Dynamic Memory Optimization using Pool Allocation and Prefetching

Qin Zhao[1], Rodric Rabbah[2], and Weng-Fai Wong[1,3]
[1]National University of Singapore, Singapore-MIT Alliance
[2]MIT Computer Science and Artificial Intelligence Laboratory
[3]National University of Singapore, Department of Computer Science

Abstract

Heap memory allocation plays an important role in modern applications. Conventional heap allocators, however, generally ignore the underlying memory hierarchy of the system, favoring instead a low runtime overhead and fast response times. Unfortunately, with little concern for the memory hierarchy, the data layout may exhibit poor spatial locality, and degrade cache performance. In this paper, we describe a dynamic heap allocation scheme called pool allocation. The strategy aims to improve cache performance by inspecting memory allocation requests, and allocating memory from appropriate heap pools as dictated by the requesting context. The advantages are two fold. First, by pooling together data with a common context, we expect to improve spatial locality, as data fetched to the caches will contain fewer items from different contexts. If the allocation patterns are closely matched to the traversal patterns, the end result is faster memory performance. Second, by pooling heap objects, we expect access patterns to exhibit more regularity, thus creating more opportunities for data prefetching. Our dynamic memory optimizer exploits the increased regularity to insert prefetch instructions at runtime. The optimizations are implemented in DynamoRIO, a dynamic optimization framework. We evaluate the work using various benchmarks, and measure a 17% speedup over `gcc -O3` *on an Athlon MP, and a 13% speedup on a Pentium 4.*

1. Introduction

Memory latency is an important performance bottleneck in modern high-performance architectures. Current practices for hiding the memory latency range from architectural innovations to compiler optimizations. In this paper, we describe our memory optimizer (MO), with an emphasis on locality-aware heap management, and data prefetching. In the former, dynamic memory allocation requests in the program are replaced with customized memory management routines. The custom allocators reserve large memory pools of the heap space, and reassign smaller portions within the pool with successive heap allocation requests. The end result is a segregation of different data types into distinct regions of the heap, often leading to improved reference locality. In the case of data prefetching, future memory requested are anticipated and fetched ahead of time in order to mask long memory access latencies.

MO is implemented in DynamoRIO, a state of the art dynamic optimization framework [2]. MO can be readily applied to programs running directly on stock hardware, without any modification or even knowledge of the program source code. The locality aware heap allocator is designed to be simple and lightweight. It operates in two stages. First, as part of the dynamic instrumentation system, it inspects all heap requests, and creates a memory request monitor for each requesting call site. When a code region runs often enough to be designated as *hot*, its monitors are inspected. If a monitor is found to have recorded a consistent calling context, the corresponding memory request is routed to a custom pool allocator. Otherwise, it is routed normally to the standard system allocators. Thus, object instances that are likely to be of the same type are grouped in continuous memory regions, while dissimilar objects are segregated into different pools.

Such data layout optimizations not only improve the data locality, but also affords more aggressive optimization opportunities via data prefetching. The key insight and contribution in this paper is that data prefetching techniques that are effective for scientific programs can now be easily applied to pointer-chasing applications—because their data access strides become far more predictable. To perform data prefetching, MO instruments the code traces extracted by DynamoRIO to discover delinquent loads and collect stride information with extremely lightweight profiling. The profiles then serves as a basis for dynamic prefetch injection. Our results suggest that pool allocation creates more opportunities for data prefetching with strides.

We evaluate MO using benchmarks collected from Olden, Ptrdist, and SPEC. The benchmarks were compiled with `gcc -O3`. MO yields a performance gain of 17% on average for an Athlon MP and 13% for a Pentium 4. For applications with little dynamic allocation and deallocation, MO contributes a performance slowdown averaging less than 1% beyond that contributed by DynamoRIO alone.

