# Data Remapping for Design Space Optimization of Embedded Memory Systems

RODRIC M. RABBAH and KRISHNA V. PALEM
Georgia Institute of Technology, Atlanta

In this article, we present a novel linear time algorithm for *data remapping*, that is, (i) lightweight; (*ii*) fully automated; and (*iii*) applicable in the context of pointer-centric programming languages with dynamic memory allocation support. All previous work in this area lacks one or more of these features. We proceed to demonstrate a *novel application of this algorithm as a key step in optimizing the design of an embedded memory system.* Specifically, we show that by virtue of locality enhancements via data remapping, we may reduce the memory subsystem needs of an application by 50%, and hence concomitantly reduce the associated costs in terms of size, power, and dollar-investment (61%). Such a reduction overcomes key hurdles in designing high-performance embedded computing solutions. Namely, memory subsystems are very desirable from a performance standpoint, but their costs have often limited their use in embedded systems. Thus, our innovative approach offers the intriguing possibility of compilers playing a significant role in exploring and optimizing the design space of a memory subsystem for an embedded design. To this end and in order to properly leverage the improvements afforded by a compiler optimization, *we identify a range of measures for quantifying the cost-impact of popular notions of locality, prefetching, regularity of memory access and others.* The proposed methodology will become increasingly important, especially as the needs for application specific embedded architectures become prevalent. In addition, we demonstrate the wide applicability of data remapping using several existing microprocessors, such as the Pentium and UltraSparc. Namely, we show that remapping can achieve a performance improvement of 20% on the average. Similarly, for a parametric research HPL-PD microprocessor, which characterizes the new Itanium machines, we achieve a performance improvement of 28% on average. All of our results are achieved using applications from the DIS, Olden and SPEC2000 suites of integer and floating point benchmarks.

Categories and Subject Descriptors: B.3 [**Hardware**]: Memory Structures; D.2 [**Software**]: Software Engineering; D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms: Algorithms, Desgin, Performance

Additional Key Words and Phrases: Design space exploration, embedded systems, memory hierarchy, memory subsystem, caches, data remapping, compiler optimization

This work was supported in part by DARPA contract F33615-99-1499, Hewlett-Packard Laboratories and Yamacraw.

Authors' address: R. M. Rabbah and K. V. Palem, Center for Research on Embedded Systems and Technology, School of Electrical and Computer Engineering, Georgia Institute of Technology, 777 Atlantic Drive, Atlanta, GA 30332; email: {rabbah, palem}@ece.gatech.edu

## 1. INTRODUCTION

The memory hierarchy has been a ubiquitous component in the design of computing platforms since the introduction of the von Neumann machine [Burks et al. 1987; Taub 1963]. It has widely served to bridge the performance gap between processor and supporting memory subsystem, usually by employing deep cache hierarchies where each level trades off capacity for access speed. As processors are increasingly used in the context of embedded systems, the cost of this memory hierarchy is increasingly becoming a limiting factor in its ability to play as central a role. Quite often, this limitation is because of the physical size, as well as the implications to the complexity of the processors used in the embedded domain. Thus, for example, the StrongARM processor which is widely used in portable multimedia devices is equipped with an 8-Kb primary data cache. By contrast, the emerging high-performance EPIC (explicitly parallel instruction computing) Itanium processor—targeted at enterprise and technical applications—is supported by a three-tiered memory hierarchy comprised of a 32-Kb primary data cache, a 96-Kb secondary cache, and a 2-4-Mb tertiary cache.

Although the Itanium is an example of a general-purpose, very long instruction word (VLIW) processor,[1] the VLIW methodology has widely proliferated the embedded computing market tiers. For example, Texas Instruments has shipped large volumes of its TI-C6 VLIW processor; StarCore has announced a VLIW DSP core; and Phillips is developing its TriMedia high-performance VLIW products. The evolution of VLIW architectures into viable embedded computing solutions is largely attributed to the simplicity of the design and the significant instruction level parallelism (ILP) which it affords [Rau and Schlansker 2000]. Unfortunately, the same reasons that make VLIW architectures appealing also magnify the needs for an adequate memory subsystem capable of sustaining high rates of data delivery to the processing units. Hence, while VLIW processors offer opportunities for performance enhancements via ILP, it is evident that when the available memory bandwidth is saturated, an additional increase in the processing speed will not yield any benefits [Cragon 1996]. Thus, it is imperative to carefully manage the movement of data across the memory hierarchy in order to deliver the promise of Moore's law to the end user. This is especially true in VLIW architectures where significant losses in processing throughput are at stake when data is found at lower levels of the hierarchy.

In addition to the above limitation posed by the physical size of the memory subsystem—important to data-intensive programs—the problem is further exacerbated by pointer-intensive applications common to the embedded domain. In these programs, the *memory access patterns* (MAPs) are quite often highly irregular, and hence the typical prediction and replacement schemes tend to mispredict more often than in the case of array-based applications with regular MAPs. The inability of these schemes to cope with unpredictability further increases the pressure on the memory hierarchy. To understand this better, consider a kernel application that repeatedly processes the elements of an array in sequence. In such a case, the address stream generated by the program is highly ordered. Most modern architectures are

[1]EPIC includes VLIW as a subset.

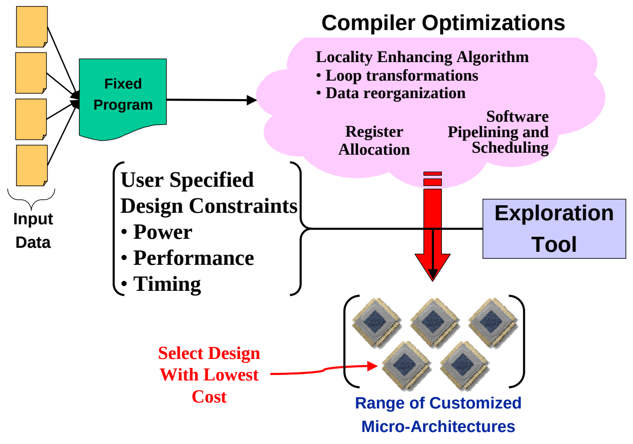

Compiler Optimizations
Locality Enhancing Algorithm
• Loop transformations
• Data reorganization
Register Allocation
Software Pipelining and Scheduling
Fixed Program
Input Data
User Specified Design Constraints
• Power
• Performance
• Timing
Exploration Tool
Select Design With Lowest Cost
Range of Customized Micro-Architectures

Fig. 1. Design space exploration and compiler optimizations may be viewed as duals of each others. Here, design space exploration is shown as the primal.

capable of accurately predicting and prefetching such regular MAPs ahead of their actual demand [Chilimbi and Hirzel 2002; Wu 2002; Fu and Patel 1992]. By contrast, the same prediction strategies are generally vulnerable and do not perform well in the context of irregular MAPs intrinsic to pointer-centric dynamic applications. Specifically, the strategies that work so well with regular applications waste bandwidth and pollute the hierarchy when data is prefetched unnecessarily.

In this article, we present a novel *data remapping* scheme that helps overcome the challenges stated above. Namely, remapping enables a more effective use of the memory hierarchy in data-starved embedded systems, particularly in the context of pointer-heavy applications. As an example, *we are able to show that our technique allows a program to achieve the same overall running time with just half the memory resources, when compared to a program that has not been remapped.*

## 1.1 Data Remapping and Design Space Exploration

The above result offers an intriguing possibility, which is a highlight of this article: *remapping, traditionally a compiler optimization, can play a significant role in optimizing the memory subsystem design, and hence performance of an embedded application.* As shown in Figure 1, design space exploration involves exploring alternate hardware or architectural solutions to meet a specified performance constraint for a *fixed program* $\mathcal{P}$. Thus, while the program is fixed, the optimization techniques are applied to find a hardware solution, subject to the given performance or execution time constraint. By contrast, and as shown in Figure 2, the dual of this problem is the domain of a traditional compiler optimization, wherein the applications or programs $\mathcal{P}_1, \mathcal{P}_2 \ldots, \mathcal{P}_k$ are optimized to achieve the best possible performance on a *fixed target processor* whose memory subsystem consists of several sophisticated *cache memories.*

From a design space exploration perspective, our technique may be regarded as a tool which can help lower memory needs significantly without compromising execution time. In particular, given a “user-specified” performance goal, remapping can

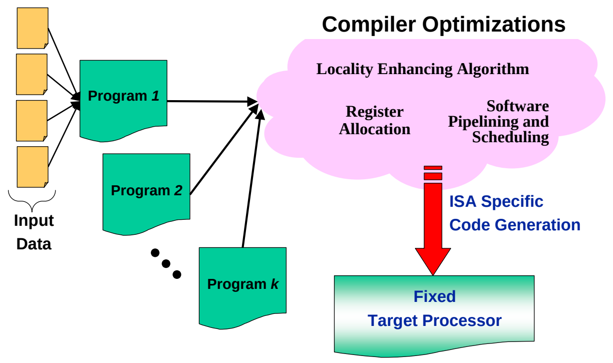


Fig. 2. Compiler optimizations for a fixed target, shown as the dual of design space exploration.

Table I. Example Impact of Data Remapping on Cost During Design Space Exploration

| Benchmark | Suite | Goal $10^6$ Cycles | Before Remapping L2 Size | Before Remapping L2 Cost | After Remapping L2 Size | After Remapping L2 Cost | Saving |
|---|---|---|---|---|---|---|---|
| 179.art | SPEC | 13,000 | 1.0-Mb | \$19.38 | 0.0-Mb | \$0.00 | \$19.38 |
| TreeAdd | Olden | 880 | 1.0-Mb | \$19.38 | 0.5-Mb | \$17.80 | \$ 1.58 |
| Perimeter | Olden | 520 | 2.0-Mb | \$48.00 | 1.0-Mb | \$19.38 | \$28.62 |

yield a system meeting this goal whose cost is significantly lower than that of a corresponding system meeting the same goal, but without the benefits of remapping. As shown in Table I, remapping preserves the application's performance with half the cache size for the three example pointer-centric benchmarks. Namely, given an execution time goal of 600 million cycles for the application Perimeter, remapping allows us to use a secondary (L2) memory size of 1-Mb instead of 2-Mb for a total saving of \$28.62 which is 60% of the cost.[2] In another example (179.art), data remapping allows the use of a memory hierarchy that only consists of a primary cache (L1). More generally, we have demonstrated that the improvement is consistent for several applications, including floating-point and integer applications from the DARPA DIS[3] suit as well as the well-known Olden and SPEC2000 suites.

In order to use the savings afforded by data remapping as a crucial step in exploring the design space of an embedded cache system, it is important to quantify the potentials for improvement using some fundamental variables over which the exploration and optimization can proceed. *In this article, we provide a novel characterization of a set of such variables.* For example, while prefetching is widely accepted in memory systems, its efficacy is very dependent on its ability to match its prediction and replacement policies with the actual needs of the program [Nystrom et al. 2001]. In this context, we quantify the degree of mismatch between

[2]For the 2-Mb L2 cache, we use two 1-Mb Toshiba TC55W800FT-55 SRAM chips, each at a cost of \$24. For the 1-Mb L2 cache, we use two 0.5-Mb Toshiba TC55V400AFT7 SRAM chips, each at a cost of \$9.19. For the 0.5-Mb L2 cache, we use four 128-Kb Cypress CY62128VL-70SC SRAM chips, each at a cost of \$4.425.
[3]The Data Intensive Systems (DIS) were developed at the Atlantic Aerospace Electronics Corporation, in conjunction with the Boeing Company and ERIM International.

