Exposing Memory Access Regularities Using Object-Relative Memory Profiling

Qiang Wu Artem Pyatakov Alexey Spiridonov
Easwaran Raman Douglas W. Clark David I. August
Department of Computer Science
Princeton University
Princeton, NJ 08544

ABSTRACT

Memory profiling is the process of characterizing a program's memory behavior by observing and recording its response to specific input sets. Relevant aspects of the program's memory behavior may then be used to guide memory optimizations in an aggressively optimizing compiler. In general, memory access behavior has eluded meaningful characterization because of confounding artifacts from memory allocators, linker data layout, and OS memory management. Since these artifacts may change from run to run, memory access patterns may appear different in each run even for the same input set. Worse, regular memory access behavior such as linked list traversals appear to have no structure.

In this paper we present *object-relative translation and decomposition* techniques to eliminate these artifacts and to expose previously obscured memory access patterns. To demonstrate the potential of these ideas, we implement two different memory profilers targeted at different sets of applications. These profilers outperform the existing ones in terms of profile size and useful information per byte of data. The first profiler is a lossless profiler, called WHOMP, which uses object-relativity to achieve a 22% better compression than the previously best known scheme. The second profiler, called LEAP, uses lossy compression to get highly compact profiles while providing useful information to the targeted applications. LEAP correctly characterizes the memory alias rates for 56% more instruction pairs than the previously best known scheme with a practical running time.

1. INTRODUCTION

Feedback-directed memory optimization (FDMO) in compilers is widely accepted as an important means to improve memory performance [1]. Previous research has suggested optimizations such as prefetching [2], speculative load reordering [3], cache-conscious data layout reorganization [4] and others, all of which use memory profiles to direct them. Memory profiling is the process of characterizing a program's memory behavior by observing and recording its response to specific input sets, and then using the relevant aspects of the behavior to guide memory optimizations. Thus a good memory profile is a key factor in the success of a particular FDMO. In this paper we present techniques to improve memory profiling, describe their incorporation into two different memory profilers, and demonstrate their effectiveness in practice.

Typically, memory profilers collect memory access information in terms of raw addresses (such as the trace-based memory profilers [5] and others [4] [6]). However, there are significant barriers to implementing an efficient profiler using this approach. In the raw address space memory access patterns are obscured by confounding artifacts from memory allocators and linker data layout. Figure 1 shows memory references in a linked list traversal and update. While these references are simple and intuitive in the program, they appear irregular in the profile due to seemingly arbitrary heap allocations. Worse, these artifacts may change from run to run. First, even a slightly different input set could lead to radically different data footprint. Second, even for the same input set, a different allocator library could lay out the memory differently. Third, even if the input and allocator are all the same, the insertion of probes could change the code segment size and thus the linker data layout of static data. Thus memory access patterns may appear different in each run, and regular memory access behavior such as linked list traversal may appear to have no structure at all.

The primary contribution of this paper is a new technique to eliminate these artifacts that obscure memory access regularities. Specifically, we present an *object-relative translation and decomposition* method for effective memory profiling. Memory accesses are translated into instruction, group (object type), object serial number, and offset to reflect the true nature of data objects in programs. This object-relative approach enables decomposition of a memory access stream to separate regular and interesting information from the irregular and provide the compiler with information useful for optimization. This translation and separation is conceptually depicted in Figure 2. In that figure, the original raw address stream appear to have an irregular pattern. The next block in the figure represents the translated object-relative stream, which looks slightly more regular but is a mixture of patterns. The right side of the figure depicts the separation of the translated stream to identify regular access patterns. The methods used to perform the conversion to object relative references and to separate the patterns are described in the next section.

The second contribution of this paper is the implementation of two profilers, called WHOMP and LEAP, based on the object- relativity to evaluate its potential benefits. The first profiler, WHOMP, uses

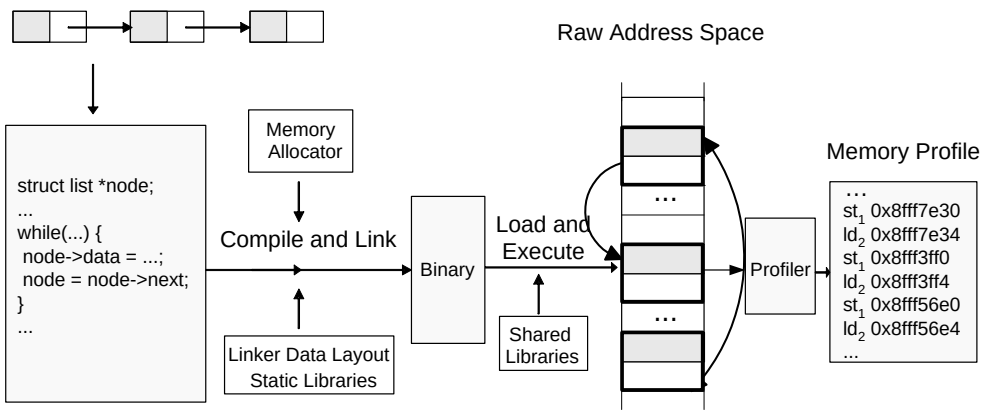


Figure 1: An example illustrating confounding artifacts in memory references.

the Sequitur compression scheme [7] to collect a lossless profile which records the entire data stream during the run of a program. This profiler produces more compact profiles than the best known existing lossless profiler using raw addresses. The second profiler, LEAP, uses a lossy linear compression scheme to obtain a compact profile indexed by load and store instructions. This profiler outperforms the best known practical profilers [6] in identifying load-store alias frequencies.

