ABSTRACT

Data access latency, a limiting factor in the performance of chip multiprocessors, grows significantly with the number of cores in non-uniform cache architectures with distributed cache banks. To mitigate this effect, it is necessary to leverage the data access locality and choose an optimum data placement. Achieving this is especially challenging when other constraints such as cache capacity, coherence messages and runtime overhead need to be considered. This paper presents a compiler-based approach used for analyzing data access behavior in multi-threaded applications. The proposed experimental compiler framework employs novel compilation techniques to discover and represent multi-threaded memory access patterns (MMAPs). At run time, symbolic MMAPs are resolved and used by a partitioning algorithm to choose a partition of allocated memory blocks among the forked threads in the analyzed application. This partition is used to enforce data ownership by associating the data with the core that executes the thread owning the data. We demonstrate how this information can be used in an experimental architecture to accelerate applications. In particular, our compiler assisted approach shows a 20% speedup over shared caching and 5% speedup over the closest runtime approximation, "first touch".

Categories and Subject Descriptors
B.3.0 [Memory Structures]: General; D.3.4 [Programming Languages]: Processors—Compilers, Optimization

1. INTRODUCTION

As the number of cores in proposed future CMPs (Chip Multi-Processors) continues to increase, the predominant baseline organization to allow scaling converge towards a tiled processor core and cache architecture in which memory accesses are interleaved among distributed cache banks. Although this organization balances cache memory accesses and effectively utilizes cache capacity, it causes a significant amount of accesses by a core to remote tile’s level-2 (L2) cache banks. A private cache organization remedies this by absorbing every access to the tile’s local L2 cache bank, but creates cache coherence problems and potentially wastes cache capacity since cache blocks are often replicated in multiple banks. It is desirable to distribute most data accessed in such a way that locality of access is preserved, remote accesses are reduced and cache capacities are efficiently utilized.

Many multi-threaded applications, such as those from the SPLASH-2 [2] and PARSEC [4], exhibit regular data access patterns that can be exploited at compile time. Among the parallel benchmarks from SPLASH-2, 10 out of 12 are data-parallel and among the 13 benchmarks from PARSEC, 9 are data-parallel. In these benchmarks, accesses to arrays (memory accesses) are usually split by a logical thread identifier (ID) and show a high degree of regularity. Even for irregular parallel applications, a recent study has shown that significant memory access patterns exist [17]. These patterns imply how multiple threads parallelize the data accesses and computations. Detecting these patterns is critical to gain an insight of how data should be distributed among multiple CMP tiles to promote locality of access implied in the program.

Several run-time oriented caching policies and schemes [9,
have been proposed to distinguish certain characteristics of data access patterns in order to achieve better performance. In particular, these schemes distinguish between data that is accessed in a private manner by one core and data that is shared by two or more cores and adjust the caching methodology accordingly. They cleverly use the page table and TLB (translation look-aside buffer) to find data during address translation rather than requiring a directory to maintain coherence. Unfortunately, these schemes work at the page granularity rather than a cache-block granularity allowed by a directory. Forcing coherence at the page granularity can result in one data pattern being polluted by another and data that is actually privately accessed can appear to be shared. We conducted experiments to determine the impact of classification granularity of 4K pages and sub-pages and our results indicate that a granularity of less than a 4K page is desirable. For example, using a classification granularity of a 4K page, 8% of the data accesses for the OCEAN benchmark of SPLASH-2 are private, while if classification is possible at 1K sub-pages, 33% of the accesses are private. For the BARNES benchmark 0.2% of accesses are private at the 4K page level but 22% are private for 1K sub-pages. For larger page sizes such as 16K, 64K, and up, this problem is amplified.

We propose to use the compiler to determine the data partitioning and to enforce this partitioning in the runtime system to determine data placement in order to promote locality of access. For the compiler to detect the memory access pattern and find the data partition, the data allocation, definition and usage must be detected and analyzed. Among all the data used by a program, the stack and text sections usually exhibit predictable patterns. However, most applications predominantly rely on dynamically allocated memory to store the data the application uses for its computations, which is the most difficult type of data for the compiler to analyze. Thus, our focus is memory allocations and the corresponding usage of the dynamically allocated memory. We focus on the widely used memory allocator malloc(), which is representative of most dynamic memory allocation routines.