Table II. Summary of Average Results for Concrete Architectures After Remapping

| Processor | CPU Speed | L2 Size | % Speedup |
|---|---|---|---|
| Pentium III | 750 MHz | 256-Kb | 26 |
| Pentium II | 400 MHz | 512-Kb | 24 |
| UltraSparc II | 400 MHz | 2048-Kb | 9 |
| Itanium HPL-PD | – | 96-Kb | 28 |

the prefetchers in the architecture and the memory access patterns of the program. Furthermore, using our measures, we can quantify the improvements achieved by our optimization along this dimension. More generally however, the measures can serve as a rigorous foundation geared to facilitate the exploration and optimization of the memory hierarchy design space with respect to power, performance, size, and cost. While the processor industry paradigm is undergoing a fundamental change, similar to the evolutionary patterns driven by customization that occurred in other industries [Bass and Christensen 2002], the design of embedded memory systems remains an ad hoc art. It often relies on intuition and extensive simulations to choose the best match for a particular processing unit [Rau and Schlansker 2000; Abraham and Mahlke 1999]. By contrast, the automatic exploration and synthesis of application-specific processing cores and systolic accelerators is the focus of various systematic studies [Schreiber et al. ; Aditya et al. 1999; Lee and Kedem 1990]. Thus, it is our contention that memory organization is a major specialization dimension and warrants carefully consideration when application-specific computing solutions are sought. A formal definition of the proposed measures and their application to deduce the improvements offered by data remapping is the topic of Section 5.

Our methodology extends the current state-of-the-art design-space exploration and optimization techniques to pointer-centric applications ubiquitous to *C*-based applications. By contrast, important related work has been reported that is relevant in the context of array-based applications. For example, Catthoor et al. [1998] perform an extensive exploration of memory organization for embedded systems, with an emphasis on storage and bandwidth optimization. In addition, Panda et al. [2001] recently published a thorough survey of data and memory optimization techniques for embedded systems. Notably, researchers [Kulkarni et al. 2000; Panda et al. 1997] have explored a coordinated data and computation reordering for array-based data structures in multimedia applications.

### 1.2 Data Remapping as a Compiler Optimization for COTS Architectures

In addition to the focus on design space exploration of embedded memory systems, we illustrate the utility of our technique in the context of a traditional compiler optimizations framework (Figure 2). Specifically, data remapping achieves speedups in the context of existing commercial-off-the-shelf (COTS) processors such as Pentium II, Pentium III and Sun UltraSparc II, as well as experimental EPIC processors modeled as variations of the HPL-PD architecture [Kathail et al. 2000]. The results are highlighted in Table II and subsequently detailed in Section 4. A noteworthy result is the average speedup due to remapping for the fastest processor (750 MHz Pentium III) which exceeds that of the slower 400 MHz UltraSparc by a factor of

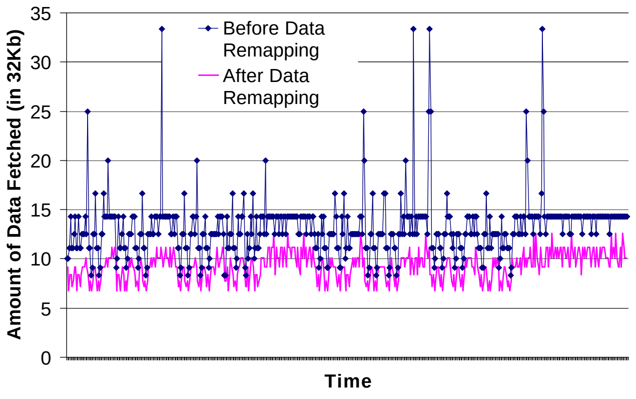


Fig. 3. Amount of data fetched before and after data remapping.

three. Such a result reiterates the importance of bridging the gap between processor and memory speeds, particularly since the size of the secondary cache used in the Pentium III is eight times smaller than the one used in the UltraSparc. In addition to the concrete computing platforms, we model an Itanium processor using the HPL-PD architecture. As noted earlier, the Itanium memory hierarchy is three-tiered, with a 32-Kb primary cache, 96-Kb secondary cache, and 2-Mb tertiary cache. The performance improvements for an Itanium-like HPL-PD processor are 28% on average.

## 1.3 Overview of Our Methodology

At the heart of our approach is a *compile-time data-remapping* algorithm to enhance locality for dynamic programs with irregular memory access patterns. Stated in simple terms, remapping is a reorganization of the application's data in memory, such that memory elements that are accessed contemporaneously are in fact placed together in memory. *Thus, remapping aims to improve the spatial locality of memory elements that in fact also share temporal locality.*

The simple remapping transformation achieves its impressive improvement by ensuring that the ratio of the number of items found in cache (cache hits) to those that are fetched from main memory (cache misses) remain the same, with half the memory size—and hence the execution time of the application is not compromised. Specifically, in the absence of remapping, much of the data delivered to the cache is often needlessly fetched because of a lack of address locality. In Figure 3 we plot the amount of data that is delivered to the cache for successive (nonoverlapping) time slices throughout the execution of a representative benchmark (TSP from the Olden benchmark suite) The graph demonstrates an average 30% reduction in the amount of data delivered to the cache as a result of our locality enhancement. Such an improvement is subsequently leveraged to vary the cache parameters (e.g., reduce the cache size) and achieve the same overall performance, but with less cache investments.

Traditionally, a limited form of remapping known as data reorganization has been applied in the context of improving the execution time of applications for a fixed target processor. However, all previous techniques for the data reorganization of pointer-centric applications [Kistler and Franz 2000; Chilimbi et al. 1999; Calder et al. 1998; Truong et al. 1998; Panda et al. 1997] are characterized by one or more of the following limitations. First, they are not completely transparent to the programmer and require some manual retooling of the application. Second, they incur significant runtime overhead as objects may be dynamically relocated in memory—the cost of dynamic data relocation is often prohibitive from a performance point of view, and certainly from an energy perspective which is of great importance in the embedded systems domain [Palem et al. 2002]. Third, they may violate program correctness in pointer-heavy applications. By contrast, our approach is (i) completely automated; (ii) does not perform any runtime data movements; (iii) preserves correctness for a much larger scope of applications in the context of pointer-based programming languages; and (iv) is lightweight with a running time linear in the size of the program. The details of our technique and its engineering into a compiler are outlined in Sections 2 and 6, respectively.

## 2. DATA REMAPPING ALGORITHM

The targets of our optimization are record data types common to real-world, pointer-heavy applications. A record is a set of diverse data types grouped within a unique declaration; we refer to elements of the set as *fields* and instances of a record as *objects*. The specific focus on data records is self-evident. Consider, for example, a function that searches through a linked list of records and replaces a certain data item matching a search key. Each record consists of a `key` field, a `datum` field, and a `next` field pointing to the next record in the list. Here, the `key` and `next` fields are accessed in succession, and more frequently (hot fields) than the `datum` field (cold field). Therefore, it would prove beneficial to fetch and cache as many hot fields as possible with each memory access. To this end, a remapping strategy that collocates the `key` and `next` fields of various objects in the same memory block and allocates all the of `datum` fields to a separate block will improve the program spatial locality, which inturn favorably impacts memory system behavior. Note that packing field-pairs in the same block (i.e., a block containing `key` and `next` fields) does not offer an advantage over individual field packing (i.e., a block containing only `key` fields) since the same number of blocks will eventually be fetched from memory.

Our schema is an innovative combination of field reordering and customized placement such that the new data layouts exhibit better spatial locality. *The remapping optimization consists of three phases.*

(1) *Gathering phase.* An analysis of the application memory access patterns is performed to identify record types that will benefit from remapping.
(2) *Remapping of global data objects.* We first present the remapping strategy in the context of global data objects, since they are often encountered in large applications. Next, we generalize the technique to dynamically allocated objects. We do not consider stack-allocated objects for remapping, as they are often small and exhibit good locality.
(3) *Remapping of dynamic data objects.* The key technical features of this work are geared toward pointer-centric applications and aim to preserve program seman-

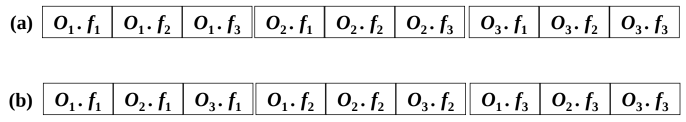

(a) $O_1.f_1$ $O_1.f_2$ $O_1.f_3$ $O_2.f_1$ $O_2.f_2$ $O_2.f_3$ $O_3.f_1$ $O_3.f_2$ $O_3.f_3$
(b) $O_1.f_1$ $O_2.f_1$ $O_3.f_1$ $O_1.f_2$ $O_2.f_2$ $O_3.f_2$ $O_1.f_3$ $O_2.f_3$ $O_3.f_3$

Fig. 4. Two example memory access patterns ($|T| = 9$) for three objects $O_1$, $O_2$, and $O_3$ of record type $R$ with fields $f_1$, $f_2$, and $f_3$. $O_j.f_k$ represents the *kth* field of the *jth* instance of a record $R$.

tics in the presence of pointer variables; a pointer variable is a variable whose value is the memory location (address) of another variable. Our optimization applies to programming languages such as *C* which associate physical meaning with the syntactic declaration of a record.

## 2.1 Gathering Phase

An arbitrary application of the remapping strategy to all data objects in a program does not necessarily increase spatial locality. Some data structures may not exhibit the requisite reference behavior to justify remapping. Although it is desirable to reorder data in memory to match all reference sequences, it is not computationally tractable [Petrank and Rawitz 2002]. To this extent, we analyze memory access patterns along program hot spots [Ball and Larus 1996] and select candidates for data remapping accordingly. The analysis is geared to characterize how well a traditional data layout is suited for various program memory access patterns or MAPs.

Consider the example memory access patterns shown in Figure 4, and let us assume that a cache may accommodate three fields at a time and that a block of the same size is used to deliver data from memory. In case (a), the reference pattern is such that the best data layout would assign the fields of object $O_1$ to one block, those of $O_2$ to another, and similarly for $O_3$. This leads to a total of three cache misses, occurring on the access to $O_1.f_1$, $O_2.f_1$ and $O_3.f_1$. In case (b), the reference pattern warrants either an alternate layout or a larger cache. Otherwise, a total of nine misses will occur, one for each reference. That is, the access to $O_1.f_1$ will lead to the delivery of $O_1.f_2$ and $O_1.f_3$, which fills our cache. The next access, however, is to $O_2.f_1$, which will lead to a cache miss and the eviction of the currently cached data (and so one for the other references). In order to avoid redundant memory accesses, a larger cache is necessary, and in this case, one that is three times the current capacity. However, as noted earlier, larger caches incur greater investments. Hence, it is more desirable to modify the data layout, such that data items in the block to be fetched are replaced with those that are more likely to be used.