In the next section, we describe the object-relative technique and show how it reveals previously hidden regularities. Sections 3 and 4 describe and evaluate the implementation of the WHOMP and LEAP object-relative memory profilers. The paper concludes with some related work and summary.

2. OBJECT RELATIVITY

Programmers using high-level languages generally think about data not in terms of addresses, but in terms of *objects*[1]. *Objects* range from simple types (such as integers), to aggregate types (arrays and structured records), to instances of classes. All programming languages have the notion of objects, although their uses might vary. In addition, most programs have many objects of the same type. This is another source of regularity: objects of the same type tend to be allocated and used in similar ways. We call the collection of all objects of the same type a *group*. Intuitively, memory accesses qualified by objects and groups can eliminate the memory artifacts and reflect the true nature of the data in programs.

Using the definitions above, one can convert the raw-address form of accesses to an *object-relative* form. Such a representation of memory accesses allows interesting aspects of the memory behavior of the program to be extracted.

2.1 Object-relative address translation

The concept of an object and a group are used to translate the raw-addresses into a form which is more meaningful to the programmer. Objects which are created at the same program point belong to the same group and this can be identified by the compiler. Given a raw address, a translation mechanism identifies the group and the object being accessed (as represented by the group identifier and the object serial number). We also refer to the specific memory location by the offset from the start of this object. Thus, for a memory access collected by trace as a (instruction-id, address) pair, object-relative translation computes the 4-tuple:

(instruction-id, group, object, offset)

This translation can be achieved using a variety of techniques which maintain a database of allocated objects indexed by their raw addresses.

As an illustrative example, Figure 3 shows memory accesses during a linked list traversal. Solid lines represent accesses to the data fields of the list element (such as node→data) and dashed lines represent accesses to the pointer fields (such as node→next). The figure shows both the raw address stream and the corresponding object-relative stream (with the group identifier 0 referring to objects of the linked list). It is observed that, compared to the raw address stream, the object relative stream better highlights the program's memory behavior. For example, it shows that both instructions are accessing objects in the same group, which may imply that all nodes traversed in the linked list are of the same type. Also, all accesses by the same instruction refer to the same offset within different objects, which is characteristic of accessing the same field of structural records.

This example illustrates that the object-relative stream reveals additional information not readily available in the raw address stream. With the object-relative stream, the system can easily identify useful patterns.

2.2 Object-relative decomposition: separating the regular from the irregular

In a memory access stream, regular and interesting memory access information may be mixed with the irregular, which makes profile-directed optimizations more difficult. To identify the regular accesses two manipulations, *horizontal decomposition* and *vertical decomposition*, are applied.

Horizontal decomposition separates the stream into its dimensions, namely instruction-id, group, object, and offset. This is illustrated by the linked list example in Figure 3. Here we see that horizontal decomposition allows one to consider one dimension at a time across all the memory accesses. In other words, a single stream of four tuples is split into four streams of individual tuple elements.

[1]Object here refers to a group of data stored as a unit (specific structs and arrays in C for example) and should not be taken in the object-oriented programming language sense.

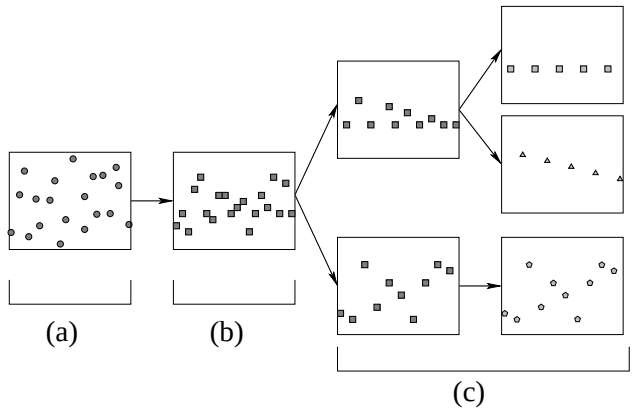


Figure 2: A conceptual illustration of identification of regular access patterns. (a) Original stream (b) Object-relative stream (c) Regular accesses separated from the irregular ones.

The resulting pattern tends to be simple and more regular. This regularity in the pattern makes the resulting profile amenable to good compression as will be shown in the later sections. This kind of decomposition is beneficial when optimizations use only one or a few dimensions of the stream. For example, object clustering [4] is only concerned with the object dimension of the stream.

Vertical decomposition collects objects which share the same value in one dimension (the same instruction-id, for example) and looks at the values of these objects in the other dimensions. Figure 3 shows an example of vertical decomposition by instruction. Here the original stream becomes two simpler sub-streams. Note that, though not shown in Figure 3, these sub-streams can be further decomposed into simpler sub-substreams. For example, further decomposition by group gives a number of simpler (object, offset) streams.

The decision to use horizontal or vertical decomposition depends on the nature of the profile and how the profile information is used. For example, memory dependence optimizations such as load speculation [3][8] require dependence information for specific load-store pairs or data types. Vertical decomposition is more suitable in this case as it can collect all information for accesses with the same instruction-id. Contrast that with optimizations requiring information of a specific granularity, such as object clustering or field re-ordering. In such cases, horizontal decomposition is more appropriate. Multi-purpose memory profilers can employ a hybrid of both techniques.

