Design Space Optimization of Embedded Memory Systems via Data Remapping

Krishna V. Palem, Rodric M. Rabbah, Vincent J. Mooney III, Pinar Korkmaz and Kiran Puttaswamy

Center for Research on Embedded Systems and Technology

Georgia Institute Of Technology

http://www.crest.gatech.edu

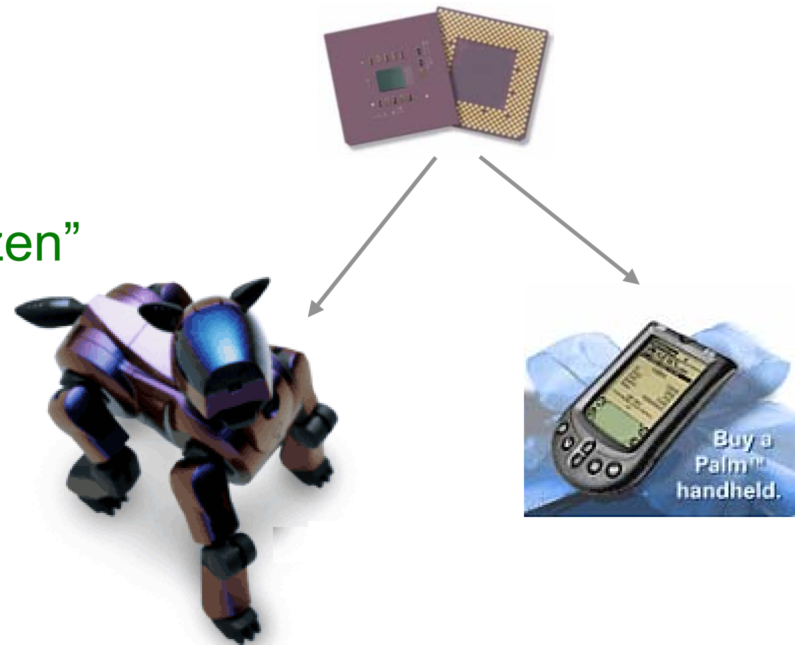
This research is funded in part by DARPA contract No. F33165-99-1-1499, HP Labs and Yamacraw

Continuing Emergence of Embedded Systems

- Favorable technology trends
  - From hundreds of millions to billions of transistors

- Projected by market research firms to be a $50 billion space over the next five years

- *Stringent constraints*
  - Performance
  - Power as “a first class citizen”
  - Size and cost

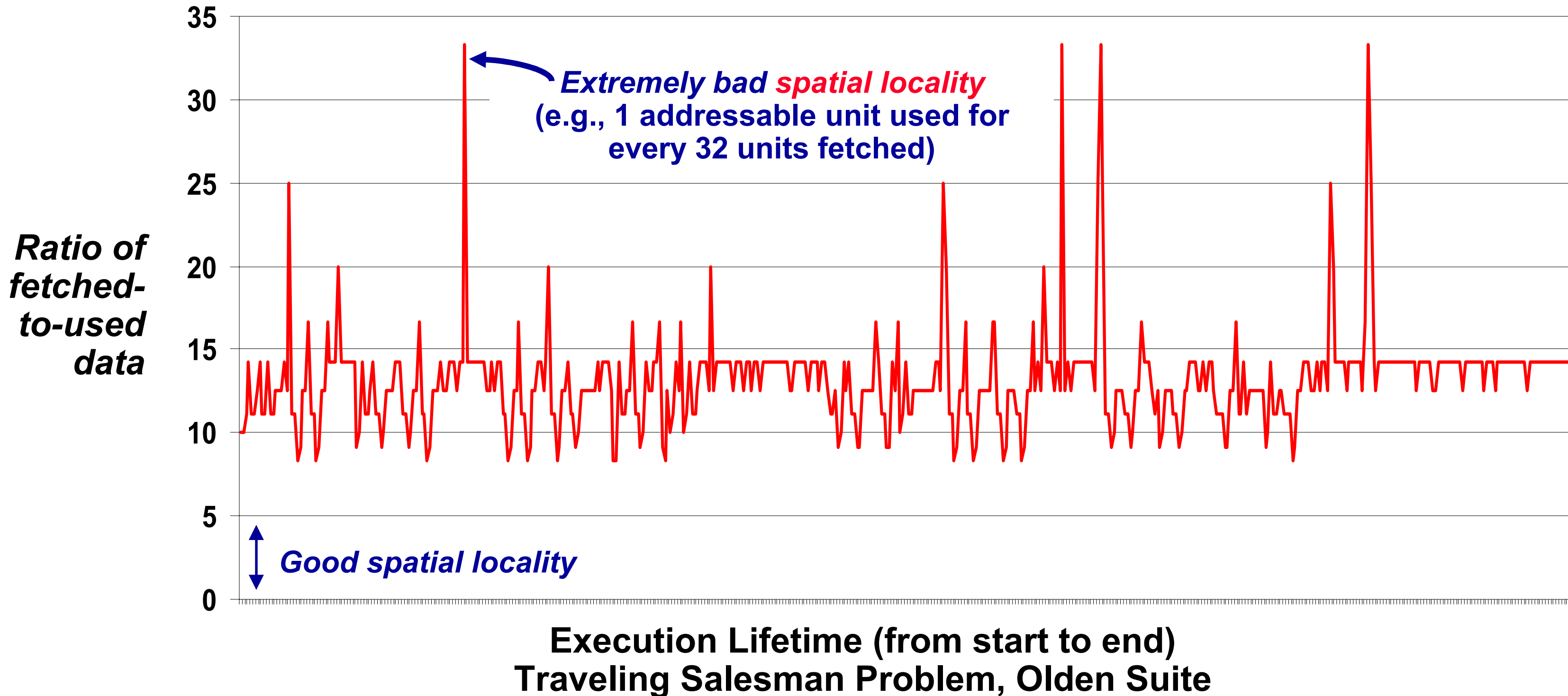


Importance of A Supporting Memory Subsystem

- Disparity between processor speeds and memory access times is increasing
  - Custom embedded processors afford massive instruction level parallelism
    - **A cache miss at any level of the memory hierarchy incurs substantial losses in processing throughput**
- Deep cache hierarchies help bridge the speed gap, but at a cost
  - Trade-off capacity for access latency
  - Significant microarchitecture investment
    - **Power requirements, size and cost**
  - Caches are vulnerable to irregular access patterns