Although what is described is a pathological example, it illustrates the need for a proper characterization of the mismatch between traditional data layouts and the application memory access patterns. Our proposed analysis characterizes the mismatch as the *neighbor affinity probability* or NAP. The measured value may range from zero to one, where the latter indicates that the data layout is well suited for the analyzed reference pattern (i.e., high probability of a block containing successive data accesses). The other extreme indicates that the data layout does not exhibit any correlation to the memory access pattern (i.e., low probability of a block containing successive data accesses) and strongly warrants an alternate

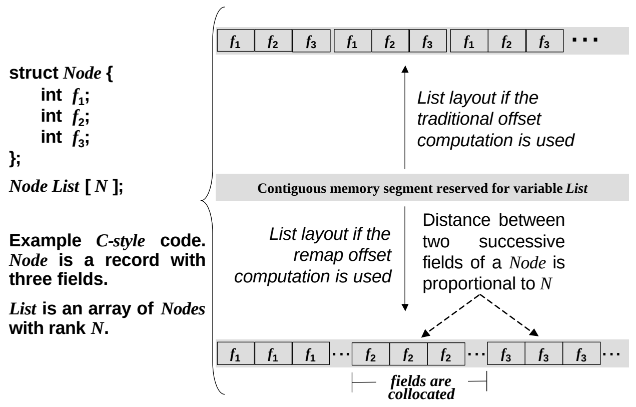
**Input:** Program $P$, Cache Block Size $B$ and Trace $T_R = (k, f)^*$ is a memory trace of all accesses to objects of record type $R$. The trace consists of a list of tuples $(k, f)$, such that $T[i]$ for $0 < i$ represents the $i^{th}$ tuple occurring in $T$, and it is an access to the $f^{th}$ field of the $k^{th}$ instance of record $R$.
**Output:** NAP for record type $R$ occurring in Program $P$.

```
01. for $j := B$ to $|T|$ do
02.     for $i := B - 1$ downto 1 do
03.         $(k_c, f_c) \leftarrow T[j]$
04.         $(k_p, f_p) \leftarrow T[j - i]$
05.         if $(k_c \neq k_p)$ then
06.             if $address(k_c, f_c)$ and $address(k_p, f_p)$ are
                less than $B$ addressable units apart then increment $NAP(R)$
07.         end if
08.     end for
09. end for
10. $NAP(R) \leftarrow \frac{NAP(R)}{(B-1)(|T|-B+1)}$
```

Fig. 5. Algorithm to compute the NAP for records in a program.

arrangement. The algorithm for computing the neighbor affinity probability is shown in Figure 5. For a fixed cache block size $B$, it analyzes an object reference trace ($T$) and computes the NAP for record types encountered in the program with an $O(|T|)$ running time. The block size enables a window-based analysis that searches for any occurrence of an access pattern resembling the one shown in Figure 4(a). Specifically, field references to different objects of the same record type are counted when the fields are within close proximity of each other (line 6). Thus in Figure 4(a)—for a block size of three addressable units—the NAP equals 14/14, which implies that successive memory references are located at nearby memory locations. Hence, the traditional data layout is effective and may not benefit from a reorganization. If, on the other hand, the MAP resembles that of Figure 4(b), where the NAP equals 4/14, it may prove worthwhile to collocate the fields of different objects to increase the likelihood of a cache hit. Once the NAP is computed, it is normalized (line 10) and may be combined with affinity information gathered using a different memory profile. Following the analysis, a simple processing step *marks* all record data types with an affinity lower than some threshold for remapping. All other record types are left unmarked and are subsequently organized using the *traditional* memory layout strategies specified by the programming language.

Our analysis does not distinguish the relative order of fields in an access pattern. This is in contrast to previous work where the temporal behavior of data fields is tracked [Chilimbi et al. 1999]. Such extensions may enhance the proposed analysis. In addition, since the NAP analysis is profile-driven, it is sensitive to the input workload selected for training the optimization. Furthermore, we note that an application may exhibit competing memory access patterns along different program hot spots. Thus, in this context, the NAP analysis above is geared to discover the most dominating pattern, and our framework performs the data remapping accord-



```
struct Node {
    int f1;
    int f2;
    int f3;
};
Node List [ N ];
```

Example C-style code. Node is a record with three fields.
List is an array of Nodes with rank N.
f1 f2 f3 f1 f2 f3 f1 f2 f3 ...
List layout if the traditional offset computation is used
Contiguous memory segment reserved for variable List
List layout if the remap offset computation is used
Distance between two successive fields of a Node is proportional to N
f1 f1 f1 ... f2 f2 f2 ... f3 f3 f3 ...
fields are collocated

Fig. 6. Layout of fields in a reserved memory segment using the *traditional* and *remapped* layout strategies for a sample source code.

ingly; efforts to accommodate a range of MAPs along different program regions may prove worthwhile, but warrants careful attention, especially when concerns for curtailing the application's power consumption are rated as "first-class citizens." In Section 5.4.2, we elaborate on the optimality of our analysis as a guide to data remapping.

## 2.2 Remapping of Global Data Objects

Once remapping candidate are identified, the global program variables are filtered to isolate arrays of records. Each such object is traditionally allocated in a contiguous memory segment (*cluster*) with a statically known starting location (*base*) and size (*rank*). The location of a field within a cluster is computed using an *offset computation function* (OCF) which determines the offset to the target relative to the base. For example, consider a record with fields $f_1$, $f_2$, $f_3$, and a cluster of such records with a rank of one. The offset to field $f_1$ relative to the base is zero—the base location of the cluster is the same as the location of field $f_1$ for the first record of the array. The offset to field $f_2$ is equal to the size of field $f_1$. Similarly, the offset to field $f_3$ is equal to the size of fields $f_1$ and $f_2$. This can be generalized to clusters of any rank, as shown in Eq. 1.

In order to improve spatial locality within a cluster, our data remapping strategy manipulates the offset computation function to yield a desired object and field layout. To this end, we introduce the *remap offset computation function* shown in Eq. 2 and illustrate the data layout that results from the traditional and remap offset expressions in Figure 6. The remapping transformation is desirable for record types with low NAP, as the respective fields of various objects in the cluster are now adjacent—that is, the remapped data layout correlates well with the reference patterns shown in Figure 4(b). In effect, if the NAP for a record type is low, then it follows that successive data references will likely not access fields of the same object.

The algorithm for remapping global data arrays is outlined in Figure 7. First, we attribute arrays of records in a program with either the traditional or remap

```
Input: NAP-annotated Program $P$.
Output: Data remapped $P$.

01. for each global variable $V$ in $P$ do
02.     if $V$ is an array of records $R$ then
03.         if $R$ is marked for remapping then
04.             Attribute $V$ with $GDRemap$
05.         else
06.             Attribute $V$ with $GDNomap$
07.         end if
08.     end if
09. end for
10. perform code generation
```

Fig. 7. Algorithm for remapping global data objects.

offset expressions. Subsequently, during code generation, the associated expression is evaluated to compute the memory location of a referenced data item. Since the remapping is completely automated and performed by the compiler, expensive data relocation at runtime is not necessary. The two offset computation functions used for the purposes of this article are

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

$$GDRemap(R_k.f) \;=\; (k-1) \times FieldSize(R.f) + N \times \sum_{i=1}^{f-1} FieldSize(R.i) \qquad (2)$$

where $R_k.f$ represents the $fth$ field of the $kth$ instance of a record $R$. We define $FieldSize(R.f)$ as the number of consecutive addressable units required to store field $f$, and $RecordSize(R)$ as the sum of $FieldSize(R.f)$ for all fields $f$ in a record $R$. The essential difference between the two OCF is the last term. The latter staggers any two fields of a single object by a distance proportional to the size ($N$) of the cluster—we refer to $N$ as the *stagger constant* and the term as a whole is called the *stagger distance*. However, since the remapping strategy is strictly applied to global data objects, the rank is readily available to the compiler, and hence the stagger distance is statically computed. Therefore, the traditional and remapping strategies contribute the same runtime overhead.

## 2.3 Remapping of Dynamic Data Objects

The need for cache-conscious data placement is ever more important as applications increasingly rely on dynamically allocated objects [Chilimbi et al. 1999; Calder et al. 1998]. It is common for traditional allocation strategies to ignore the underlying memory hierarchy in favor of low runtime overhead. Unfortunately, such a scenario often results in poor interactions between data layout and program access patterns. Our methodology is to leverage the NAP analysis to identify suitable data types that would benefit from a controlled placement of newly allocated objects. The

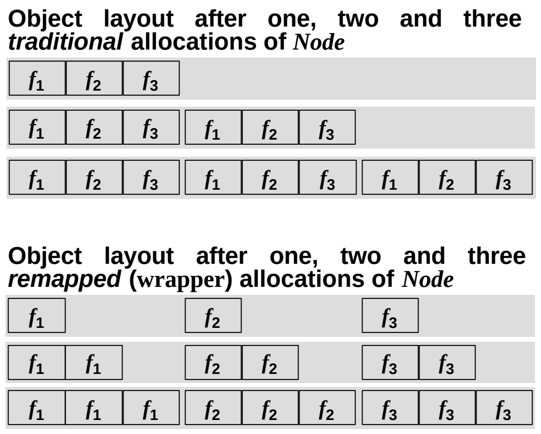

Object layout after one, two and three *traditional* allocations of *Node*
$f_1$ $f_2$ $f_3$
Object layout after one, two and three *remapped* (wrapper) allocations of *Node*

Fig. 8. Layout of fields in a reserved memory segment for the traditional and remapped dynamic layout strategies.

goal is to produce a field collocation layout as illustrated in Figure 8 and introduced previously for arrays of records. To this end, we use automatically generated light-weight *wrappers* around traditional memory allocation requests in the program—much like customized memory management mechanisms used in many applications where a large memory pool is allocated, and smaller portions within the pool are reassigned with successive allocation requests. However, unlike traditional custom memory management modules which tend to be complex [Chilimbi et al. 1999; Calder et al. 1998; Truong et al. 1998], the generated wrappers are simple and efficient. Furthermore, the innovative combination of a custom memory allocator and new OCF allows for fine-grain control of field placement as opposed to object placement alone.

The algorithm for remapping dynamic data objects is shown in Figure 9. The first steps of the algorithm intercepts memory allocation requests and substitute custom allocation requests, viz, a wrapper (lines 1–8). An example wrapper function is illustrated in Figure 10. The automatic generation of such wrappers is trivial and not discussed here. Note that the algorithm targets repeated single object allocations, rather than dynamic allocations of arrays of records. The remapping strategy used for global arrays may be applied to dynamic ones. In this case, however, the size of a dynamically allocated array may not be available to the compiler. Furthermore, an application may allocate several such arrays, each of a different size. Therefore, the optimization will incur some runtime overhead, as the stagger distance is necessarily computed online. A special scenario arises when the compiler is able to determine that all dynamic arrays of a given record type are of the same size, or alternatively, that a suitable maximum size can be used. In such a case, the stagger distance is statically fixed and the wrapper adjusted accordingly.