One problem with vertical decomposition is that it eliminates the ability to directly index into the stream based on time. To cope with this inconvenience, we extend the object-relative stream with an additional dimension - time:

(instruction-id, group, object, offset, time-stamp)

where time-stamp is a counter starting from 0 at the beginning of the program and incremented after every collected access. Any tuple in the substream obtained by vertical decomposition is tagged by its time-stamp and is therefore uniquely identified.

2.3 Incorporating object-relativity into a memory profiling framework

This sub-section presents a general framework from which specific object-relative profilers can be viewed. Figure 4 shows this object-relative framework. The primary addition to a standard memory profiler is the object-management component (OMC). The OMC records information about every object allocated in the program: the time when it is allocated and de-allocated, the address range used by the object, and the type of the object. Additionally, this component assigns an identifier to every group and object that can be used to identify them later in the program. Given an address, the OMC identifies the group and object, and translates the raw address into a *(group, object, offset)* triple.

The program is instrumented by inserting instruction and object probes into the target program. The instruction probes are inserted next to every load and store instruction. Every time a memory instruction executes, the adjacent probe passes the instruction ID and the memory address accessed to the control and decomposition component (CDC, the details of which are described shortly). Object probes are introduced at object creation and destruction points. They collect the creation and destruction time, size, and type of every object in the execution of the program, and pass this information to the OMC.

The CDC acts as a hub to the profiling process. It receives information from the instruction probes, and queries the OMC to make the information object-relative. It then passes on the object-relative stream to the separation and compression component (SCC).

As the name suggests, the SCC first separates the stream into multiple substreams (by horizontal decomposition, vertical decomposition, or both). It then sends the substreams into a steam compressor. Examples of such compression schemes include linear compression [9], Sequitur compression [10], and others.

The output of the SCC is a compressed raw-address profile in an object-relative format. This can then be optionally fed into a post-processor to get more refined, optimization-specific information. The profiler can also output the object lifetime and other auxiliary information from the OMC unit. This run- and alloc-dependent information is separated from the invariant object-relative tuples when it is useful to certain profile consumers.

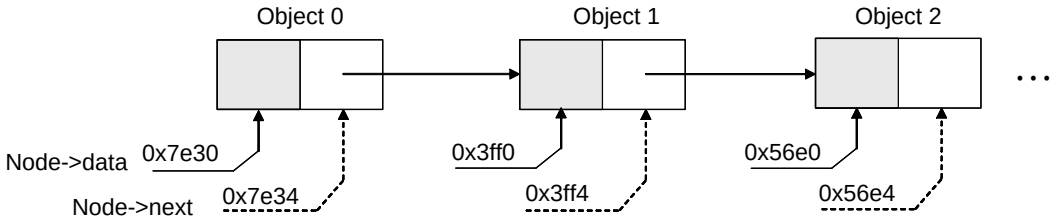
The presented framework can be used as a foundation to many different object-relative memory profilers. Two such implementations are described next.

3. WHOLE-STREAM MEMORY PROFILER

The object-relative profiling framework is used to create a lossless whole-stream memory profiler (WHOMP), the first of two object-relative memory profilers presented in this paper. This profiler uses the lossless Sequitur compression scheme to record the entire stream in object-relative format. This section describes WHOMP and quantitatively measures the benefits it receive from the use of object-relativity. The results of the implementation quantitatively support our intuition that a good compression is a natural result of the object relative approach.

3.1 WHOMP implementation

The WHOMP implementation follows the framework presented in Section 2. WHOMP instruments the program by inserting instruc-



		access 1	access 2	access 3	access 4	access 5	access 6	Pattern
Raw-address stream:								
(inst-ID, address)		(1, 0x7e30)	(2, 0x7e34)	(1, 0x3ff0)	(2, 0x3ff4)	(1, 0x56e0)	(2, 0x56e4)	–
Object-relative stream:								
(inst-ID, group, obj, offset)		**(1, 0, 0, 0)**	(2, 0, 0, 4)	**(1, 0, 1, 0)**	(2, 0, 1, 4)	**(1, 0, 2, 0)**	(2, 0, 2, 4)	–
Horizontal decomposition:	inst-ID	1	2	1	2	1	2	(12)*
	group	0	0	0	0	0	0	(0)*
	object	0	0	1	1	2	2	$(0011\cdots nn)$
	offset	0	4	0	4	0	4	(04)*
Vertical decomposition:								
(by instr-ID)	inst-1	(0, 0, 0)	–	(0, 1,0)	–	(0, 2, 0)	–	(0, k, 0)
	inst-2	–	(0, 0, 4)	–	(0, 1, 4)	–	(0, 2, 4)	(0, k, 4)

Figure 3: An example illustrating the object-relativity, horizontal decomposition and vertical decomposition. The table presents several ways of representing accesses to the linked structure at the top.