Shortcomings of A Memory Hierarchy

- Caches Are Not Well Utilized



- Bandwidth from memory to cache is also limited
  - When data is fetched but not used, bandwidth is wasted
- Important to maximize resource utilization

Impact of Spatial Locality on System Design

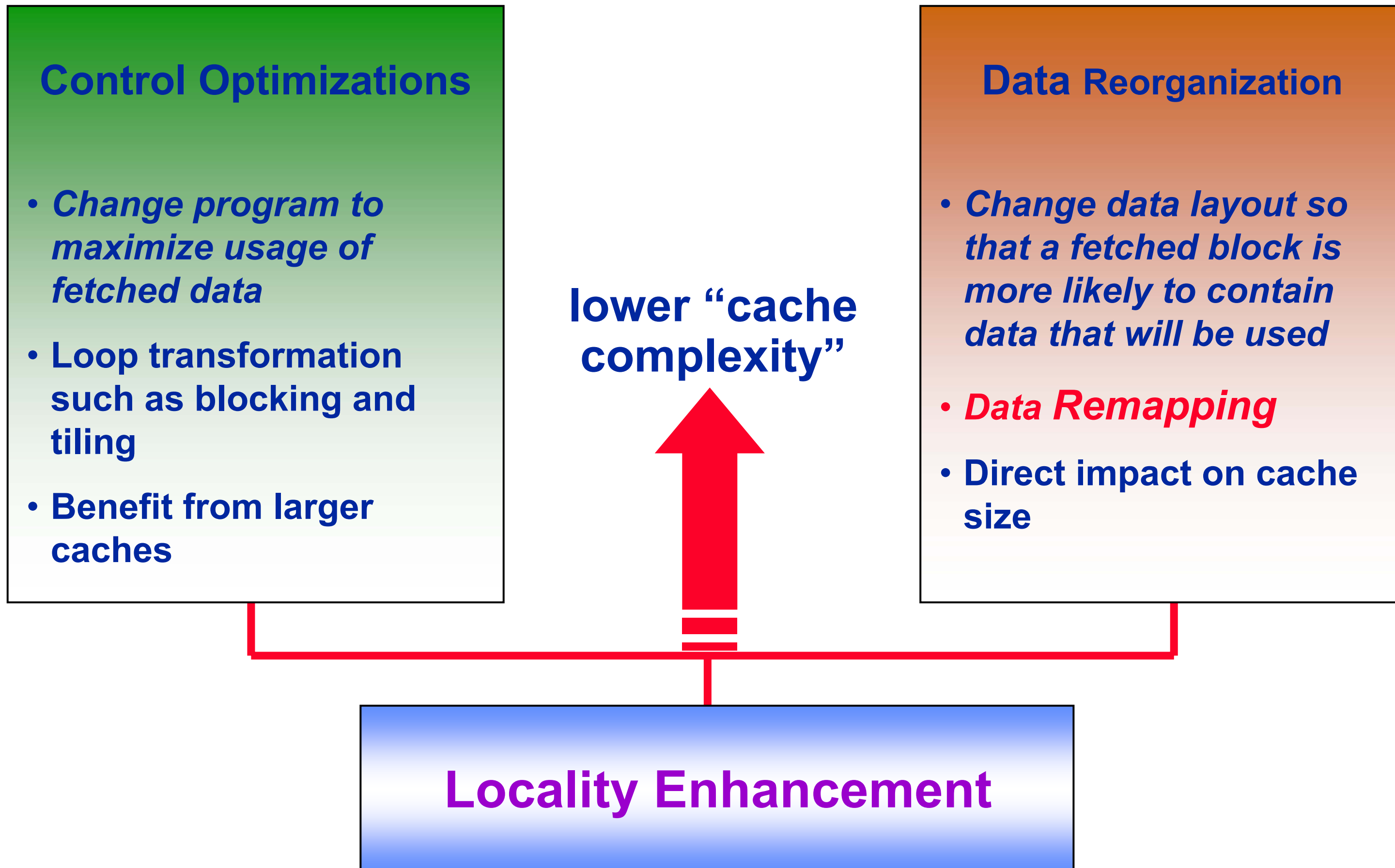
- When the application has low spatial locality, then the usable cache size is less than its actual capacity
  - If ¼ of the fetched data is used then most of the cache resource is used to store unnecessary data
    - **For a 512 Kb cache, only 128 Kb are effectively used**
  - To compensate for wasted storage, a larger cache is necessary
  - Unfortunately, cost and logic complexity are proportional to size
    - **This is particularly undesirable in embedded systems where profit margins and system area are low**
    - **In addition, larger circuits are undesirable from an energy perspective**

Brand	Cache Size	$ Cost
Cypress CY62128VL-70SC	**128 Kb**	**4.43**
Toshiba TC55V400AFT7	**512 Kb**	**9.19**
Toshiba TC55W800FT-55	**1024 Kb**	**24.00**