Once all wrapper allocations are in place, the code generator calculates the field offset for a given pointer access. If it can be determined that a pointer aliases a dynamically allocated record, the compiler evaluates the remapping offset expression shown in Eq. 4. Similarly, the code generator uses the traditional offset calculation

```
Input: NAP-annotated Program $P$.
Output: Data remapped $P$.

01. for each statement $S$ in $P$ do
02.     if $S$ is of the form $x \leftarrow Allocate(RecordSize(R))$ then
03.         if $R$ is marked for remapping then
04.             replace $S$ with $x \leftarrow Wrapper(R)$
05.             generate $Wrapper(R)$ if necessary
06.         end if
07.     end if
08. end for
09. for each recordtype $R$ in $P$ do
10.     if $R$ is marked for remapping then
11.         reorder the fields of $R$ such that the most frequently
            used field has field index equal to 1
12.     end if
13. end for
14. perform code generation
```

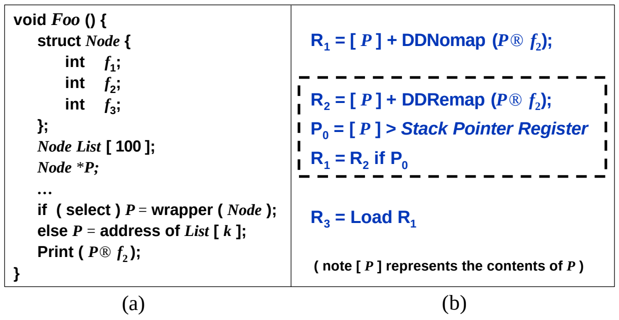
Fig. 9. Algorithm for remapping dynamic data objects.

```
Input: Record Type $R$ and Stagger Constant $N$.
Output: Valid heap (base) address where $R$ is allocated.

/* Cluster, Base and Limit are persistent variables */
Initialize $Cluster$, $Base$ and $Limit$ to 0
if $Base = Limit$ then
    $Cluster \leftarrow$ reserve heap segment of size $N \times RecordSize(R)$
    $Base \leftarrow$ base address of $Cluster$
    $Limit \leftarrow Base + N \times MaxFieldSize(R)$
end if
$Address \leftarrow Base$
$Base \leftarrow Base + MaxFieldSize(R)$
return $Address$
```

Fig. 10. An example wrapper function.

(Eq. 3) for pointer variables that alias static records. When the compiler is unable to disambiguate a data alias, we evaluate both expressions and rely on a runtime comparison of the pointer value against the *stack pointer register* to determine the proper offset (Figure 11). This is possible because the remapping is restricted to heap objects and does not alter the layout of stack objects. We used a variation of Steensgaard points-to analysis [Steensgaard 1996] to statically disambiguate an application's pointer references. The simple runtime disambiguation that supplements the compiler analysis was found to be highly effective, contributing less than a 5% increase to the total dynamic instruction count of an application; a more detailed discussion of pointer-related issues can be found in Section 6.1. The offset



```
void Foo () {
    struct Node {
        int   f1;
        int   f2;
        int   f3;
    };
    Node List [ 100 ];
    Node *P;
    ...
    if  ( select ) P = wrapper ( Node );
    else P = address of List [ k ];
    Print ( P® f2 );
}
```

(a)

```
R1 = [ P ] + DDNomap (P® f2);

R2 = [ P ] + DDRemap (P® f2);
P0 = [ P ] > Stack Pointer Register
R1 = R2 if P0

R3 = Load R1

( note [ P ] represents the contents of P )
```

(b)

Fig. 11. In (a) the value of *select* may not be statically known. Thus, $P$ may alias a remapped record or a static record. The code generated for the expression $P \rightarrow f_2$ is shown in (b) and the dynamic disambiguation code is highlighted.

computation expressions used for dynamically allocated objects are

$$DDNomap(P \rightarrow f) \;=\; \sum_{i=1}^{f-1} FieldSize(*P.i) \tag{3}$$

$$DDRemap(P \rightarrow f) \;=\; \sum_{i=1}^{f-1} StaggerConstant \times MaxFieldSize(*P) \tag{4}$$

where $P$ is a pointer to a record of type $R = *P$ and $MaxFieldSize$ is the maximum $FieldSize$ of all fields $f$ in a record $R$. The stagger constant is a compiler-defined value that is equivalent to the rank of an array used earlier to remap global data objects. The maximum field size is used in the remapping OCF to ensure that fields from different record instances do not overlap. However, when a record consists of fields with widely varying sizes, it is necessary to refine the remapping approach, as eluded to in Section 6.4. Note that a runtime disambiguation is not necessary for the first field of a record (i.e., when $f = 1$), since $DDNomap$ and $DDRemap$ evaluate to zero. Hence, the remapping algorithm modifies the record layout such that the most frequently used field has an index of one (lines 9–13 in Figure 9).

## 3. EXPERIMENTAL FRAMEWORK

Benchmarks from the DIS, Olden and SPEC2000 suites were selected for detailed analysis. The Olden benchmarks provide a common frame of reference with previous work on data reorganization [Kistler and Franz 2000; Chilimbi et al. 1999; Truong et al. 1998]. The others provide insight into larger programs. The benchmarks were executed using large reference input sets, whereas profile information was gathered using much smaller training workloads.[4]

A short description of each benchmark is as follows. 164.gzip is an integer SPEC benchmark. It utilizes a dynamically allocated array of records during decompres-

[4]Some simulation-based results are reported for less time-intensive workloads. However, the results reported for commercial processors are all based on reference input sets.

Table III. Benchmarks, Workloads and Main Memory Footprints

| Name | Workload | Memory Footprint |
|---|---|---|
| 164.gzip | test | 15Mb |
| 179.art | test | small |
| Field | 11654 Tokens | small |
| Health | 8 Levels, 100 Units 100 | 41Mb |
| Perimeter | 11Kx11K | 146Mb |
| TSP | 1M Cities | 40Mb |
| TreeAdd | 22 Levels, 20 Iteration | 64Mb |

sion. 179.art is a floating point benchmark from the SPEC suite. It dynamically allocates an array of records at startup, which is heavily used throughout execution. DM and Field are benchmarks from the DIS suite. The former is a database management system, with many different dynamically allocated objects that are repeatedly updated, deleted, and reallocated. The latter uses an array of records that is repeatedly searched and modified at random. The remaining benchmarks are provided by the Olden suite. They are memory intensive and allocate substantial amounts of heap objects. The primary data structure used in Health is a linked list to which elements are added and removed. Perimeter and TreeAdd, respectively, allocate quad and binary trees at program start-up and do not subsequently modify them. TSP creates a quad-tree at program startup that is repeatedly updated. Additional details for each benchmark, such as the input workload and memory footprint, are tabulated where appropriate (Tables III and VIII).

## 4. EXPERIMENTAL RESULTS

We present experimental results in two parts. First, we use data remapping as a design exploration tool. To this end, our experiments were conducted using a robust research infrastructure. Second, we highlight the use of data remapping as a traditional compiler optimization, wherein the applications are optimized to achieve the best possible performance on various commercial processors.

### 4.1 Design-Space Exploration

The remapping algorithms were implemented within Trimaran, a publicly available infrastructure for compiler research. It provides a common and uniform platform for verification and validation of results. Trimaran includes an optimizing compiler, a parametric processor simulator, and a smart memory and cache hierarchy simulator. The algorithms were implemented in the compiler front-end, where type information is available. The benchmarks (Table III) were compiled using classic and high-level optimizations which include loop unrolling, copy propagation, common subexpression elimination, dead code elimination, and aggressive register allocation. In addition, both superblock [Hwu et al. 1993] and hyperblock [Mahlke et al. 1992] optimizations—designed to increase instruction-level parallelism—were applied and it was determined that the former resulted in the best baseline performance (i.e., prior to remapping). For the results that follow, all benchmarks were completely simulated. Various memory hierarchy configurations, with respect to primary and secondary cache organization, were used and are reported throughout where appropriate. The results reported in this section assume a bandwidth of

Table IV. Summary of Execution Statistics Before Remapping
(L1=32 Kb, L1 block size=16 bytes, L2=1 Mb, L2 block size=32 bytes)

| Benchmark | Execution Cycles | L1 Read Hits | L1 Read Misses | Total L2 Requests |
|---|---|---|---|---|
| 164.gzip | 1,106,079,932 | 4,077,168 | 328,996 | 554,515 |
| 179.art | 704,713,706 | 22,119,661 | 38,059,226 | 57,595,798 |
| Field | 1,047,393,960 | 653,993,052 | 4,517,423 | 4,954,030 |
| Health | 2,616,712,073 | 104,512,992 | 74,405,616 | 115,343,232 |
| Perimeter | 813,958,394 | 129,045,017 | 10,798,610 | 39,561,696 |
| TreeAdd | 877,485,849 | 133,145,917 | 20,994,686 | 24,221,730 |
| TSP | 1,077,624,556 | 274,867,547 | 34,111,685 | 51,383,672 |

Table V. Percent Improvements After Remapping Compared to Table IV
(L1=32 Kb, L1 block size=16 bytes, L2=1 Mb, L2 block size=32 bytes)

| Benchmark | Execution Cycles | L1 Read Hits | L1 Read Misses | Total L2 Requests |
|---|---|---|---|---|
| 164.gzip | 2.00 | 0.28 | -0.01 | -0.06 |
| 179.art | 69.23 | 38.11 | 75.89 | 71.55 |
| Field | -0.02 | 0.00 | 0.00 | 0.00 |
| Health | 21.90 | -16.47 | 23.13 | 31.24 |
| Perimeter | 22.82 | -0.65 | 7.82 | 26.87 |
| TreeAdd | 10.50 | 8.15 | 24.88 | 26.61 |
| TSP | 14.07 | -37.08 | 43.72 | 42.23 |
| *Average* | 20.07 | – | – | 28.36 |

8 bytes per cycle throughout the hierarchy, a memory access latency of 30 cycles, and 4-way associative caches. In addition, the memory is configured to provide streaming support and uses read/write-allocate semantics.