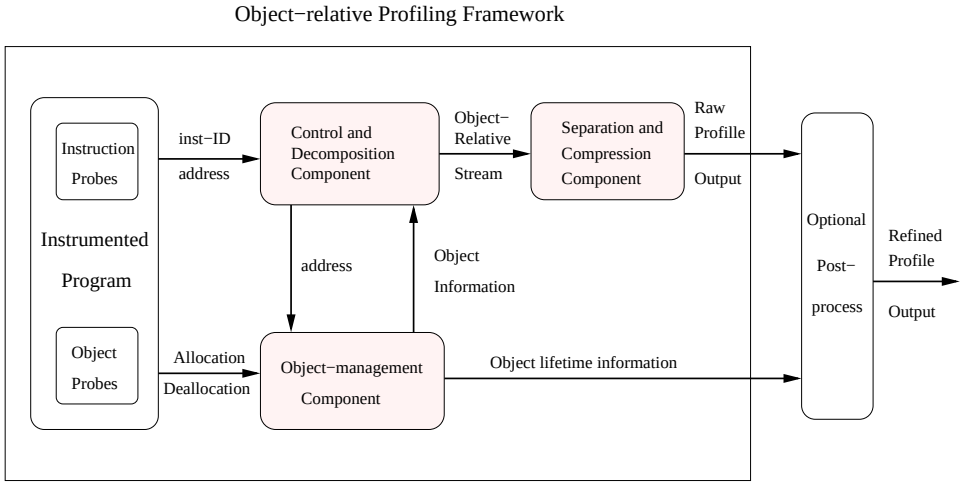


Figure 4: Object-relativity in a memory profiling framework

tion and object probes directly into the assembly code. The instruction probes are inserted next to every load/store instruction. The profiler inserts object creation/destruction probes at allocation/de-allocation points for dynamically allocated objects[2], or at the beginning and end of the program for all statically allocated objects. Interactions between the instrumented program and the CDC/OMC components take place via thread-to-thread communication. The profiler groups allocated dynamic objects by static instruction. The compiler can provide type information to further refine this strategy. WHOMP uses the exported symbol table from the gcc compiler to determine the size and group of statically-allocated objects. Since static analysis handle stack variables very efficiently, we chose not to profile them. To speed up the lookup process in the OMC, the profiler uses an auxiliary B-tree-like data structure which stores the range of addresses that each object takes up. When the program de-allocates an object, the profiler removes elements from this tree.

The CDC translates the raw address stream into an object-relative stream, representing addresses as a five tuple (instr-ID, group, object, offset, time-stamp). It then passes the stream to the SCC. The SCC first decomposes the object-relative stream horizontally along all four dimensions (instruction ID, group, object and offset). Each of these streams is then fed into a separate Sequitur compressor.

The Sequitur compression scheme used in WHOMP was developed by Nevill-Manning and Written [7]. It encodes input data stream as a context-free grammar based on its repeating patterns. The compressor scans the input sequence and builds the grammar incrementally. Each repetition gives rise to a rule in the grammar and every repeated subsequence is replaced by a non-terminal symbol, producing a more concise representation of the whole sequence. For example, the sequence of symbols "abcbcabcbc" will be compressed into the grammar:

$$S \rightarrow AA; \quad A \rightarrow aBB; \quad B \rightarrow bc$$

We refer to the above grammar as *Sequitur grammar* in later discussions.

3.2 Evaluating WHOMP

We evaluated the performance of WHOMP by collecting WHOMP profiles for 7 SPEC benchmarks. We believe these programs form a representative set as they display a range of memory behavior. All experiments were run on an IA-64 Itanium machine (HP-I2000 Itanium workstation, with 1GB of RAM, Linux RedHat7.0, Kernel version 2.4.3), using the training input sets.

The output of WHOMP is an object-relative multi-dimensional Sequitur grammar (OMSG), with one grammar formed for each of the decomposed streams. Both the OMSG and the conventional raw address Sequitur grammar (RASG) are lossless. They both contain information about repeating memory access patterns, which is useful for a class of correlation-based memory optimizations including clustering, custom heap allocation[4], and hot data stream prefetching [11]. However, OMSG provides some additional benefits over the traditional RASG, as discussed next.

[2]We choose to treat custom alloc pools as single objects. An alternative is to manually target the custom alloc/dealloc functions rather than target the standard malloc/free, depending on the application and use of the profile information. The profiler can be parameterized to handle this.

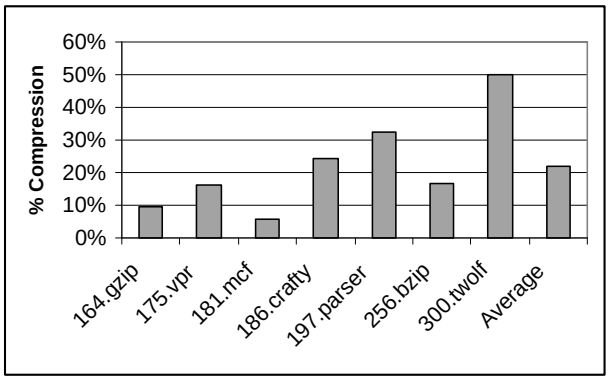


Figure 5: The compression ratio of the OMSG over the conventional raw address Sequitur grammar.

OMSG can be more compact than a Sequitur-compressed raw address stream profile. This is due to the exposed regularity and the multi-dimensional decomposition, which causes the substreams to have a much simpler pattern to compress, as compared to the original raw address stream. Simpler and more regular pattern lead to a better and smaller grammar, and hence a higher compression ratio. Intuitively, the linked list example in Figure 3 illustrates this idea as simple regular expressions can be used do describe the object sub-stream. Of course, for this simple example, since the patterns are short, the effect of compressing into multiple grammars is not pronounced. However, for a real data stream in practice, the additional regularity can yield better compression and make the total size of OMSG smaller.

To compare the performance of OMSG, we also generate the conventional RASG using the raw address stream (similar to the grammars in [4]). We then compute the percent of compression achieved by OMSG over RASG, using the RASG size as the base. Figure 5 shows the experiment results. On average, OMSG is 22% more compact than RASG.

Even though the object-relative method has the additional overhead of translating raw addresses into the object-relative format, on average profile collection time is roughly the same (OMSG is 1% faster than RASG). This indicates that the time spent extracting the object relativity information is offset by the gains in Sequitur's pattern search due to the regularity in the stream. The details on the Sequitur algorithm and its pattern search can be found in [7].

