MPADS: Memory-Pooling-Assisted Data Splitting

Stephen Curial
Xymbiant Systems Inc.
scurial@gmail.ca

Peng Zhao
Intel Corporation
peng.zhao@intel.com

José Nelson Amaral
Department of Computing Science,
University of Alberta
amaral@cs.ualberta.ca

Yaoqing Gao Shimin Cui Raul Silvera Roch Archambault
IBM Toronto Software Laboratory
{ygao,scui,rauls,archie}@ca.ibm.com

Abstract

This paper describes Memory-Pooling-Assisted Data Splitting (MPADS), a framework that combines data structure splitting with memory pooling.[1] MPADS relies on pointer analysis to ensure that splitting is safe and applicable to type-unsafe language. MPADS makes no assumption about type safety. The analysis can identify cases in which the transformation could lead to incorrect code and thus MPADS abandons those cases.

To make data structure splitting efficient in a commercial compiler, MPADS is designed with great attention to reduce the number of instructions required to access the data after the data-structure splitting. Moreover the implementation of MPADS reveals that architecture details should be considered carefully when re-arranging data allocation. For instance one of the most significant gains from the introduction of data-structure splitting in code targetting the IBM POWER architecture is a dramatic decrease in the amount of data prefetched by the hardware prefetch engine without a noticeable decrease in the cache utilization. Triggering fewer hardware prefetch streams frees memory bandwidth and cache space. Fewer prefetching streams also reduce the interference between the data accessed by multiple cores in modern multicore processors.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors

General Terms compilers, optimization

Keywords memory management, allocation strategies, memory pooling

1. Introduction

Lattner and Adve and Shin *et al.* have convincingly demonstrated that significant performance improvements can be realized by improving the memory placement of linked-based data structures (9; 17). Zhong *et al.* and Rabbah and Palem proposed expensive affinity-based analysis, based on trace or profile information, for controlling the layout of pointer-based data structures (24; 15). However, Zhao *et al.* found that a simple and inexpensive *maximal splitting* strategy, which splits an array of aggregated data structures into many arrays, with each element containing a single field, works very well in standard benchmarks (23).

An open question is whether maximal splitting can be applied to pointer-based data structures that are not organized into arrays. Moreover, is it possible to combine maximal splitting with Lattner and Adve's memory pooling in a commercial compiler for type-unsafe language? Targeting a commercial compiler is important because such an environment imposes constraints that are not seen in optimizing compilers developed for research purpose: the analysis cannot rely on memory traces — traces are too large for commercial applications; the analysis must be conservative — whenever there is a possibility that the transformation will change the semantics of the program, the transformation must be abandoned; and the analysis must work with any legal construct in the language, including unusual code idioms that may not appear in benchmarks.

Splitting data structures is a data reorganization technique that can significantly increase the spatial locality of data and reduce the runtime of programs that use link-based data structures (2; 4; 7; 20). Memory-Pooling-Assisted Data Splitting (MPADS) is a framework designed to safely and automatically split pointer-based data structures without adding padding. MPADS' pointer analysis guarantees that it is safe to split a given structure even when the program is written in a weakly-typed language like C or C++.

MPADS introduces two different splitting techniques that trade address computation and storage overhead for increased data locality that reduces the number of cycles that processors spend stalled waiting for data. MPADS extends Lattner and Adve's Data Structure Analysis (DSA) that is used to identify type-homogeneous structures (8; 9).

The implementation of MPADS includes Lattner and Adve's pool allocation; a simple annotation to the existing pointer analysis to keep track of allocation sites and to identify structures whose layout can be safely transformed; and the introduction of new address computation code to every statement in the program that contains a reference to transformed data structures.

Some of the highlights of MPADS are that it does not require profile information or collection and analysis of memory trace — the importance of implementing this code transformation without such requirements cannot be understated. Also, MPADS does not

[1] Although it MPADS may call to mind "memory padding," a distintion of this framework is that is *does not* insert padding.

ISMM '08 June 7–8, 2008, Tucson, Arizona, U SA.

require padding of fields, and the address computation code introduced is designed to minimize its impact on runtime performance.

MPADS is not the first implementation of the combination of memory pools and splitting of pointer-based data structures. In April of 2007 Jeon, Shin and Han published their structure-splitting (6). While the technique presented in this paper was developed before their work appeared and in a completely independent fashion, their structure splitting framework is similar to the non-uniform splitting described in Section 2.2.2. Section 2 describes MPADS. A detailed comparison of their system and MPADS is given in section 4.

MPADS improves upon the state of the art data splitting systems (6; 15; 24) by reducing the overhead of the address computation. MPADS also shows that, for some benchmarks, using a much faster, and less precise, pointer analysis is feasible. The experimental results in Section 3 shows that while a series of microbenchmarks showcase the potential performance improvements due to MPADS, it still is not delivering its full potential on standard benchmarks in the IBM XL compilers. The culprit is the unification-based alias analysis that the compiler uses and that prevents the analysis from identifying profitable opportunities.

2. MPADS: Memory-Pooling-Assisted Data Splitting

2.1 Memory Pooling

When dynamically allocated objects of different types may are mapped to the same cache line the result is poor locality, a polluted cache and an increase in the number of cache misses.[2] Memory pooling is a memory allocation policy for dynamically allocated objects that places objects of the same type together in a *memory pool* to improve data spatial locality. Memory pools also tend to result in frequent cache reuse and less capacity misses.

2.1.1 Memory Allocation Library

Standard memory allocation functions do not provide support to allocate a group of objects together. Thus, a memory allocation library that can allocate similar objects in a pool must be created. The memory allocation calls are similar to those provided in the standard C library. The difference is that each object is allocated in a pool that is managed by the memory allocation library.

The memory allocation functions take a *structure identifier* as a parameter. The structure identifier is used to tell the memory allocation library which allocations should be grouped together.

2.1.2 APIs

The Memory Allocation Library must provide support for the common memory allocation calls found in the standard C library: malloc, calloc and free. The APIs currently supported are:

- `void* pool_alloc(unsigned int struct_id, size_t struct_size, size_t pool_size);`
- `void* pool_calloc(unsigned int struct_id, size_t num_objs, size_t struct_size, size_t pool_size);`
- `void pool_free(void* ptr);`

2.1.3 Memory Pools

The Memory Allocation Library manages a set of pools for each data structure. Distinct data structures are identified by the structure identifier that is passed into the allocation function. The pools have a fixed size, typically the same size as, or larger than, a page. Data is allocated contiguously within the pool until the pool is full. When the pool is full, another pool is allocated and more memory can be allocated in this newly created pool.

Memory can be freed from the pools by using the Memory Allocation Library's free function. The freed objects are stored in a list and the memory can then be assigned to another allocation. When all of the memory in the pool is freed, the pool can be reclaimed.