- Similarly, when the application has low spatial locality, the system bandwidth is not used effectively
  - Bandwidth is wasted
  - Longer memory access times

Enhancing Spatial Locality

- Compiler optimizations can alleviate the amount of investment in caches



Scope of Control and Data Optimizations

- Control optimizations work well for numerical computations that stream data
  - Applications such as FFT, DCT, Matrix Multiplication, etc.
  - Data stored in arrays
  - Programs are optimized to use current data set as much as possible
  - *Ding and Kennedy in PLDI 1999*
  - *Mellor-Crummey, Whalley and Kennedy in IJPP 2000*
  - *Panda et al. in ACM Transactions on Design Automation of Electronic Systems 2001*
- However, a large class of important real world applications extend beyond number crunching
  - Complex data structures or records
    - **Sets of variables grouped under unique type declarations**
  - Difficult to modify program to maximize fetched data usage

Advantage of Data Optimizations

- Control optimizations break down in the presence of complex data structures

  Example
  - Linked list of records, each record has three fields
    - ***Key***, ***Datum*** **and** ***Next*** **(a pointer to the next record in the list)**
  - Search for a record with special *Key* and replace *Datum*
    - **The search will need the** ***Key*** **and** ***Next*** **fields of many records**
    - **By contrast, only one** ***Datum*** **field is necessary**

- Not clear how to modify a program to maximize use of fetched *Datum* field
  - Many similar examples in real world applications

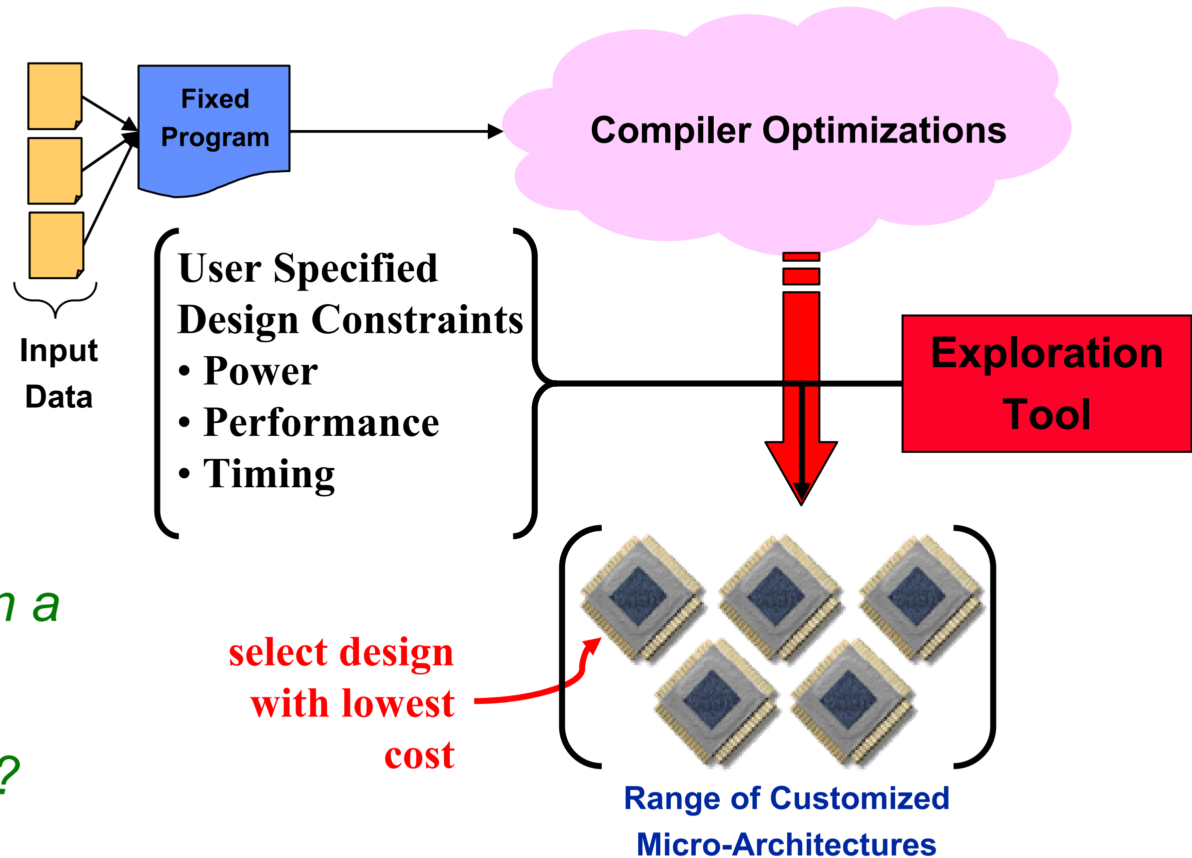
- Best to reorganize the data so that each block contains more items that will be used together
  - *Chilimbi and Larus in PLDI 1999*
  - *Kistler and Franz in PLS 2000*

Realizing Systems With Simpler (Smaller) Caches via Data Remapping

- *Data remapping* is a novel data reorganization algorithm
  - *Fully automated whereas previous work requires manual retooling of applications*
  - Linear time complexity
  - *Pointer-friendly, a show stopper for related work*
  - Uses standard allocation strategies
    - **Previous work uses complex heap allocation strategies**
  - *Compiler directed, does not perform any dynamic data relocation*
    - **Previous work incurs dynamic overheads because they move data around (not desirable from a power/energy perspective)**
- Reduce the "workingset" and enhance resource utilization
  - Influence cache size and bandwidth configurations during system design for a fixed performance goal