Another benefit of OMSG is the production of refined, fine-grained information for optimizations. The grammar for each dimension is useful for a different set of optimizations. For example, the offset-level grammar can be used for optimizations like field-reordering [4]. A frequently repeated offset sequence, say $(0, 36)*$, along with the object lifetime information (recall from the last section that this was not discarded), may reveal field-reordering opportunity to the compiler to take advantage of spatial locality in the reference stream. Another example is the use of object-level grammar for object clustering or global variable re-mapping [4]. Optimizations also have the option to simultaneously access two or more dimensions of the OMSG. Therefore, the fine-grained, object-relative, multi-dimensional grammar provides extra flexibility and adaptability to memory optimizations.

4. LOSS-ENHANCED ACCESS PROFILER

The previous section illustrated an inherent benefit of the object-relative approach, namely good compressibility of the profile. However, the lossless profile obtained with WHOMP comes at the price of long running times and huge profile sizes even after compression. With the observation that losing some portion of the profile information may not affect the outcome of a given task, we implement a Loss-Enhanced Access Profiler (LEAP) targeted to two specific applications. We evaluate LEAP in the context of these applications to demonstrate the benefits of object-relativity in a lossy profiler.

LEAP uses linear compression and collects profile indexed by load and store instructions. We target two optimization techniques: speculative load reordering [3, 8] and stride-based prefetching [2]. Speculative load reordering is a technique that speculatively schedules a load instruction ahead of a preceding store to hide the memory. This reordering is beneficial only if the load is independent of the store or is dependent with a low frequency, because of the relatively high recovery overhead. Hence this optimization requires a very good estimate of dependence frequencies between loads and stores. The other optimization, stride-based prefetching, performs prefetching for strided memory accesses. To facilitate this, strongly strided instructions - instructions which access memory with one particular stride most of the time - must be identified.

4.1 LEAP implementation

The implementation of this profiler is similar to that of the lossless WHOMP discussed in the previous section. In WHOMP, after translating the raw addresses into the object-relative form, the SCC decomposes the stream vertically by instruction id and then by group to get a number of (object, offset, time) streams. These streams are then sent to a linear compressor, which is discussed in detail later. Since some complex streams may be difficult to represent compactly, the profiler discards some information so that the resulting stream is compactly represented. This makes the profiler lossy. The (object, offset, time) sub-streams are also decomposed horizontally. We use these mixed sub-streams to record additional aspects of the memory reference behavior.

This profiler uses a simple linear compressor, which is based on the linear memory access descriptor (LMAD) model in [9]. A LMAD is described by the triple *[start, stride, count]*. Note that *start* and *stride* can be n by 1 vectors, where n is the number of dimensions in the stream that is compressed. For example, for the (object, offset, time) sub-streams, the value of n is 3, while for the (offset) sub-streams, the value of n is 1. The LMAD compression reads each symbol in the data stream and attempts to describe the stream using its linear descriptors. If the new symbol does not fit into the current linear pattern, it will start a new LMAD for this symbol. For example, an offset stream of

$$0, 8, 16, ..., 80, 4, 8, 12, ..., 36$$

will be described by two LMADs as

$$[0, 8, 10], \quad [4, 4, 9]$$

To get a compact profile with a practical running time, we can only allow a finite number of LMADs. Reducing the number of LMADS will reduce the running time, but affect the profile quality. Increasing the number of LMADs gives a less lossy profile but increases the running time. In our implementation, we chose a maximum of 30 LMADs for a given (instruction-id, group) pair. This number was found to be suitable for our applications and to keep the running time low.

This linear compression scheme has several attractive properties. It is simple and fast, and it works quite well if the data stream has predominantly linear patterns. We notice that, in practice, a significant portion of instructions do exhibit linear access behavior and hence can be captured by a small number of LMADs. The biggest disadvantage of the linear compression scheme is that it cannot handle an elaborate access pattern, especially if the stream exhibits predominantly non-linear behavior. In such a case, the allowed maximum number of LMADs will be quickly exhausted. The compressor will then discard the new symbols in the stream, and only record some overall information such as max, min, and $granularity$. So, for such a case, the information stored in LMADs is essentially a sample of the initial part of the original data stream. We use the term *sample quality* to refer to how much information is captured in LMADs, with respect to all information in the original data stream. For example, the sample quality for a predominantly linear stream can be $\approx 100\%$, while $\approx 0\%$ for a predominantly non-linear stream. A low sample quality may also be acceptable if most of the linear access patterns are captured as the targeted optimizations rely mainly on the linear access patterns.

4.2 Evaluation of the LEAP

This subsection presents the evaluation of the LEAP profiler in the two target applications. Since the profiler is lossy, the potential benefits accrued to the targeted optimizations are a more suitable evaluation metric than the compression ratio.

The experiments are conducted for the same set of SPEC2000 benchmarks as before. For each benchmark, LEAP is run once to collect the LMADs. Two different post-processors use these LMADs to compute memory dependence frequency and strongly strided instructions respectively.

4.2.1 Application 1: Memory dependence frequency

As described earlier in this section, the memory dependence frequency profile is useful for memory optimizations such as speculative load re-ordering and loop-invariant load removal. In this sub-section, we evaluate how well the LEAP profiler can capture the memory dependence frequency information for all store-load pairs.