Using multiple pools to store the data for each data structure allows MPADS to use only a small amount of additional memory while not limiting the framework to a fixed number of structures that can be allocated.

2.1.4 Compiler Transformation

MPADS uses the results of the pointer analysis to differentiate objects and allocate each structure in its own pool. To do this, MPADS must first determine which structures should be grouped together and then must replace the memory allocation calls with calls to the custom-made Memory Allocation Library.

MPADS uses a Steengard's style unification-based analysis (19). Thus an *alias set* is a set of pointers that may point to the same object(s). An *allocation site* in the program is a call to malloc, calloc, realloc, alloca, valloc, strdup, memcpy, memalign or posix_memalign.[3] The alias analysis has been modified to collect the allocation sites during the same pass that performs the pointer analysis. When two alias sets are unified during the analysis the list of associated allocation sites is also unified. All objects that may be pointed to by a pointer in an alias set are allocated in the same pool. After the pointer analysis is complete the compiler iterates through the list of allocation sites for each candidate, and transforms the allocation to the corresponding call from the memory allocation library. The structure identifier from the alias set is passed as the `struct_id` parameter to the allocation function. MPADS also creates a list of the deallocation sites during the alias analysis. The same process that was performed for the allocation sites is performed for deallocation sites. If an object is flagged to be transformed, the deallocation sites are also changed to use the corresponding functions in the Memory Allocation Library.

If a non-standard allocation function is used to allocate an object that is associated with an alias set, that set is not a candidate for pool allocation. The usage of non-standard memory allocators suggests that the programmer may be manually tuning the application. In this case interfering with the placement of the objects in memory may actually degrade performance.

2.2 Structure Splitting

Memory pooling increases the locality of data by grouping similar structures together. Memory pooling works well when a traversal of a data structure references several fields of each object before moving to the next object. However, often a traversal of linked-based objects only accesses a small fraction of the fields in each object. The non-referenced fields are likely to share a cache line with the fields that are referenced. These non-referenced fields pollute the cache and waste valuable memory bandwidth. Structure splitting is a technique that addresses this problem.

When a structure is split, all of the similar fields in each structure are grouped together. For example, all of the first fields in a structure are allocated near each other, all of the second fields in a structure are allocated near each other and so on. The result is that the fields with the same offsets in different instantiations of the structures have good spatial locality.

[2] In this paper *object* refers to an instantiation of an aggregated data structure.

[3] The current implementation of MPADS only transform objects that are allocated through a call to malloc or calloc in the standard C library.

Figure 1 gives an example of how three structures, `A`, `B` and `C` are allocated both with and without splitting. Each structure in the example has 4 fields, `f1`, `f2`, `f3` and `f4`. When structures are allocated without splitting, as shown in Figure 1(a), the fields of each structure are located next to each other in memory: `A.f1`, `A.f2`, `A.f3` and `A.f4`. MPADS organizes the data as shown in Figure 1(b). In the split version, fields `A.f1`, `B.f1` and `C.f1` now have good spatial locality.

Splitting data structures may improve performance in several ways. If the traversal of a data structure only accesses a few fields of the structure, then splitting greatly increases locality, reduces the size of the working set, reduces the memory traffic, reduces the number of capacity misses and does not pollute the cache. Splitting the data also creates data streams that can be prefetched by hardware prefetchers. Most hardware prefetch engines support prefetching multiple data streams simultaneously.

There are three common methods to split structures: affinity-based splitting, frequency-based splitting and maximal splitting. Affinity-based splitting typically requires a profiling run to analyze and determine the affinity of the fields in a structure. Fields with high affinity are grouped together and then the structure is broken into groups based on the field affinity. Frequency-based splitting also needs information about how often each field is accessed and this is typically obtained from a profile or memory trace. The fields are grouped into frequently-accessed fields, known as *hot fields*, and infrequently-accessed fields, or *cold fields*. The structure is split to separate the hot and cold fields. Maximal splitting does not group any of the fields in a structure together, it completely separates every field in the structure.

Zhao *et al.* studied data splitting techniques and found that maximal splitting can achieve the best, or near-best, performance when compared with affinity- and frequency-based splitting (23). MPADS uses maximal splitting.

Splitting a data structure changes the address computation required to access a field of an object. The new address computation must be efficient because memory references occur frequently. Without careful design, the additional overhead from adding instructions for the new address computation may not be offset by performance improvement from increasing data locality. To reduce the overhead of address calculation MPADS uses two different techniques for structure splitting depending on the layout of the structure

2.2.1 Splitting Structures with Uniform Layout

If all of the fields in the structure are of the same length then the address computation is simpler and more efficient than the case where the fields are different lengths.

The access to a field via a pointer p, before splitting, is calculated as: *(p + *offset*) where the value of *offset* is typically small. After splitting the pointer dereference is still computed as *(p + *offset*) but now the *offset* is a much larger value. When uniform splitting is used, the new offset for the i-th field in a data structure, f_i, can be calculated as:

$$num\_structs\_per\_pool = \frac{pool\_size}{\sum_i sizeof(f_i)} \quad (1)$$

$$f_i\_offset = length(f_i) * num\_structs\_per\_pool * (i-1) \quad (2)$$

If the target processor has a base-plus-offset addressing mode, there is likely a limited number of bits available to use for the offset. In such architectures, either the pools can be made small enough to ensure that the offset can be accommodated in the base-plus-offset addressing mode or an additional add instruction can be used before the memory access.

2.2.2 Splitting Structures with Non-Uniform Layouts

A structure with non-uniform layout is a structure comprised of fields that have different lengths. MPADS still allocates fields into a pool and splits them maximally, but the address calculation required to reference a field is more complicated.

A drawback of using multiple pools for splitting structures with fields of multiple sizes is that for each field access, the start address of the sub-pool in which the field resides must be computed at runtime. For example, consider a pool that has several objects allocated in it, shown in Figure 2. Let the length of fields 1, 2, 3 and 4 be 2, 4, 4 and 8, respectively.

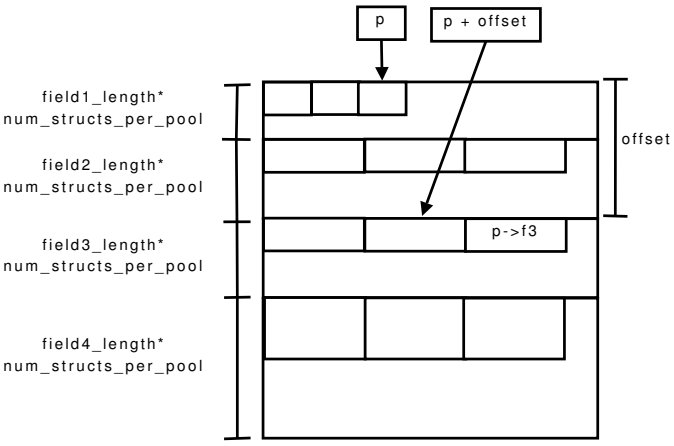

Figure 2. Example illustrating why the index in the pool must be known for non-uniform splitting.

