Georgia Institute of Technology
CREST

Center For Research in Embedded Systems and Technologies
Technical Report

Bridging Processor and Memory Performar in ILP Processors via Data-Remapping

Rodric M. Rabbah and Krishna V. Palem

Georgia Institute of Technology
Atlanta, Georgia

CREST-TR-01-002
GIT-CC-01-14
June 2001

Bridging Processor and Memory Performance in ILP Processors via Data-Remapping

Rodric M. Rabbah and Krishna V. Palem
rabbah@ece.gatech.edu, palem@ece.gatech.edu
Center for Research on Embedded Systems and Technology
Georgia Institute of Technology

Abstract

Current system design trends continue to magnify the disparity between processor and memory performance. Thus, as microprocessors perform increasingly better than the memory systems supporting them, it is ever more important to bridge the performance gap to help translate the promise of Moore's law into overall performance delivered to the end applications. This gap in performance between the processor and the memory is further exacerbated in the context of modern processors with high-levels of instruction level parallelism (ILP), especially for data-intensive applications. In these processors, increased demands for data delivery lead to concomitant needs for higher memory bandwidth and cache sizes. In this paper we provide a fast compile-time data-remapping technique which helps in bridging the gap between the ILP processor and its memory system, by enhancing the spatial locality of data-access. Our strategy is the first automatic approach applicable to pointer-intensive dynamic applications for which existing optimizations are mostly inadequate. We demonstrate an average performance improvement of 27% for several data-intensive applications. This is attributed to enhanced data locality, resulting in lowered demand on the bandwidth between cache levels, as well as between the cache subsystem and main memory. We also show that with increasing levels of ILP and fixed memory bandwidth, our remapping technique enables very high levels of performance with smaller cache sizes. For example, as much as a factor of 15 reduction in multi-level caches can be tolerated without a loss in performance. Although we use cycle-accurate simulators to detail the benefits of our remapping, we also measure 24% performance improvements for the Intel Pentium II and III processors, and a 9% yield on the Sun UltraSparc-II.

1 Introduction

A well-known performance bottleneck in the current and forthcoming computer architectures is the increasing gap between processor and memory speeds[4, 27]. This disparity is further aggravated by the continuing trend in processor design to extract greater instruction level parallelism as evident in the recent emergence of EPIC architectures and the release of the first such processor - the IA64 Itanium[13]. As ILP continues to increase, faster data delivery and greater cache through-put will be required to achieve better performance (Figure 1). This phenomena, compounded by irregular memory access patterns common to dynamic applications, increases the pressure on the memory system and further magnifies the long latencies associated with memory accesses.

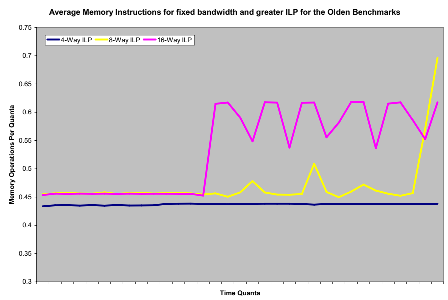

Figure 1: For fixed data bandwidth and a perfect cache model, a more parallel processor requires data at a faster rate. This motivates the need for bandwidth ameliorating strategies.

A number of strategies have been advocated to address the memory bottleneck and improve overall system performance. The majority of these techniques either attempt to hide long latencies or enhance data locality. Examples of latency masking optimizations include prefetching[24, 20, 6] and load-sensitive scheduling algorithms[15, 30]. However, such strategies are vulnerable to unpredictable memory reference patterns and may degrade performance. Specifically, prefetch strategies waste bandwidth and pollute caches when data is unnecessarily requested. Similarly, poor or pessimistic operation characterization during scheduling often leads to reduced ILP. On the other hand, locality enhancing optimizations amortize the cost of expensive memory accesses by improving data reuse. Loop-tiling, loop-skewing and numerous other control-flow transformations[2, 9, 21, 19, 11, 12] have significantly improved the performance of applications with predictable access patterns. Unfortunately, these optimizations fare poorly when applied to important pointer-intensive scientific and dynamic real-world applications[26, 22].

In this paper, we introduce a fast and light-weight *locality enhancing algorithms* (*LEA*) to improve overall system performance. In particular, we focus on better utilization of the memory hierarchy, and explore the impact of bandwidth constraints for high ILP processors. The proposed optimization

- Enhances temporal data locality,
- Reduces memory bandwidth requirements and
- Applies to programming languages with dynamic memory allocation support, such as *C and C++*.