Novel Use of A Compiler
A Focus On Embedded System Design

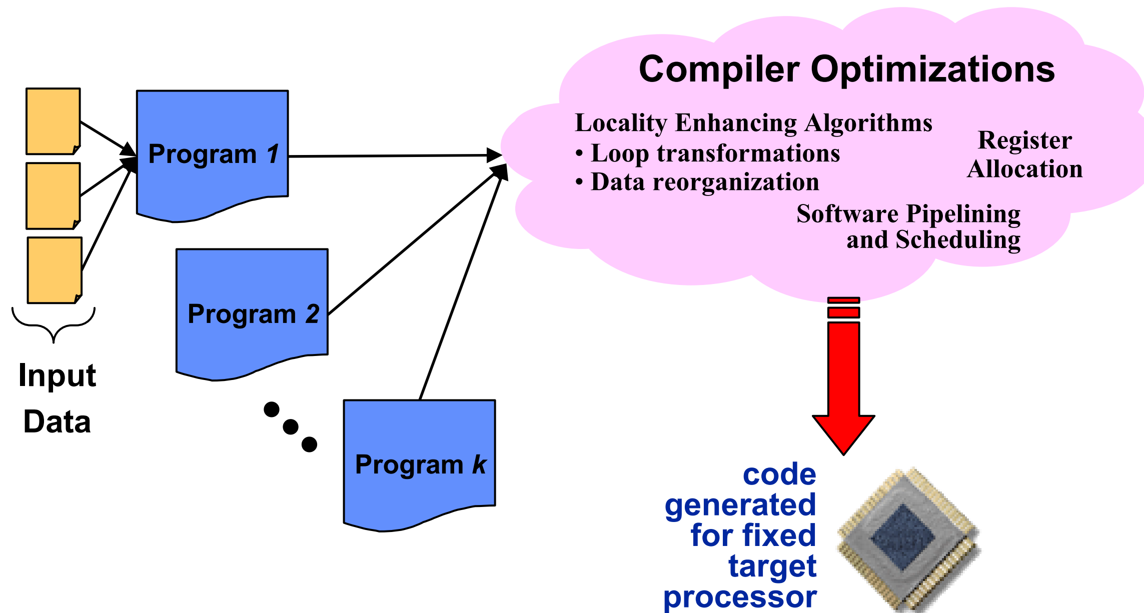
- *Fix program*
- *User specifies design constraints*
- *Optimizations and exploration tools search design space*
- *Best design is chosen*



For a desired performance goal, can a system be designed with a smaller cache and hence lower cost?

Traditional Role of a Compiler

- Compiler optimizations such as locality enhancing techniques are well-known in traditional compiler optimizations
  - Fixed target processor
  - Optimize program for performance



Presentation Outline

- Introduction
- Data Remapping Algorithm
  - Overview
  - Remapping of Global Data Objects
  - Remapping of Heap Data Objects
  - Analysis for Identifying Candidates for Remapping
- Evaluation Framework and Results
  - Design Space Exploration via Data Remapping
- Concluding Remarks

Data Remapping Overview

- Focus of data reorganization is on data records where the program reference pattern does not match the data layout in memory
  - Data is fetched in blocks
  - If the fields of a record are located in the same block but they are not all used at the "same" time, then some fields were unnecessarily fetched
    - **Need to filter out such record types for remapping**

- When we have identified records how do we remap?
  - Runtime data movement is expensive

Remapping Arrays Via Offset Computation

remap data fields for collocation →

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

apply traditional data layout →

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

```
struct Node {
    int  A;
    int  B;
    int  C;
};

Node List [N];
```

Example *C-style* code. Node is a record with three fields. List is array of Nodes.

the fields of Node are adjacent

Traditional List layout

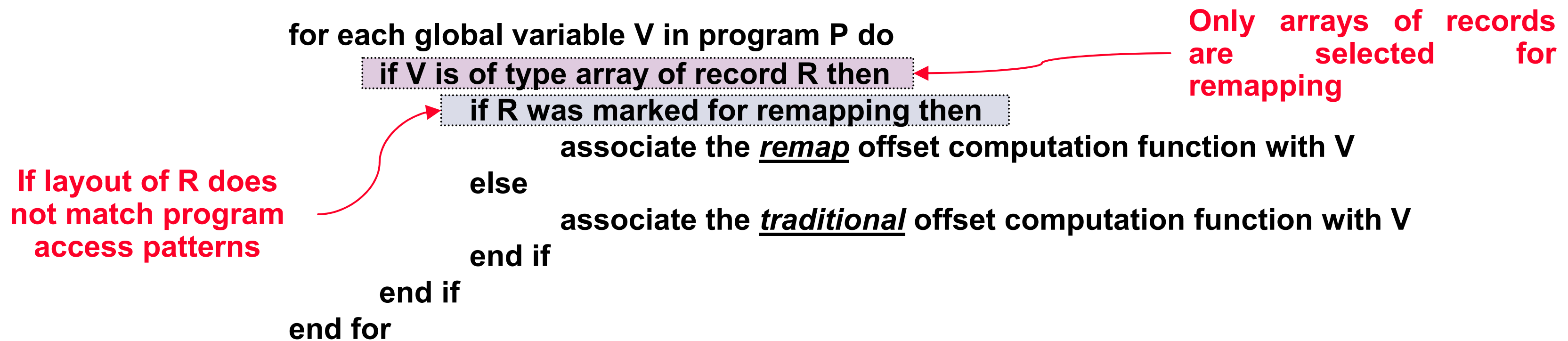
Remapped List layout

the fields of Node are staggered by Rank(List) or *N*

the fields of List[k] to List[k+N] Nodes are co-located

Algorithm for Remapping Global Data Objects

- The algorithm for remapping global data structures selectively attributes global data objects with the *remap offset computation function*