Assume that there is a pointer `p` and that the program contains a reference to field `f3`: `p->f3`. Assume that the pool can store 100 objects and that p points to the third object allocated in the pool. If MAPS were to use equation 2 to calculate the offset, it would obtain $(2*100)+(4*100) = 600$. However, this offset is actually 4 bytes short of the location that should be accessed. The dotted arrow in Figure 2 shows the data that would be accessed if the offset was 600 bytes. Thus, to access the correct location MAPS needs to compute how many objects have been allocated in the pool before the structure referenced by the pointer.

Using the runtime library to search for the start of the pool that the pointer belongs to and then returning its index in the pool is extremely expensive. To make address calculation inexpensive, MPADS aligns the memory allocated for the pools on boundaries that are multiples of the size of the pool. If the pools are aligned then a simple bitwise *and* operation can be used to find the index of the object. The bit mask, $mask$, can be calculated by the expression $\sim (poolSize - 1)$.[4] The index of the object into the pool is given by:

$$index = \frac{p \,\&\, mask}{sizeof(f_1)} \quad (3)$$

where $\&$ stands for the bitwise logic AND operation and f_1 is the first field in the data structure. Let N be the maximum number of structures that can be allocated in the pool, and $S(f_i)$ be the size, measured in bytes, of field f_i. The offset of f_1 is zero. The offset of all other fields is given by:

$$offset(f_i) = S(f_i) * index - p\&mask + \sum_{j=1}^{i-1} S(f_j) * N \quad (4)$$

[4] This calculation of the bit mask requires that the pool sizes be a power of 2.

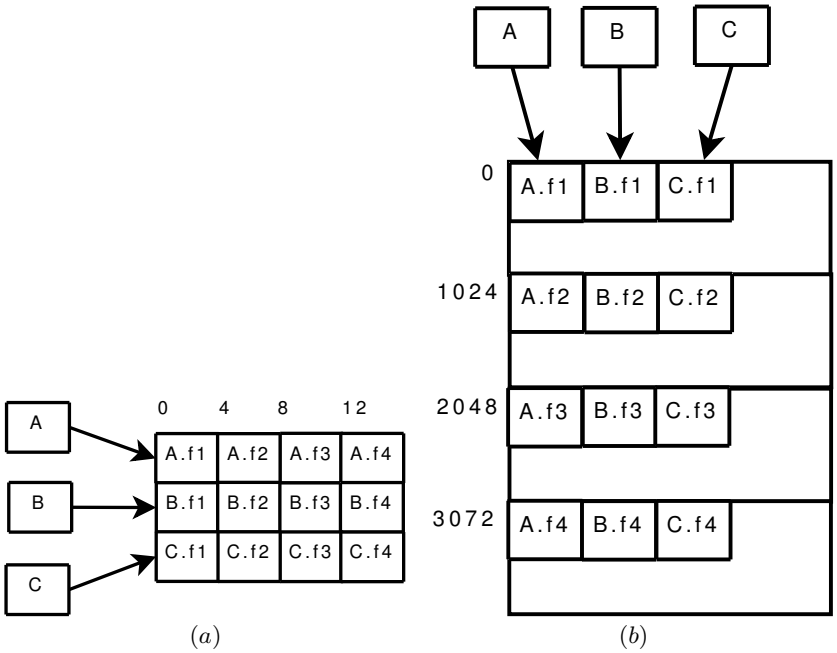

Figure 1. An example of (a) three structures allocated without splitting and (b) three structures allocated with MPADS Maximal Splitting.

The calculation of $offset(f_i)$ using equation 4 is shown graphically in Figure 3.

All of the sub expressions in equation 4, except for $index$ and p, are known at compile time and can be folded to further reduce overhead. Moreover, when $S(f_1)$ is a power of 2 the compiler can use strength reduction to replace the division with a bit-shift operation. [5] The compiler can reorder the fields in the data structure to place a frequently referenced field whose size is a power of two as the first field in the structure. Moreover, if the referenced field has the same length as the first field in the structure, then $S(f_i) * index = p\&mask$ and the offset computation is greatly simplified.

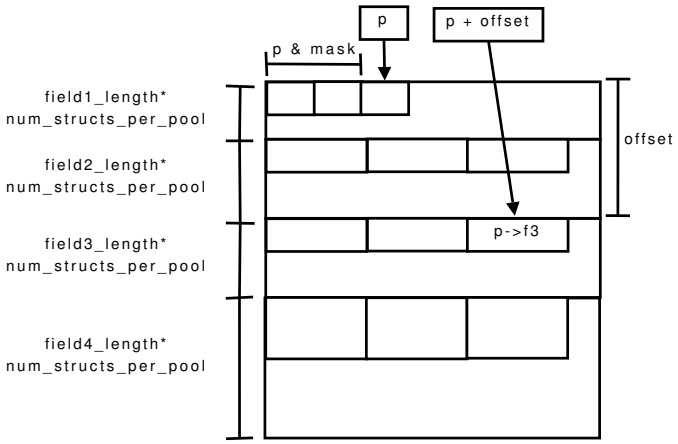

Figure 3. Pointer access with MPADS Non-Uniform Maximal Splitting

[5] The $S(f_1)$ is known at compile time because the length of each field in the structure must be known for the transformation to be identified as safe.

2.3 Identifying Structures to Transform

MPADS must discover data structures that are candidates for the transformation and ensure that the transformation will not alter the semantics of the program. For a transformation of an object X to be safe, MPADS must identify all the pointers that may point to X. Then MPADS must determine that all the objects pointed-to from this alias set have the same layout.

In MPADS two objects have the same layout if their byte-level view is the same. Formally, given two structures s_1 and s_2 with n fields in each structure, s_1 and s_2 have the same layout if and only if for $0 \leq i < n$, $lengthof(s_1.f_i) = lengthof(s_2.f_i)$ AND $offset(s_1.f_i) = offset(s_2.f_i)$, where f_i is the i-th field of s_1 and s_2. MPADS is not concerned about the data type of the fields because splitting the structures only changes where the data is located and not how it is used. To identify safe candidates, MPADS combines the alias sets computed by the alias analysis with the object layout information from the compiler's symbol table. Currently MPADS does not split structures where one of the fields is itself an aggregated data structure. MPADS can be extended to handle this case by moving the fields of the nested structure into the parent.