We specifically focus on pointer-intensive applications with extensive dynamic allocations of data *records*. These applications are challenging, mainly due to their unpredictable and often dynamic memory access patterns. Using detailed simulations of EPIC architectures, we demonstrate the significant performance impact of our LEA in reducing bandwidth requirements and increasing locality. Similarly, we measure the overall performance yields for existing computing platforms with different processor speeds and cache configurations.

The contributions of this paper are two-fold. First, we present a fast automated software tool[1] for data-remapping to effectively reduce bandwidth consumption. Second, we detail simulations and analysis of the effects of our optimization on data requirements and overall performance for EPIC architectures.

We implement our data-remapping algorithm in the Trimaran[28] EPIC compiler and use its configurable simulation infrastructure[29] to demonstrate an average 18% reduction in bandwidth requirements for all levels of the memory hierarchy. We also show considerable improvements in various cache statistics, and an average effective increase of 15% in IPC.

Our performance analysis is reported for benchmarks from the SPEC2000, Data Intensive Systems[10] and Olden suites, benefiting applications such as neural network simulation, large database management, image matching, and scientific computation. We also demonstrate the same trends in performance on existing Intel X86 and Sun UltraSparc platforms.

[1] Our algorithms run in time linear in the size of the source code.

1.1 Summary of Results

We make the following contributions.

- **Data-Remapping Algorithm**. The innovative aspects of this work are geared at significantly improving performance for high ILP processors and in the context of data-intensive applications. We propose a fast and fully-automated algorithm to achieve better reference locality in dynamic and pointer intensive applications. The algorithm running time is linear in the size of the program.

- **Significant Bandwidth Reduction**. We apply our optimization to several well know data-intensive applications and achieve an average 2x reduction in overall bandwidth requirements for a two level memory hierarchy.

  - *Lower Bus Demand*. The bandwidth improvements are attributed to a 18% reduction in data requests between level one and level two of the memory system. The bus connecting the second level cache and main memory contributes another 18% reduction.

  - *Better Locality*. Our data-remapping strategy better utilizes the lowest level cache and reduces the miss ratio by an average 22%.

- **Working Set Reduction**. Using incremental bandwidth measurements, we demonstrate a factor of two reduction in the application workingset. This is attributed to effective data colocation strategies applied by our optimization.

- **Performance Gains**. We demonstrate the effectiveness of our optimization on a wide range of EPIC processors with varying levels of ILP. We obtain a 15% improvement in effective IPC and a 22% reduction in execution time.

We also demonstrate the same trends in performance for concrete architectures. We achieve as much as a 26% improvement in execution time for the Intel Pentium III processor, a 24% improvement for the Intel Pentium II and a 9% improvement for the Sun UltraSparc II.

The remainder of this paper is organized as follows.

- **Data-Remapping Strategy.** Section 2 presents an overview of data-remapping, then details our locality enhancing algorithm and heuristics.

- **Experimental Results and Analysis.** Section 3 demonstrates the performance benefits of our locality enhancing algorithms. We detail the impact of our LEA on the performance of various EPIC, X86 and UltraSparc architectures.

- **Related Work.** Section 4 discusses related work and compares this approach with known techniques.

2 Data-Remapping Strategy

Traditional data-layout strategies typically attempt to minimize the total space requirements of a data structure[25]. This is often adequate to achieve small memory footprints and benefit regular applications with well defined access patterns. However, for larger dynamic programs where interactions amongst data structures vary over time, the relative order of objects and their placement in memory becomes an issue[26]. Although it is desirable to reorder data in memory to match the access sequences, it is simply not feasible. Not only would the cost of dynamic data movements far outweigh any benefits, but finding the best data layout for a set of objects over numerous computation paths is NP-complete[17].

Our LEA is a simple, efficient and highly effective strategy that implicitly assigns data fields that are most often referenced together to the same cache block. This is achieved without any actual data relocation and has the benefits of amortizing the cost of a block fetch from memory, reducing bandwidth consumption, and avoiding detrimental cache conflicts. Our approach focuses on a coordinated placement of fields during record allocations and relies on compiler generated *remapping functions* to compute their location during subsequent usage. We assume that a record is defined as a set of diverse data types grouped within a unique declaration. We shall refer to elements of the set as *fields*. We will also refer to a record as a *structure* or *object*.

The algorithm is presented in two stages. First, a feedback-driven *gathering* phase analyzes profile information to select candidates for remapping. Next, a *reorganization* module applies fixed remapping strategies to the candidates identified in the earlier step. There are two variations of the reorganization algorithm: one for static data reorganization and the other for dynamic data reorganization.

2.1 Gathering Phase

The gathering phase analyzes profile information to characterize record types and guide in the selection of candidates for remapping. The information is used to selectively remap frequently referenced data types that exhibit poor cache behavior along program hot-spots[23].