- The offset function is evaluated during *code generation* to locate a target field
- The traditional function is associated with all other global and stack-allocated structures
  - Stack objects are often small and exhibit good temporal locality

Data Remapping Overhead

- The remapping of global data objects does not contribute run-time overhead

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

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

  – Both functions require the *same* computation overhead for the first term
    – ***K*** **may or may not be available to the compiler**
  – The second term does not incur any run-time cost
    – **The value of *N* is available to the compiler**

Remapping Technique for Heap Objects

- What if we have dynamically allocated records, is it still possible to remap using offset expressions?
  - Yes, first we introduce a wrapper around standard allocation tools in the language
    - **Wrapper is very simple, it allocates a memory pool to hold a few records**
  - Code generator handles offset computation
- By contrast, traditional allocation tools are oblivious to the memory hierarchy
  - The resulting layout may interact poorly with the memory access pattern
  - To resolve the poor layout to access interaction, the objects can be reorganized at specific intervals during execution
    - **After a large tree is built, the nodes can be reordered**
    - **Reorganization of objects during execution is limited**
    - **High cost and unsafe in the context of pointer-centric languages**

Locality Enhancing Placement

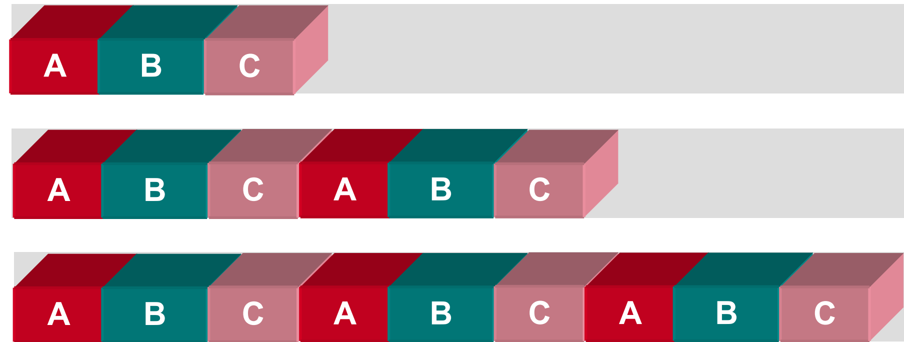
```
struct Node {
        int     A;
        int     B;
        int     C;
};
...
Node* P;

while (condition) {
        P = Allocate (Node);
}
```

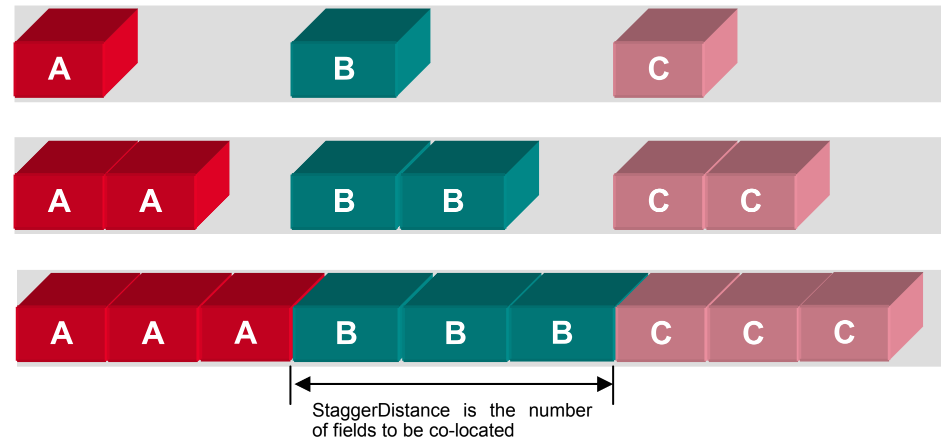
Example *C-style* code. Node is a record with three fields. P is a pointer to a Node.

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

Object layout in Cluster after one, two and three *traditional* allocations of Node



Object layout in Cluster after one, two and three *remapped* allocations of Node



Placement is controlled by an automatically generated light-weight Wrapper

Remapping Heap Objects Via Offset Computation

- Dynamically allocated objects are accessed through *pointer variables*
  - A pointer variable *P* is a variable whose value is a memory location
  - $P \rightarrow x$ refers to the xth field of some record instance
- The code generator must determine which record layout is aliased by a pointer variable
  - If a pointer aliases a dynamically allocated record then the remap offset computation function must be used
  - If a pointer aliases a static or global record then the traditional function must be used
  - In cases where static disambiguation is not possible, a run-time check is necessary

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

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

**P = Record type*

Resolving the Alias Issue

- Since dynamic data reorganization does not affect global objects, a run-time check is used to determine which offset computation function to use
  - The compiler evaluates the remap and traditional expressions
  - The results of both computations are inserted in the instruction stream
  - A run-time comparison of the pointer value to the *stack register pointer* selects the correct offset

```
struct Node {
        int     A;
        int     B;
        int     C;
};
Node List[100];      <- Not Remapped
Node* P;             <- Remapped

if ( select )   P = allocate(Node);
else            P = &List[k];
Print (P->B);
```

R_1 = [P] + Traditional ($P \rightarrow B$);

R_2 = [P] + Remap ($P \rightarrow B$);

P_0 = [P] > Stack Pointer Register

$R_1 = R_2$ if P_0

R_3 = Load R_1

Computation of the proper offset to access element *B* of Node can not be determined at compile time

- The reorganization algorithm reorders the fields of a record such that access to the most frequently used field does not require a run-time disambiguation
  - Both offset expressions evaluate to 0 for the first field of a record

