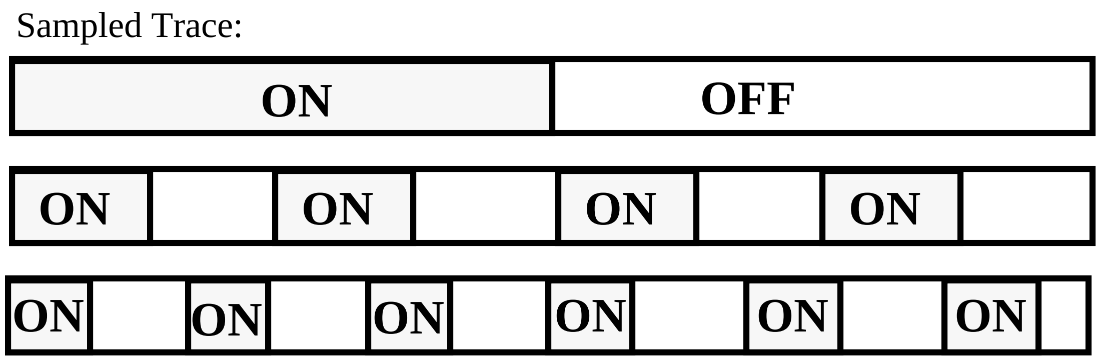


Figure 6.2: Varying number of samples taken, while holding sampling ratio constant.

Simulating references in a single large sample from the beginning of the trace, as shown in the top of the figure, can lead to non-representative results because the behavior of the program's initialization phase may carry disproportionate weight in the cache miss rate estimations. Rather, it is preferable to divide the references into smaller samples interspersed throughout the trace, so that the behavior of all the different program phases can be representatively captured. However, because of the other error described below, one cannot extend this to samples of arbitrarily short length.

Error due to unknown references: This error occurs because at the beginning of each sample, the cache state is not known. Until a cache set (or an individual line in a direct-mapped cache) has been primed with known state, it will be unknown whether references to it are cache hits or cache misses. The error due to unknown references is the deviation between the application's estimated miss rate (including an estimate of the miss rate of unknown references), and the application's true miss rate *during the sampled region*.

The situation leading to this second error is illustrated in Figure 6.3. At the beginning of each sample, the cache state is unknown. Then, as references are processed, cache lines are primed with known values, and subsequent references to those primed lines are known to be either cache hits or cache misses.

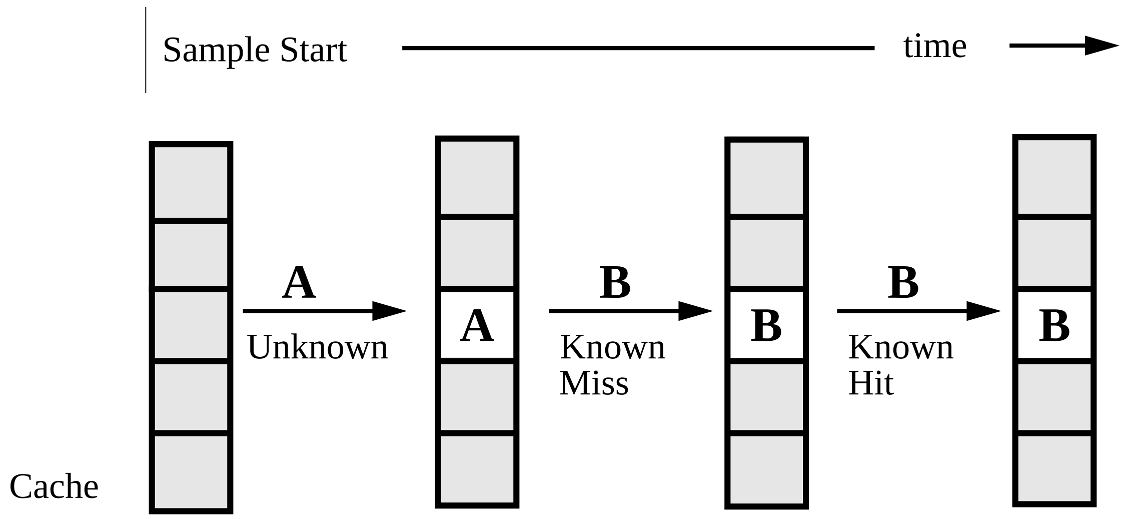


Figure 6.3: Cache priming and unknown references in time sampling.

For a particular sample of references, the cache miss rate within the sample, m, can be expressed as:

$$m = \frac{M_k + \mu U}{H_k + M_k + U}$$

where H_k is the number of *known hits* in the sample, M_k is the number of *known misses* in the sample, U is the number of unknown references, and μ is the fraction of unknown references which are actually cache misses. Knowing H_k, M_k, and U, we can estimate the sample's miss rate, m, by estimating the unknown reference miss rate, μ, and plugging it into the equation.

Wood et al. [WHK91] show that miss rates for unknown references are typically higher than the overall application miss rate, so assuming that μ is equal to the steady state miss rate (or the miss rate on known references) will result in overly optimistic performance estimates. One can always compute a range of estimated sample miss rates, m', by allowing μ to vary from 0 to 1. Thus, we can express an m' for $\mu = 0.5$ and give symmetric error bounds for ranges of μ from 0 to 1:

$$m' = \frac{M_k + 0.5U}{H_k + M_k + U} \pm \frac{.5U}{H_k + M_k + U} \qquad (6.1)$$

For all accuracy results shown in this dissertation, we use this simple estimate with $\mu = 0.5$. Wood et al. have introduced a more sophisticated method for estimating μ

by estimating the fraction of time that lines in the cache are dead (that is, will not be referenced again before a new line replaces them in the cache). For our benchmarks, we have found that this method slightly improves the accuracy of the sampling estimates, but requires significantly more processing time to compute.

Given these two types of errors, (i) error due to non-representative samples and (ii) error due to unknown references, an accuracy tradeoff exists when determining how to set the sampling parameters. In order to reduce the error due to non-representative samples, one should increase the number of samples taken, so that they are spread throughout the trace. In order to reduce the error due to unknown references, one should increase the length of each sample taken. However, to maintain a constant sampling ratio and performance, increases in the *number* of samples taken must be accompanied by decreases in the *length* of the samples. Similarly, increases in the *length* of a sample (to reduce the error due to unknown references) must be accompanied by decreases in the total *number* of samples gathered. Thus, one of the main goals of this chapter is to determine appropriate tradeoffs between these two errors.

6.3 Time Sampling in Sequential Programs

This section presents our results on the accuracy of time sampling for uniprocessor programs. In general, we find that time sampling is quite effective at accurately reproducing cache statistics from a full simulation.

As a preview, Table 6.3 compares estimated cache miss rates from sampling to (i) the program's true miss rate calculated over all references and (ii) the program's true miss rate calculated over only those references occurring during a sample. These results come from a simulation of a 128KB direct-mapped cache with 32 byte lines as in the previous chapter. The samples taken are 0.5M references long, with a sampling ratio of 10%. (That is, 0.5M references are simulated, and then simulation is turned off for 4.5M references, before it is turned on again.) In each of the applications, the absolute error in miss rate never exceeds 0.5%.

Table 6.3: Estimated and true miss rates for sequential applications with 128KB direct-mapped cache. Measured using a 10% sampling ratio and 0.5M references per sample.

Appl.	Overall True Miss Rate	True Miss Rate During Samples (%)	Estimated Miss Rate During Samples (%)	Number of Samples Gathered
MATMUL	18.2	17.9	17.8	28
ESPRESSO	0.23	0.14	0.20	57
TRI	6.1	6.4	6.1	22
MP3D	8.5	8.3	7.8	22

While the data show reasonable accuracy for one set of parameters, a more general evaluation of time sampling for performance debugging must examine accuracy over a range of parameters. Important questions to be answered are:

- How does the accuracy vary with the number of samples?
- How does the accuracy vary with cache size?
- How does the accuracy vary with the length of each sample?

If we understand each of these trends and tradeoffs, we can decide when, and to what extent, small amounts of sampling error may be accepted in exchange for better performance.

6.3.1 Accuracy vs. Number of Samples

The number of samples taken partly determines how representative the trace will be of the overall program performance. Intuitively, a single large sample will not reproduce the overall program's behavior as well as several smaller ones. The reason for this is that program behavior can vary substantially over the run time of the program, with some phases having very poor memory system performance while other phases have much better performance.

Laha et al. [LPI88] mention the importance of capturing representative samples, and present data indicating that 35 samples was generally sufficient to characterize the miss rates for their Lisp benchmarks. Here, we present more comprehensive data showing how accuracy varies across a wide range of values for number of samples. To do this, we fix the total number of references simulated and vary the number of samples taken. This approach allows us to study accuracy's dependence on the number of samples taken while also holding the sampling ratio (and therefore performance) constant. To study the representativeness of the samples collected, we examine the deviation between the true miss rate during sampled regions, and the application's overall true miss rate. (This is the error due to non-representative samples mentioned in Section 6.2.2.) Note that we are *not* studying the effects of unknown references in this section.

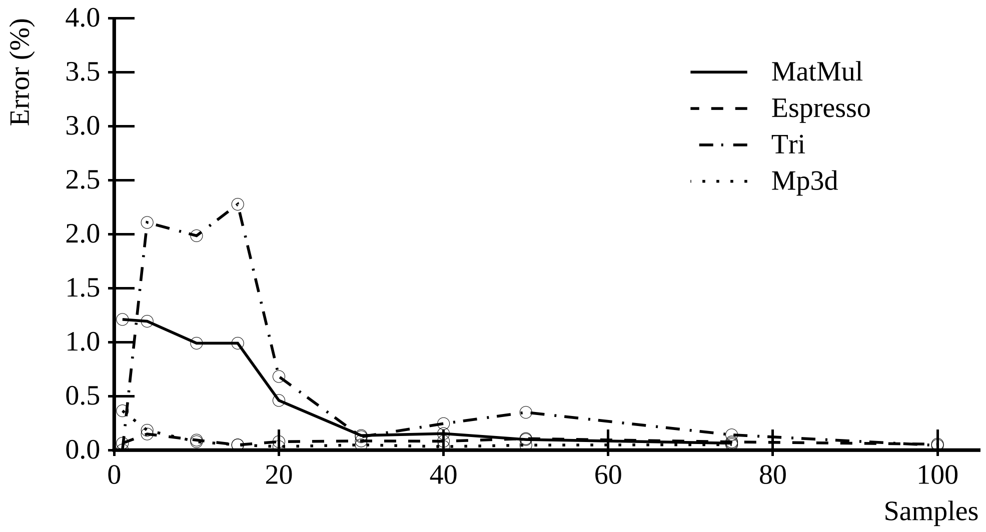


Figure 6.4: Absolute sampling error as a function of the number of samples taken. Measured using a 10% sampling ratio and simulating a 128KB direct-mapped cache.