MPADS' pointer analysis is an unification-based, field-sensitive, flow-insensitive and context-insensitive inter-procedural Steensgaard-style analysis (19). This analysis was chosen because it scales to large programs. It is important for the alias analysis to be field-sensitive because structures often contain many fields of different types and the coarse granularity of a field-insensitivity analysis could result in many missed opportunities.

The pointer analysis provides more information than simply a check for safety, it is also used to determine in which pools the candidate structures should be allocated (9). Using a unification-based alias analysis results in all the pointers that access a particular data structure to be in the same alias set. Different alias sets represent different objects and must be allocated in different pools.

2.3.1 Combining Structure Splitting with Memory Pooling

When structure splitting is combined with memory pooling, calls to allocation and deallocation functions are still intercepted the same way as in the implementation of pool allocation but now a slightly different function must be used. The allocation function for structure splitting still groups similar objects together, but the location and pattern of the memory for each field that is allocated differs from the pool allocation routines.

The main difference between the allocation function for structure splitting and pool allocation is that the pool allocation library returns addresses that are separated by the size of the object allocated while the allocation function for structure splitting returns addresses separated by the size of the first field of the object.

Consider memory pooling without structure splitting. For instance, assume that the pool size is 4k and that the program is allocating a 16-byte structure consisting of four 4-byte fields. A call to the pool allocation function returns memory address m and allocates memory in locations $[m, m + 15]$. Locations $[m, m + 3]$ are reserved for the first field, $[m + 4, m + 7]$ for the second field and so forth. The second call to the pool allocation function returns $m + 16$ and allocates $[m + 16, m + 31]$.

Using the same example with structure splitting, each of the four fields will occupy one quarter of the pool or 1024 bytes. The first field of the first object is allocated at m and the allocation function would return the address m. The second, third and fourth fields of the first object are located at $m + 1024$, $m + 2048$ and $m + 3072$, respectively. The first field in the second object allocated in the pool is located at $m + 4$ with the second, third and fourth fields of the second object located at $m + 1028$, $m + 2052$ and $m + 3076$, respectively.

For non-uniform splitting, pools must be aligned by the pool size and are allocated using the `posix_memalign` system call. MPADS requires that the pool size be known at compile time to reduce the cost of address computation. To make the memory library more flexible, the pool size can be passed as a parameter. MPADS automatically generates this parameter and uses the same value for the address calculation.

The APIs for the splitting functions include parameters for the size of the first field in the structure. The size of the first fields in the structure must be known for the allocation function to return the correct address. Further description of pool allocation is given by Lattner and Adve (9).

- `void* split_alloc(unsigned int struct_id, size_t first_field_size, size_t struct_size, size_t pool_size);`
- `void* split_calloc(unsigned int struct_id, size_t first_field_size, size_t num_objs, size_t struct_size, size_t pool_size);`
- `void split_free(void* ptr);`

2.3.2 Code Transformation

Once the candidates for safe splitting are identified, MPADS replaces the corresponding calls to memory allocation and deallocation functions with the calls in the API described above.

To change pointer accesses MPADS recursively traverses the abstract-syntax-tree representation of the code searching for an indirect load or store — a load of an address from the stack followed by a load or store. Once an indirect load or store is found the compiler determines which alias set the pointer is a member of. If the corresponding alias set has been flagged as a candidate for splitting, the address calculation used in the indirect load or store is changed to use either the uniform split or non-uniform splitting addressing described in sections 2.2.1 and 2.2.2.

The offset for the first field in each structure is always 0 and can be accessed without a costly address computation. To try to improve the performance of the transformed program MPADS should put the most frequently accessed field at offset 0. Since profile information is not available, MPADs assumes that recursive fields are accessed more frequently and thus MPADS makes a recursive field the first field. If there are multiple recursive fields MPADS arbitrarily picks one of them to be the first field. Because recursive fields usually contain addresses, whose size is a power of two, this strategy also simplifies the division operation in the address computation for non-uniform splitting.

Although MPADS doesn't require fields to be aligned, some archictectures require memory access to be aligned. An extension of MPADS could pad fields so they are aligned.

2.4 Dynamic Memory Footprint vs. Static Memory Footprint

Memory pooling can increase the memory requirements, also known as the static memory footprint, of the application because the last allocated pool may not be full. However, the amount of extra space required will be amortized when many pools are allocated. The static memory footprint of an application have little or no effect in the application's memory performance.

The motivation behind data structure splitting is to trade increased space requirements for faster application execution. To improve the cache performance, the developer has to reduce the size of the dynamic memory footprint. A working definition for dynamic memory footprint is the number of distinct cache lines (or memory pages) that the application actually references at runtime. Smaller working sets allow more data to fit into cache and can reduce the number of cache misses. The MPADS transformation is designed to reduce the size of the working set by splitting the data structures to place fields that are accessed together close to each other in memory.

2.5 Implementation in the IBM XL Compiler

The MPADS transformation is implemented in the Toronto Portable Optimizer (TPO) in the IBM XL compiler. MPADS required an inter-procedural pointer analysis to guarantee safety and thus it is a natural choice to implement MPADS in the TPO, which performs whole program optimization and analysis.

The TPO performs two passes over the program, the first pass collects information and analyzes the code while the second pass modifies the program. The MPADS framework was easily integrated into the 2 passes that the TPO performs. On the first pass the pointer analysis is performed and candidate structures are identified. On the second pass the candidate allocation sites and pointer de-references are modified.

MPADS added very little additional overhead to the compiler. The pointer analysis that MPADS uses is already performed by the TPO as part of the *Forma* array reshaping transformation (23). Additionally, MPADS does not need to make any additional passes over the code because the pointer analysis provides enough information for the actual code transformation process to be done locally, almost as though it is a peep-hole optimization.

3. Experimental Evaluation

The results of the experimental evaluation of MPADS can be summarized as follows. For all of the larger benchmarks tested, MPADS outperforms memory pooling. The memory pooling optimization that is being used for comparison is MPADS without structure splitting: the calls to the memory allocation functions are replaced but the pointer calculations are not changed (3). For `llu`, MPADS cut the execution time in half. However, the results for the rest of the benchmarks are mixed. Many potential opportunities

were abandoned because the pointer analysis did not have enough precision and thus the transformation did not have as large an impact as expected. As well, the transformation caused one of the benchmarks, `health`, to have worse cache behavior and run 9% slower than the baseline.

3.1 Benchmarks

Benchmarks from 3 sources were used: SPEC 2000, Olden (16) and LLU (25). The Olden and LLU benchmarks were chosen because they have been used to evaluate code transformations that aim to improve cache performance and because they contain pointer-based data structures (2; 9; 20). The SPEC 2000 benchmarks were chosen because they are the *de facto* standard for performance measurement in the industry. None of the benchmarks are multithreaded.