We apply a memory dependence detection post-process to the collected LMADs. We restrict our attention to the *read after write* dependence. So we define a pair of st/ld instructions as conflicting if the st accesses location A at a time t_1 while the ld accesses the location A at a later time t_2 ($t2 > t1$). The memory dependence frequency (MDF) for a (st, ld) pair is computed as

$$\text{MDF for } (st, ld) = \frac{\text{\# of conflicts with } st}{\text{Total \# of exec for } ld}$$

Because of the linear structure of LMADs, the above computation can be sped up using some *omega-test-like* linear programming algorithms [12]. For example, detecting the location conflicts involves solving integer solutions k_1, k_2 for

$$\begin{array}{ccc} start_1 + stride_1\, k_1 & = & start_2 + stride_2\, k_2 \\ k_1 \leq count_1 & & k_2 \leq count_2 \end{array}$$

Because the LMADs profile is relatively small, our post-process computation is fast. For SPEC2000 benchmarks, the post-process time ranges from seconds to several minutes.

The output of the analysis is a list of dependent store/load pairs along with the computed dependence frequency for each. For example, $(st_2, ld_1, 10\%)$, $(st_3, ld_1, 90\%)$ shows that a static ld_1 is dependent on a static st_2 for 10% of its execution, while it is dependent on st_3 for 90% of its execution.

Since LMAD compression is lossy, we evaluate the error distribution with respect to a lossless profiler. We used a lossless raw-address based profiler which records the dependence information of all the memory operations in a program as the baseline. Such a profiler is extremely slow and produces huge profiles. Figure 6 shows the error distribution for LEAP for all benchmarks. Notice that a dominating majority (75%) of the dependent pairs either have frequencies that are completely correct (center point) or off by no more than 10% (the points on either side of center).

We also compare the performance of LEAP to another existing memory dependence profiling approach by Connors [6], which uses raw address streams instead of object-relative streams. This profiler is the best known memory dependence profiler with a practical running time and memory footprint. We perform exactly the same evaluation as the one above but this time we compare the ideal with our re-implementation of the instruction-indexed profiler in [6]. (Instruction-indexed is recommended by authors in [6], and we chose a window size such that it exhibits a running time similar to LEAP.) The error distribution produced is shown in Figure 7. While not overestimating the frequency for any dependent pairs, this scheme often misses some of the dependences as it identifies dependences only in a small window of instructions based on addresses recorded in a small history window.

To provide an easy to read comparison, we show the average error distribution for all benchmarks using both approaches in Figure 8. Observe that LEAP out-performs the other approach by a large margin. In particular, note the 56% improvement in the number of pairs detected completely correct or off by no more than 10%. This is likely to result in improved optimizer performance as Chen [3] demonstrates that better dependence analysis leads to better optimization/scheduling. Thus, due to the exposed regularity of accesses, the object-relative approach proves to be more amenable to compression (as shown later) - without sacrificing accuracy significantly in this application.

4.2.2 Application 2: Memory stride patterns

Here, we show that LEAP collects memory stride profile nearly as accurately as a specialized stride profiler (like the one in [2]), despite the fact that LEAP also collects information which is amenable to many optimizations but not particularly well-suited for the purpose of collecting memory strides. We do this by evaluating how well the LEAP profiler captures memory stride patterns and identifies the strongly-strided instructions. We adopt the definition of the (single) strongly-strided instruction in [2] – an instruction for which one stride accounts for $\geq 70\%$ of its total accesses. With the collected LMADs, identifying strongly strided instructions requires a trivial post-process which examines all offset strides captured for a given instruction. We choose to consider only those strongly strided instructions within objects (i.e. with identical group and object IDs), as we found an overwhelming majority of strongly-strided instructions occurred within objects (One of the reasons

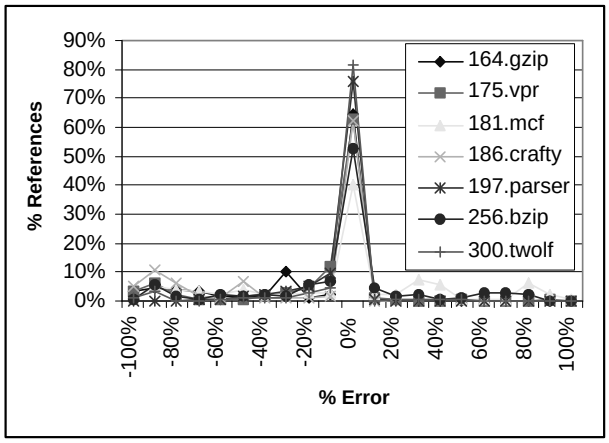


Figure 6: The error distribution of the LEAP memory-dependence results.

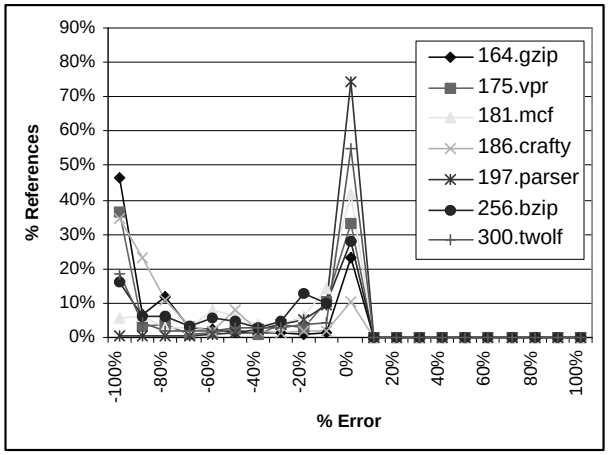


Figure 7: The error distribution of the Connors memory-dependence results.