Remapping Heap Objects Via Offset Computation

- Dynamically allocated objects are accessed through *pointer variables*
- The code generator must determine which offset expression to use since different record layouts require different expressions
  - If a pointer aliases a dynamically allocated record then the remap offset computation function must be used
  - If a pointer aliases a static or global record then the traditional function must be used
  - In cases where static disambiguation is not possible, a run-time check is necessary
    - **The compiler evaluates the remap and traditional expressions**
    - **The results of both computations are inserted in the instruction stream**
    - **A run-time comparison of the pointer value to the *stack register pointer* selects the correct offset**

```
struct Node {
        int     A;
        int     B;
        int     C;
};
Node List[100];
Node* P;

if ( select )   P = allocate(Node);
else            P = &List[k];
Print (P→B);
```

Not Remapped (→ Node List[100];)
Remapped (→ P = allocate(Node);)

Computation of the proper offset to access element *B* of Node can not be determined at compile time

R_1 = [P] + Traditional (P→B);

R_2 = [P] + Remap (P→B);

P_0 = [P] > Stack Pointer Register

$R_1 = R_2$ if P_0

R_3 = Load R_1

Comment About Alias Disambiguation

- Clearly dynamic disambiguation of all pointer accesses is not efficient
- Steensgaard points-to analysis is used to resolve as many pointer aliases at compile time
  - Analysis does not discriminate between aliases of a pointer and fields of a record
    - **Pointer to any field of a record is classified as an alias of the entire record**
    - **By contrast to Andersen point-to analysis**
  - Linear time algorithm
- Combination of compile time and dynamic disambiguation empirically observed to be effective
  - On average, 3-5% increase in dynamic instruction count

Algorithm for Remapping Dynamic Data Objects

- The algorithm for dynamic data reorganization focuses on repeated single object allocations
  - The algorithm for global data reorganization can be extended for dynamic array-of-record
- Methodology is to automatically generate a light-weight wrapper around traditional memory allocation requests
  - Wrapper controls the placement of new objects relative to existing ones

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

Eliminates overhead for most frequently accessed field

Replace traditional object allocator with *locality enhancing allocator*

Selecting Candidates for Remapping

- Profile information is analyzed to characterize how well the data layout correlates with the program reference patterns
  - Identify data types with poor memory performance along program hot-spots
  - Build a model of data reuse for extensively used objects

- Analysis computes the Neighbor Affinity Probability (NAP) for each object type
  - NAP ranges from 0 to 1, indicating the probability (from low to high) of a cache block successfully prefetching data

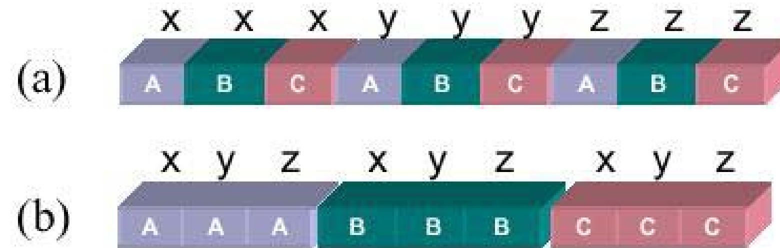
- The neighbor affinity probability is used as a criteria for selecting candidates for data remapping

NAP Computation

```
struct Node {
        int     A;
        int     B;
        int     C;
};

Node x, y, z;
```

Example *C-style* code. Node is a record with three fields.



two example access patterns

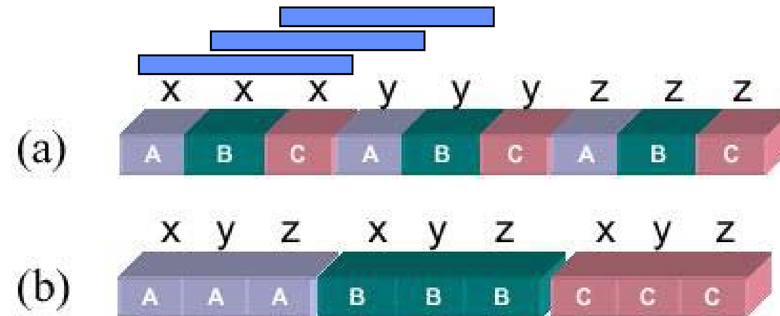
- For a cache block of size $B = 3$
  - Fields of *x* are in one block, those of *y* are in another and similarly, the fields of *z* belong to yet another block
- For an access *j*, does the current layout and block size deliver data that will be used in access $j+1$, $j+2$, …, $j+B-1$?

NAP Computation

```
struct Node {
        int     A;
        int     B;
        int     C;
};

Node x, y, z;
```

Example *C-style* code. Node is a record with three fields.



two example access patterns

Given a program P, a memory access profile trace $T_R = (k, f)^*$ of accesses to fields of a record of type R, and a block size B, let T[i] for $0 < i \leq |T|$ represent the ith pair occurring in T

procedure ComputeAffinity (Program P, Trace T, RecordType R, BlockSize B)

for $j \leftarrow B$ to $|T|$ do
  for $i \leftarrow B - 1$ downto 1 do
    $(k_1, f_1) \leftarrow T[\,j\,]$
    $(k_2, f_2) \leftarrow T[\,j - i\,]$
    if $(k_1 \neq k_2)$ and if f_1 and f_2 may map to the same block, then increment NAP(R)
  end for
end for
$NAP(R) \leftarrow NAP(R) / B\,(|T| - B)$

end ComputeAffinity

In (a) the data layout matches the access pattern well – for $B = 3$, NAP = 7/9

In (b) an alternate layout is necessary – for $B = 3$, NAP = 0

- B is a history window
- Running time is O(|T|)
  - Incremental computation
- Records with NAP values less than a threshold are selected for remapping

Frequently Asked Questions

- How to handle pointer arithmetic?
  - Indexing into a record or indexing into an array
  - Often possible for the compiler to adjust computation
- What to do about precompiled libraries?
  - Blocked operations such as MEMCPY or QSORT
  - May require recompilation or field-level alternative implementation
- What about profile sensitivity?
  - Incomplete and competing memory access patterns
  - Generalized matching problem is NP-complete
  - Finer-level analysis of NAP
- How does data remapping compare to previously published efforts?