Figure 6.4 shows the absolute deviations between the true miss rate during sampled regions and the overall true miss rate, plotted versus the number of samples taken. For each application, the total number of references simulated is held constant at 10% of the total application references, and the number of samples collected is varied from 1 to

100. In each case the number of references per sample is adjusted to maintain the 10% sampling ratio. Overall the four applications show excellent agreement. For 30 samples or more, absolute errors never exceed 0.5%.

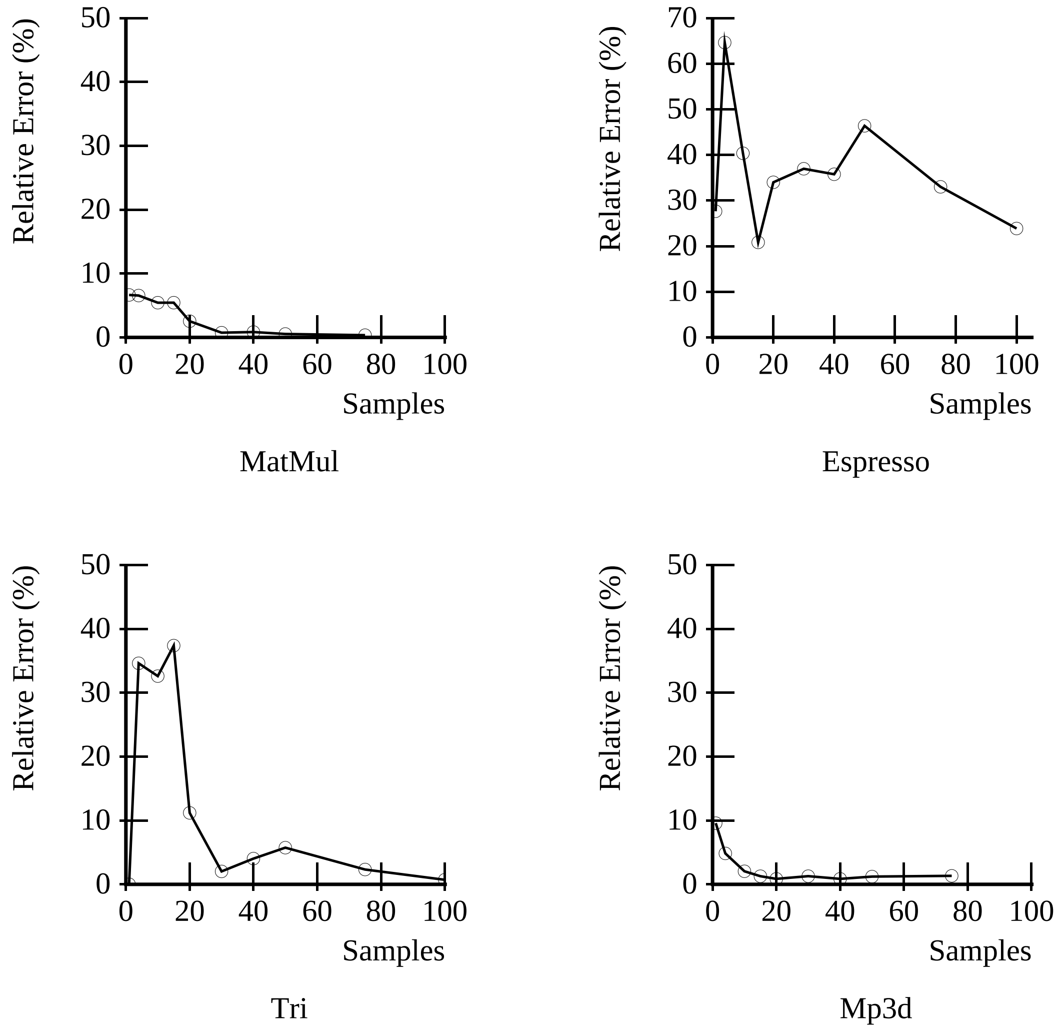


Figure 6.5: Relative sampling error as a function of the number of samples taken. Measured using a 10% sampling ratio and simulating a 128KB direct-mapped cache.

To view the accuracy trends of the four applications more closely, Figure 6.5 presents the same data plotted as *relative* deviations, normalized by each application's overall true true miss rate. For two of the applications, MatMul and Mp3d, as few as 10 samples suffice to bring the relative error down to roughly 5% or less. As shown in Figure 6.6,

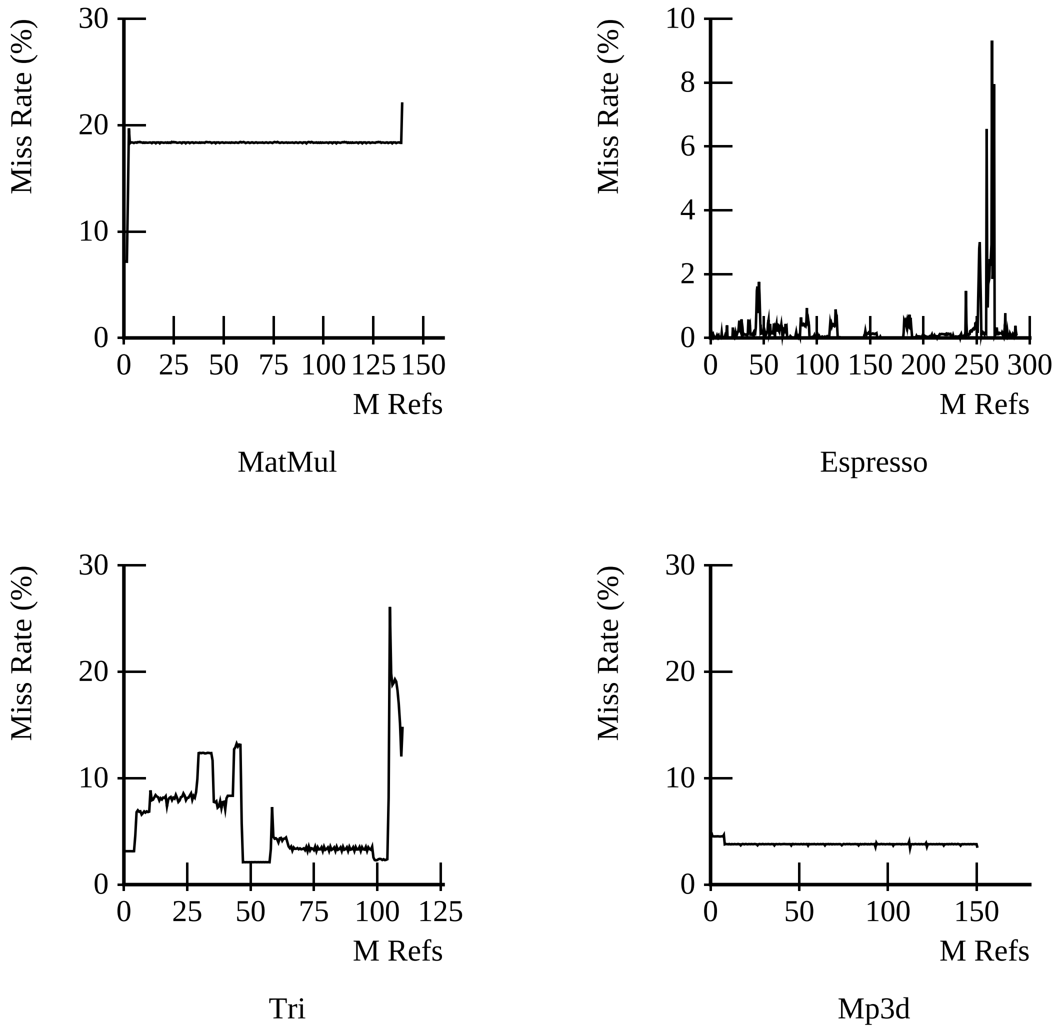


Figure 6.6: True miss rate over time in the sequential applications. The time axis is numbered as a cumulative count of program references.

these applications have memory behavior which is fairly uniform over time, and for these types of programs, fewer samples are required to capture the “typical” memory behavior.

The third program, Tri, shows large relative fluctuations in error for less than 20 samples, but achieves accuracy within 10% relative error when using more samples. Viewing Tri’s true miss rate over time in Figure 6.6, we see that it fluctuates much more with time than that of the Mp3d or MatMul. Thus, more samples are needed to capture its behavior.

Finally, the fourth application, Espresso, shows relative errors greater than 20% even at 100 samples. Espresso has a very low miss rate, so even fairly small fluctuations in the absolute miss rate estimate show up as large relative swings. Since Espresso has very few known misses, it is more difficult for the samples to collect representative numbers of them and this can lead to large relative errors. The absolute error from sampling is, however, still quite small. Furthermore, while we include Espresso to understand sampling's accuracy limits on low-miss-rate applications, we also note that low-miss-rate applications typically do not require memory system tuning, so MemSpy's accuracy on them is less relevant than on applications with higher miss rates.

To summarize, the presence of phases in the memory behavior of a program (such as Tri) mandates the use of more samples to accurately represent its memory behavior. Here MemSpy can leverage off users' knowledge of how their programs behave. When programmers know that their application has memory behavior which is fairly uniform over time, they can request that a reduced number of samples be taken, for higher performance. While users do not want *full* control of the sampling setup, directives like these allow them to speed up the tuning process in specific cases. Furthermore, since tuning is iterative, users can choose to have only a few samples collected in early runs, and then simulate a higher fraction of references as they move to more detailed tuning.

6.3.2 Accuracy vs. Cache Size

As previous studies have shown [KHW91, WHK91], the accuracy of miss rate estimates is also a function of the cache size used. For our sequential benchmarks, Figure 6.7 shows the relative error in cache miss rate estimates for 16KB, 128KB, and 1MB caches at a 10% sampling ratio, with a sample length of 0.5M references. In the figure, these errors are shown relative to the true miss rate *during the sample*. Note that within each sample, the best we can hope to achieve is to re-create the true miss rate of that sample. The error with respect to the overall miss rate may be slightly more or less than the error shown here, depending on how representative the sampled regions are of the behavior of the full trace.