In this paper we describe a compilation methodology for multi-threaded applications that uses analysis to discover the data access patterns and determines the most appropriate partitioning of the data for the application. A novel representation is introduced to describe the memory access partition, which is used to guide the data placement at run time. We call this assigning data ownership to a particular processor core. As described in Section 4, data ownership can be used by a cache organization scheme to enforce that data is assigned to the L2 cache in the same tile as the processor executing the thread that owns the data.

2. BACKGROUND AND RELATED WORK

To maximize utilization of multi-core system, not only the resources such as cache capacity and interconnections should be efficiently used, but also multiple cores should partition the work and data in such a way that locality is highly preserved and the volume of inter-processor messages is minimized. This requires a study of data accesses behaviors among multiple processors.

Some early attempts have been made to analyze data accesses in a nested loop and find a data or loop partitioning among parallel threads. Ramanujam and Sadayappan [20] used matrix representation to formulate a data partitioning applied to multiprocessors without caches. Ju and Dietz [12]'s efforts focused on finding a data layout (row or column major) for memory block in a uniform memory access (UMA).

In Gupta’s PARADIGM [8], data usage in parallelizable FORTRAN 77 and HPF (high performance fortran) code was analyzed and partitioned across machines with distributed memories. Optimized communication operations were generated for the parallelized code. Another similar effort has been made by Kremer [13], in which an automatic data layout specification was produced using 0-1 integer programming technology.

In Wilson and Lam’s work [24], data dependence was studied using a linear algebra approach. They distributed loop iterations and thus data that carry no dependence among multiple processors. Barua [3] developed a heuristic method to deal with data partitioning in a way that avoids NP-complete linear programming solutions.

More recently, Paek and Padua [19] presented an advanced compiler framework for non-coherence cache machine based on array access regions. They reduced the data exchange by finding a suitable iteration/data distribution. Chu et al. proposed a profiling based scheme [6] to determine a fine-grain data access partitioning. Another technique of Chu and Malhike [5] describes an approach to partition data objects among multi-cluster processors. In this work, size and usage information of each object is collected using data flow analyses. Objects associated with the same computation are merged together to form a group. Groups of objects are then partitioned for the multi-cluster machine to balance workload and improve performance.

In another attempt [15], a polyhedral model is used to perform localization analysis. The goal of this analysis is to find a data layout transformation to promote locality of access, which, as we have previously noted, would otherwise be destroyed by finely interleaving data among the tiled banks. In essence the work of [15] is attempting to accomplish the same goal as our work but in a different way. They restructure the program to change the array indexing such that the shared cache policy is retained, but the addresses accessed by a tile are mapped to that tile. Instead we communicate the data’s owner detected by the compiler to the runtime system and use a cache policy to enforce the compiler’s partitioning.

While the above methods largely focus on compile-time data layout, several schemes have also been proposed to capture data access behavior at run-time and improve cache performance through data placement. Jin and Cho designed SOS [10] to place data among L2 caches based on data access patterns. Another attempt [21] has been made to reduce access latency by assigning pages to cache banks local to the core which touch that page first (first touch). To balance cache pressure, the number of pages assigned to a particular tile is tracked and used to make page migration decisions. Reactive NUCA (R-NUCA) [9] classifies a data page as private upon the initial access. If another processor subsequently requests the page, that page is reclassified as shared and moved to a shared cache bank cluster to reduce average access latency by multiple processors.

The uniqueness of our work is that we both analyze multi-threaded source code in the compiler and tightly couple the analyzed result with cache performance through interaction.
between the compiler and run-time system. To the best of our knowledge, few attempts have been made in this direction. The goal of our analyses is to expose the data access relationship among multiple threads and to determine the data partitioning implied by the multi-threaded applications. By applying the data partition discovered by the compiler across multiple cache banks, each core is associated with the data on which it will likely operate most.

3. PRELIMINARIES

Our compilation methodology requires the use of several known compiler techniques that are well understood. In this section we mention a few of these optimizing passes and their applications in order to provide a background for the compiler methodology described in the following sections. Due to the complexity of analyzing multi-threaded code, we not only develop new optimizing and analyzing passes, but rather utilize existing packages commonly contained in compiler infrastructures such as SUIF [24].