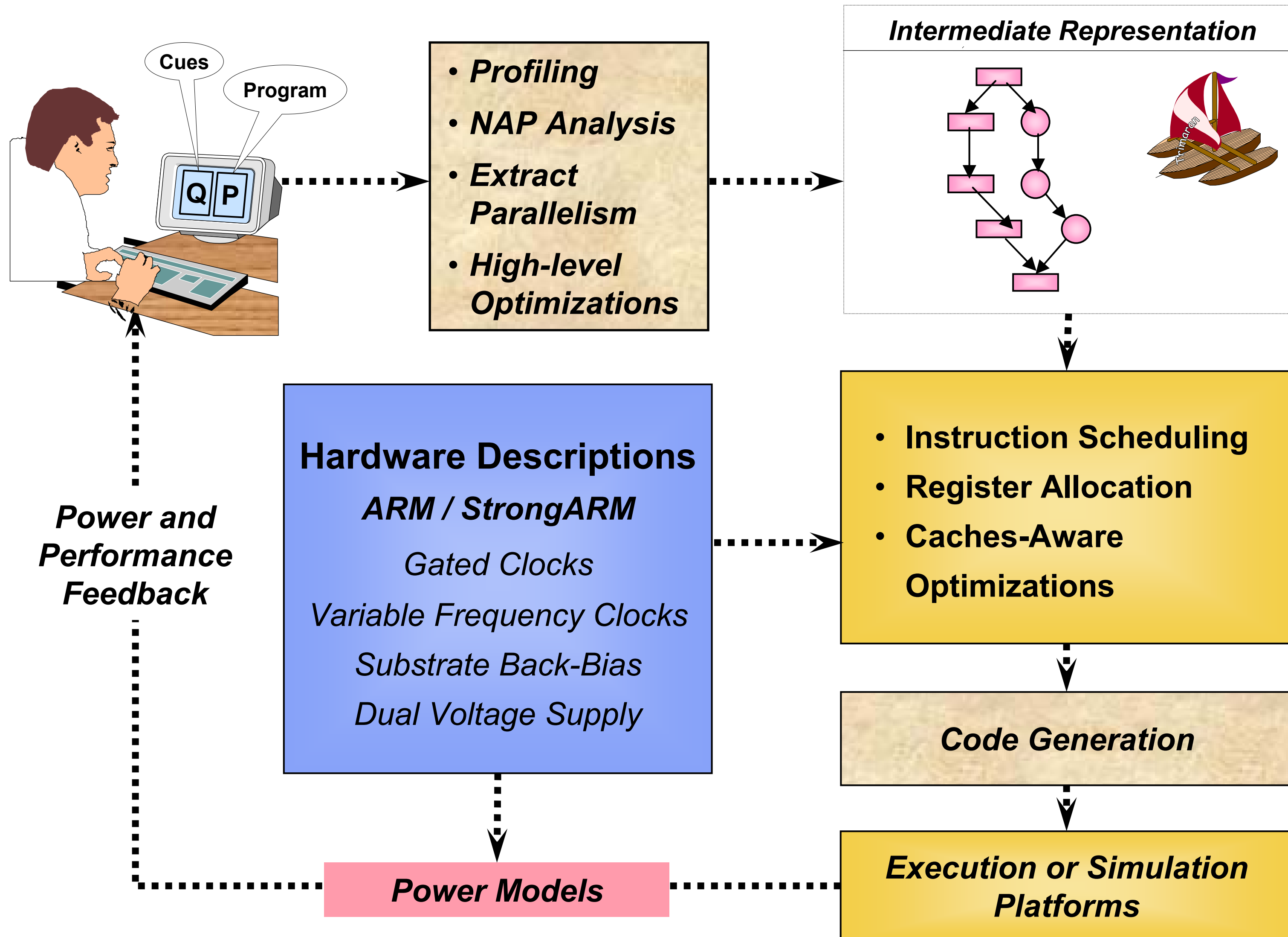
Summary of Data Reorganization Strategies

	Access Function Overhead	Object Allocation Overhead	Dynamic Object Relocation	Requires Programmer Assistance
Field Reordering	None	N/A	N/A	No
Object Co-location	None	Moderate various heuristics proposed	Yes	Yes
Data Remapping	Negligible 3-5% increase in dynamic instruction count	None	No	No

Presentation Outline

- Introduction
- Data Remapping Algorithm
  - Overview
  - Remapping of Global Data Objects
  - Remapping of Heap Data Objects
  - Analysis for Identifying Candidates for Remapping
- Evaluation Framework and Results
  - Design Space Exploration via Data Remapping
- Concluding Remarks

Implementation and Evaluation Framework



Modeling Energy Dissipation of the Caches

- Kamble and Ghose analytical models to measure energy dissipation
  - *International Symposium for Low Power Electronics and Design, August 1997*
  - Bit and word lines, input and output lines, sense amplifiers
  - Estimation within 2% of dissipation for conventional caches
    - **About 30% error for some complex caches**
    - **These organizations are not considered for the experiment**
  - Leakage current and I/O pads dissipation is not accounted for
  - Require run-time statistics, cache organization
    - **Total cache accesses**
    - **Hit/miss counts for read and write accesses**
    - **The number of write-backs**
  - Also require various capacitance values
    - **Bit and word lines**
    - **Gate and drain of a 6-transistor SRAM cell**
    - **Input and output lines**