In Table IV, we summarize the processor and memory hierarchy performance for a baseline architecture, before remapping. Specifically, we tabulate the total execution cycles of the application and the number of primary cache (L1) hits and misses. In addition, we provide the number of requests that reached the secondary cache (L2). In Table V, we illustrate the impact of data remapping in the traditional sense of a compiler optimization relative to a fixed target processor. The average performance improvement for all the benchmarks after data remapping is 20%. A noteworthy statistic is the reduction in traffic between the first and second levels of the memory hierarchy – 28.36% on average, an indication of increased data locality; in Section 5.4.2, we investigate the notion of locality in greater detail. While data remapping affords significant reductions in execution time, the objective here is to explore the possibility of trading-off some of the benefits in order to reduce the memory needs of the application. Thus, in Tables VI and VII, we illustrate some examples of the memory design space that we explored. Specifically, the former tabulates the results for the same set of experiments where the only variable is the size of the secondary cache—we reduced the L2 cache size from 1024-Kb to 512-Kb. Thus, while the secondary cache size is halved, the performance gains are reduced roughly 6% to 18.97%. In Table VII we show the result of applying data remapping to the same set of benchmarks, however, in this case, both cache sizes

Table VI. Percent improvements after remapping compared to Table IV.
(L1=32 Kb, L1 block size=16 bytes, L2=512 Kb, L2 block size=32 bytes)

| Benchmark | Execution Cycles | L1 Read Hits | L1 Read Misses | Total L2 Requests |
|---|---|---|---|---|
| 164.gzip | 2.00 | 0.28 | -0.01 | -0.06 |
| 179.art | 64.38 | 38.11 | 75.89 | 72.04 |
| Field | -0.02 | -0.01 | 0.00 | 0.00 |
| Health | 21.88 | -16.47 | 23.13 | 31.24 |
| Perimeter | 22.81 | -0.65 | 7.82 | 26.87 |
| TreeAdd | 10.49 | 8.15 | 24.88 | 26.61 |
| TSP | 11.25 | -37.08 | 43.72 | 42.23 |
| *Average* | 18.97 | – | – | 28.43 |

Table VII. Percent improvements after remapping compared to Table IV.
(L1=16 Kb, L1 block size=16 bytes, L2=512 Kb, L2 block size=32 bytes)

| Benchmark | Execution Cycles | L1 Read Hits | L1 Read Misses | Total L2 Requests |
|---|---|---|---|---|
| 164.gzip | -0.66 | 1.38 | -13.68 | -11.64 |
| 179.art | 64.36 | 38.11 | 75.89 | 72.04 |
| Field | 0.00 | 0.00 | 0.00 | 0.00 |
| Health | 21.77 | -16.18 | 22.73 | 30.28 |
| Perimeter | 22.79 | -0.64 | 7.70 | 26.75 |
| TreeAdd | 10.46 | 8.19 | 24.67 | 26.26 |
| TSP | 9.51 | -36.29 | 37.40 | 30.43 |
| *Average* | 18.32 | – | – | 24.87 |

are halved. The results demonstrate a 9% trade-off in savings due to remapping in exchange for significantly smaller caches. Concomitantly, when the memory hierarchy components are reduced in size, the corresponding power and energy requirements are also reduced [Palem et al. 2002].

In Section 5, we investigate the impact of the block size and bandwidth on the memory hierarchy and processor performance. Briefly however, when we increased the width of a cache block from 16 bytes to 32 bytes—for an 8-issue EPIC processor with a 32-Kb primary cache—we measured an average reduction of 38% in cache misses before remapping. Similarly, when we further increased the width of the block to 64 bytes, we observed a 73% reduction in cache misses. Intuitively, a larger block is more likely to contain data that is actually used compared to a smaller block. However when we applied data remapping and simulated the benchmarks using the narrower block (i.e., 16 bytes per block), we measured a 36% reduction in cache misses. This is comparable to the result previously achieved when the block size was 32 bytes (and in the absence of remapping). Thus, the added system complexity is not easily justifiable. Similarly, we applied our optimization, simulated the benchmarks using a 32-bytes per block cache configuration, and measured a miss ratio reduction of 85% on average, considerably better than simply doubling the size of a cache block to 64 bytes. In this context, data remapping again affords a memory subsystem with lesser complexity. Furthermore, we found that similar trade=offs are possible with respect to processor ILP. Namely, a 4-issue processor

Table VIII. Benchmarks, Workloads and Main Memory Footprints

| Name | Workload | Memory Footprint |
|---|---|---|
| 179.art | ref1 and ref2 | small |
| DM | set14 and set24 | 24Mb |
| Field | 11654 and 54860 Tokens | small |
| Health | Levels 3-6, Units 1000-10000 | 123Mb |
| Perimeter | 11Kx11K and 12Kx12K | 147Mb |
| TreeAdd | 20 and 25 Levels | 512Mb |
| TSP | 3M and 8M Cities | 320Mb |

in the presence of data remapping may outperform an 8-issue processor without the benefits of remapping. In the sequel, we demonstrate why such trade-offs are possible.

## 4.2 Optimization for a Fixed Architecture

To demonstrate the data remapping impact in a real-world setting, the benchmarks were compiled using a remapping-augmented industry-strength compiler and executed on commercial processors. This serves to demonstrate the applicability of data remapping in a broader context and presents a high-level overview of performance. It does not however allow a fine-grain analysis of various system components.

The remapping algorithms were implemented within the GNU Compiler Collection (GCC, version 2.95.2). Profile information was gathered using Trimaran and shared with GCC, since the latter lacks the necessary tools for feedback-driven optimizations. The algorithms were implemented in the compiler front-end, where type information is available and source-level transformations are possible. The benchmarks in this context, as well as the input workloads and memory footprints, are listed in Table VIII.

The benchmarks are compiled using the standard and the highest available levels of optimization ($-O$ and $-O3$). Similarly, the benchmarks were compiled with our remapping-augmented compiler. Standard GCC optimizations are geared to reduce code size and execution time. Aggressive optimizations add function inlining and other techniques that do not involve a time-speed trade-offs. Two Pentium and a Sun UltraSparc II processors were used to measure user execution times. The processor configurations were summarized earlier in Table II.

Table IX summarizes execution time speedups for the Pentium III system for aggressive (O3) and data remapping (R) optimizations compared to the baseline (O). For example, in the second column we report the percent execution-time speedup when aggressive optimizations (O3) are enabled, relative to the baseline performance (O optimizations in this case)—hence, a positive percentage indicates an improvement and a negative percentage is indicative of performance degradation. Similarly, in the third column, we present the percent speedup when baseline optimizations are used with data remapping (O+R), compared to baseline optimizations alone. In some cases (DM, Health, Perimeter, TreeAdd) data remapping alone outperformed aggressive compiler optimizations. Results for the Pentium II and UtlraSparc II are reported in Tables X and XI, respectively. Speedups are generally similar, with some exceptions. Results for the Pentium II system in-

Table IX. Results for Pentium III Processor

| Benchmark | % Execution Speedup | | | |
|---|---|---|---|---|
| | O3/O | O+R/O | O3+R/O | O3+R/O3 |
| 179.art | 54.97 | 36.84 | 74.72 | 43.87 |
| DM | -17.95 | 15.12 | -3.26 | 12.45 |
| Field | 78.01 | 37.67 | 79.25 | 5.63 |
| Health | 8.96 | 36.35 | 49.13 | 44.12 |
| Perimeter | 19.10 | 28.65 | 46.15 | 33.44 |
| TreeAdd | 12.82 | 28.02 | 38.33 | 29.26 |
| TSP | 29.28 | 10.20 | 37.80 | 12.04 |
| *Average* | 26.46 | 27.55 | 46.02 | 25.83 |

Table X. Results for Pentium II Processor

| Benchmark | % Execution Speedup | | | |
|---|---|---|---|---|
| | O3/O | O+R/O | O3+R/O | O3+R/O3 |
| 179.art | -9.33 | 30.94 | 74.32 | 76.51 |
| DM | -25.39 | 14.44 | -11.84 | 10.81 |
| Field | 58.05 | 32.83 | 45.52 | -29.86 |
| Health | 12.03 | 24.07 | 39.22 | 30.91 |
| Perimeter | 21.96 | 31.91 | 53.87 | 40.89 |
| TreeAdd | 22.36 | 10.09 | 31.05 | 11.19 |
| TSP | 2.78 | 22.69 | 26.81 | 24.71 |
| *Average* | 11.78 | 23.85 | 36.99 | 23.59 |

Table XI. Results for UltraSparc II Processor

| Benchmark | % Execution Speedup | | | |
|---|---|---|---|---|
| | O3/O | O+R/O | O3+R/O | O3+R/O3 |
| 179.art | 62.03 | 4.05 | 61.57 | -1.19 |
| DM | 24.13 | -4.67 | 18.75 | -7.09 |
| Field | 86.23 | 22.08 | 82.79 | -24.98 |
| Health | 16.42 | 12.80 | 32.38 | 19.10 |
| Perimeter | 24.14 | 25.00 | 51.72 | 36.36 |
| TreeAdd | 28.89 | 22.91 | 50.51 | 30.39 |
| TSP | 45.04 | 1.43 | 48.89 | 7.00 |
| *Average* | 40.98 | 11.94 | 49.52 | 8.51 |

dicate performance degradation occurs when aggressive optimizations are applied to 179.art and DM. In contrast, applying data remapping alone reduces execution time by 31% and 14% respectively. Surprisingly, when 179.art was compiled with aggressive optimizations in conjunction with data remapping, execution time was reduced by a factor of four. In contrast, the Field benchmark benefits from data remapping alone. However, when other optimizations are applied, execution time is lengthened. We made no attempt to determine which compiler optimizations benefited from or were inhibited by data remapping; our ongoing research will consider the interactions among different optimizations. The speedups for the UltraSparc II were generally lower than either Pentium architectures, despite the largest secondary cache size (see Section 6.3 for a possible explanation).

## 5. MEASURES FOR QUANTIFYING MEMORY AND PROCESSOR PERFORMANCE

The proposed data-remapping schema is an example of a highly effective locality-enhancing algorithm. In general, locality, temporal or spatial, is a well-recognized phenomenon, though it lacks concrete characterization. To this end, we introduce four novel measures that completely encapsulate locality and its performance implications. These measures characterize the intrinsic behavior of a program and quantify the impact of various microarchitecture constraints on the realized or observed program behavior. As a result, the measures provide a deeper analysis for the reasons why data remapping is effective.

Informally, the measures are *turnover factor, packing factor, demand bandwidth, and balance factor*. The turnover factor is a measure of rate of change of the application. The packing factor measures the application workingset and the efficacy of a data layout relative to a memory hierarchy configuration. The demand bandwidth measures the rate at which the program accesses new data. The balance factor relates the demand bandwidth to processor throughput. As a byproduct of the measures, we show why remapping allows a program to execute and achieve a fixed performance point with a lower hardware investment. Thus, simpler machines can achieve the same level of performance as a more complex and expensive machine in the absence of remapping.

The proposed methodology is based on the premise that the bandwidth—the rate of data transfer across a data bus between two objects in the memory hierarchy—is the main bottleneck to faster processing rates. Namely, when the available bandwidth is saturated, any further increase in the clock frequency will not yield a performance improvement [Cragon 1996]. Hence, the balance between both the availability of and demand for memory bandwidth must be carefully managed to deliver the promise of Moore's law to the end-user.

While the measures and the methodology detailed below are applicable to any memory hierarchy, we restrict our focus to the ubiquitous and elaborate cache memories. It will be readily apparent how the proposed measures may be applied in the context of less sophisticated memory structures such as scratchpad memories, on-chip local memories, and similar custom data buffers.