Algorithm 1 Algorithm for computing the neighbor affinity of record in a program. *w* is the temporal locality window. We use a value of *w* equal to the size of a cache block scaled by the cache associativity. Normalization and squaring of the NAP is done to favor most often used record type. The running time for the algorithm is O(w—T—).

```
let Trace T = (R, k, f)*, is a memory access profile trace
T[i] for 0 < i <= |T| represents the ith triple occurring in T

procedure ComputeAffinity(Program P, Trace T, w)
    for j := 1 to |T| do
        for i := w - 1 downto 1 do
            (R, k, f) := T[j]
            (R', k', f') := T[j-i]
            if (R' = R) and (k' <> k) and (f' <> f) then
                NAP(R) += 1
            end if
        end for
    end for

    for each record type R in P do
        NAP(R) := square(NAP(R) / (w (|T| - w)))
    end for
end ComputeAffinity
```

Our analysis seeks to identify and build a model of data reuse for the extensively used object types in a program and to quantify the probability that multiple occurrences of a field *f* within a chosen time span w belong to different objects of the same data type R. We shall refer to this measure as the *neighbor affinity probability* or *NAP*. A low NAP implies that on the average, the fields of a particular record instance exhibit good temporal locality. In other words, for any particular time span of length w, if one field of a record instance is encountered, then the other fields of the same record instance are also likely to be encountered. In contrast, a moderate to high neighbor affinity suggests that if a particular field f is encountered, the same field but from a different record instance will be temporally accessed. High NAP is used as a criteria for applying our remapping strategy. Namely, record types that exhibit this phenomena are remapped such that for a cluster of these objects, field colocation is achieved. The method for computing the affinity of a record type is given in Algorithm 1.

The gathering algorithm computes the neighbor affinity for each record type in a program. It subsequently *marks* those with high NAP for remapping. Our reorganization strategy is to use a coordinated allocation technique to achieve field colocation for global and dynamic data structures.

2.2 Reorganization Phase

In the gathering phase of the algorithm, we isolate the most often used record types and characterize the interactions of fields among different objects of the same type. During this stage, we focus strictly on global and dynamic data structures of marked types. We ignore all stack-allocated objects, as they are often small and exhibit good temporal locality; in order to preserve these characteristics, we maintain the traditional object layout native to the language.

Although we present a strategy for remapping global data variables, the key technical features of our method are geared towards preserving program semantics in the presence of pointer variables[2]. This applies to programming languages that associate physical meaning with the declaration layout of a record. Notable examples are *C and C++*. The majority of the encountered difficulties are due to *pointer arithmetic*, which arise from non-standard but common programming practices. In essence, the programmer uses knowledge of the size and physical layout of a

[2]A pointer variable is a variable whose value is the memory location *(address)* of another variable.

data type to access its various elements. In the interest of clarity, we shall restrict the focus of the remaining discussion to the *C* programming language. We expect that extensions of this work to object oriented languages will yield comparable improvements in data locality and performance.

Global Data Reorganization

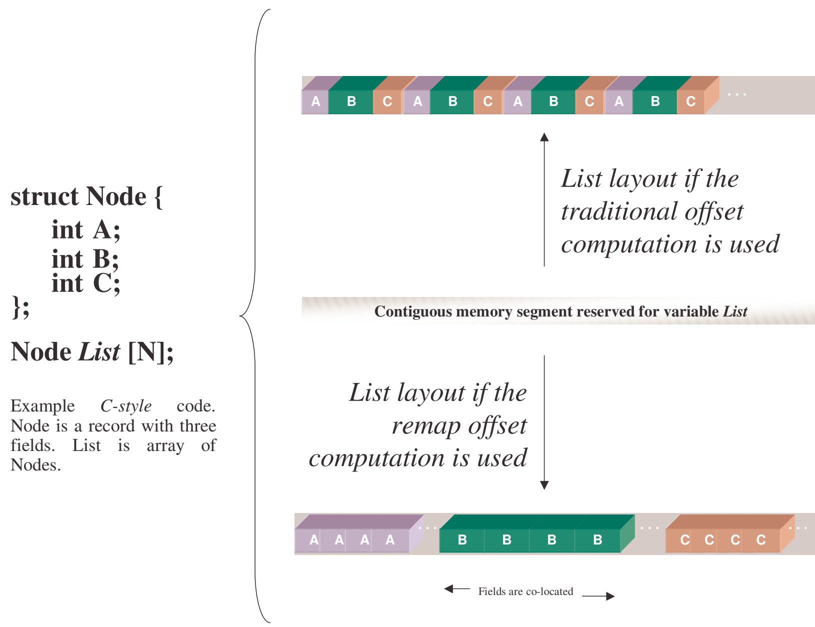

Figure 2: Layout of fields in a reserved memory segment for the remap and traditional strategies for a sample source code.