The benchmarks tested are comprised of C and C++ programs that use linked data structures. The size and layout of the data structures in the benchmarks varies. Some benchmarks use a standard linked list while others use structures such as a linked list of linked lists, or quad-trees. MPADS performs an analysis to identify candidates to split and should be able to split the structures regardless of the layout. Optimization opportunities are not discovered in several of the benchmarks. Those benchmarks are not included in the results because a transformation was not performed on them and accordingly there is no change in their performance.

Opportunities are identified in only 5 of the SPEC 2000 benchmarks and the opportunities that were identified were responsible for referencing only a small fraction of each application's data. As a result the transformation did not have a measurable impact on any of the SPEC 2000 benchmarks. Although MPADS did not identify any significant opportunities in SPEC we believe that opportunities exist. Lattner and Adve's Data Structure Analysis (DSA) has successfully identified candidates in SPEC 2000 (11; 12). They use a context-sensitive, field-sensitive, flow-insensitive unification-based pointer analysis that is more precise then the Steensgaard's style analysis used in MPADS.

3.2 The Potential for Gain

Several simple programs containing data structures that could be split by MPADS were written for initial testing of the code transformation. Even though the performance gains in these simple programs is not representative of what should be expected from complete applications, the performance improvements in the POWER4 processor speed improvements shown in Figure 4 are indicative for the potential for performance improvement due to data structure splitting.[6] The labels in the graph columns describe what these simple programs do (3).

3.3 Experimental Setup

The benchmarks are evaluated on two different hardware architectures and are compiled with the IBM XL compiler at the highest optimization level, -O5. The machines used for evaluation are a 1.7 GHz POWER4 machine and a 1.9 GHz POWER5 machine. The pertinent information about the memory hierarchy configuration of each machine and the memory latencies are given in Table 1.

All of the timing results are calculated by taking the smallest running time from 10 runs of the application. The performance metrics are gathered using the *tcount* tool that monitors the hardware counters and are gathered during a separate run so that they do not affect the timing results (21).

3.4 Results

Given the tight integration of MPADS in the two passes of TPO and the use of a pointer analysis that the compiler already performs, there was no noticeable increase in compilation time with the introduction of MPADS.

In all bar graphs the Baseline is the program compiled with the -O5 flag, Pool Allocation is a measurement for the application compiled with the -O5 flag and the pool allocation optimization, and MPADS use the MPADS optimization with the -O5 flag.

The MPADS transformation either outperformed or tied the performance of memory pooling on every benchmark. The speedup for each of the benchmarks after the transformations is given in Figure 5. Both memory pooling and MPADS had larger impacts on the POWER4 processor than on the POWER5. On the POWER4, MPADS improved 5 benchmarks and memory pooling only improved 3. On the POWER5 MPADS and memory pooling only improved 2 benchmarks but MPADS improved LLU by 27% more then memory pooling.

[6] POWER5 results show similar trends.

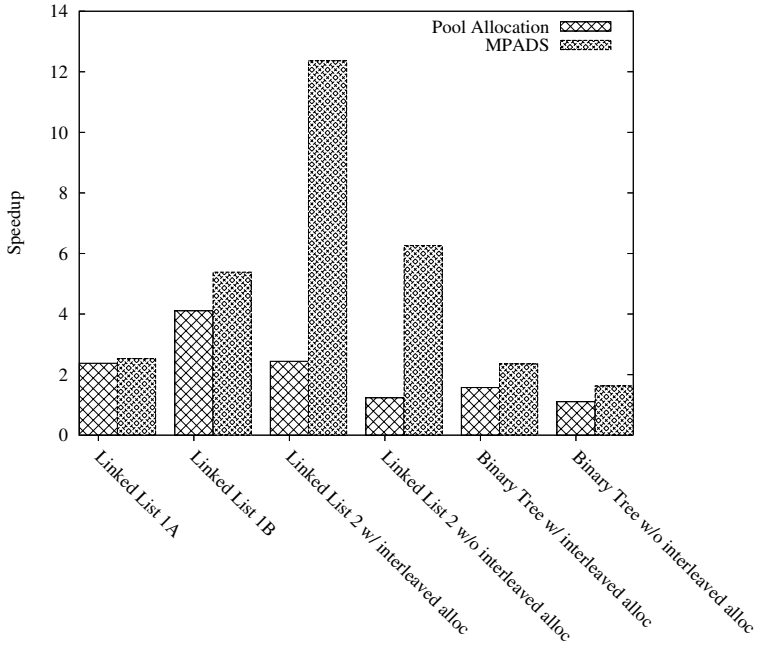

Figure 4. Speedup for small programs on a POWER4.

	POWER 4	POWER 5
L1 Data Cache	32kb 2-way associative 128 byte cache line Latency: 1 cycle	32kb 4-way associative 128 byte cache line Latency: 4 cycles
L2 Cache	1.44Mb shared per chip 8-way associative 128 byte cache line Latency: 8-12 cycles	1.9Mb shared per chip 10-way associative 128 byte cache line Latency: 14 cycles
L3 Cache	32Mb per chip 8-way associative 512 byte cache line Latency: 118 cycles	32Mb per chip 12-way associative 256 byte lines Latency: 80 cycles
TLB	1024 entries 4-way set-associative Latency: 250 cycles	1024 entries 4-way set-associative Latency: 351 cycles

Table 1. Cache Configuration

Benchmark	Instr. Count (in 100,000,000)			CPI		
	baseline	Pool	MPADS	baseline	Pool	MPADS
bh	503	0.02	-1.1	1.6	-0.1	0.4
em3d	980	0.1	-6.9	0.6	4.9	13
health	343	5.3	2.3	2.4	-15	-7.4
power	16	-0.1	-0.3	4.6	0.3	5.8
tsp	1215	5.5	-6.2	2.4	-0.7	13
llu	966	1.0	-11	8.9	43	62
Total/avg	17378	2.0	-3.9	5.5	17	29
	L1D Misses (in 100,000)			L2 Misses in 100,000)		
	baseline	Pool	MPADS	baseline	Pool	MPADS
bh	7272	-11	-5.4	442	-2.5	-1.1
em3d	1372	13	13	59	-20	14
health	769	-20	-14	63	0.94	4.1
power	7.1	6.3	-3.6	0.25	12	2.6
tsp	397	-3.8	1.8	64	25	58
llu	18934	13	-46	527	62	42
Total/avg	28751	-2.7	-9.0	1156	13	20
	L3 Misses (in 100,000)			DTLB Misses (in 100,000)		
	baseline	Pool	MPADS	baseline	Pool	MPADS
bh	14.7	22	13	218	-2.3	-2.0
em3d	10.5	-13	7.8	410	12	13
health	39.7	-13	-19	22	8.4	-5.5
power	0.05	8.2	3.4	0.24	-36	0.43
tsp	25	33	61	14	18	11
llu	0.8	19	75	5.4	-5.1	57
Total/avg	91	9.4	24	670	7.4	7.6

Table 2. Hardware counter measurements in POWER 4 architecture. The baseline columns is the number of events, the Pool and MPADS are percentage decrease in relation to baseline.