2. Pool Allocation

Our pool allocation strategy can lead to better performance in two ways. In applications where the data structure traversal matches the data allocation order, the pooling strategy can expose more regular strides. For example, consider a search through a linked-list of records. Each record consists of a `key` field, a `data` field, and a `next` field pointing to the next record in the list. The `key` and `next` fields are accessed consecutively, until a match is found. It is only then that the `data` field is accessed. If the data stored in each record is itself a heap object, and worse, the data varies in size form record to the next, there are various undesirable effects on the spatial locality of the working set. Namely, data are unnecessarily fetched from memory, consuming valuable bandwidth. Also, since variable length data intervene between the `key` and `next` fields, the strides within a record, and across successive records will appear irregular. Thus, the segregation of different object types can improve spatial locality, and concomitantly, overall performance. Furthermore, because pool allocation may expose more regular strides, a simple prefetching strategy can anticipate future references and shorten their memory access latency.

MO is implemented in DynamoRIO, a dynamic optimization framework. In DynamoRIO, when the basic blocks are first interpreted, all calls to standard dynamic memory allocators (e.g., `malloc` and `calloc`) are replaced with calls to an allocation *wrapper*. The wrapper, in addition to the standard parameters passed to the dynamic allocators, requires one additional parameter: the *memory request monitor* (MRM). We associate a unique MRM with each call site. The MRM is a data structure used to record the size of previous dynamic memory allocations originating at the call site. If the last entry matches the size of the new request, the MRM is said to contain a valid context.

DynamoRIO occasionally promotes chains of basic blocks to optimized traces. This is done to improve the performance of frequently executed code. When a region is promoted, MO inspects the monitors in the region. If the monitor has recorded a valid context (i.e., recent requests from the call site were identical with respect to the size), the optimizer replaces the call to the wrapper with a call to a custom allocator. Otherwise, the call is replaced with the original call to the standard allocator provided by the system. The custom allocator reserves a large memory pool to service incoming requests. With each new request, it reassigns an appropriately sized region in the pool.

MO requires that all allocations within a pool are of the same size. Thus, allocations from a pool are as simple as incrementing a pointer. In the case of deallocation from a pool, the custom allocator uses a stack to maintain the list of free regions. Newly freed regions are pushed onto the stack, and incoming requests are eventually serviced form the stack. When the free list is exhausted (i.e., the pool is completely assigned), a new pool twice the size of the previous pool is reserved. Similarly, when all objects in the pool are freed, the pool is deallocated.

On the occasion that a pool request does not match the expected data size, the requested is routed to the standard system allocators. The rational is that allocations of the same object type will often match in size, and MO should focus on the common case scenarios. Any requests to reallocate a dynamic data structure are also handled by the standard allocators, with the caveat that if the reallocated data were organically resident in a custom pool, the space is reclaimed for future requests.

3. Data Prefetching

Our data prefetching strategy is motivated by previous research [1, 17] which has shown that in many programs, long latency cache misses are dominated by only a small number of static loads. These loads are commonly known as *delinquent loads*. MO identifies delinquent loads at runtime using a low-overhead profiler.

The memory optimizer does not instrument and profile all memory operations, but rather only a specific subset. Since the profiling is done online, the alternative is prohibitively expensive. MO profiles memory instructions that

- Execute frequently — delinquent loads must also be frequently executed instructions, so our profiler only instruments instructions found in traces, which are hot code fragments derived from the executing program by DynamoRIO.

- Reference the heap — stack and global data references usually exhibit good locality behavior, and hence they are not instrumented. In the x86 ISA, instructions accessing memory through `ESP` or `EBP` are considered non-heap references.

- Do not use the index register — in the x86 ISA, array references typically use an index register. The profiler is thus biased toward pointer dereferences, rather than array accesses which are likely to exhibit relatively better cache performance.

When DynamoRIO constructs a trace, MO examines the code fragment and adds a few new instructions per delinquent load to write the base address referenced by the load to a designated profile in memory[1]. A counter is also added to the trace. It is incremented on every entry to the code region, and when it exceeds a threshold t, it triggers an analyzer to inspect the profiles associated with the trace.