The algorithm for global data reorganization is shown in Algorithm 2. It associates one of two *offset computation functions* (*remap* or *traditional*) with each globally declared array-of-record program variables. The functions are subsequently used during code generation to calculate the offset of the specified field relative to the base address of a *Cluster*[3].

The global data remap function is GDRemap (1). It interprets the *Cluster* as a record of arrays. This interpretation, illustrated in Figure 2, yields an implicit transformation that is desirable for record types with moderate to high NAP. Specifically, the respective fields of various records in

[3] A *Cluster* is a contiguous memory segment reserved for N objects. This is equivalent to an array of *rank* N.

the *Cluster* are now located in adjacent memory addresses. In contrast, the GDNomap Equation (2) is the traditionally used offset computation function[14]. It interprets the *Cluster* as an array of records.

Algorithm 2 Algorithm for Static Data Reorganization. The algorithm running-time is linear in the size of the program.

```
procedure GDReorg (Program P)
    for each global variable V in P do
        if V is of type array of record R then
            if R was marked for reorganization then
                Associate the Remap offset computation function with V
            else
                Associate the Traditional offset computation function with
            end if
        end if
    end for
end GDReorg
```

$$GDRemap(R_k.f) = (k-1) \times FieldSize(R.f) + N \times \sum_{i=1}^{f-1} FieldSize(R.i) \tag{1}$$

$$GDNomap(R_k.f) = (k-1) \times RecordSize(R) + \sum_{i=1}^{f-1} FieldSize(R.i) \tag{2}$$

For the equations above, $R_k.f$ represents the f^{th} field of the k^{th} instance of a record R, $f \geq 1$, and $R \in Cluster$ with $N \geq 1$ records. We define $FieldSize(R.f)$ as the number of consecutive addressable units required to store $R.f$ and $RecordSize(R) = \sum_{i=1}^{|R|} FieldSize(R.i)$.

During code generation, the compiler evaluates the variable-associated offset functions when appropriate. In terms of run-time overhead, observe that the essential difference between the GDRemap and GDNomap is in the last term. The size of the *Cluster* (N) is necessary to carry out the GDRemap computation. The value is however readily available to the compiler, as the remapping strategy is strictly applied to global arrays. This implies that the third term of GDRemap can be computed statically. Thus our remapping strategy contributes the same run-time overheads as the traditional offset computation.

Dynamic Data Reorganization

The methodology for remapping dynamic data objects focuses on repeated single object allocations rather than dynamic array of record allocations. The remapping strategy used for global arrays readily applies for dynamic arrays of structures. Unfortunately, efforts to accommodate the remapping of theses arrays would entail a run-time retrieval of the array size and a subsequent multiplication operation. The associated overhead may be small compared to the overall benefits, but we have made no efforts to quantify it. We did however apply the global data reorganization scheme in special case scenarios where the compiler was able to

- Determine that all dynamic arrays of a given record type are of the same size,
- Statically disambiguate all pointer variables that alias these arrays.

This alleviates the need for redundant dynamic disambiguations. Finally, we will discuss the remapping strategy applied to repeated single record allocations. Recall that the GDRemap function assumes record fields are part of a cluster of records of the same type. Our approach here is to automatically generate a light-weight *wrapper* around the memory allocation requests in the application. The wrapper is used to control the placement of new objects in dynamically reserved clusters. Once the cluster has been completely allocated, the resultant data layout resembles that of a global array of records, achieving field colocation. The algorithm for dynamic data reorganization is given in Algorithm 3 and an implementation of an example wrapper function is shown in Algorithm 4. The automatic generation of wrappers is trivial and not discussed here. We use a *StaggerDistance* equal to the size of a cache block for all the experiments reported in Section 3.

Notice that the algorithm does not associate offset functions with pointer variables. Instead, it is left to the code generator to determine which expression to use for address computations. Our approach is to try and determine if an encountered pointer aliases a marked record. If so, the compiler evaluates the DDRemap (3) expression. Similarly, the code generator uses the DDNomap (4) computation for pointer variables that alias static records. We found the inter-procedural dataflow analysis algorithm of Aho et al.[1] to be adequate for resolving alias issues for the benchmarks