### 5.1 Factors that Impact Memory and Processor Performance

A well-known performance bottleneck in the current and forthcoming computer architectures is the increasing gap between processor and memory speeds [Burger et al. 1996]. A popular strategy advocated to address the bottleneck entails the prediction of a data reference and its delivery to the cache (from memory) in advance of the actual request. This process, known as *prefetching*, serves to mask the long latency associated with a memory access (cache miss). In Figure 12, we illustrate a hypothetical prefetching process. Specifically, a central processing unit (CPU) executes a specific program, $\mathcal{P}$. When a demand for a noncached data item (address) is issued, the processor stalls until the item is delivered to the cache and subsequently forwarded to a register. The address is snooped by the prefetching engine and analyzed in the context of a local history which the engine maintains to guide its decision. The prefetcher may then initiate the retrieval of additional data items from memory and their transfer across the pipeline (bus) to the cache. An

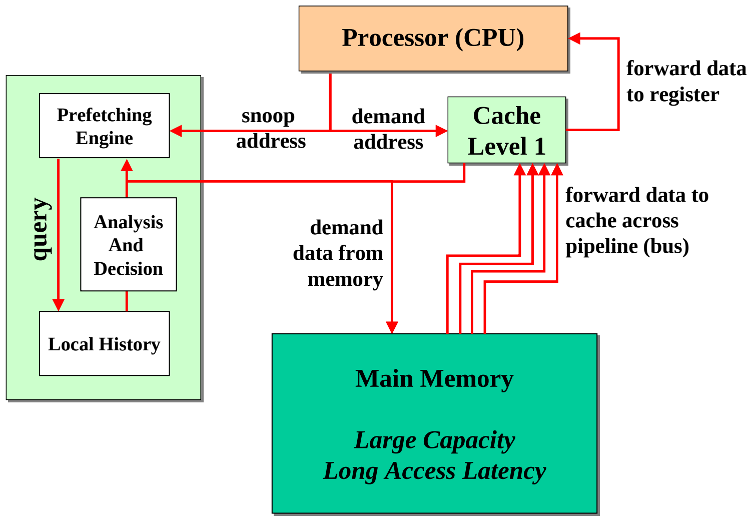


Fig. 12. Prefetching overview.

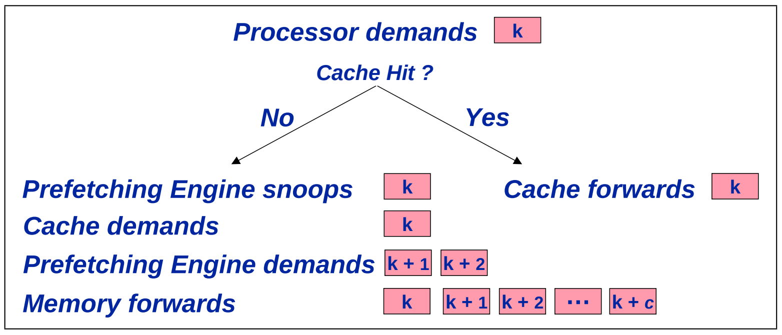

Processor demands k
Cache Hit ?
No
Yes
Prefetching Engine snoops k
Cache forwards k
Cache demands k
Prefetching Engine demands k + 1 k + 2
Memory forwards k k + 1 k + 2 ⋯ k + c

Fig. 13. Example of a block-prefetch engine.

accurate prediction will increase processor throughput by increasing the cache hit rates at lower levels of the hierarchy, thus obviating the need to access the slower memory components. It is worthwhile to note that the success of a prefetching engine hinges on two important factors. The first reflects the predictability of the program, which is the degree to which the anticipated addresses match those demanded by the processor. The second factor is the rate of data transfer (in bytes per cycle) from one object in the hierarchy to another.

### 5.2 Predicting Program Behavior in Hardware

A form of prefetching available in all architectures (including embedded computing systems) is the *block-fetch* [Cragon 1996]. In this scheme, illustrated in Figure 13, given an address $k$, the prefetching engine will initiate the delivery of data at locations $k{+}1$, $k{+}2$, $\ldots$, $k+c$, for $c < \mathcal{B}$, and $\mathcal{B}$ is a design-specific *block size*. Without a loss of generality, we assume the units of the block size to be in terms

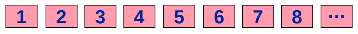


Fig. 14. MAP with spatial locality.

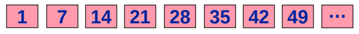


Fig. 15. MAP with no spatial locality.

of a generic addressable unit ($au$) such as a byte. We may think of the physical memory space as if it were partitioned into nonoverlapping sets (blocks), each of size $\mathcal{B}$. Every address $k$ in the physical memory space is mapped to a specific and unique block by virtue of the hardware configuration. In practice, the block-prefetcher does not cross block-boundaries when it initiates data demands (i.e., up to $c < \mathcal{B}$ addresses are fetched, where $k$ and $k + c$ are members of the same block). The advantages of a block-fetch strategy are twofold. First, it serves to amortize the cost of expensive memory accesses by leveraging the capabilities of modern memories to quickly and efficiently retrieve adjacent data items [Cragon 1996]. Second, and more importantly, it masks the latency of a cache miss. The strategy, however, relies on the premise that the program will exhibit address adjacency or *spatial locality*.

In Figure 14, we illustrate an example memory access pattern with an ordered reference stream of adjacent data locations. Consequently, a block-fetch strategy proves highly beneficial as the prefetching engine will accurately predict the program MAP. On the other hand, pointer-centric applications may lack address adjacency as an artifact of poor placement of data during allocation. Figure 15 illustrates an example program reference stream for which a block-fetch strategy is ineffective (when the block size consists of five or fewer addressable units). Note that although the addresses in the stream are separated by a fixed stride, we consider such a MAP irregular (in the context of our analysis) for two specific reasons. First, in order to discover the pattern (stride), the prefetching engine requires additional and more complex logic [Fu and Patel 1992]. However in the context of embedded system design, the additional investments in hardware complexity may be prohibitive. Thus, while a stride-discovering prefetcher may prove beneficial, we show that data remapping enables similar effects using the simpler block-fetch mechanism. Second, we wish to point out the consequences of inaccurate mispredictions by the prefetcher. Namely, incorrect prefetching artificially increases the contention and demand for bandwidth. Furthermore, items that are unnecessarily fetched and cached needlessly occupy valuable storage. Thus, assuming that the reference pattern in Figure 15 is irregular relative to the block-fetch scheme is justifiable.

## 5.3 Impact of Data Remapping on Predicting Program Behavior

The proposed optimization is a remapping of data elements into new blocks, such that data items that are likely to be used contemporaneously belong to the same block. Consequently, the mismatch between the predicted and the actual reference stream is reduced. A possible remapping of the data shown in Figure 15 is illustrated in Figure 16. Observe that the remapped reference stream is similar to that shown in Figure 14 where the predicted set of data references (using a block-prefetcher) exactly match the actual set. In effect, by increasing the spatial locality of a reference stream, data remapping improves the probability of a correct prediction. A direct and desired consequence is a lowering of the amount of data that is

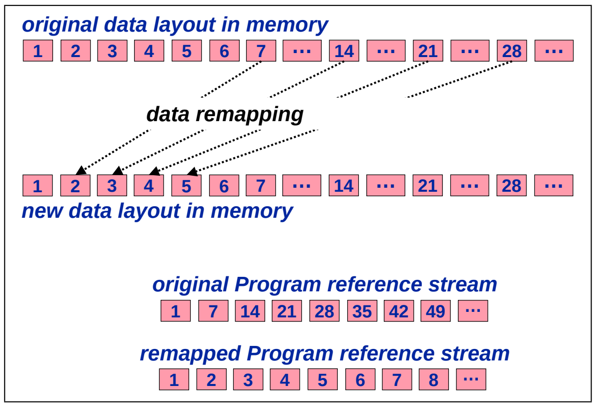


Fig. 16. Example of how remapping improves the accuracy of a block-prefetch engine.

needlessly fetched and cached. Recall in Figure 3 we showed that the amount of data that is unnecessarily fetched is 30% less when data remapping is applied. It trivially follows that with a decrease in the amount of data fetched, the bandwidth demands of the application will be lower.

## 5.4 Performance Metrics

We now formally introduce our novel measures, which (i) completely characterize the intrinsic or virtual behavior of a program; and (ii) relate the virtual characteristics to the realized or observed behavior when a microarchitectural model is imposed during execution.

5.4.1 *Intrinsic Program Behavior.* Consider a data reference trace $\mathcal{T}$ for a specific application as a string of addresses. We partition $\mathcal{T}$ into smaller nonoverlapping substrings, $s_1$, $s_2$, ..., $s_n$. The number of unique characters in each substring $s_i$ equals $\mathcal{V}$, the virtual workingset size.[5] In addition, the substrings do not overlap, and therefore $\mathcal{T} = s_1|s_2|\ldots|s_n$. We now define the cost of a transition between two substrings $s_i$ and $s_{i-1}$ as the *turnover factor*,

$$\Gamma(s_{i>0}) = \mathcal{V} - |\hat{s}_i \cap \hat{s}_{i-1}|,$$

where $\hat{s}_i$ is the set formed from $s_i$, and $\hat{s}_0$ is the empty set; we refer to $\hat{s}$ as the virtual workingset. The turnover factor quantifies the temporal locality inherent in an application. That is, the average turnover factor of an application with perfect temporal locality—complete data reuse—would equal zero. On the other hand, a turnover factor equal to $\mathcal{V}$ is indicative of an application that does not exhibit any temporal locality. Furthermore, given a workingset's turnover factor, we can define its *demand bandwidth*,

$$\mathcal{D}(s_{i>0}) = \Gamma(s_i)/|s_i|,$$

[5]Here again we assume the units of a workingset size are in terms of a generic addressable unit such as a byte.

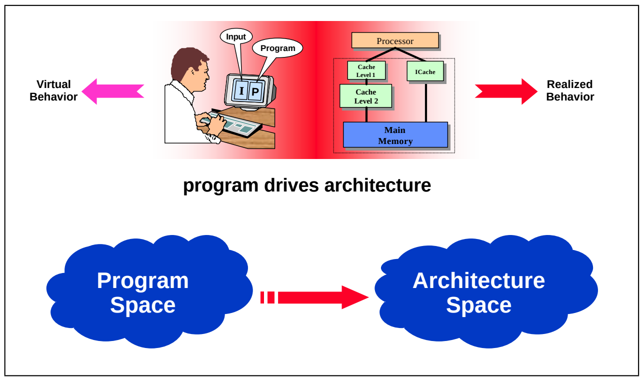


Fig. 17. The program drives the architecture.

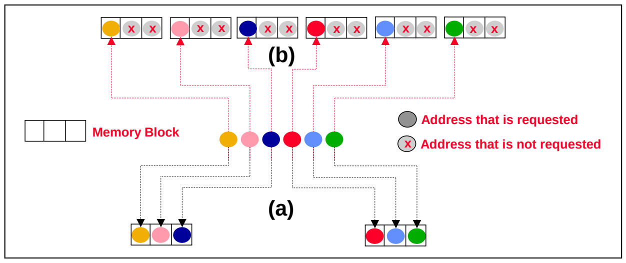


Fig. 18. Examples of mappings addresses to memory blocks.

as the rate at which new data elements are referenced by the program. Here, we use the length of each substring as a measure of time, such that a virtual workingset spanning a long period of time requires less bandwidth than a workingset that is changing very rapidly (i.e., $|s_i| = \mathcal{V}$).

Whereas the turnover factor and demand bandwidth capture intrinsic program behavior, it is necessary to interpret their implication in context of an architecture model as shown in Figure 17. That is, a fixed program-input workload pair may manifest different *realized behaviors* when executed on various computing platforms. Of particular importance and focus is the notion of spatial locality and its impact on bandwidth demand and processor throughput.