[1]The number of new instructions varies with the number of delinquent loads in the trace. We only record the base address of a memory reference. For example if the instruction is `ebx` ← `[eax]`16, we only record the value of `eax`.

benchmark	native	DR	pool	pft	all
em3d	**10.444**	10.57	11.02	10.63	10.97
health	11.95	11.77	8.88	11.89	**8.85**
mst	12.7	13.28	13.14	12.29	**12.07**
treeadd	338.76	384.88	322.28	279.83	**262.05**
tsp	42.96	43.43	41.56	43.74	**41.44**
ft	99.33	99.63	26.79	38.37	**8.42**
ammp	754.68	762.15	746.65	722.3	**694.35**
art	**780.12**	789.14	812.18	787.66	813.227
equake	343.01	354.54	**275.14**	353.8	275.36
twolf	957.15	986.77	**950.45**	988.31	972.98

Table 1. Athlon MP execution time (in seconds).

benchmark	native	DR	pool	pft	all
em3d	5.54	5.29	**5.27**	5.3	5.31
health	5.46	5.54	4.741	5.52	**4.51**
mst	6.54	6.47	6.47	6.59	**6.46**
treeadd	**103.17**	115.97	112.75	126.39	121.41
tsp	20.56	20.87	**19.31**	20.81	19.34
ft	16.36	16.48	3.58	14.38	**3.18**
ammp	409.65	429.12	404.74	415.79	**397.24**
art	389.58	376.57	300.758	362.23	**280.87**
equake	122.5	125.94	110.15	124.67	**109.21**
twolf	**394.44**	427.03	403.36	434.88	405.95

Table 2. Pentium 4 execution time (in seconds).

In each profile, the analyzer finds the longest sequence where the stride (s) between successive references is the same. If the length of that sequence is greater than $\frac{t}{2}$, and the stride is greater than a quarter of the cache block size (b), the corresponding load is considered delinquent. In this case, the instrumentation code is replaced with an instruction to prefetch data at a distance approximately five cache blocks away. A stride satisfying the constraints $s \geq \frac{b}{4}$ was empirically observed to amortize the prefetch overhead most effectively. Shorter strides do not miss as often, and longer strides can lead to one miss every four (or fewer) references.

The prefetch distance (p) is determined as follows:

$$p = (n \times object\ size) + s$$

where n is the number of objects to look ahead in the reference stream, and the stride offset adjusts for the location of a field within the object. It is evident that the prefetching heuristic is geared toward recursive data structures, with the assumption that objects are located consecutively in memory (as in an array of objects). Since our optimizer cannot quickly determine the size of objects it intends to prefetch, it approximates the prefetch distance using the block size instead. In addition, we empirically determined the value of n that leads to the best performance. Hence, the prefetch distance is approximated as:

$$p = (5 \times b) + s.$$

In order to reduce prefetch overhead, MO compares the profiles of multiple delinquent loads for redundancy. If two or more loads access the same memory locations (e.g., they share the same base register), only one prefetch instruction is added to the trace.

4. Experiment Evaluation

We evaluated our memory optimizations using two different x86 architectures. The first is an SMP system with two 1.2 GHz AMD Athlon MP processors, running Linux kernel version 2.4.18-3smp. Each processor has a 64 KB primary data cache, and an equally sized instruction cache. There is also a unified 256 KB secondary cache, with 64 byte cache lines. The other system is a 3 GHz Intel Pentium 4 processor with hyperthreading. It runs Linux kernel version 2.4.2smp. The primary data and instruction caches are 8 KB and 12 KB respectively. The unified secondary cache is twice the size of the Athlon MP. It is 512 KB with 64 byte cache lines.

The benchmarks we use are `em3d`, `health`, `mst`, `treeadd`, and `tsp` from Olden, `ft` from Ptrdist, and `art`, `equake`, `ammp`, and `twolf` from SPEC 2000. All the benchmarks were compiled with `gcc -O3`. The Olden and Ptrdist benchmarks are commonly used in the literature when evaluating dynamic memory optimizations. Many of the benchmarks from these suite run too quickly for meaningful measurements, and were therefore omitted. We also evaluated *all* of the SPEC benchmarks. We report the performance results for only those benchmarks where the performance differed more than 1%. For the SPEC benchmarks we use the reference workloads. In the case of `art` which has two reference workloads, we used only one of them (namely the one with `startx` = 110).