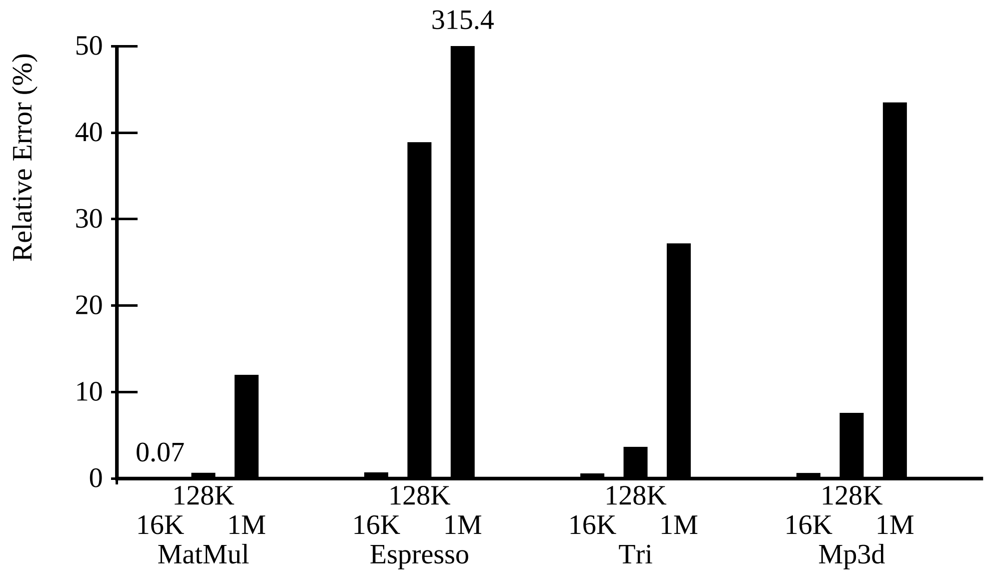


Figure 6.7: Relative sampling error as a function of cache size for a sampling ratio of 10% and a sample length of 0.5M references.

For the 16KB and 128KB caches, the relative errors are all less than 10%, with one exception. Espresso's miss rate has a *relative* error of nearly 40% for the 128KB cache. However, its absolute miss rate (0.144% within the sampled regions) is so small that these large relative errors are neither surprising nor problematic, from a performance debugging point of view. Since Espresso's memory stall time is a small fraction of its total execution time, even the sampled version of MemSpy will accurately point out that Espresso does not have any substantial memory performance bottlenecks.

With 1MB caches, the relative errors are higher – greater than 10% for all four of the applications. These larger relative errors are due to two factors. First, as the cache size increases, more references are needed to prime the cache state. This causes the number of unknown references to increase. Second, as the cache size increases, the application's cache miss rate generally decreases. This causes a decrease in the number of known misses. Equation 6.1 indicates that both of these effects tend to increase the size of the error bounds on estimated miss rate.

Note however, that despite the large relative error, the absolute errors remain quite small. Figure 6.8 shows a plot for the 1MB cache of both the true cache miss rate (stars)

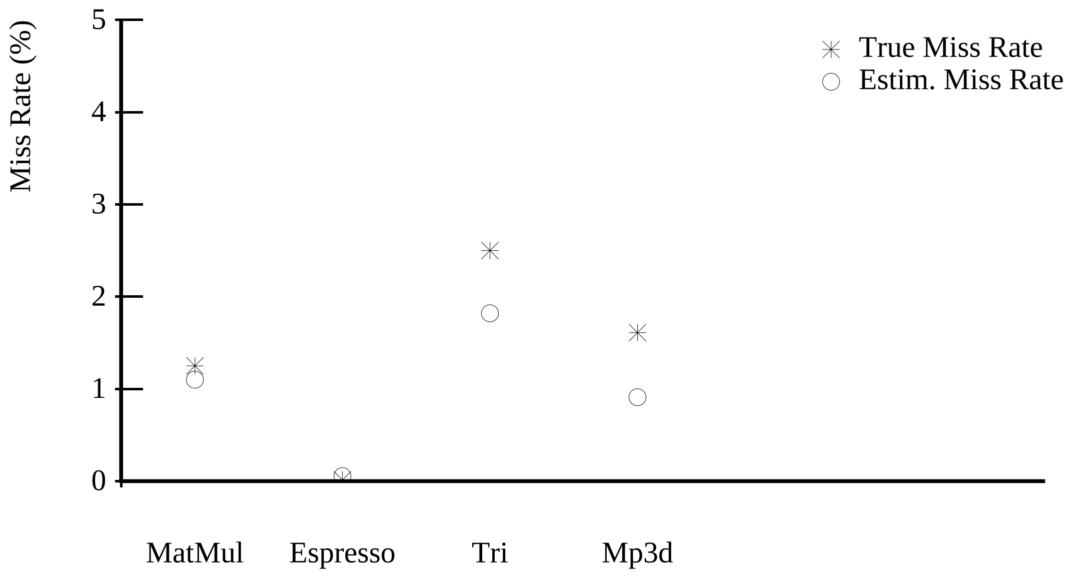


Figure 6.8: Absolute deviation between true and estimated cache miss rates for 1MB direct-mapped cache. Measured using a 10% sampling ratio and 0.5M references per sample.

and the sampled cache miss rate (circles). Absolute error of this magnitude (1% or less) may often be considered acceptable when tuning programs.

To improve these miss rate estimates for large cache sizes, we have two options. First, we can reduce the significance of unknown references in Equation 6.1, by lengthening each sample taken. Alternatively, we can improve our estimate of μ, the miss rate for unknown references. The following subsection examines the first of these issues. The second issue is more difficult in general. The model proposed by Wood et al. [WHK91] attempts to capture the behavior of unknown references but for our benchmarks, we have found that this method only slightly improves the accuracy of the sampling estimates, and requires significant extra processing.

6.3.3 Accuracy vs. Sample Length

Section 6.3.2 illustrates the fact that a single choice of sample length may not work effectively across a range of cache sizes and application behaviors. For larger caches, unknown references become significant, and one must use longer samples to mitigate their effect. This section studies the accuracy trends as we vary the length of samples taken.

In this section, we examine behavior within individual samples. For each sample, we compare the miss rate with an estimation of unknown reference behavior, to the true miss rate *within the sample*. Thus, in this section we are focusing on the error due to unknown references and do not account for the error due to non-representative samples. To gather the data shown here, we divide the trace into contiguous samples, and collect data for all of them, averaging the results. For example, in a program which makes 100M references total, a sample length of 5M references will lead to 20 data points which we average. This allows us to study more samples per application than if we restricted ourselves to a sampling ratio of, say one tenth, which would only yield 2 data points for that configuration.

Figure 6.9 shows relative errors in miss rate estimates versus sample length for the four sequential benchmarks each simulated with a 1MB cache. As expected, longer samples dramatically improve accuracy. At 8M references per sample, all applications have relative errors less than 10%. Absolute errors on these applications are all under 0.1%.

One can also examine how the sample length required for good accuracy varies with the cache size. For each of the benchmark applications simulated with a variety of cache sizes and sample lengths, Figure 6.10 plots the sample length required for 10% relative error (or better) versus cache size. Note that required sample lengths never exceed 8M references per sample, and are often significantly shorter.

For even moderately long running applications, 8M references per sample is not a prohibitively long sample length. To collect 30 samples of this size with a sampling ratio of one tenth, the application would need to have 2.4 billion simulated data references. If a simulated reference occurs once every 3 instructions, this is roughly 7.2 billion instruction cycles, or less than a minute and a half of execution time on a 100 MIPS machine. These requirements are not prohibitive, either in terms of the run length or the number of references required.

Finally note that unlike the error due to non-representative samples, the error due to unknown references that we have studied in this section can be *bounded*. These error

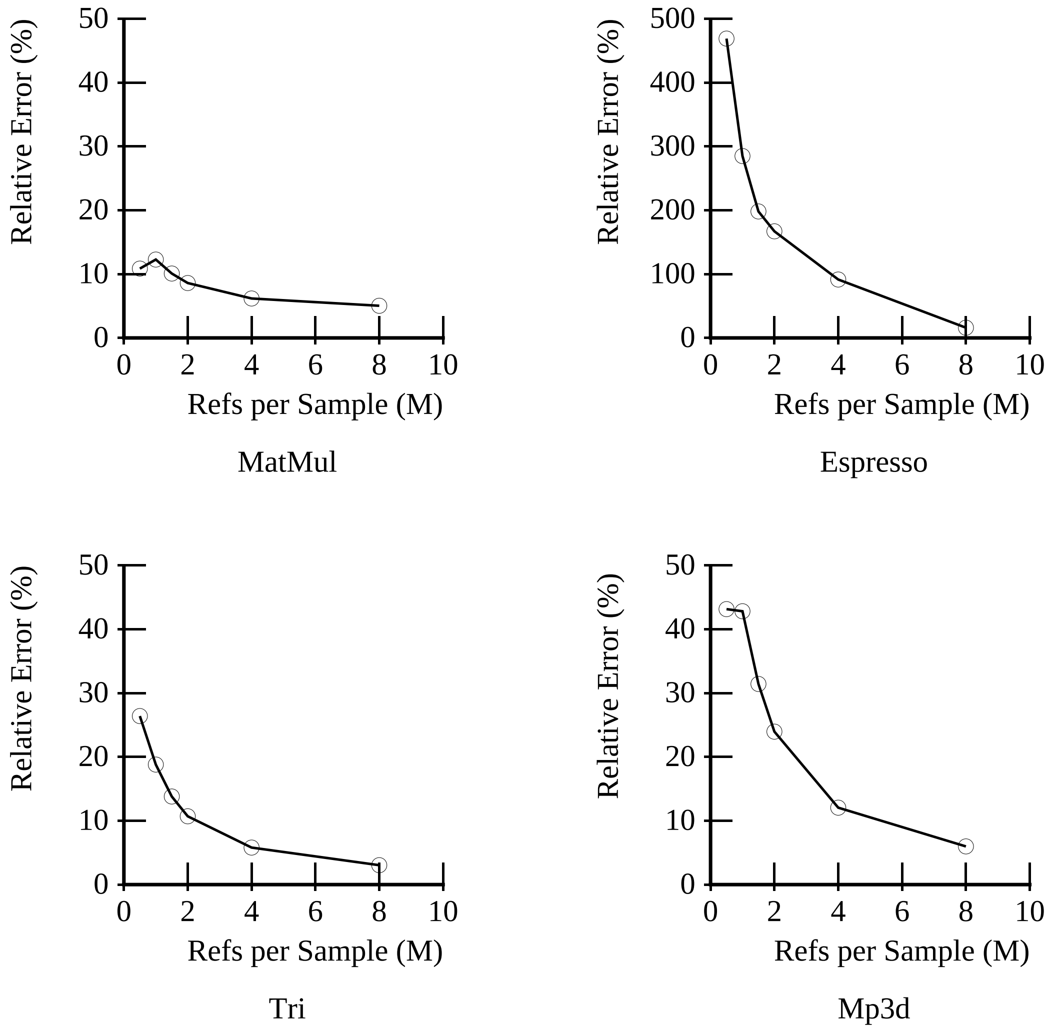


Figure 6.9: Relative error as a function of sample length for 1MB direct-mapped caches.

bounds are expressed in Equation 6.1 and are based on the fact that the mis-estimation of unknown references cannot exceed the total count of unknown references. Thus, if a tool presents users with error bounds in addition to its miss estimate, the users can evaluate whether the miss estimate is likely to be "close enough" for their purposes. When it is not close enough, the users can request longer samples on subsequent measurement runs.