Algorithm 3 DDReorg is a source level transformation. It reorders the fields of a record such that the most frequently used field is located first in the layout. It also replaces requests for dynamic allocations of a record R with a type-specific allocator (the *wrapper*). The algorithm runs in time linear in the size of the program.

```
procedure DDReorg (Program P)
    for each record type R in P do
        if R is marked for reorganization then
            reorder the fields of R such that the most
            frequently used field has field index 1
        end if
    end for
    for each statement S in P do
        /* single out allocations of a single object */
        if S is of the form x := Allocate(R, 1) then
          1. replace S with x := Wrapp_R()
          2. generate Wrapp_R
        end if
    end for
end DDReorg
```

$$DDRemap(P \rightarrow f) = \sum_{i=1}^{f-1} StaggerDistance \times MaxFieldSize(*P) \tag{3}$$

$$DDNomap(P \rightarrow f) = \sum_{i=1}^{f-1} FieldSize(*P) \tag{4}$$

In the equations above, P is a pointer to a record of type R and $(*P) = R$. We also define $MaxFieldSize(R) = max\{\forall f \leq |R|, FieldSize(R.f)\}$. Note that since $DDRemap(P \rightarrow f)$ and $DDNomap(P \rightarrow f)$ evaluate to 0 for the first field of a record ($f = 1$), the run-time alias disambiguation is not necessary. We exploit this characteristic for the most frequently accessed field of a record. We achieve this by reordering the field layout, such that the most frequently used field is located first.

used in our experiments. However, for pointers where the compiler is unable to disambiguate alias information[25], we evaluate both offset expressions and rely on a run time comparison of the pointer value against the *stack pointer register* to determine which value is to be used. This is a simple and effective solution that exploits novel predication features and advanced comparators in EPIC architectures.

Algorithm 4 An example wrapper function. When clusters are fully consumed, new ones are allocated.

```
function Wrapp_R () returns address
    /* Cluster C is a persistent variable */
    if Cluster C is full then
        /* Allocate a new Cluster */
        C := Allocate(R, StaggerDistance)
    end if
    address := C
    /* Update cluster usage */
    C := C + DDRemap(R.1)
end Wrapp_R
```

2.3 Implementation Notes

Our remapping algorithms were implemented in the Trimaran C Compiler. We also implement the global data and dynamic data reorganization algorithms in the GNU C Compiler (GCC). Profile information gathered by Trimaran was shared with our GCC implementation, as GCC does not provide tools for feedback-driven optimizations. Trimaran is comprised of the IMPACT[28] compiler as the front-end and the ELCOR[28] compiler as the system back-end. We incorporate our remapping algorithms in the compiler front-end, since type information and some source level transformations are required.

3 Experimental Results

We evaluate the impact of our LEA in three ways. First, we analyze the bus traffic across all level of the memory hierarchy using cycle-accurate simulation tools in Trimaran. We anticipate that a reduction in traffic implies better data and cache usage and hence better locality. We support this claim by illustrating the average decrease in workingset size over the lifetime of an applications in the presence of data-remapping. Lastly, we look at the impact of better memory system performance on execution cycles.

3.1 Benchmarks and Methodology

We performed detailed simulations of four Olden, two DIS and one SPEC2000 benchmark. The application characteristics are shown in Table 2. We simulate the benchmarks using both input data sets listed in Table 2. Profile information used to determine which data types to remap was obtained using much smaller data sets. We compile all applications with hyperblock and superblock optimizations enabled. All benchmarks, with the exception of TreeAdd[4], were simulated for the EPIC configurations shown in Table 1. Because of the space constraints, we report average results over all simulations, unless otherwise noted. The interpretation of our results follow.

Issue Width and ILP	**L1 Size, Block Size, Way**	**L2 Size, Block Size, Way**
1, 4	16K, 16, 1	256K, 128, 2
1, 4	16K, 128, 1	256K, 128, 2
1, 4	16K, 128, 1	1024K, 128, 2
1, 4	32K, 128, 4	1024K, 128, 2
4, 4	32K, 128, 4	1024K, 128, 2
8, 8	32K, 128, 4	1024K, 128, 2
8, 8	64K, 128, 4	1024K, 128, 2

Table 1: EPIC Configurations used in simulations.

[4]Trimaran could not successfully simulate this benchmark. We do report the performance of TreeAdd for concrete architectures.

Name	Suite	Description	Main Data Structures	Input Data sets	Memory
ART	Spec 2000	Simulates neural networks	Dynamic array of records	ref1 and ref2	small
DM	DIS	Data archive management	Dynamic records and arrays of records	set14 and set24	24Mb
Field	DIS	Random token search and string replacement	Dynamic array of records	11654 and 54860 token replacements	small
Health	Olden	Simulation of the Colombia Health Care System	Doubly-linked list	levels 3-6 and time units 1000-10000	123Mb
Perimeter	Olden	Computer image region perimeter	Quad-Tree	11Kx11K and 12Kx12K images	147Mb
TSP	Olden	Traveling salesman shortest path	Dynamic records	3M and 8M cities	320Mb
TreeAdd	Olden	Tree node processing	Binary tree	Tree depth of 20 and 25 levels	512Mb

Table 2: Benchmark Characteristics.