In Table 1 (Athlon MP) and Table 2 (Pentium 4), we report the execution times (i.e., user and system time) of each benchmark. There are five columns per benchmark. The first, labeled `native` is the measured runtime for the application running natively. The column labeled `DR` represents the program running in the DynamoRIO environment. The column labeled `pool` is the runtime measured when using DynamoRIO and pool allocation. The column labeled `pft` corresponds to the runtime when using DynamoRIO and data prefetching. The last column is labeled `all`, and it reports the runtime when using DynamoRIO along with pool allocation and data prefetching. For each benchmark, we highlight in bold the best performing optimization strategy. Note, we report the best of three measurements for each column.

The relative performance of each scenario, as compared to native execution, is illustrated in Figure 1 (Athlon MP), and Figure 2 (Pentium 4). These graphs have four bars per benchmark, and they are labeled in accord with the tables. The native performance serves as the baseline, and hence bars greater than unity imply degradation, bars equal to unity im-

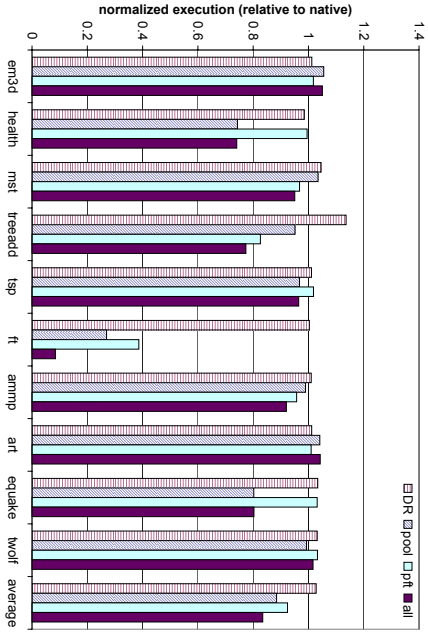

Figure 1. Athlon MP normalized execution time.

ply no change in performance, and bars shorter than unity imply a performance improvement.

On the Athlon MP system, our memory optimizations offer an average (arithmetic mean) speedup of 17% compared to native execution. In the best case, `ft` ran 10x faster. This benchmark first generates a graph and then constructs a minimum spanning tree. The pooling collocates all of the vertices in one pool, and aggregates the edge data structures in another. The reference patterns in the latter part of the computation are identical to the graph construction order, and as a result, the pool allocation strategy works quite well. The prefetching heuristics are also successful at reducing the execution time, and when combined with pool allocation, the prefetching efficacy improves further. In almost all cases where prefetching is successful in improving performance, there is usually an added benefit when the optimization is combined with pool allocation—even when pool allocation alone does not lead to significant performance gains. We believe this is due to the new prefetching opportunities exposed as a result of the custom allocation strategies. For example, in the case of `mst`, `treeadd`, and `ammp`, the performance gains when both optimizations are enabled exceed the additive advantage of each optimization alone.

In the case of the Pentium 4, the average speedup is 13%. The trends are generally similar, with two exceptions. In the case of `art`, the pool allocation strategy improves performance by 20% compared to native execution. In contrast, the same strategy does not affect the performance of `art` on the Athlon MP. This benchmark does not have any pointer data structures, but rather dynamically allocates arrays of various sizes for a neural network simulation. We believe the performance gains on the Pentium 4 are attributed to its large secondary cache, and reduced cache conflicts. We will use detailed simulations to further investigate this issue. In this benchmark, prefetching is not very effective because our heuristic ignores index memory operations. The profiling only records the base registers used in the memory access, whereas in array based computation, we will need to also record and inspect the index registers.