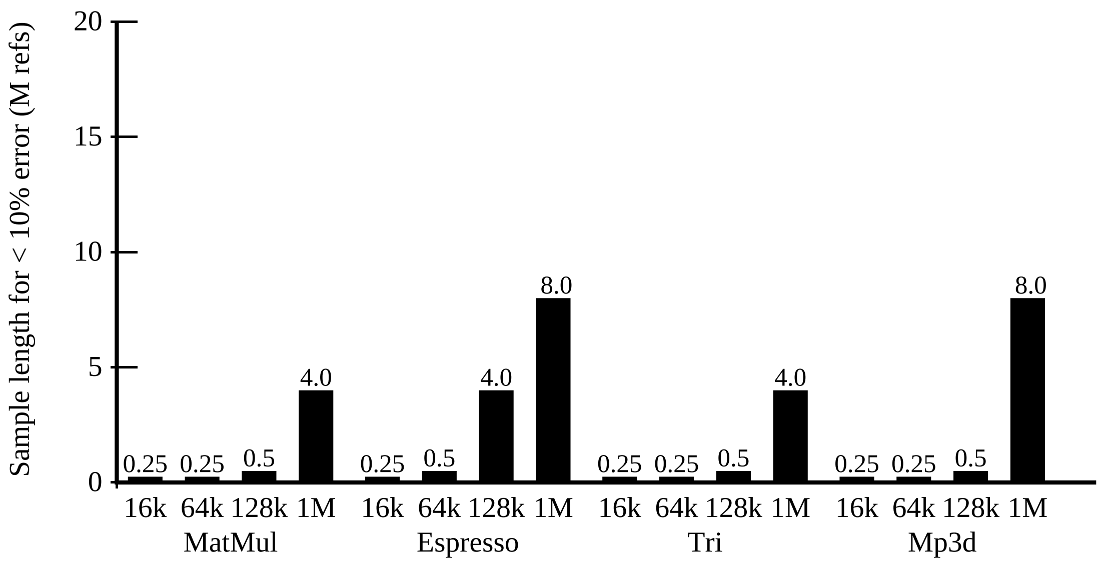


Figure 6.10: Sample length required to achieve less than 10% relative error, as a function of cache size.

6.4 Time Sampling in Parallel Programs

The previous section presented an analysis of accuracy issues arising when applying sampling to sequential benchmarks. Several of these issues (such as selecting the sample length and the number of samples to collect) continue to apply in the parallel domain. In addition, new issues arise in collecting accurate samples on shared memory multiprocessors. In this section we briefly summarize initial accuracy results for parallel benchmarks, and then present results on how this accuracy varies with the number of processors.

Table 6.4 gives accuracy results for our set of parallel benchmarks on a simulated machine with 16 processors, each with a 64KB cache. A 10% sampling ratio is used, and each sample is 3M references long. (A sample in this case is a contiguous group of references taken from the *interleaved* reference traces of *all* 16 processes.)

In general, the samples taken are quite accurate. The largest *absolute* deviation between the true and estimated miss rates during samples (columns 3 and 4 respectively) is only 0.3%. Water has a relative deviation greater than 10%, but as with Espresso in the sequential benchmarks, the absolute miss rate in this case is so low that for this application, the memory performance is unlikely to be the bottleneck, and such a deviation is acceptable.

Table 6.4: Estimated and true miss rates for parallel applications. Measured with 16 processors and 64KB direct-mapped cache per processor, using a 10% sampling ratio and 3M references per sample.

Appl.	Overall True Miss Rate	True Miss Rate During Samples (%)	Estimated Miss Rate During Samples (%)
MP3D	8.53	8.27	7.79
CHOLESKY	3.29	3.02	2.92
WATER	0.38	0.42	0.43
LOCUS	1.03	1.47	1.40

When moving to the parallel domain, two opposing trends will affect the accuracy of sampling as compared to the sequential domain. First, extra communication in the form of coherence traffic can lead to higher miss rates than a uniprocessor execution with the same total cache. One might expect that these increased miss rates would tend to decrease the sample lengths required for a given desired accuracy.

On the other hand, parallel machines typically have more total cache than sequential machines, proportional to the number of processors in the machine. Overall, a multiprocessor's larger total cache space can lead to (i) lower miss rates and (ii) a larger number of cache lines that need to be primed. Both of these effects tend to increase the sample lengths required for a given accuracy.

Note that throughout the discussion in this section, the size of problem being solved is held constant. Thus a particular data set is more likely to fit in the machine's caches in a 16 processor run, than in a 1 processor run, since it has more cache. In another type of scaling, *time-constrained scaling*, the problem size is increased along with the number of processors used, such that the execution time for the problem remains roughly constant regardless of the number of processors used. In time-constrained scaling, the cache misses generally scale with the number of processors, rather than generally decreasing.

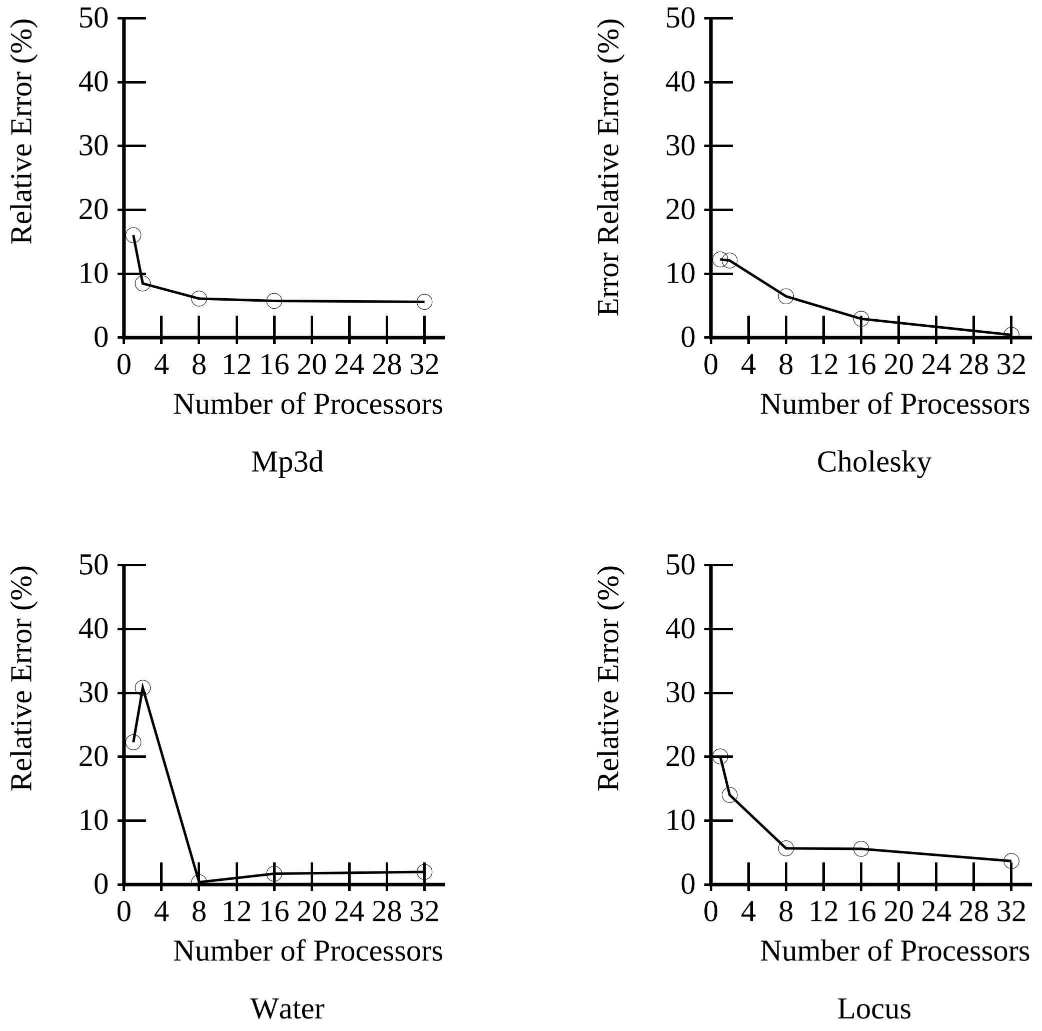


Figure 6.11: Relative error as a function of the number of processors, while holding total cache constant at 1MB. Measured using a 10% sampling ratio and 3M references per sample.

Table 6.5: Relative error versus number of processors, where each processor has a 64KB direct-mapped cache. Measured using a sampling ratio of 10% and 3M references per sample. The error is presented relative to the true miss rate within samples.

Application	Relative Miss Rate Error (8 procs) (%)	Relative Miss Rate Error (16 procs) (%)	Relative Miss Rate Error (24 procs) (%)
MP3D	3.2	5.8	8.0
CHOLESKY	1.0	3.1	4.4
WATER	0.04	1.8	3.4
LOCUS	3.7	4.5	6.2

Studying sampling under this sort of scaling should lead to even more favorable accuracy trends with increases in processors than the ones we report here.