5.4.2 *Impact of Microarchitecture on Program Behavior.* Recall from Section 5.2 that for every address $k$ in the memory space, there exists a unique block $\mathcal{X}$ such that $k \in \mathcal{X}$. Therefore, given a virtual workingset $\hat{s}_i$, let $\mathcal{R}_i$ equal the total number of blocks necessary to map every address $k$ in $\hat{s}_i$ to its corresponding $\mathcal{X}$. As an example, consider a virtual workingset consisting of six (unique) addresses and a block of three addressable units. In Figure 18(a), we illustrate one possible mapping such that $\mathcal{R}$ equals two. However, the architecture may impose an alternate mapping such that each address is located in a different block (Figure 18(b)) and as a result, $\mathcal{R}$ equals six. Therefore, in the context of a concrete machine model, the size of a physical workingset may exceed that of the virtual workingset. This implies that some data is needlessly, but necessarily, fetched and cached. There are two important consequences that follow as a result. First, to accommodate the unnecessary data items that are fetched, a cache with a greater capacity may be necessary. Second, it trivially follows that the demand bandwidth will proportionally increase, as demonstrated below. In order to quantify the extent to which these two consequences impact the memory design, we first define the *packing factor*:

$$\Phi(s_i) = \mathcal{R}_i \times \mathcal{B}/\mathcal{V}$$

as the ratio of the physical workingset size ($\mathcal{R} \times \mathcal{B}$) to that of the virtual workingset. When the ratio is greater than one, the bandwidth demand will proportionally increase, and hence

$$\mathcal{D}(s_{i>0}) = \Gamma(s_i) \times \Phi(s_i)/|s_i|.$$

Revisiting the example in Figure 18(b), for a virtual workingset size $\mathcal{V} = 6$ *au* and block size $\mathcal{B} = 3$ *au*, the packing factor is equal to three. Therefore, the rate of delivery of every block fetch must equal three units per cycle, lest the processor incur additional stall cycles as it awaits the delivery of data (i.e., the next processor-initiated memory fetch cannot proceed until a previous bus transaction has completed). By contrast, the mapping of addresses to memory blocks in Figure 18(a) leads to a packing factor of one. As a result, the rate of data delivery across the memory hierarchy need not exceed a rate of one unit per cycle.

In Figure 19, we plot the physical workingset size ($\mathcal{R}_i * \mathcal{B}$) as a function of time for a representative benchmark before and after remapping for $\mathcal{V} = 32$ *au* and $\mathcal{B} = 32$ *au*. Observe that in the absence of data remapping, the application suffers from a severe lack of spatial locality, evident by the numerous broad peaks throughout the application's lifetime. By contrast, when locality enhancement is applied via data remapping, the physical workingset sizes are significantly and consistently smaller. As a consequence, it is possible to achieve better performance using data remapping in conjunction with a smaller cache , as we noted earlier in Section 4. The lower graph of Figure 19 reinterprets a figure shown earlier in Section 1.3. Specifically, we plot the packing factor for the same application before and after remapping as a function of time; also shown is the enhancement ratio due to remapping. Note the prominent peaks that occur at several time periods for the baseline reference trace (before remapping). They are an indication of how poorly the data layout correlates with the application MAPs. In contrast, the reorganized layout is dramatically better, with little jitter (i.e., relatively uniform over time), thus reducing unpredictable memory hierarchy behavior and delivering better overall performance.

The utility of the packing factor as a guide to memory system design is self-evident. Specifically, if a locality-enhancing algorithm (LEA) fails to lower the packing factor to an acceptable value, it will prove worthwhile to design a memory subsystem that does not fetch data in blocks (which in turn lowers the complexity of a design). Alternatively, the measure may be used to evaluate the merits of various LEAs and select the optimization that yields the best overall results in terms of system complexity and performance.

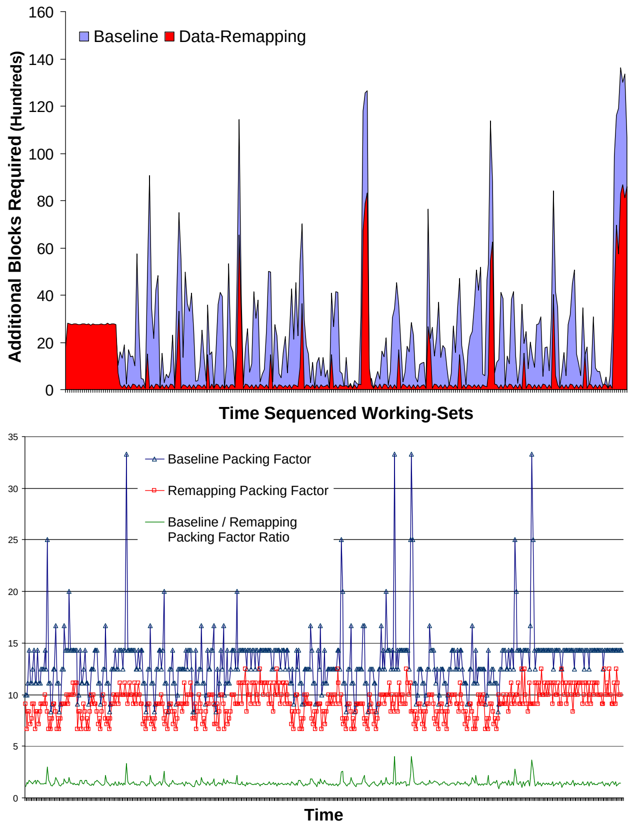


Fig. 19. Demonstrating the relationship between the physical workingset size (top) and the packing factor (lower graph) as a function of time.

5.4.3 *Impact of Bandwidth on Processor Throughput.* Revisiting our initial claim that the processing rate is limited by the amount of available bandwidth, we define the *balance factor* as $\Psi = \widehat{\mathcal{D}}/\mathcal{W}$, where $\widehat{\mathcal{D}}$ is the average demand bandwidth for the entire memory reference trace and $\mathcal{W}$ is the available system bandwidth. Hence, when the demand bandwidth of the application exceeds the available bandwidth, the processor stall penalty will lengthen by a factor of $\Psi$.

In Table XII we illustrate how the block size $\mathcal{B}$ and hardware bandwidth $\mathcal{W}$ impact overall performance before and after remapping. The first column lists the block size and system bandwidth (in bytes and bytes per cycle, respectively). For each row, we report the unoptimized and optimized balance factors in the second and third columns, respectively. Similarly, columns four and five report the unoptimized and optimized execution times in cycles. Each of the values is properly normalized such that the balance factor (and execution cycles) of the unoptimized case with the minimum bandwidth for a chosen block size is one. For example, the results reported in the first row correspond to a block of size 16 bytes and an available bandwidth of 8 bytes per cycle. The balance factor or stall penalty before

Table XII. Memory and Processor Performance for Various Block and Bandwidth Configurations

| | S | tall Penalty ($\Psi$) | E | xecution Cycles |
|---|---|---|---|---|
| $\mathcal{B}$ / $\mathcal{W}$ | Before remapping | After remapping | Before remapping | After remapping |
| 16 / 8 | 1 | .48 | 1 | .78 |
| 16 / 16 | .98 | .48 | .98 | .78 |
| 32 / 8 | 1 | .34 | 1 | .80 |
| 32 / 32 | .96 | .34 | .98 | .78 |
| 64 / 8 | 1 | .35 | 1 | .89 |
| 64 / 64 | .90 | .32 | .95 | .85 |

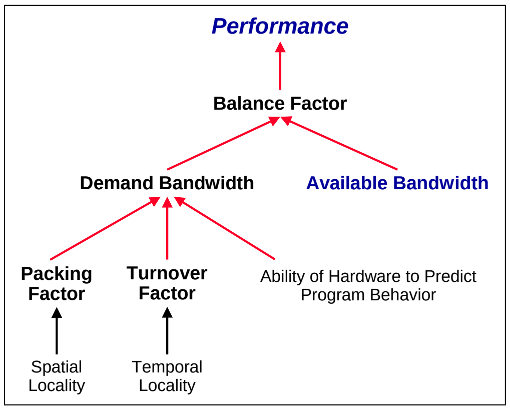


Fig. 20. Summary of metrics for design space exploration of memory systems.

optimization is one. After applying data remapping and for the same hardware configuration, the stall penalty ($\Psi$) is halved (i.e., 0.48 of the unoptimized penalty for the same configuration). Concomitantly, the execution time is reduced 22% (i.e., 0.78 of the unoptimized execution time). In the second row, we consider a scenario where the bandwidth is increased while the block size is held constant (i.e., $\mathcal{B} = 16$ bytes and $\mathcal{W} = 16$ bytes per cycle). In this case, the stall penalty of the application before remapping is only marginally lower (2%) compared to that where the bandwidth was less. From an embedded system perspective, the additional hardware complexity invested to increase the system bandwidth would not be justifiable. Specifically, data remapping yields a significantly better reduction in stall penalties with lesser hardware investments. The tables demonstrate that this trend is consistent for several block and bandwidth configurations.

## 5.5 Summary and Impact

In Figure 20 we summarize our measures, which we believe can serve as the quantitative foundations of a design space exploration system in which remapping and other novel compiler techniques can play a crucial role for optimizing the costs associated with the cache subsystem.

While we have demonstrated how data remapping can be used to navigate the memory design space and reduce memory needs, we believe the methodology is applicable to a wide range of locality-enhancing and latency-masking techniques. For example, loop-tiling, loop-skewing, and numerous other control-flow transformations [Ding and Kennedy 2000; 1999; Carter et al. 1998; McKinley et al. 1996; Lam et al. 1991] have been shown to significantly improve the temporal locality (and, inturn, performance) of applications with predictable access patterns. Whereas such optimizations are rather complex and often fare poorly when applied to important dynamic and pointer-intensive real-world applications, our measures will remain important design tools that can quantify the improvements achieved along the range of identified dimensions.

The proposed measures will also quantify the utility of the latter class of latency-masking techniques such as data prefetching. In this context, we observe that a prefetch is a degenerate form of a regular memory access, and will therefore consume bandwidth. Hence, an effective prefetching strategy will not increase the application's demand bandwidth requirements. By contrast, a prefetching strategy that tends to mispredict program behavior will inflate bandwidth requirements. Note that it is trivial to characterize cache management policies (e.g., data placement and eviction strategies) as prefetching mechanisms such that the impact of the executed decisions is accounted for as part of the demand bandwidth. When one considers the efficiency of a prefetching engine, an interesting question arises: *what is the simplest strategy that best matches the program behavior?* From the perspective of a custom embedded system, a simple prefetching mechanism is desirable, as its logic complexity and cost are likely to be low. A quantitative characterization of the prefetching engine in terms of its logic complexity is beyond the scope of this article and is the topic of ongoing research.

## 6. ENGINEERING DETAILS AND OTHER ODDITIES

In the following section, we highlight some noteworthy engineering details and oddities we have encountered during our study of data remapping.