Table 2 summarizes the study of performance variations due to the implementation of pool allocation and MPADS in the POWER 4 architecture. The number of instructions for the baseline is expressed in 100 millions. The baseline for all other events is expressed in 100,000s. The numbers under the Pool and MPADS columns represent the percentage reduction in the number of events in relation to the baseline for that version of the program. For instance, for the `health` benchmark the use of pool allocation results in a decrease of -13% in the number of L3 cache misses, which means that the pool allocation increases the number of misses in this program by 13% while MPADS eliminates 8.4% of the DTLB misses in the same benchmark. The Total/avg row reports the total number of events in all benchmarks and the corresponding average variations in this total for pool allocation and MPADS. For CPI the Total/avg row has the average CPI for baseline, and the average percentage variation in CPI for the other columns.

The results on the top section of Table 2 are as expected: MPADs increased the number of instructions executed by 3.9% on the whole suite of benchmarks because structure splitting increases the number of instructions executed during the address calculation of most references, but reduces the average number of clocks per instruction by 29% because there are fewer stall cycles.

Even though MPADS increased the number of L1D misses for many benchmarks it still obtained a speed up in this benchmarks. The number of L2 and L3 misses indicate that MPADS decreased the number of misses at the lower levels of cache and these reductions outweighed the increases in L1D misses.

Benchmarks `bh`, `em3d`, `power` and `tsp` had much smaller performance improvements on the POWER5 compared to the POWER4 because there was a much less significant reduction in misses on the POWER5.[7] As a result the number of cycles spent stalled only sightly decreased on the POWER5 and most of that gain was eaten up by the overhead of the extra address-calculation instructions.

The LLU benchmark simulates a linked list and was proposed as a replacement to the `healths` benchmark that may not be representative of a typical linked-list data structure (25). LLU received the largest speedup from MPADS, 2.07 on the POWER4 and 1.72 on the POWER5. It's interesting that MPADS increased the number of L1 cache misses but decreased the number of DTLB, L2 and L3 cache misses. The L1 misses increased because the benchmark accesses many of the fields in the list nodes at the same time and the reorganization of the data caused poorer L1 cache performance. However, for the larger L2 and L3 caches the data reorganization allowed them to implicitly prefetch more data and significantly reduced the number of cache misses.

3.5 Hardware Prefetching

Most modern processors – including Intel's Core Duo (22), AMD's Athlon (5), Sun's UltraSPARC (13) and IBM's POWER series (18) – use hardware-based stride prefetchers to automatically identify strided data accesses. Once the prefetch engine has identified a strided data access, it can efficiently issue prefetch instructions. Typically, hardware prefetch engines only detect streams within a page and need to issue a new prefetch stream as the access move between pages.

The MPADS transformation increases spatial locality and reduces the size of the working set. This increased locality causes fewer prefetch streams to be allocated and fewer prefetches to be issued. Requiring fewer prefetch streams frees up resources to

[7] Due to space constraints the performance counter measurements for the POWER 5 architecture are not shown in the paper.

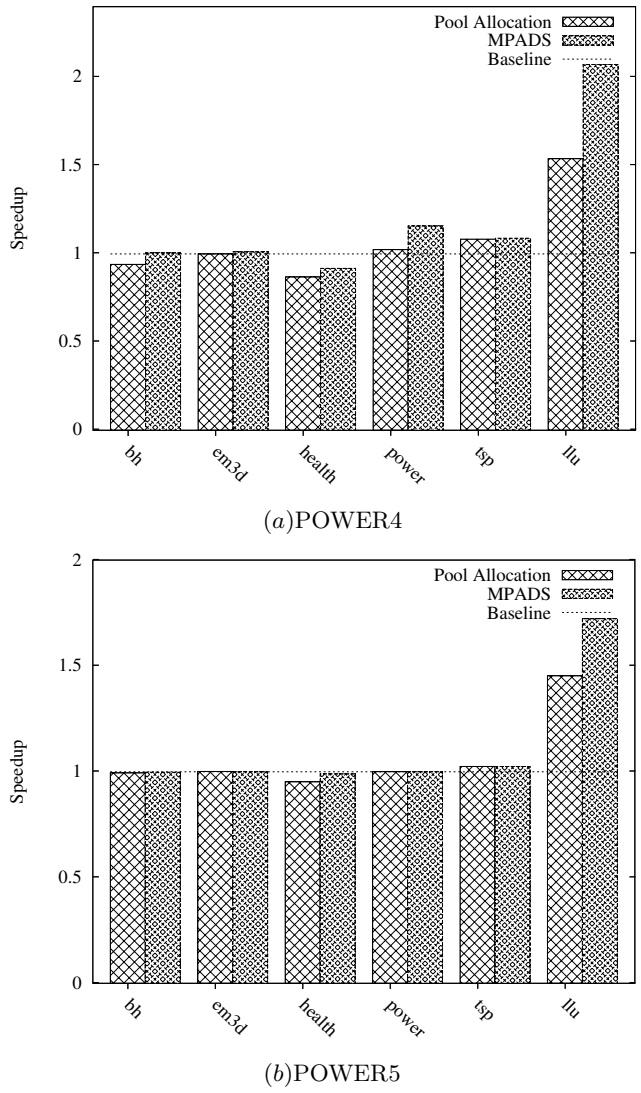

Figure 5. Speedup over baseline.