Our goal is to understand the effects of the opposing trends on sample length with increases in processors. To accomplish this, we first isolate the effect of coherence traffic in parallel runs. We present data in which the total cache in the machine is held constant, and we scale the number of processors. In contrast to real machines which scale memory as they scale processors, this study allows us to eliminate effects stemming from scaling the cache size. Figure 6.11 gives a summary of the estimated cache miss rates obtained for identical problems running on machines where the number of processors is varied, but the total cache in the machine is held constant at 1MB. Thus, we simulate a 1 processor machine where the processor has a 1MB cache, a 16 processor machine where each processor has a 64KB cache, and so on. When holding total cache size constant, a 16 processor run is much more accurate than a 1 processor of the same problem. This is due to the increased miss rates of the multiprocessor run. Thus, coherence traffic in multiprocessor simulations reduces the need for long samples.

In real systems, as the number of processors in a system increases, the total cache will usually scale linearly. This generally means that more priming references will be needed to fill the cache. Table 6.5 shows the accuracy trends as we vary both the number of processors being simulated and the total cache size of the machine. That is, in these simulations, each processor has its own 64KB cache, regardless of how many processors are in the system. In all cases, the miss rate estimate for 24 processors has a slightly higher relative error than that for 8 processors. This indicates that the effect of adding more lines to prime outweighs the effect of increased coherence traffic that was previously discussed. Thus, when processors and cache are added to the system proportionally, samples need to lengthen in order to maintain a given accuracy. However, the presence of coherence traffic means that the samples can grow sub-linearly.

Overall, the conclusion to be drawn from the results in this section and the preceding results in Section 6.3 is that the accuracy of trace sampled cache miss rate estimates can be quite good, even at fairly aggressive sampling ratios such as 10%. The crucial issues that remain to be discussed in the following sections are (i) the effect of trace sampling on particular MemSpy performance metrics and (ii) the effect of trace sampling on MemSpy's execution time overhead.

6.5 Sampling-Induced Error in MemSpy Metrics

Until this point, we have presented our results in terms of their effect on the overall cache miss rate. However, MemSpy presents more detailed statistics than simply the cache miss rate. In this section, we discuss the sampling implementation's accuracy at estimating the percentages of memory stall time incurred in different procedure-data pairings, as well as its accuracy at estimating the causes of cache misses.

6.5.1 Memory Stall Breakdown by Procedure-Data Pairings

An important MemSpy feature is the ability to break down memory stall time by pairings of procedure and data objects. In this way, MemSpy allows the user to see which data structures and procedures are most responsible for a program's poor memory performance. The potential pitfall here is that as we subdivide statistics to view procedure-data subsets, we may be more prone to non-representative samples, because each procedure-data pair will have fewer simulated references on which to estimate behavior.

A key attribute of sampling, however, is that the procedure-data pairs which have the largest impact on program performance will also be the easiest on which to obtain accurate statistics using sampling. The reason for this is that the statistics bins with a large impact on performance will be responsible for a large number of references and cache misses. First, this makes it more likely to capture representative numbers of events for the bin during sampling. Second, with a large number of misses, the effect of unknown references (U) in Equation 6.1 will be small compared to the effect of known misses (M_k). This reasoning is analogous to the motivation for program-counter sampling as used in Gprof, Quartz, and Mtool [GKM83, AL90, GH91b].

To gauge sampling's accuracy within procedure-data pairings, we collect bottleneck information from a full MemSpy run, and compare it to that from a sampled MemSpy run. For sequential applications, we once again simulate a 128KB cache, with a sample length of 0.5M references and a sampling ratio of one tenth. For parallel applications, we simulate 16 processors with a 64KB cache per processor, with a sample of length 3M references and a sampling ratio of one tenth. (As before, this sample length refers to the number of references taken from the full (not per-processor) trace.)

For both sequential and parallel benchmarks, we find excellent agreement between the sampled and true statistics. For two of the sequential applications studied, MatMul and Mp3d, the orderings of bottlenecks reported by the sampling version exactly match the orderings for the true version for all bins incurring 2% or more of the total memory stall time. As an example, Table 6.6 shows the excellent agreement in bottleneck orderings and percentage stall time breakdowns for MatMul. For the two main bottlenecks, memory stall time is accurate to within a tenth of a percent. In a third application, Tri, the bottleneck orderings match exactly, except for small reorderings in procedure-data pairings incurring less than 10% of the stall time.

Only in Espresso, which has an extremely low miss rate and essentially no memory bottlenecks, are there poor estimates of procedure-data bottlenecks. Here, the statistics bin responsible for 20% of the cache misses is mis-estimated to be responsible for only 3% of the misses. This causes it to be ranked ninth instead of first. However, because Espresso has no significant bottlenecks, it is not representative of the sort of application expected

Table 6.6: MemSpy memory bottleneck listing for MatMul.

Data Object	Procedure	Memory Stall Time (%)	
		True	Sampled
Matrix Y	block	78.3	78.4
Matrix Z	block	18.6	18.6
Matrix X	block	2.1	1.7

to be tuned using MemSpy. When an application truly has memory bottlenecks, we have found that sampling is able to estimate their significance with reasonable accuracy.

For the parallel applications, the results were also good. We omit Water from this discussion since, like Espresso, it has a very low miss rate. For the other three, all procedure-data pairs incurring 10% or more of the stall time are correctly ordered. Mp3d is sampled with no reorderings. Cholesky and LocusRoute have reorderings in relatively unimportant procedure-data pairs.

For example, Table 6.7 shows the bottleneck orderings for Cholesky. We see very good agreement both in bottleneck orderings, and in percentage stall time breakdowns, with only insignificant reorderings in procedure-data pairs with small program effects. The top three bottlenecks in the code are identified as such. Further down the list, the bottleneck reorderings occur in variables that the programmer is not likely to tune anyway. For example, the stall time contribution of the vector `v` in `TriangularSolve` is significantly underestimated. However, since this statistics bin is only responsible for 6.4% of the total memory stall time, it is not a significant bottleneck anyway.

6.5.2 Causes of Cache Misses

MemSpy also presents a breakdown indicating the causes of a bin's cache misses. Unfortunately, reproducing these statistics with sampled simulation data can be less accurate than simply estimating the cache miss rate. The reason is that estimating the causes of cache misses requires priming information about the previous activity for a particular *memory line*. Contrast this with estimating the cache miss rate, which requires priming

Table 6.7: MemSpy memory bottleneck listing for Cholesky.

Data Object	Procedure	Memory Stall Time (%)	
		True	Sampled
Vector v	DoEightModify	31.9	41.0
Vector v	ScatterUpdate	21.5	14.8
M.row	ScatterUpdate	8.7	11.3
Vector v	TriangularSolve	6.4	0
Vector v	ModifyColumn	5.8	0
Vector v	DoFourModify	5.5	5.5
Vector storage	DoEightModify	4.1	7.2
Vector v	FillElements	3.2	0
Vector v	DoOneModify	1.9	0.7
Vector storage	ScatterUpdate	1.5	1.8

information about a particular *cache* line. While one can prime a cache line with references to several different memory lines, priming state information for causes of cache misses requires references to one particular memory line.

In general, we have found that for important bottlenecks, cache misses occur often enough that the information on causes of cache misses is able to indicate the nature of the problem. Difficulties arise when viewing the cause of misses in statistics bins that incur fewer references. In these cases, more of the misses may be due to unknown causes. Since the users can see what fraction of miss causes are unknown, they can determine whether the cause of miss information is sufficiently accurate or not. As the users begin to to fine tune their code, they may choose to simulate a higher fraction of references, to detect the more subtle performance bugs.

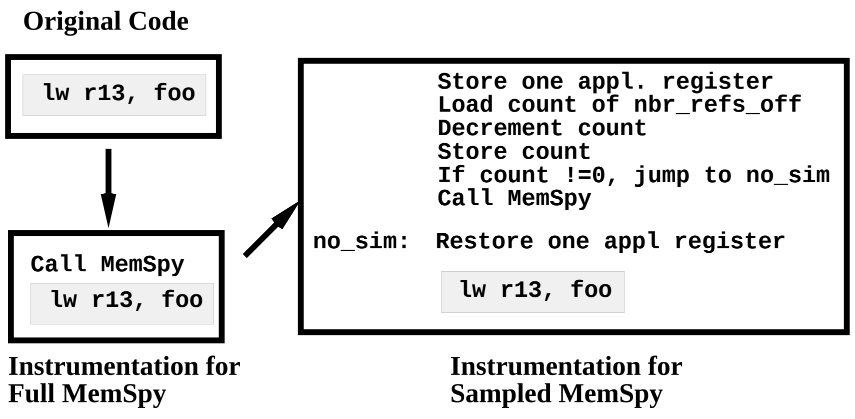
6.6 MemSpy Performance Using Sampling

Ultimately the goal of implementing sampling within MemSpy is to reduce the tool's execution time overhead. Having presented results on the accuracy of a sampling version of MemSpy, we now describe the implementation of sampling and its performance.

6.6.1 Implementation

An important goal in implementing sampling within MemSpy is to avoid most of the per-reference overhead incurred in the baseline and hit bypassing versions of the tool. This overhead includes context switching into the simulator, as well as performing the cache hit check. Thus, it is not sufficient to simply turn off statistics when sampling indicates simulation is off. Rather, our implementation should also circumvent the context switching and hit check overhead as well.

To accomplish this, we modify the normal MemSpy assembly time instrumentation of memory references. Figure 6.12 illustrates the original and new instrumentation. In addition to the usual call to the MemSpy memory simulator, additional instrumentation is added. In this extra instrumentation, a sampling counter is decremented and checked against zero to see if simulation is currently on or off. This decrement-and-check introduces an overhead of 6 instructions per instrumented memory reference. If simulation is off, control branches around the memory simulator procedure call. If simulation is on, there is additional overhead to save more application registers and perform the cache hit check as described in Section 5.3.2 for hit bypassing. Thus, we expect this implementation to offer a modest performance improvement on cache hits (which could be bypassed anyway) and a large performance improvement on cache misses. Section 6.6.3 discusses possible further improvements on this approach.


Original Code
Instrumentation for Full MemSpy
Instrumentation for Sampled MemSpy

```
lw r13, foo
```

```
Call MemSpy
 lw r13, foo
```

```
         Store one appl. register
         Load count of nbr_refs_off
         Decrement count
         Store count
         If count !=0, jump to no_sim
         Call MemSpy

no_sim:  Restore one appl register

          lw r13, foo
```

Figure 6.12: Inlined assembly code for sampling.