In the case of `treeadd`, data prefetching degrades performance significantly on the Pentium 4. The optimizer adds one prefetch instruction to the code. It is to prefetch the root node passed to a recursive function responsible for nearly 95% of the total execution time. The prefetch is quite effective on the Athlon MP. On the Pentium 4 however, the same prefetch instruction is responsible for more than a 20% degradation in performance. The difference may be due to the way prefetching is implemented in the two architectures. We used the `prefetchnta` opcode for prefetching, but in various test kernels, the impact of prefetching on performance varied across the two machines. In general, prefetching was more effective on the Athlon MP compared to the Pentium 4.

In Table 3 we report the number of prefetch instruction that are injected into the instruction streams of the various benchmarks. When pool allocation is used, the optimizer in some cases was able to perform more prefetching. The profiling analysis suggests the new opportunities are a direct result of the custom allocation strategies. For some benchmarks such as `treeadd` and `tsp` no prefetch instructions were added. As we continue this research, we will explore new heuristics to identify delinquent loads.

Finally, we note that the pool allocation strategy increases the memory footprint of the applications. This is largely due to the delayed deallocation of pools (i.e., while objects within a pool are deallocated, the heap space is not released until the pool is empty). Also, the pool allocator may reserve more heap space than necessary, especially since we double the size of pools when they are filled. However, while the memory footprint is larger, pool allocation leads to smaller working sets because of increased spatial locality.

benchmark	without pooling	with pooling
em3d	13	13
health	7	7
mst	4	6
treeadd	1	1
tsp	0	0
ft	8	10
ammp	41	44
art	11	11
equake	4	11
twolf	168	237

Table 3. Number of prefetch instructions.

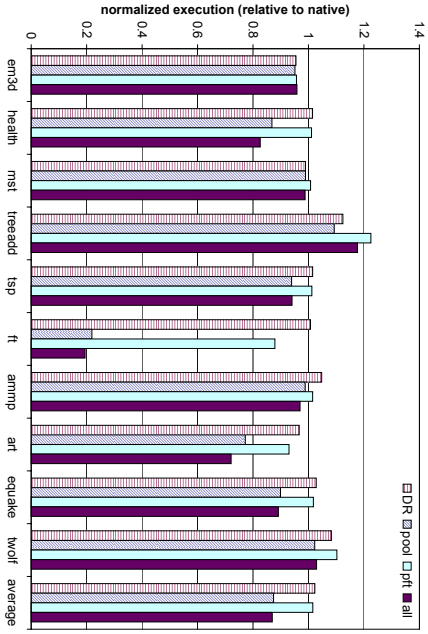

Figure 2. Pentium 4 normalized execution time.

5. Related Work

Various strategies for locality-aware heap management exist [10, 6, 3, 20, 16, 18, 12]. They are characterized by one or more of the following. First, they are not completely transparent to the programmer and may require some manual retooling of the application. Second, they may incur significant runtime overhead as objects are migrated in memory. Third, they may violate program correctness in pointer-heavy applications, due to pointer arithmetic. Some techniques try to overcome the latter with source code analysis to discover pointer aliasing, and attempt to guarantee correctness. By contrast, our approach is (i) completely automated and transparent; (ii) does not require access to the source code; (iii) does not perform any runtime data migration; (iv) always preserves correctness; and (v) is relatively lightweight, and amenable to runtime adaptation and customization. Note that if the application implements its own custom memory allocator, all of the techniques, including ours, either require a modification to the source code, or knowledge of the function names for the allocators (so that they are intercepted at runtime).

In terms of data prefetching, there is also a large body of work exploring static [4, 11, 14, 15, 21] and dynamic [5, 8, 9, 19] strategies. The static technique usually require access to the source code, whereas the dynamic strategies require architectural extensions. There are also online prefetching strategies [7] that analyze memory access patterns to predict future reference patterns and appropriately prefetch data. Others [13] uses performance counters to identify delinquent loads and predict a stride between successive references. Our work complements hardware based strategies, especially those geared toward stride prefetching. In contrast to software techniques, our work does not require access to the source code and uses simple heuristics to identify delinquent loads. Other heuristics are possible, and we continue to explore various ideas to improve our system.

Our memory optimization strategies are implemented in DynamoRIO. There are however other dynamic optimization systems. We believe our work is equally applicable in such systems.