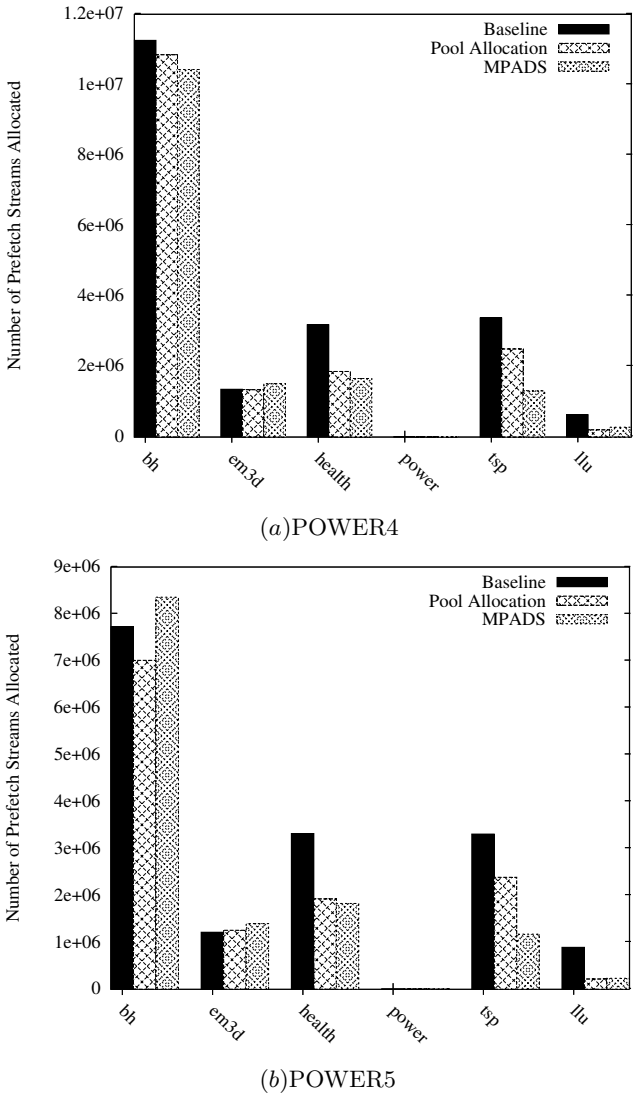

Figure 6. Number of prefetch streams allocated.

prefetch data accesses that may have been previously ignored. It also reduces the memory bandwidth requirements and cache pollution — an important feature in multi-core processors that share lower levels of the memory hierarchy.

Both the POWER4 and POWER5 processors support up to 8 simultaneous prefetch streams. Figure 6 shows that the number of prefetch streams allocated after the MAPS transformation is reduced on both the POWER4 and POWER5 processors for most of the benchmarks — exceptions are *em3d* and *bh* on the POWER5.

Figure 7 shows the number of prefetches from L2 to the L1 cache and from main memory into the L2 cache in the POWER4.[8]

4. Related Work

The area of research most closely related to this work is the area of automatic data transformations. Automatic data transformations are appealing because they are transparent to the programmer and the compiler can often optimize programs better than the average programmer.

Finding a good data layout is a difficult problem. Even if we know the order that memory locations are accessed, the problem of organizing data in memory to minimize the number of cache misses can't be solved efficiently or even approximated very well unless P = NP (14). Thus all of the proposed solutions are heuristics designed to improve the naive layout that is commonly used in production compilers.

Lattner and Adve developed one of the first fully automatic and safe data transformations to successfully transform dynamically allocated objects for general purpose programs written in type unsafe languages (9). They created an analysis called Data Structure Analysis that is based on a context-sensitive pointer analysis. Their pool allocation automatically identifies safe candidates to transform and allocates them in pools based on the objects that they were aliased with. The pool allocation idea forms the base for MPADS pool allocation and other structure splitting frameworks (6; 17).

[8] The variations are very similar in the POWER5 processor.

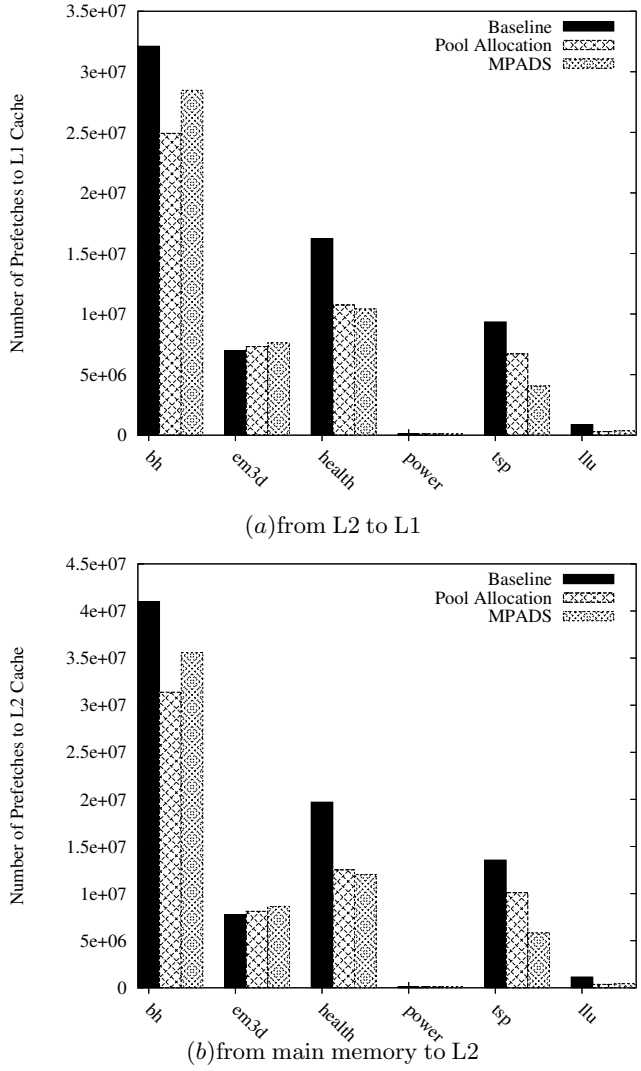

Figure 7. Number of Prefetches on POWER4.

Lattner and Adve used their Data Structure Analysis and pool allocation to safely compress pointers in linked data structures (10). Their system reduces 64-bit pointers to 32-bit pointers by allowing pointers in the same pool to index other objects in that pool using an offset from the base of the pool. The system reduces the size of objects and allows more objects to fit in cache resulting in smaller working sets and improved application performance. MPADS attempts to reduces the size of the working set by reorganizing how data is placed in memory, opposed to modifying the size of the data.

Zhong *et al.* define a model to measure the closeness of references in a memory trace, the model is known as *reference affinity* (24). Zhong *et al.* show how reference affinity can be used for structure splitting and array regrouping. Fields with high affinity are grouped together and then the structure is split into groups. Although they perform structure splitting in a compiler they assume that the language is type-safe and use programmer intervention to ensure that the transformation does not alter the semantics of the program. MPADS does not require a program trace and guarantees that the transformation is safe. As well, MPADS performs maximal splitting instead of affinity-based splitting.

A precursor of MPADS is *Forma*, a maximal splitting framework also implemented in the IBM XL compiler to automatically and safely reshape single-instantiated arrays (23). The maximal splitting implemented in *Forma* is limited to the transformation of arrays of data structures and does not combine splitting with memory pooling. Instead of using the affinity-based splitting proposed by Zhong *et al.*, Forma uses maximal splitting and discovers that maximal splitting achieves best, or near-best, performance on the SPEC 2000 and Olden benchmarks.