6.6.2 Performance

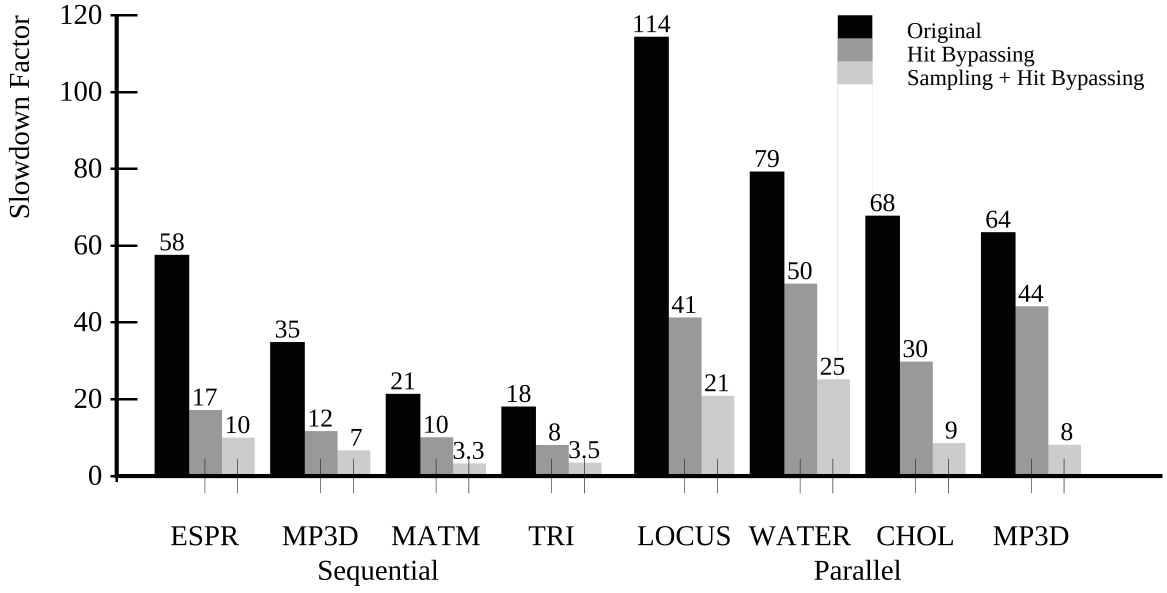


Figure 6.13: MemSpy performance overhead using sampling and hit bypassing.

Figure 6.13 gives the simulation overhead using the time sampling approach, and compares this overhead to that of the previous MemSpy implementations evaluated in Chapter 5. The performance numbers are shown for a sampling ratio of 10%. As with previous data in this chapter, the sequential applications are measured for a 128KB cache using samples of 0.5M references each, and the parallel applications assume a 64KB cache per processor and samples of 3M references each.

For the sequential benchmarks, the use of sampling in addition to hit bypassing results in a 1.7 to 3.1 fold performance improvement over hit bypassing alone. For the parallel benchmarks, the improvements are slightly larger, ranging from 2 to 5.4. For several reasons, the improvement does not reach the potential factor of ten allowed by a 10% sampling ratio. First, sampling targets only reference simulation, not simulation of other events such as synchronization events and procedure call and return logging. In fact, procedure logging, which represents only a small fraction of the overhead in the baseline implementation presented in Chapter 5, is now responsible for over half of the overhead time. Second, the additional benefits from sampling are greatest in applications which benefit the least from hit bypassing. For this reason, the observed improvement is larger in applications like parallel Mp3d; since Mp3d has a large number of write misses, fewer references can be bypassed using hit bypassing, and thus more speedup remains to be achieved using sampling.

6.6.3 Discussion

At this point, to further reduce the overhead of this approach, one could attempt to reduce (i) procedure logging overhead or (ii) sampling instrumentation overhead. Procedure logging overhead could be reduced to some degree through simple optimizations of the logging code; however, procedure events cannot be sampled as with memory events, since procedure calls and returns must occur in matched pairs to maintain the state of the stack.

The second source of overhead, sampling instrumentation, is defined as the additional instructions needed to switch simulation on and off. In the implementation presented here, this overhead is primarily the six additional instructions per memory reference that allow control to branch around the memory simulator when simulation is OFF.

To avoid this overhead, we have also implemented a preliminary version of a more aggressive approach. In this new approach, control alternates between two different versions of the application. One version is fully instrumented to simulate all memory references, and the second version is instrumented only to log synchronization and procedure entries and exits, not memory references. The program executes in the fully instrumented version when simulation is turned ON, and then switches to the minimally instrumented version when simulation is turned OFF. These mode switches are determined by virtual timer interrupts using the UNIX `setitimer` call.

The performance benefits of this approach thus far have been moderate at best. It offers no better than a 20% speedup over the more straightforward sampling implementation for the benchmarks presented here. That reasons are first that it is still subject to the overhead of procedure event logging. Second, it has double the application code size (because there are two versions of all application code), which has a detrimental effect on instruction cache behavior. However, further work on efficiently implementing the mode switch in this approach, as well as improving procedure logging in general, may make this an attractive alternative to our current implementation.

6.7 Discussion

To restate our conclusions thus far, this chapter has shown that time sampling methods allow accurate estimations of cache behavior with significant improvements in simulation performance. For moderate size caches, cache statistics in sequential and parallel programs can be estimated with less than 10% relative error, even when simulating only one tenth of the full application reference trace. For the applications studied, this leads to performance improvements of 1.7 to 5.4 fold. At this point, we touch on several additional details and extensions: (i) the issue of periodic application behavior and (ii) the incorporation of other forms of sampling.

6.7.1 Avoiding Periodic Behavior

One of the pitfalls of time sampling is the possibility that the samples will repeatedly coincide with periodic application phases, resulting in a cache miss estimate that is not necessarily representative of the program as a whole, even when gathering a large number of samples. While we have not implemented it here, a straightforward solution is to use samples whose length varies randomly around a chosen mean, with a specified variance. This would introduce some randomness into the sampling interval, to make it less likely to repeatedly coincide with a particular phase of the application.

6.7.2 Incorporating Other Forms of Sampling

To this point, this chapter has only treated issues related to time sampling. Other forms of sampling, such as set sampling or processor sampling in parallel programs, are also available.

In set sampling [KHW91], a fixed collection of cache lines (or *sets* of cache lines in caches with set associativity greater than 1) are simulated for the entire program run. Set sampling is promising from an accuracy standpoint because it is not subject to error due to unknown references as time sampling is. However, set sampling is subject to inaccuracies if the lines chosen for simulation are not representative of the whole program behavior. That is, lines in the cache that are “hot spots” (i.e., have more activity than other lines) may affect the cache statistics more than less active lines in the cache. For this reason, the accuracy of set sampling actually improves as the number of lines in the simulated cache increases, since a fixed sampling ratio will increase the number of lines being simulated, and can decrease the extent to which any one line affects the cache statistics.

In processor sampling [CD93], the trace is filtered to only examine the effects of a subset of the processors in the machine. Events occurring at processors outside of this subset are only simulated to the extent that they may affect (through invalidations for example) the processors in the studied group. This approach might also be useful for improving the performance of simulations for large-scale parallel machines.

Approaches that combine set, time, and processor sampling may allow one to take advantage of particular accuracy characteristics of each to achieve the best possible performance for a given accuracy. For example, in an approach combining set and time sampling, one could reduce set sampling’s susceptibility to hot spotting by using the technique of *time varying sampled sets*. This technique recognizes that the main weakness in set sampling is that the sampled sets are constant for the entire run. If the group of lines being simulated were changed several times during the run, the particular choice of lines to sample would become less significant. However, whenever the new group of lines came into use, these new lines would have unknown state (and thus unknown references, similar to time sampling). Thus, when combining set and time sampling one could take advantage of the fact that the first references to each set in each time sample will be unknown. Using this, one could redefine the *groups of lines* to be sampled at the beginning of each *time sample*. In this way, the same references would be unknown for both set and time sampling. Thus, one could reduce the susceptibility of set sampling to hot spotting, without introducing any *additional* unknown references.

6.8 Chapter Summary

Overall, this chapter has shown that reference trace sampling is an effective and accurate technique for optimizing MemSpy's performance while retaining good accuracy.

We first presented results on the accuracy of time sampling in sequential benchmarks. For small and moderately sized caches, miss rates can be estimated with very small absolute deviations ($< 0.5\%$) and relative deviations of 10% or less. These estimations can be achieved with sample lengths of only 0.5M references or less. Thus, gathering 20-30 such samples at a sampling ratio of 10% means that applications need only 100M-150M references in order to take advantage of sampling. For good accuracy when simulating larger caches ($\geq$ 1MB), longer samples are required– 4M references or more. However, this still allows for aggressive sampling ratios on many applications. In parallel simulations, slightly longer samples are required to account for the fact that parallel machines generally have more total cache. However, the presence of coherence traffic mitigates this trend. We find a sublinear growth in required sample length as processors are added.

Finally, we examined the performance of MemSpy with time sampling implemented in addition to the optimizations described in Chapter 5. Overheads for sequential applications dropped to the range of 3 to 10 fold. Overheads for parallel applications vary from 8 to 25. With these overheads sequential programs with runtimes of around a minute can be run with MemSpy in roughly 10 minutes or less. Our experiences have shown that programmers are often happy to tolerate moderate overheads, given the detail and utility of the statistics presented. MemSpy's performance approaches that for tools such as Mtool [GH91b] and Gprof [GKM83], despite the fact that MemSpy provides much more detailed statistics.

Chapter 7

Related Work

Previous chapters have discussed MemSpy's contributions in several areas. This chapter relates these contributions to previous research in similar areas. In Section 7.1 we present representative examples of other performance monitoring tools and contrast them to the detailed, data oriented approach embodied by MemSpy. Following this in Section 7.2 we discuss related work in the field of reference trace sampling.

7.1 Performance Monitoring Tools

Many performance monitoring tools have been developed, with a wide variety of tuning goals and presentation styles. An extensive but nonetheless incomplete list would include the work described in [AL90, AG88, AGS90, DBKF90, Gol92, GH91a, GH91b, GKM83, LW92, GGJ$^+$89, SR85, LSV$^+$89, MRA$^+$89, MGA92, Mil88, MCH$^+$90, NSS$^+$88, Smi91]. These tools embody a variety of approaches. Some, like Gprof [GKM83] and Pixie [Smi91], give high level statistics on sequential program behavior, without attempting to isolate memory bottlenecks. Others such as Quartz [AL90], PIE [LSV$^+$89], and IPS2 [MCH$^+$90] are high level tools targeted for parallel programs. They provide statistics on application synchronization behavior in addition to computational statistics; they still, however, provide no specific support for memory behavior. Tools like Mtool, SHMAP, and CPROF do provide memory statistics, at varying levels of detail. The following subsection describes Gprof as an example of a high-level computation oriented tool.