6. Conclusion and Future Work

In this paper, we described a runtime memory optimizer (MO) to perform locality-aware data layout, and data prefetching. Our framework does not rely on static compiler analysis, and is implemented in DynamoRIO, a dynamic optimization infrastructure. As part of DynamoRIO, we can run a broad range of applications directly on commercial off the shelf processors. The applications include pointer-chasing codes written in weakly-types languages such as C and C++. MO yields performance gains between 13-17% on average for an Athlon MP and a Pentium 4.

One of the distinguishing characteristics of our memory optimizer is that it does not require any knowledge of the program source code and data structures. Instead, it implements simple heuristics to spatially cluster objects that are likely instances of the same data type. In applications where dynamic data allocation patterns are similar to subsequent traversal patterns, the clustering boosts spatial locality and leads to improved memory system performance. In addition, the pooling together of objects leads to greater regularity in the access patterns, thereby creating more opportunities for data prefetching. Our optimizer leverages these opportunities to perform data prefetching. MO implements a lightweight profiler to identify delinquent loads, and injects prefetch instructions into the code stream when doing so is profitable.

We are currently extending our infrastructure to allow for *cumulative optimizations* where dynamic optimizations are carried forward from one run to another. We are also experimenting with new heuristics for pool allocation and delinquent loads identification. Lastly, while our infrastructure is Linux based, we plan to target systems running Microsoft Windows, and evaluate our memory optimizations using desktop applications.

Acknowledgements

We thank the anonymous referees for their comments and suggestions on an earlier version of this paper. This research was supported in part by the Singapore-MIT Alliance.

References

[1] S. G. Abraham and B. R. Rau. Predicting load latencies using cache profiling. Technical Report HPL-94-110, Hewlett Packard Labs, Dec 1994.

[2] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, September 2004. http://www.cag.csail.mit.edu/rio/.

[3] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–149, Oct. 1998.

[4] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, 1991.

[5] M. Charney and A. Reeves. Generalized correlation-based hardware prefetching. Technical Report EE-CEG-95-1, Cornell University, Feb. 1995.

[6] T. Chilimbi, M. Hill, and J. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, May 1999.

[7] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 199–209. ACM Press, 2002.

[8] J. Fu and J. Patel. Stride directed prefecthing in scalar processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 102–110, Dec. 1992.

[9] D. Joseph and D. Grunwald. Prefetching using Markov predictors. *IEEE Transactions on Computers*, 48(2):121–133, Feb. 1999.

[10] T. Kistler and M. Franz. Automated data-member layout of heap objects to improve memory-hierarchy performance. *ACM Transactions on Programming Languages and Systems*, 22(3):490–505, May 2000.

[11] A. C. Klaiber and H. M. Levy. An architecture for software-controlled data prefetching. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 43–53, 1991.

[12] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 129–142, New York, NY, USA, 2005. ACM Press.

[13] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen. The performance of runtime data cache prefetching in a dynamic optimization system. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 180, Washington, DC, USA, 2003. IEEE Computer Society.

[14] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Oct. 1992.

[15] T. Ozawa, Y. Kimura, and S. Nishizaki. Cache miss heuristics and preloading techniques for general-purpose programs. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, Nov. 1995.

[16] P. Panda, N. Dutt, and A. Nicolau. Memory data organization for improved cache performance in embedded processor applications. *ACM Transactions on Design Automation of Electronic Systems*, 2(4):384–409, 1997.

[17] R. Rabbah, H. Sandanagobalane, M. Ekpanyapong, and W.-F. Wong. Compiler orchestrated prefetching via speculation and predication. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 189–198, New York, NY, USA, 2004. ACM Press.

[18] R. M. Rabbah and K. V. Palem. Data remapping for design space optimization of embedded memory systems. *Trans. on Embedded Computing Sys.*, 2(2):186–218, 2003.

[19] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, pages 42–53, 2000.

[20] D. Truong, F. Bodin, and A. Seznec. Improving cache behavior of dynamically allocated data structures. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 322–329, Oct. 1998.

[21] Y. Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 210–221, 2002.