3.2 Epic Platforms

We first consider the impact of data-remapping in the context of bandwidth usage. Our algorithms are geared to improving data locality and increasing the reuse of fetched data items. We thus expect to observe significant bandwidth reductions manifested in reduced bus usage across the memory hierarchy. Figure 3 compares the average bus utilization for all benchmarks in the presence of remapping. The graph is normalized relative to the baseline simulation results. As we expected, the bus usage between first and second levels of cache were reduced by 18%. The reduction in bus demand was also felt between the second level cache and main memory, where an average reduction of 18% was also observed in the presence of remapping.

We attribute the reduction in bandwidth requirements to better cache utilization and improved data locality. Figure 4 compares the level one and level two cache performance in terms of hit and miss ratios. Here we see an increase in the level one load hits and a corresponding reduction in misses. Although the average number of L2 misses increases in the presence of remapping, there is an equivalent reduction in L2 cache requests (see Figure 7). This suggests that not only is the level

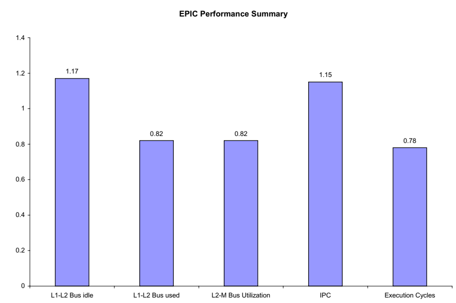

Figure 3: The normalized graph shows the average bus utilization at all level of the memory hierarchy.

one cache better utilized, but that the level two cache requests are for the most part compulsive.

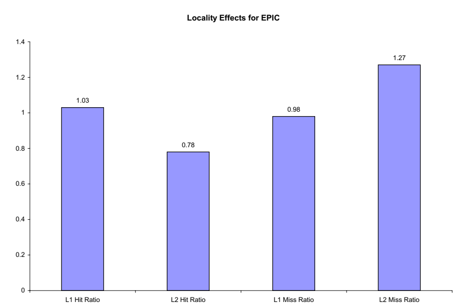

Figure 4: The normalized graph compares the average baseline L1 and L2 hit and miss ratios to those in the presence of remapping.

To get a better grasp of the data locality phenomena, we investigate the variations in workingset[26] size for the level one cache. Due to space constraints, we can only illustrate two such examples.

In Figure 5 we divide the lifetime of execution into fixed segments. We then measure the incremental size of the workingset required to perform the computation. Thus, an increase in bus demand implies a proportional increase in the workingset. The average reduction in the workingset size is slightly greater than a factor of two in the presence of remapping. We show a similar trend for the ART SPEC benchmarks in Figure 6. A factor of two reduction was also observed in the presence of remapping.

A summary of all simulation results is shown in Figure 7.

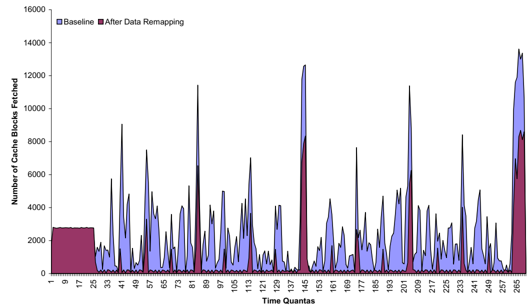

Figure 5: Incremental bandwidth requirements for TSP

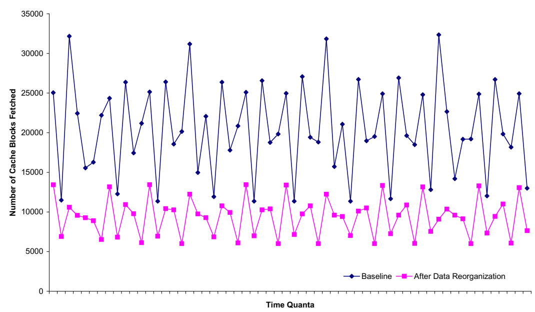

Figure 6: Incremental bandwidth requirements for ART. A representatitve time window is shown for clarity.

3.3 Existing Platforms

Figure 8 summarizes performance gains for three concrete architectures. The benchmarks were compiled using GCC and the highest level of optimizations (-O3).

Figure 9 illustrates the importance of remapping as processor speeds increase. Observe that for the faster Pentium III processor, despite a factor of 8 reduction in the second-level cache size, we attain 20% improvement compared to the UltraSparc.