Following that we describe each of the memory oriented performance monitoring tools: Mtool, SHMAP, and CPROF. We compare these four tools to MemSpy based on the information they present, their runtime overhead, and their possible limitations.

7.1.1 Gprof

Gprof [GKM83] is a commonly used execution profiler for sequential programs.

Information Presented Gprof's prime metric is an assignment of processor time to procedures in the code. Gprof presents procedures ordered by the fraction of total time charged to them. It also presents hierarchical information which attributes time spent in activations of *callee* procedures to the procedures that called them. This execution time profile of a program's procedures offers a high level view of which procedures have the greatest potential for optimization.

The main limitation in Gprof's statistics is that they do not distinguish between "useful" computation time and time spent in memory stalls or other delays. The tool simply ranks procedures by the total time spent in them, and provides no help in locating memory system bottlenecks.

Runtime Overhead Gprof, offering simple statistics, has a low execution overhead. At compile time it instruments the program to generate a dynamic call graph as it executes. This requires instrumentation in the prologue of each monitored procedure, similar to MemSpy's procedure logging. In addition during the program run, it uses program counter sampling to periodically determine the current state of the processor. This lightweight monitoring can give a statistical estimate of the time spent in each of the program procedures. The overhead of this type of monitoring is most dependent on the frequency of procedure calls in the code. The authors do not report on tool overhead, but given the relatively low complexity of this approach the overhead is expected to be roughly a factor of two or less.

7.1.2 Mtool

Mtool [GH91a, GH91b] is a system specifically designed to detect memory bottlenecks in both sequential and parallel programs.

Information Presented Mtool's basic performance metric is the difference between a program's execution time with the actual memory system, and the execution time of the same code with an ideal memory system. This difference is the amount of execution time for which the processor was stalled due to memory system delays. This information is presented for loops and procedures within the program.

While Mtool is useful for focusing attention on the primary memory bottlenecks in the code, it provides no statistics on the specific application behavior (cold-start misses, interference, sharing, etc.) responsible for the problems. Furthermore, since Mtool's output is procedure and loop oriented, it does not provide insight into which data objects are responsible for the poor memory system behavior. This limitation is especially problematic when several data structures are all accessed, for example, on the same source code line as was the case for the MatMul example in Chapter 3.

Runtime Overhead Mtool performs monitoring using basic block counting. This is similar to the approach used by Pixie [Smi91]. However, Mtool has been optimized so that it uses profiling information to place counters only on a subset of edges in the basic block graph. It then post-processes this information to determine execution counts for all edges. With this optimization Mtool reduces counter overheads to roughly 10% on average. So for the minimal counter overhead (and intrusion), Mtool requires 1 profiling run and 1 monitoring run to generate statistics. This leads to an overhead slightly over a factor of two. Subsequent monitoring runs without code changes have roughly a factor of 1.15 overhead. However, when the code changes, the programmer must either reprofile the code, or run the risk of high inaccuracy.

Note that MemSpy's overheads for sequential benchmarks (factors of 3 to 8) are quite competitive with Mtool's overheads, and MemSpy offers much more detailed information. In moving to the parallel domain, Mtool's overheads increase only slightly though, while MemSpy's increase a great deal. These results illustrate the tradeoffs in tools between detailed output and low execution time overhead. For initial program analysis, Mtool, Gprof, or Pixie are natural choices. On the other hand when programmers cannot diagnose a performance bug based on such high level output, MemSpy offers a more detailed, but still acceptably fast, alternative.

7.1.3 SHMAP

A significantly more detailed tool for studying memory referencing patterns is SHMAP [DBKF90].

Information Presented This system annotates FORTRAN programs, collects memory reference traces, and produces an animated graphical picture of references to the program's main data objects. That is, the tool displays either a representation of the cache, or a representation of a matrix in the program, and as trace references are processed, the corresponding points in the cache or matrix display are highlighted. Over time, the animation may give users clues about poor application memory behavior.

While SHMAP is useful for detecting patterns in references to array data objects, it offers little help for references to more complex data structures, such as lists and trees. SHMAP also offers little summary information about the program's behavior; miss rates are computed only on a per-processor, rather than per-data-object or per-procedure basis, and the user must glean information on cache replacements by carefully examining the animation. For long running simulations of program execution, watching the animations and discerning patterns may become quite tedious.

Runtime Overhead Like MemSpy, SHMAP is a simulation-based tool. The authors do not report specific performance overheads for their approach. However, their approach does not make use of MemSpy-style optimizations such as hit bypassing or reference sampling. In addition, SHMAP records a full trace file of memory events, rather than simply tracking counts which are written out to a summary file at the end of execution. For these reasons, we expect SHMAP has significantly higher runtime overheads than MemSpy.

7.1.4 CPROF

Finally CPROF [LW92] is a uniprocessor memory performance monitoring tool developed at the University of Wisconsin.

Information Presented Developed later than MemSpy, CPROF incorporates several similar features. Most notably CPROF provides data oriented statistics, and it presents information on the causes of cache misses. The authors lend further support to our argument that detailed, data oriented statistics are crucial for memory system performance monitoring, by presenting several case studies in which they tune the memory behavior of SPEC [SPE89] benchmarks. In these examples, the ability to focus on the behavior of particular data structures in the code was key to understanding the performance bugs.

Runtime Overhead The authors of [LW92] report no data on the runtime overhead of CPROF. However CPROF annotates code for memory simulation using the QPT system [Lar93] which reports base overhead factors of 1.4 to 12.3. These base overheads include only the time to generate a compressed basic block “witness” trace. This trace must then be post-processed and simulated to generate the memory statistics, and the memory simulation time and bin search time is expected to be similar to MemSpy’s. Thus, based on MemSpy’s initial performance in Chapter 5, memory simulation and bin searches should add roughly factors of 12 to 40 to the total CPROF overhead. Thus sequential CPROF overheads could be roughly estimated at factors of 14 to 52. MemSpy currently has lower overheads, but as one would expect many of the optimizations used by MemSpy could also be applied to CPROF.

7.2 Reference Trace Sampling

MemSpy’s overheads are optimized in part using reference trace sampling. By sacrificing small amounts of accuracy, sampling allows for substantial performance gains. Some aspects of memory reference trace sampling have been studied before, but primarily in the context of architectural studies. The following paragraphs contrast this previous work with the studies presented in this dissertation.

Laha et al. [LPI88] studied the accuracy of memory reference trace sampling using caches that were 128KB in size and smaller. Overall, their study concludes that sampling techniques allow accurate estimates of cache miss rates. However, the results in their paper were presented for sample lengths of 60,000 references every 100,000 references, or a sampling ratio of 0.6. Sampling ratios in this range offer very small performance benefits.

Laha et al. also propose a technique for handling unknown references which is based on the notion of primed sets. Statistics are not gathered for a particular cache line or set until references to the set are known references. This method becomes problematic with large caches. For large caches with the relatively short sample lengths used in this work, the vast majority of sampled references are unknown. Thus, miss rate estimates are made based on very few references. Furthermore this technique looks only at the behavior of known references, and systematically excludes unknown references. Since unknown references have higher miss rates in general than the known references, this technique will tend to produce optimistic cache miss rates. For this reason, MemSpy does not use this approach.

To overcome such limitations posed by unknown references, Wood et al. [WHK91] developed a model for estimating the miss rates of unknown references in sampled traces. Their model relies on developing statistical averages of the times between successive cache misses to individual cache lines. They found that their model predicted the behavior of unknown references better than several previous methods [LPI88, Sto90]. However, when cache misses are less frequent, this model becomes significantly less accurate. Unfortunately, it is precisely in these low miss rate applications that the behavior of unknown references becomes important. (At higher cache miss rates, the number of unknown references is negligible when compared to known misses.)

In later work, Kessler et al. [KHW91] studied trace sampling for large secondary caches of 1MB to 16MB, and with sampling ratios down to 1/10. For their benchmarks, they noted that unknown references at the beginning of a sample can dominate known misses, even using the model proposed in [WHK91]. This leads to inaccurate cache miss rate estimates. We believe this large inaccuracy arose in part because the benchmarks they used had few cache misses[1], and so for these benchmarks, the total number of known misses in each sample was often very small. For applications with larger miss rates (the type we expect to encounter with a performance tuning tool) time sampling performs well, and is less susceptible to hot-spotting than is set sampling.

7.3 Chapter Summary

This chapter has related previous work in the areas of performance tools research and reference trace sampling to the results presented in this dissertation. First, we discussed the features of four tools that are representative of different approaches to performance monitoring. In fact, one can consider tools in terms of a spectrum spanning various levels of detail. The runtime overhead of these tools is generally inversely related to the detail provided by the tools. In MemSpy we have selected a tradeoff point along this hierarchy where we provide very detailed statistics at overheads which are competitive with other less detailed approaches.

We then discussed related work studying reference trace sampling. Previous work has focused mainly on the accuracy of trace sampled results; little has been reported about the actual performance benefits of implementing sampling within a simulation system. In contrast, our work presents both accuracy and performance results. In addition, we leverage off the specific context of performance tuning tools, with its more flexible tradeoffs between the accuracy of sampled results and the performance of the sampling system.

[1]Seven of the eight traces had misses per instruction (MPI) values less than 0.003, while one had an MPI of roughly 0.02. [KHW91] does not report cache miss rates directly. With a cache miss latency of 20 cycles, an MPI of 0.003 corresponds to spending only 6% of program runtime in memory stalls.

Chapter 8

Conclusions

Both sequential and parallel applications are currently facing a growing gap between processor and memory speeds, and performance lost due to memory stalls can substantially limit overall application performance. Despite this trend, performance monitoring tools have lagged in providing support for identifying and characterizing memory bottlenecks. This dissertation has examined the issues inherent in tuning program memory bottlenecks. It presents our arguments on what kinds of information are useful, how to gather such information efficiently, and how to present such information in ways that isolate and highlight program memory bottlenecks. These ideas are embodied in our implementation of the MemSpy performance monitoring tool.

8.1 Contributions

In the face of a growing processor–memory performance gap, new techniques are warranted, both to efficiently gather detailed information on program memory performance and to effectively organize and manage the often large volume of information collected. This dissertation has made contributions in both of these categories, as outlined below.