### 6.1 Static Alias Analysis

The proposed optimizations are geared towards preserving program semantics in the presence of pointer variables. In order to ensure program correctness, alias analysis is necessary to guide the selection of the proper offset computation function during code generation. So far, we have defined two expressions for remapping (*GDRemap* and *DDRemap*) and one for traditional data layouts. Therefore, alias analysis in this context aims to statically characterize pointer accesses into one of three classes: *points-to static object*,[6] *points-to heap object*, or *unknown*. Clearly, it is the class of unknowns that has been a *show-stopper* in the past. For our purpose, pointer references are classified using Steensgaard points-to analysis [Steensgaard 1996] applied to the application as a whole. Steensgaard analysis was chosen because it does not discriminate among the fields of a record. Specifically, a pointer that may alias a field within a record is classified as an alias to a record. By contrast, Andersen-style [Andersen 1994] analysis does discriminate between field accesses. Clearly, the former is desired for the purposes of data remapping.

[6]A slightly more detailed analysis would refine this category into two subclasses: one for stack objects and the other for global data.

Unresolved aliases mainly arise from nonstandard programming practices such as pointer arithmetic and extensive type coercion. However, we found that the compiler is often capable of inferring the intended *meaning* of various pointer arithmetic manipulations. For example, let $P$ represent a pointer variable pointing to a record with three fields, each four bytes in size. Meaningful pointer arithmetic expressions to access the first, second, and third fields of the aliased record first cast $P$ into an absolute address (i.e., an integer) then apply some addition of the form $(cast(P) + c)$ where $c \in \{0, 4, 8\}$. Program behavior for other values of $c$ would be undefined [Kernighan and Ritchie 1988; ANSI ]. The compiler may use the specified offset $c$ and solve for the field index $f$ using the traditional OCF. The expression is then adjusted accordingly using the remapping OCF. An alternate, but related, form of pointer arithmetic does not coerce (cast) the type of $P$. In this context, an expression of the form $(P + c)$ translates to $P + c \times RecordSize(*P)$, where $*P$ is the type of the record aliased by $P$. Again, this form of pointer arithmetic can be easily addressed using the OCFs introduced in Section 2.3. Namely, the expression translates to $P + c \times MaxFieldSize(*P)$.

## 6.2 Precompiled Libraries

Large applications will often use precompiled libraries to reduce development time. If data remapping is partially applied, the modified layout may not be propagated properly to the libraries, and program correctness cannot be preserved. A viable solution is to undo the effects of remapping (data duplication) at a library function call site and re-remap upon return. However, if this is done frequently, the cost of data duplication may not be tolerated, and the optimization should be inhibited. In general, any library function that operates on objects as a whole is an issue (e.g., `quicksort` and `memcpy`). In practice it is rather obvious how new remapping-compatible variants of such functions could be implemented (for example, `memcpy` in our case is actually replaced by a fine-grained copy). Furthermore, in an embedded system design setting it is common to have access to the entire application source code, and thus it is reasonable to assume that the complete program may be compiled and hence remapping consistently applied.

## 6.3 Object Deallocation

Many applications do not use heap objects that are persistent throughout execution. Rather, they may frequently deallocate old objects and subsequently allocate new ones. To enable the remapping of dynamic data objects, wrappers reserve relatively large clusters that are used for record allocation. Hence, when a request to deallocate an object is made, a nonempty cluster may not be freed. The implemented resolution is to maintain a bit vector per cluster that indicates when the cluster is no longer in use and hence may be safely deallocated. In other words, the object deallocation is delayed. This may in some cases result in a larger program memory footprint. Some operating systems (e.g., SunOS) actually use a delayed deallocation strategy, which may explain why DM (for example) did not benefit from data remapping on the Sun UltraSparc II (Table XI). The operating system for both Pentium machines is Linux RedHat version 6.2.

### 6.4 Remapping Variations

The proposed offset computation functions may be generalized to express other complex remapping strategies. For example, the optimization discussed here collocates the respective fields from various objects. It may be desirable however to collocate several fields from the same record as well. An example where such a collocation might be applicable is in pointer-heavy applications, where the stagger distance between any two fields must be statically known. In this context, failing to collocate fields may lead to excessive padding between successive data elements and thus diminish the benefits due to remapping. Therefore, for data record types with numerous fields of varying sizes, it will prove useful to collocate several fields to form an aggregate of constant size. A *characteristic function* may then be used as a *second-order* OCF to locate fields within an aggregate. In order to achieve an effective field-level collocation, it shall be necessary to augment our NAP analysis such that the interactions of data fields in a memory access pattern are tracked.

## 7. REMARKS

In this article, we presented a novel data remapping algorithm and demonstrated its surprising effectiveness in lowering the memory needs of a range of floating point and integer applications. *A notable contribution of our article is the use of data remapping as a significant step in optimizing the cache memory needed to achieve a given performance goal.* To achieve this, we quantify the dynamics of memory behavior through a range of cost measures capturing the program rate of change and its impact on performance. *Thus, given a particular application, optimizing the size and cost of a cache will reduce to "navigating" over a space of these measures*—a topic for future research. Here our cost measures can serve as the foundation for a design space exploration system where data remapping and other conventional optimizations such as loop transformations can be used as key optimization steps. Furthermore, for several COTS microprocessors with fixed cache architecture, such as the Pentium and UltraSparc, we show that remapping can achieve a performance improvement of 20% on the average. In addition, for a parametric research HPL-PD microprocessor, which characterizes the new Itanium machines, we achieve a performance improvement of 28% on average. All of our results are achieved using the DIS, Olden and SPEC2000 family of integer and floating-point benchmarks.

ACKNOWLEDGMENTS

The authors wish to thank Richard Lipton, Santosh Pande and Yannis Smaragdakis for their comments and suggestions. In addition, the authors thank Lakshmi Chakrapani, Mongkol Ekpanyapong, Benjamin Goldberg, Roy Ju, Ken MacKenzie, Scott Mahlke, Trevor Mudge, Marc Saint, Suren Talla, Gary Tyson, P. S. Thiagarajan, Wong Weng-Fai, and the reviewers for their valuable advice and feedback.

## REFERENCES

Abraham, S. and Mahlke, S. 1999. Automatic and efficient evaluation of memory hierarchies for embedded systems. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture.* 114–125.

Aditya, S., Rau, B. R., and Kathail, V. 1999. Automatic architectural synthesis of VLIW and EPIC processors. In *Proceedings of the International Symposium on System Synthesis.* 107–113.

Andersen, L. 1994. Program analysis and specialization for the C programming language. Ph.D. thesis, University of Copenhagen.

ANSI. The ANSI C standard. www.lysator.liu.se/c/.

Ball, T. and Larus, J. 1996. Efficient path profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture.*

Bass, M. and Christensen, C. 2002. The future of the microprocessor business. *IEEE Spectrum 39,* 4 (Apr.), 34–39.

Burger, D., Goodman, J., and Kagi, A. 1996. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architectures.* 78–89.

Burks, A., Goldsteind, H., and von Neumann, J. 1987. Preliminary discussion of the logical design of an electronic computing instrument. Papers of John von Neumann.

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

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

Catthoor, F., Wuytack, S., DeGreef, E., Balasa, F., Nachtergaele, L., and Vandecappelle, A. 1998. *Custom Memory Management Methodology. Exploration of Memory Organization for Embedded Multimedia System Design.* Kluwer Academic Publishers.

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

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

Chilimbi, T. and Hirzel, M. 2002. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* 199–209.

Cragon, H. 1996. *Memory Systems and Pipelined Processors.* Jones and Barlett Publishers.

Ding, C. and Kennedy, K. 1999. 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.* 229–241.

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

DIS. Data intensive systems benchmark suite. www.aaec.com/projectweb/dis/.

Fu, J. and Patel, J. 1992. Stride directed prefetching in scalar processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture.* 102–110.

Hwu, W., Mahlke, S., Chen, W., Chang, P., Warter, N., Bringmann, R., Ouellette, R., Hank, R., Kiyohara, T., Haab, G., Holm, J., and Lavery, D. 1993. The superblock: An effective technique for VLIW and superscalar compilation. *Journal of Supercomputing.*

IPF. The Intel ITANIUM Processor Family. www.intel.com/products/server/processors/server/itanium/index.htm.

Kathail, V., Schlansker, M., and Rau, B. R. 2000. HPL-PD architecture specification: Version 1.1. Tech. Rep. HPL-9380 (R.1), Hewlett Packard Laboratories. Feb.

Kernighan, B. and Ritchie, D. 1988. *The C Programming Language.* Prentice Hall.

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

Kulkarni, C., Catthoor, F., and Man, H. 2000. Advanced data layout organization for multimedia applications. In *Proceedings of the Workshop on Parallel and Distributed Computing in Image Processing, Video Processing, and Multimedia*.

Lam, M., Rothberg, E., and Wolf, M. 1991. The cache performance of blocked algorithms. In *Proceedings of the Fourth International Conference in Architectural Support for Programming Languages and Operations Systems*. 63–74.

Lee, P. and Kedem, Z. 1990. Mapping nested loop algorithms into multidimensional systolic arrays. *IEEE Transactions on Parallel and Distributed Systems 1,* 1, 64–79.

Mahlke, S., Lin, D., Chen, W., Hank, R., and Bringmann, R. 1992. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*. 45–54.

McKinley, K., Carr, S., and Tseng, C. 1996. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems 18,* 4 (July), 424–453.

Nystrom, E., Ju, R., and Hwu, W. 2001. Characterization of repeating data access patterns in integer benchmarks. In *Proceedings of the Workshop of Memory Performance Issues held in conjunction with the 28th International Symposium on Computer Architecture*.

Olden. The olden benchmark suite. www.cs.princeton.edu/˜mcc/olden.html.

Palem, K., Rabbah, R., Mooney, V., Korkmaz, P., and Puttaswamy, K. 2002. Design space optimization of embedded memory systems via data remapping. In *Proceedings of the Languages, Compilers, and Tools for Embedded Systems and Software and Compilers for Embedded Systems*.

Panda, P., Catthoor, F., Dutt, N., Danckaert, K., Brockmeyer, E., Kulkrani, C., Vandercappelle, A., and Kjeldsberg, P. 2001. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems 6,* 2 (Apr.), 149–206.

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

Petrank, E. and Rawitz, D. 2002. The hardness of cache conscious data placement. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages.*

Rau, B. R. and Schlansker, M. 2000. Embedded computing: New directions in architecture and automation. Tech. Rep. HPL-2000-115, Hewlett Packard Laboratories. Sept.

Schreiber, R., Aditya, S., Mahlke, S., Kathail, V., Rau, B. R., Cronquist, D., and Sivaraman, M. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI Signal Processing*.

SPEC. The Standard Performance Evaluation Corporation benchmark suite. www.spec.org.

StarCore. Leadership in DSP technology for communication applications. www.starcore-dsp.com/files/SC140pres.pdf.

Steensgaard, B. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*. 32–41.

Taub, A. 1963. *Collected Works of John von Neumann.* The Macmillan Company, New York.

Texas Instrument TI-C6 VLIW Processor. TMS320C600: A high performance DSP platform. www.ti.com/sc/docs/products/dsp/c6000/index.htm.

TriMedia. The TriMedia processor cores. www.trimedia.com/products.html.

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

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