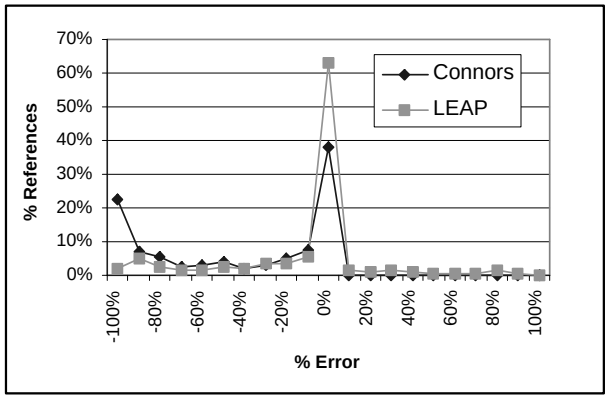


Figure 8: A comparison between the average error distributions of the LEAP and Connors profilers. The higher the peak at 0% error, the better.

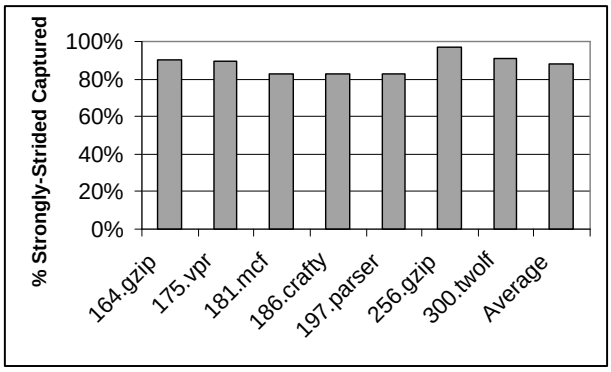


Figure 9: Stride score for LEAP.

Table 1: LEAP profile size, speed, and sample quality.

Benchmarks	Compre--ssion Ratio	Dila-tion Factor	Sample Quality	
			Accesses captured(%)	Instructions captured (%)
164.gzip	1169x	15	57.1%	40.8%
175.vpr	3935x	16	34.7%	52.8%
181.mcf	9993x	7	6.5%	40.8%
186.crafty	967x	9	50.3%	41.7%
197.parser	667x	7	76.3%	8.2%
256.bzip	7152x	14	31.6%	50.6%
300.twolf	856x	15	66.5%	39.8%
Average	3539x	11.5	46.5%	40.5%

might be treating the custom allocation pools as single objects – see the footnote in Section 3.1). An extension to include strongly strided instructions across objects can be implemented by using the auxiliary object lifetime information, which is run/alloc dependent though.

To evaluate the quality of the stride data, we compare the identified strongly-strided instructions to the "real" ones, which are found by the equivalent lossless version. We re-implement the stride profiling in [2] with a setting to make it lossless and track all the strides for a given instruction (which is extremely slow because of the huge amount of stride information to be tracked). The bars in Figure 9 show the percent of correctly identified instructions over the "real" ones. We see that on an average 88% instructions are correctly identified over all benchmarks.

4.2.3 LEAP profile size and speed

For the sake of completeness, we now present some more basic metrics about the LEAP profiler. These metrics provide some insight into the efficiency of the LEAP implementation in terms of profile size, speed, and sample quality. From Table 1, several interesting observations can be made.

First, the LEAP profile is quite compact. LEAP achieves an average of 3 orders of magnitude compression relative to the trace. Second, the time dilation (or the slowdown) over the native is acceptable. At 11.5 times the native, this number was acceptable for experimentation. Further optimization for speed (by specializing for a particular architecture's performance counters for example) was deemed unnecessary for our purposes. In practice, improvements in speed can be made by probe optimizations. Here, we inserted probes directly into the IA-64 assembly even when other means could have been used to determine behavior statically and used multiple threads to collect and analyze data. Thread synchronization added profiling overhead, but this was done for ease of implementation.

The third observation is that LEAP captures nearly half of all memory accesses. The *sample quality* metric is composed of the fraction of captured accesses and the fraction of captured instructions. The former is the fraction of all memory accesses in the program what were captured by LMADs at the level of offsets inside objects (not including the timing information). The latter is the fraction of instructions whose behavior could be completely captured by all LMADs. These numbers are significantly lower (on average 47% and 41%) than fractions presented for correct memory dependence and stride data determination in the last two sub-sections (on average 75% and 88%). This observation shows that the LEAP profile is relatively efficient at representing useful information per byte of output. In addition, it illustrates the fact (also stated in [10]) that it is not necessary for a good profile to capture all memory access information, as long as it captures useful information relevant to optimizations.

5. RELATED WORK

In this section, we highlight important related work. Previous work in [13], [4] and [10] are closely related to this work in the sense that they all deal in profiling using objects. This work, however, directly addresses and generalizes the use of objects in memory profiling and studies this approach in depth.

Calder et al. [13] proposed a novel data object placement algorithm, called *Cache-Conscious Data Placement (CCDP)*, for the purpose of reducing the frequency of data cache misses. *CCDP* reassigns virtual addresses to data objects, where a data object can be a global variable, a heap object, or the whole stack. To facilitate *CCDP*, a profile is generated listing all data objects encountered during execution along with information of data objects' reference count, size, and other life-time information. Clearly, the *object profile* in [13] examines how objects are managed and used during the execution. By contrast, the *object-relative* profile proposed in this paper is a generalization with emphasis on object-relative translation, decomposition, and compression in the context of a wider set of optimizations.