Data Oriented Statistics

The first contribution of this work is in introducing the concept of presenting performance statistics in terms of familiar program data structures. Orthogonal to previous code oriented statistics, data oriented statistics form an important new dimension in viewing and analyzing program behavior. Together, data and code oriented statistics are a powerful approach for understanding and tuning memory performance. Despite the natural connection between an application's memory performance and its data structures, no previous tools had offered general support for data oriented statistics.

Heuristics for identifying program data to monitor, and for aggregating statistics for similarly-used data structures are a crucial component in implementing data oriented statistics. Our heuristics attempt to group statistics into "natural" presentation granularities. To do this we note that in most programs, ranges of memory allocated at a particular point reached by identical call paths are likely to be used similarly. Taking advantage of this fact, MemSpy presents its data oriented statistics in terms of aggregations of such memory allocations.

Chapter 3's matrix multiply case study highlighted the usefulness of these data oriented statistics. Here data oriented statistics were key in understanding that the poor memory system behavior was due to self-interference within a single matrix, and not due to cross-interference between matrices. Without such statistics even this relatively simple application would have been difficult to analyze and reason about.

Detailed Statistics on Causes of Cache Misses

By characterizing the predominant cause of misses for each data structure, the tool can give users important insights as to whether memory bottlenecks are occurring due to poor spatial locality, cache interference, inter-processor sharing, or other effects.

In Chapter 3's second example, we showed MemSpy's use in pointing out instances of poor spatial locality and interference in a parallel sparse matrix application, Tri. Detailed information on the causes of cache misses was vital to understanding the nature of these performance bugs and devising appropriate program fixes.

Simulation Performance

A key issue in implementing such detailed statistics is gathering data efficiently. This thesis shows that for gathering such detailed statistics, simulation based performance monitors can offer a feasible, effective, and inexpensive alternative to other data collection methods. In Chapter 5 we presented baseline performance measurements for our approach, and proposed techniques to optimize simulation performance. The first of these techniques streamlines the processing of cache hits by maintaining statistics only on misses. This optimization leads to roughly 20% to 50% improvements in MemSpy execution time.

While that initial optimization eliminates the statistics bookkeeping overhead for cache hits, it does not address the context switch overhead required to switch from the application to the simulator. In an additional optimization called *hit bypassing*, we target this overhead as well. In hit bypassing, the memory reference instrumentation saves only a minimal set of registers before determining if the reference is a hit or a miss. The performance of cache hits is thus optimized, because for them the simulator need only save and restore the minimal register set, rather than incurring the overhead of saving and restoring the full register set. This technique further reduces MemSpy's overhead, down to factors of roughly 8 to 17 for sequential code and 30 to 50 for parallel code.

Reference Trace Sampling

Finally, this work has shown that reference trace sampling can even further optimize the performance of tools like MemSpy. We primarily explored the use of *time sampling*, in which simulation of the reference trace is intermittently turned on and off such that only chunks of references out of the full trace are simulated. Because only a partial simulation is performed, some loss of accuracy might be expected.

For small and moderately sized caches, miss rates can be estimated with very small absolute deviations ($<$ 0.5%) and relative deviations of 10% or less. These estimations can be achieved with sample lengths of only 0.5M references or less. For good accuracy when simulating larger caches ($\geq$1MB), longer samples are required– 4M references or more. However, this still allows for aggressive sampling ratios on many applications.

We find that reference trace sampling can lead to significant performance improvements. When simulating a total of only one tenth of the application references, overheads for sequential applications dropped to the range of 3 to 10 fold. Overheads for parallel applications vary from 8 to 25. In fact overheads in this range make MemSpy an attractive alternative to less detailed tools such as Mtool and Gprof. Furthermore, since MemSpy's statistics provide more detail than other tools, programmers may require fewer profiling runs to tune their code.

8.2 A Broader Perspective

The scope and applicability of this work is delineated by both the accuracy of the monitoring techniques, and the execution time required to get results. That is, a performance monitoring tool is only useful if it gives "acceptable" simulation accuracy with "tolerable" execution time overheads. A key observation is that one often has considerable flexibility in defining what is acceptable accuracy or tolerable overhead for a particular context.

Considering a spectrum of tradeoffs in tool usage, users can often choose where to position themselves between the two extremes of (i) high accuracy statistics with slow collection speeds or (ii) lower accuracy statistics with faster collection speeds. In some cases, users may decide that the performance improvements offered by certain abstractions or optimizations are not worth possible decreases in accuracy (or the intellectual effort of determining *if* they affect accuracy). In other cases, users can harness particular application characteristics to obtain good simulation performance without significant decreases in accuracy.

Figure 8.1 shows a taxonomy of different application characteristics and MemSpy's use in each category. This table divides programs according to two characteristics, cache miss rate and execution time. In the left column, the table shows applications where the execution time is naturally quite short, so MemSpy overheads are generally quite tolerable, even without optimizations like trace sampling. For some of the applications in this regime, the memory performance is naturally quite good as well, and programs are unlikely to need tuning. For others, the applications may need tuning, but the execution

Cache Miss Rate	**Execution Time**	
	Short →	**Long**
Low ↓	**No tuning needed.**	**Use MemSpy. May use sampling.**
High	**Use MemSpy. May use sampling.**	**Use MemSpy with sampling.**

Figure 8.1: Application characteristics and MemSpy usage.

times are still short enough that MemSpy's performance overheads do not necessarily warrant special optimizations like trace sampling. For example, Vrender, a real-time graphics program described in Appendix A, generates image frames in a few seconds or less. As such, monitoring the program takes very little time, only a minute or two, even without trace sampling. Finally, at the bottom of the left column, applications may have very high miss rates, so that although the MemSpy overheads are already quite small, sampling could also be used to further reduce them.

As we move towards applications with higher execution time overheads (towards the right hand side of Figure 8.1), we note first that many applications have characteristics that allow programmers to shorten the tool's execution time without significantly affecting accuracy. For example, in many iterative applications one can choose to simulate fewer iterations than the full application calls for. Furthermore, our implementation of MemSpy also allows programmers to turn monitoring on and off for particular procedures in the code. This lets users indicate to the tool which procedures require no performance analysis, giving them further control over the tradeoff between the completeness of the data produced by MemSpy and the simulation runtime.

Of course, some programs are not amenable to such optimizations. With that in mind, we next consider the upper right quadrant of Figure 8.1 – applications with long execution times, but low cache miss rates. Users may choose to tune the performance of these programs, because with high memory latencies, even low cache miss rates may affect performance. We first note that even without trace sampling, we expect MemSpy's overheads to be reasonably small in these cases. This is because most references are hits, and hit bypassing allows them to be simulated cheaply. On top of this, the programmer can also use sampling. Because of the longer execution times, a large number of samples can easily be collected. The low cache miss rates may increase the effect of sampling error due to unknown references, but since this error can be bounded, users may still choose to take advantage of sampling in this regime.

Finally, as we move down to the lower right corner of the table, we encounter applications that are likely to be both (i) most in need of tuning due to their high execution times and cache miss rates and (ii) very conducive to sampling for precisely the same reasons. These applications are conducive to sampling, first because the long reference trace generally allows a large, representative set of samples to be taken, with samples long enough to prime the cache. Second, the large number of known misses accelerates the cache priming and decreases the error due to unknown references. Thus, it is interesting to note how the regime in which sampling is most effective, coincides well with the regime in which it is most needed for performance tuning.

8.3 Future Directions

There are several ways in which this research could be extended. First, new techniques could significantly improve the tool's performance, making it an even more attractive alternative to less detailed approaches. Second, the scope of this research could be extended to modes of use beyond the domain of performance tuning that was the focus of this dissertation.

8.3.1 Improving Simulator Performance Through Static Analysis

An important class of tool performance optimizations may come from increasing the use of static analysis to replace dynamic monitoring when possible. For example, one such technique would focus on using static analysis to reduce context switch overhead. By incorporating more sophisticated live/dead analysis into the compile-time instrumentation phase, one could avoid doing register saves and restores on “dead” registers. This would further optimize the context switching overhead of the simulator. When available, dead registers could be used for the hit check, so that cache hit performance is optimized even more.

Compiler analysis may also allow static processing for some of the program references. This static analysis could determine sets of references that are guaranteed to be either hits or misses, and eliminate simulation for those. Similarly, compiler analysis could also be used to perform some of the program binning statically, to reduce the program’s dynamic bin search overhead.

8.3.2 Broadening Targeted Usage

Finally, one could broaden the *usage* of MemSpy’s data oriented profiles by extending them into the domains of (i) program correctness debugging and (ii) compiler optimizations.

In data oriented correctness debugging, reference profiles of different data structures could be used to indicate when illegal or unintended accesses to data structures were being made. For example, the Vrender case study in Appendix A demonstrates a case where MemSpy helped discover shared accesses being made to a data structure that was intended to be private. The discovery was made by noticing a large number of *invalidation* misses in what should have been a private data structure used by only one processor. Thus, data oriented information on the causes of cache misses was instrumental in finding this bug. MemSpy could also easily flag illegal accesses to unallocated data by unmapping statistics bins on memory deallocations, and then subsequently flagging any references to unmapped bins. Beyond this, a framework for programmer assertions about data structure usage could allow tools like MemSpy to provide further support for correctness debugging. By specifying data referencing attributes like “never written” or “read once”, programmers could request that the tool check for particular types of unintended accesses.

Another avenue for extending the work in this dissertation would be to use MemSpy profiles as feedback to compilers. For example, profile information on individual data structures could be used to determine appropriate prefetching strategies for particular code sequences. That is, the data structure profiles could augment the static analysis used for guiding the compiler inserting prefetching directives. This can allow the compiler to provide prefetching for data structures that have the most cache misses, while not wasting processor-memory bandwidth by prefetching on data structures that already have good cache behavior.

In addition, information categorizing a data structure’s performance as limited by true sharing, false sharing, interference, or spatial locality can also be useful in guiding data layout in sequential and parallel programs. For example, when the compiler notes problems of interference or false sharing, it can sometimes offset or rearrange data in memory to reduce them. Thus although the ideas developed here were presented primarily in the context of manual program tuning, similar information could also be useful in automating some of these same tuning techniques within compilers.

From a more general viewpoint, efficient techniques for evaluating application and architecture performance form the underpinning for many research studies in the areas of architectures, compilers, and software engineering. By proposing techniques that both significantly accelerate detailed memory simulations, and effectively organize the information collected, this dissertation broadens the scope of monitoring methods available to researchers in these disciplines. In the future, additional applications of the techniques proposed here can demonstrate their utility not only for tuning program performance, but also much more broadly for guiding compiler optimizations, and evaluating new computer architectures.