Rabbah and Palem develop a completely automated data remapping technique that splits pointer-based structures (15). Their system uses a trace of all the field accesses in the program to determine the field-access affinity. This affinity is then used to decide which structures to split. The candidate structures are then split maximally, which is similar to MPADS uniform structure splitting from Section 2.2.1. However, uniform splitting is the only splitting mechanism supported and the fields must be padded so that they are all the same length. If many fields in a structure are padded this can result in the data remapping polluting the cache more than the original organization. MPADS only uses uniform structure splitting if all of the fields in the structure are the same size and thus does not pollute the cache with padding. Rabbah and Palem also use a field-insensitive pointer analysis whereas MPADS uses a more precise field-sensitive pointer analysis.

Shin *et al.* restructure the field layout for dynamically allocated objects (17). Their field restructuring removes the padding in the structure, groups fields with high affinity together and performs affinity-based splitting. To determine which fields are accessed together the system uses profile information. Shin *et al.* describe the technique used to split the structures but do not describe how to integrate it into a compilation framework nor do they mention how they guarantee safety. Although their technique is similar to the Non-Uniform Splitting technique described in Section 2.2.2, MPADS is different because it is designed to be safely integrated into a production compiler and performs maximal splitting. As well, MPADS supports uniform splitting to reduce the address calculation overhead when all of the fields in a structure are the same length. The numbers obtained by Shin *et al.*'s structure reshaping are slightly better than MPADS but they were obtained on a system with a higher clock speed and smaller caches.

Jeon, Shin and Han expand on Shin *et al.*'s previous work using structure splitting to reorganize objects allocated in the heap (6). The major difference from their previous work is that this system is implemented in the CIL compiler framework and does not use profile information. The improvement to the field restructuring is the addition of a static analysis that uses regular expressions to represent the field access pattern. The regular expression can then be used to extract the access pattern and estimate the field affinity. Once again, affinity-based splitting is performed. The safety of their system is based on Lattner and Adve's observation that most pools are used in a type-consistent style (9). Jeon, Shin and Han rely on their regular expressions to select candidates if the closure only contains fields from a single node and this will likely be enough for the majority of the cases. However, without a pointer analysis it is impossible to guarantee safety because fields can be accessed through pointers that may not be captured through their regular expression framework.

An alternative to the automatic transformations described above are transformations that require programmer intervention. Truong, Bodin and Seznec investigate grouping fields that are referenced together into the same cache line and splitting data structures so that fields of different instances of a structure are grouped together (20). Chilimbi, Davidson and Larus modify the internal organization of fields in a data structure at compile time to improve the locality (1). The cahce miss rates were reduced by 10-20% and run time was

reduced by 6-18% on five of the Java benchmarks that Chilimbi, Davidson and Larus used.

5. Conclusion

The main accomplishment of this paper is a demonstration that memory pooling and pointer-analysis-based structure splitting can be effectively combined in the context of a commercial compiler infrastructure. While small program examples demonstrated very high potential for performance gains, realizing significant gains in the performance of actual programs will required a more precise, and more expensive, alias analysis than the one currently available in the compiler. The main gain of the currently implementation of MPADS is a significant reduction in the number of hardware prefetch streams that are allocated, with a consequent reduction in the memory bandwidth utilization and on the polution of the data cache.

Trademarks

Acknowledgments

This research was partially funded by the IBM Center for Advanced Studies (CAS) in Toronto and by a grant from the Collaborative Research and Development (CRD) program of the Natural Sciences and Engineering Research Council (NSERC) of Canada.

References

[1] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming Language Design and Implementation*, pages 13–24, New York, NY, USA, 1999. ACM Press.

[2] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming Language Design and Implementation*, pages 1–12, New York, NY, USA, 1999. ACM Press.

[3] S. Curial. Safe automatic data splitting for linked data structures. Master's thesis, University of Alberta, 2007.

[4] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 229–241, New York, NY, USA, 1999. ACM Press.

[5] Jack Huynh. The AMD AthlonTM MP Processor. *AMD White Paper*, May 2003. http://www.cpuplanet.com/img/pdf/athlon_mp512l2.pdf.

[6] J. Jeon, K. Shin, and H. Han. Layout transformations for heap objects using static access patterns. In *Compiler Construction*, pages 187–201, March 2007.

[7] M. Kandemir and I. Kadayif. Compiler-directed selection of dynamic memory layouts. In *CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign*, pages 219–224, New York, NY, USA, 2001. ACM Press.

[8] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[9] 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.

[10] C. Lattner and V. S. Adve. Transparent pointer compression for linked data structures. In *MSP '05: Proceedings of the 2005 workshop on Memory system performance*, pages 24–35, New York, NY, USA, 2005. ACM Press.

[11] Chris Lattner and Vikram Adve. Data structure analysis: A fast and scalable context-sensitive heap analysis. Tech. Report UIUCDCS-R-2003-2340, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Apr 2003.

[12] Patrick Meredith, Balpreet Pankaj, Swarup Sahoo, Chris Lattner, and Vikram Adve. How successful is data structure analysis in isolating and analyzing linked data structures? Tech. Report UIUCDCS-R-2005-2658, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Nov 2005.

[13] Sun Microsystems. *UltraSPARC III User's Manual*, 2004.

[14] E. Petrank and D. Rawitz. The hardness of cache conscious data placement. In *POPL*, pages 275 – 307, Portland, OR, USA, January 2002.

[15] R. M. Rabbah and K. V. Palem. Data remapping for design space optimization of embedded memory systems. *ACM Transactions on Embedded Computing Systems*, 2(2):1–32, May 2003.

[16] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Trans. Program. Lang. Syst.*, 17(2):233–263, 1995.

[17] K. Shin, J. Kim, S. Kim, and H. Han. Restructuring field layouts for embedded memory systems. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 937–942, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[18] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5), 2005.

[19] B. Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM Press.

[20] D. N. Truong, F. Bodin, and A. Seznec. Improving cache behavior of dynamically allocated data structures. In *Seventh International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 322–329, 1998.

[21] D. Vianney, A. Mericas, B. Maron, T. Chen, S. Kunkel, and B. Olszewski. CPI analysis on POWER5, part 1: Tools for measuring performance. http://www-128.ibm.com/developerworks/library/pa-cpipower1/, April 2006.

[22] Ofri Wechsler. Inside Intel Core Microarchitecture: Setting new standards for energy-efficient performance. *Technology@Intel Magazine*, pages 1–11, March 2006. http://support.intel.com/technology/magazine/computing/core-architecture-0306.pdf.

[23] P. Zhao, S. Cui, Y. Gao, R. Silvera, and J. N. Amaral. *Forma*: A framework for safe automatic array reshaping. *ACM Transactions on Programming Languages and Systems*, 30:2:1–2:30, November 2007.

[24] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 255–266, New York, NY, USA, 2004. ACM Press.

[25] Craig B. Zilles. Benchmark health considered harmful. *SIGARCH Comput. Archit. News*, 29(3):4–5, 2001.