Rubin et al. [4] proposed an efficient profile-analysis framework for data layout optimizations. As part of their framework, the memory profiler collects a *data-object trace*. There, the data object is defined as an elementary piece of data, which can be a field of a record, a global variable or any other single-access granularity objects represented by a global identifier (a unique object ID). They use this object identifier along with the raw address as a pair to identify memory references. While this representation is useful in eliminating false aliasing due to the reuse of the memory addresses for different objects, it does not convey the relationship between an object and its fields. In contrast, objects defined in this paper are a part of the hierarchy of (group, object, offset) tuples, which yields an explicit view of the data structures, the fields, and their common properties. Also, since our scheme has factored out the raw addresses from the representation, identifying patterns becomes easier and the profile exhibits higher compression rates.

Chilimbi [10] proposed an efficient representation of the memory access stream for the purpose of quantifying and exploiting data reference locality. As part of the effort to reduce the size and processing time of a profile, he proposed an *address abstraction*, which abstracts the address to a name or an identifier. For example, a heap object is abstracted into a (start address, global identifier) pair. This identifier pair does not have a size or a type associated with it and it does not distinguish the individual fields within a heap object. Therefore, the *address abstraction* in [10] results in coarse-grained memory profile information. All offset information is discarded, making it impossible to regenerate the address trace once these abstractions have been applied [10]. In contrast, the techniques presented in this paper do not discard any information while exposing regularity.

6. CONCLUSION AND FUTURE WORK

This paper presents object-relative translation and decomposition techniques to expose previously obscured memory access regularities for effective memory profiling. It also describes and evaluates two memory profilers built using these ideas.

Object-relative translation translate raw address to a 4-tuple (Instruction ID, group, object, offset). These tuples can be decomposed in different ways to extract information useful to the target application.

To illustrate the value of these ideas, we apply them to two memory profilers, WHOMP and LEAP. WHOMP is lossless and gives a complete stream of the memory accesses in the form of the above mentioned tuples. LEAP is lossy and collects compact profile which can easily used to identify load-store dependence frequency and stride information.

Experiments show that object-relativity leads to a profile with more compact representations, and with more useful information for several common memory optimizations. WHOMP achieves an average 22% better compression over another lossless profiler using raw-addresses. The LEAP profiler, which produces a profile 3 to 4 orders of magnitude smaller than the original data trace, can correctly estimate the memory dependence frequency for 75% of instruction pairs (which is 56% more than an existing approach), as well as identify 88% of strongly-strided instructions.

Future work includes integration of this memory profiler into an industrial-strength compiler. First, the compiler can improve profile performance by eliminating the need to collect the information known statically. Second, the compiler can provide a framework for FDMO enabled with object-relativity. Another avenue to explore is to make use of recent results on phase detection and prediction [14] to profile references in a phase cognizant manner.

7. REFERENCES

[1] M. D. Smith, "Overcoming the challenges to feedback-directed optimization," in *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo '00)*, January 2000.

[2] Y. Wu, "Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, June 2002.

[3] W. Y. Chen, S. A. Mahlke, N. J. Warter, R. E. Hank, R. A. Bringmann, S. Anik, D. M. Lavery, J. C. Gyllenhaal, T. Kiyohara, and W. W. Hwu, "Using profile information to assist advanced compiler optimization and scheduling," in *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, August 1992.

[4] S. Rubin, R. Bodik, and T. Chilimbi, "An efficient profile-analysis framework for data-layout optimizations," in *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, January 2002.

[5] E. M. Nystrom, R. D. Ju, and W. W. Hwu, "Characterization of repeating data access patterns in integer benchmarks," in *Proceedings of the 28th International Symposium on Computer Architecture*, September 2001.

[6] D. A. Connors, "Memory profiling for directing data speculative optimizations and scheduling," Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1997.

[7] C.G.Nevill-Manning and I.H.Written, "Identifying hierarchical structure in sequences: A linear-time algorithm," *Journal of Artif. Intell. Research*, pp. 67–82, July 1997.

[8] C. Dulong, R. Krishnaiyer, D. Kulkarni, D. Lavery, W. Li, J. Ng, and D. Sehr, "An overview of the Intel IA-64 compiler," *Intel Technology Journal Q4, 1999*, pp. 1–15, June 1999.

[9] Y. Paek and J. Hoeflinger, "Efficient and precise array access analysis," *ACM Transactions on Programming Languages and Systems*, vol. 24, no. 1, pp. 65–109, 2000.

[10] T. Chilimbi, "Efficient representations and abstractions for quantifying and exploiting data reference locality," in *Proceedings of the ACM SIGPLAN 01 Conference on Programming Language Design and Implementation*, June 2001.

[11] T. Chilimbi and M. Hirzel, "Dynamic hot data stream prefetching for general-purpose programs," in *Proceedings of the ACM SIGPLAN 02 Conference on Programming Language Design and Implementation*, June 2002.

[12] J. Hoeflinger and Y. Paek, "A comparative analysis of dependence testing mechanisms," in *Proceedings of LCPC2000, 13th International Workshop on Languages and Compiler for Parallel Computing*, August 2000.

[13] B. Calder, K. Chandra, S. John, and T. Austin, "Cache-conscious data placement," in *Proceedings of the 8th International Symposium on Architectural Support for Programming Languages and Operating Systems ASPLOS'98*, October 1998.

[14] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *Proceedings of the 30th International Symposium on Computer Architecture*, pp. 336–349, June 2003.