4 Related Work

Previous work has attacked the processor-memory gap by computation reordering to increase spatial and temporal locality[5, 8, 7, 31, 18, 22]. Most recently, Crummey et al.[22] explore a coor-

	ART	DM	Field	Perimeter	TSP	Health	*Average*
L1 Cache Requests	0.37	1.00	1.00	0.79	1.02	0.99	0.86
L1 Store Requests	0.42	0.98	1.00	0.78	1.09	0.80	0.84
L1 Load Hit Ratio	1.09	1.00	1.00	1.02	0.99	1.06	1.03
L1 Load Miss Ratio	0.63	0.97	1.00	0.78	0.60	0.71	0.78
Cycles L1-L2 Bus Idle	1.68	1.00	1.00	1.02	1.06	1.24	1.17
Cycles L1-L2 Bus used	0.79	1.00	1.00	0.66	0.77	0.71	0.82
L2 Cache Requests	0.29	0.97	1.00	0.93	0.80	0.49	0.75
L2 Store Requests	0.59	0.96	1.01	0.95	0.88	0.89	0.88
L2 Load Hit Ratio	1.27	1.02	1.11	0.65	1.13	0.69	0.98
L2 Load Miss Ratio	3.42	1.01	1.10	0.65	0.74	0.70	1.27
L2 Cache Blocked	1.00	0.98	1.10	0.61	0.60	0.83	0.85
Cycles L2-M Bus used	0.86	0.99	1.00	0.72	0.75	0.63	0.82
Virtual Latency Stalls	0.11	0.86	1.08	0.36	0.33	0.65	0.56
L1 Cache Busy Stalls	0.35	0.99	1.01	2.92	0.77	0.70	1.12
IPC	1.26	1.02	1.00	1.05	1.26	1.31	1.15
Execution Cycles	0.27	0.98	1.00	0.86	0.82	0.77	0.78

Figure 7: Summary of results for EPIC architectures

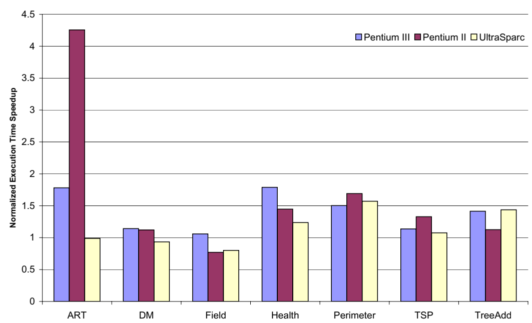

Figure 8: Performance improvements in the presence of remapping for Intel X86 and Sun UltraSparc platforms.

dinated data and computation reordering strategy based on space filling curves. Their strategies are applied to array-based languages (i.e. Fortran) and attack a different class of applications. Our focus is on pointer-intensive applications with considerable dynamic memory allocations.

Kistler et al.[18] proposed an automated field-ordering algorithm to minimize load latency in case of a cache miss. Their technique exploits a hardware feature available on the PowerPC to improve data delivery across buses in the memory hierarchy. Similarly, Chilimbi et al.[7] proposed a reordering technique that assigns temporally related fields of a record into the same cache-line. However, both algorithms offer only partial solutions, as they do not consider the interaction of fields amongst various instances of a record. Chilimbi et al.[8] described a data placement scheme to specifically address this issue. However, the proposed strategies are not completely transparent

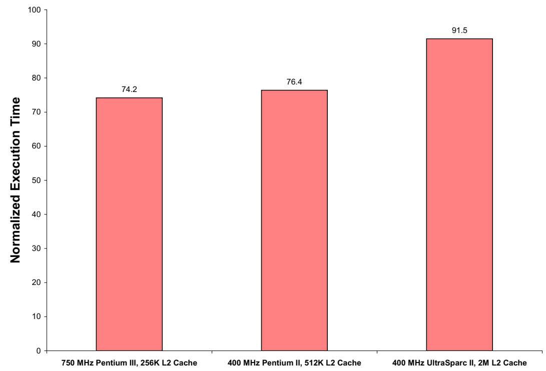

Figure 9: Summary of real hardware performance improvements in the presence of remapping compared to full GCC optimizations.

to the programmer, require some manual re-tooling of the application and can incur a significant run-time overhead as objects are dynamically relocated in memory.