Benchmarks

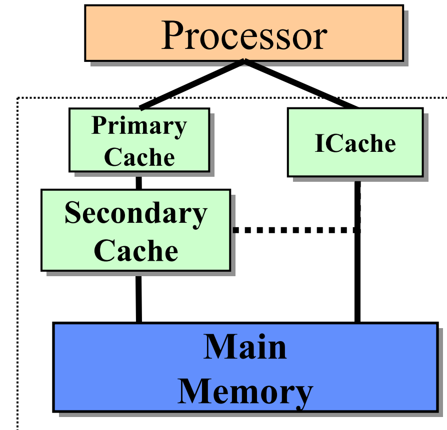
- Benchmarks from SPEC, Olden and DIS suite
- 8 different bus and cache organizations

Benchmark	SUITE	Memory Footprint	Main Data Structure
164.GZIP	SPECINT00	Small	Dynamic array of records
179.ART	SPECFP00	Small	Dynamic array of records
FIELD	DIS	Small	Static array of records
HEALTH	OLDEN	41 / 123 Mb	Linked list
PERIMETER	OLDEN	146 / 147 Mb	Quad tree
TREEADD	OLDEN	64 / 512 Mb	Binary tree
TSP	OLDEN	40 / 320 Mb	Quad tree

Data Remapping as a Compiler Optimization Impact on Performance and Energy

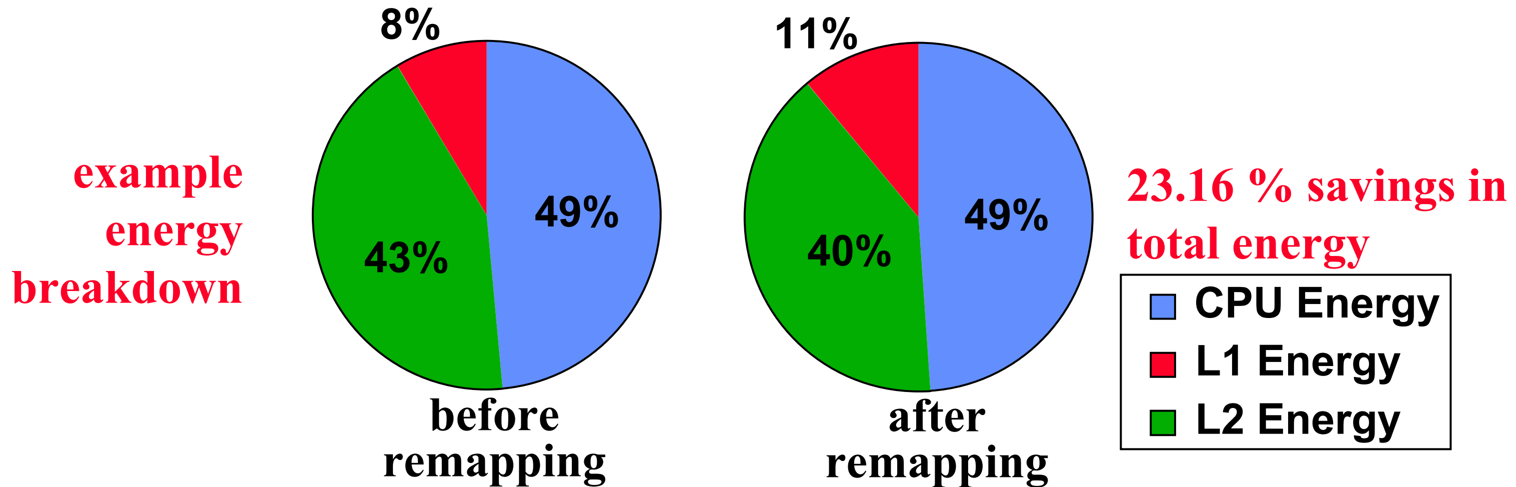
- If we consider data remapping as a compiler optimization for a fixed cache configuration, what are the performance implications?

	% performance improvement
Worst	-0.02
Best	69.23
Average	20.07



An ARM like processor with 32 Kb L1 and 1 Mb L2

Design Space Optimization Via Data Remapping



- For a fixed cache configuration
  - More primary cache hits increase energy dissipation
  - Less secondary cache accesses
    - **Significant reduction in bus traffic and secondary cache accesses dramatically offset first level energy increases**
  - Hence we can achieve the same performance goal using smaller caches
    - **Less cache entries → less energy**

Design Space Optimization Via Data Remapping

- If *we halve the sizes of the primary and secondary caches*, we can *maintain* performance goal using data remapping

	% energy reduction
Worst	38.046
Best	84.654
Average	57.141

- Performance goal satisfied using smaller primary cache size (16 Kb vs. 32 Kb) and smaller secondary cache (512 Kb vs. 1024 Kb)
- 61% saving in $ cost for the cache subsystem

Presentation Outline

- Introduction
- Data Remapping Algorithm
  - Overview
  - Remapping of Global Data Objects
  - Remapping of Heap Data Objects
  - Analysis for Identifying Candidates for Remapping
- Evaluation Framework and Results
  - Design Space Exploration via Data Remapping
- Concluding Remarks

Summary and Remarks

- *Data remapping* is a novel data reorganization algorithm
- Compiler can play a role in design space exploration of memory systems
  - Combined remapping and loop transformations

Data remapping for design space exploration of embedded cache systems. Rabbah R.M. and Palem K.V. ***To appear in the ACM Transactions on Embedded Computing Systems 2002.***

Thank You.