Truong et al.[31] have also tried to improve the cache behavior of pointer-intensive applications. They proposed a field reorganization technique and introduced a new memory allocator to support *instance interleaving*; this is the interleaving of identical fields of different dynamic objects into a cache-block. Unfortunately, the approach chosen by Truong et al. is cumbersome in that the record layout is left completely to the programmer. This includes the insertion of large pads between fields to accommodate object interleaving. Since the record reorganization is a source level transformation, it leads to a substantial waste of storage space for statically allocated records. The user must also re-tool and annotate the application to invoke a specialized memory allocator that consumes the inserted pads. Their strategy is strictly a dynamic technique and does not apply to statically allocated data. In contrast, our data-remapping algorithm is fully-automated, uses existing allocation tools and applies to a wider class of objects.

References

[1] A. Aho, R. Sethi, and J. Ullman. "Compilers Principles, Techniques and Tools". Addison-Wesley.

[2] J. Anderson, S. Amarasinghe, and M. Lam. "Data and computation transformation for multi-processors". In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.

[3] T. Ball, and J. Larus. "Efficient Path Profiling". In *IEEE Micro*, Decemeber 1996.

[4] D. Burger, J. Goodman, and A. Kagi. "Memory bandwidth limitations of future microprocessors". In *Proceedings of the 23rd Annual International Symposium on Computer Architectures*, pages 78-89, May 1996.

[5] 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, October 1998.

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

[7] T. Chilimbi, B. Davidson, and J. Larus. "Cache-conscious structure definition". In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13-24, May 1999.

[8] 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.

[9] J. Carter, W. Hsieh, M. Swanson, L. Zhang, A. Davis, M. Parker, L. Schaelicke, L. Stoller, and T. Tateyama. "Memory system support for irregular applications". In *Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers*, May 1998.

[10] Data-Intensive Systems Benchmark Suite. www.aaec.com/projectweb/dis/

[11] C. Ding, and K. Kennedy. "Improving cache performance of dynamic applications with computation and data layout transformations". In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 229-241, May 1999.

[12] C. Ding, and K. Kennedy. "The memory bandwidth bottleneck and its amelioration by a compiler". In *Proceedings of the International Parallel and Distribute Processing Symposium*, May 2000.

[13] Intel Itanium Processor. www.intel.com/itanium

[14] B. Kernighan, and D. Ritchie. "The C Programming Language". Prentice Hall.

[15] D. Kerns, and S. Eggers. “Balanced scheduling: instruction scheduling when memory latency is uncertain”. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1993.

[16] D. Gannon, W. Jalby, and K. Gallivan. “Strategies for cache and local memory management by global program transformations”. In *Proceedings of the First International Conference on Supercomputing*, June 1987.

[17] D. Kirkpatrick, and P. Hell. “On the completeness of a generalized matching problem”. In *The Tenth Annual ACM Symposium on Theory of Computing*, 1978.

[18] T. Kistler, and M. Franz. “Automated data-member layout of heap objects to improve memory-hierarchy performance”. In *ACM Transactions on Programming Languages and Systems*, Volume 22, No. 3, pages 490-505, May 2000.

[19] M. Lam, E. Rothberg, and M. Wolf. “The cache performance of blocked algorithms”. In *Proceedings of the Fourth International Conference in Architectural Support for Programming Languages and Operations Systems*, pages 63-74, April 1991.

[20] C. Luk, and T. Mowry. “Compiler-based prefetching for recursive data structures”. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222-233, October 1996.

[21] K. McKinley, S. Carr, and C. Tseng. “Improving data locality with loop transformations”. In *ACM Transactions on Programming Languages and Systems*, Volume 18, No. 4, pages 424-453, July 1996.

[22] J. Mellor-Crummy, D. Whalley, and K. Kennedy. “Improving memory hierarchy performance for irregular applications using data and computation reordering”. In *Proceedings of the ACM International Conference on Supercomputing'*, pages 425-433, June 1999.

[23] M. Merten, A. Trick, C. George, J. Gyllenhaal, and W. Hwu. “A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization”. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 136-147, June 1999.

[24] 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, October 1992.

[25] S. Muchnick. “Advanced Compiler Design Implementation”. Morgan Kaufman.

[26] E. Nystrom, R. Ju, and W. Hwu. “Characterization of Repeating Data Access Patterns in Integer Benchmarks”. To appear in *Proceedings of the 28th International Symposium on Computer Architecture*, July 2001.

[27] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keaton, C. Kazyrakis, R. Thomas, and K. Yellick. “A case of intelligent RAM”. In *IEEE Micro*, pages 34-44, April 1997.

[28] Trimaran: An Infrastructure or Research in Instruction Level Parallelism. www.trimaran.org

[29] R. Rabbah, and M. Saint. “SMACHS: Smart Memory and Cache Hierarchy Simulator”. Manuscript in preparation.

[30] F. Sanchez and A. Gonzalez. “Cache sensitive modulo scheduling”. In *IEEE Micro*, 1997.

[31] 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, October 1998.