Figure 1 shows a paradigm for our experimental compiler framework. Multi-threaded applications with optional user directives (Section 4.2.1) are fed into the front end and converted to the compiler’s intermediate representation. The interprocedural analysis (IPA) is then applied to handle global data and propagate the information from one procedure to another. The call graph generated in this pass can be used to keep track of the parameter passing when necessary. For each procedure, control flow graphs (CFGs) are generated to represent the control paths in the program. Each node in the graph is a basic block in which there is no branch instructions. The sequence of instructions in a basic block represents the data flow in that basic block. IPA and control data flow analysis form a basis for other optimizations.

One of the control and data flow analyses that is necessary for data partitioning is pointer analysis to track the behavior of dynamically allocated memory using malloc(). The starting addresses of memory blocks returned by malloc() are usually assigned directly to pointers. Other pointers can point to the same memory block through pointer assignments. To perform pointer analysis and address the memory ambiguity problem, we define:

**Definition 1.** A pure pointer is a pointer that have been directly assigned the return value of malloc().

**Definition 2.** A derived pointer is a pointer that is derived from a pure pointer either directly or indirectly based on a pointer assignment.

**Definition 3.** A reference list for an allocated memory block is a collection of pointers that can be used to refer to that block.

Pointers, together with some offsets, can be passed as function arguments, or used as basic pointers to refer to a particular memory location. Memory disambiguation is required to identify which memory block is referenced by a particular array access. Using pointer analyses, array accesses can be resolved back to their pure pointers, which are used in the remaining compiler analysis. A detailed example of using reference list is shown in Section 4.3.

Other auxiliary analyses include constant propagation, expression folding, loop transformations, symbolic analysis, etc. These will be illustrated by an example in Section 4.3.

However, a major focus of this paper is the development of a new compiler technique called multi-threaded memory access analyses, described in next section.

4. MULTI-THREADED MEMORY ACCESS ANALYSIS

Traditional data access analyses such as reuse, dependence and locality analysis [1, 18] focus on affine array subscript patterns in loop nests of a single threaded application. The primary concern of these techniques is to find the relationships of memory locations accessed by different loop iterations. In contrast, our multi-threaded analysis approach detects the memory access patterns in a parallel programming context. The pattern information is then used to determine the data partitioning implied by the application.

There are a number of efforts made to represent the pattern of the memory accesses within the program. Tu and Padua [23] approximated memory access using a triplet notation. Li and Yew proposed a reference-list based representation called atom image [14]. A more precise notation is convex regions [22, 7], which express the geometrical shape of array accesses. Paek introduced regions that use span-stride pairs with an abstract access form to represent memory accesses within a program phase such as a nested loop [18]. For our purposes, we select the region method to achieve a balance between efficiency and accuracy.

This section begins by first introducing traditional memory access regions for dealing with affine array subscript function in single threaded programs [18, 19]. The region theory forms the basis of our analysis to describe affine array access patterns in an execution phase of the application.
(For our purposes we define an execution phase as a loop nest in the code). We then extend the region concept to a MMAP (Multi-threaded Memory Access Pattern) for multi-threaded applications. Finally, multiple phases are considered to create a data partition.

4.1 Array Access Regions

Within a loop nest for a single threaded program, a reference to an array A can be generally represented as $A[f(L)]$, where $f(L)$ is the subscript function defined on a set of loop indices $L = i_1, ..., i_m$. The span of $f(L)$ resulting from $i_k$ is the maximum distance traversed by varying only $i_k$ from its lower bound $l_k$ to its upper bound $u_k$ [18, 19]:

$$span_{i_k} = |f(i_1, ..., i_{k-1}, u_k, i_{k+1}, ..., i_m) - f(i_1, ..., i_{k-1}, l_k, i_{k+1}, ..., i_m)|$$

Similarly, the stride is defined as the minimum distance across memory by changing only $i_k$ by its step $s_k$:

$$stride_{i_k} = |f(i_1, ..., i_{k-1}, i_k + s_k, i_{k+1}, ..., i_m) - f(i_1, ..., i_{k-1}, i_k, i_{k+1}, ..., i_m)|$$

Thus, an array access region can be described by the following form, where $O$ denotes the starting offset:

$$R = (A_{span_{i_1}}^{stride_{i_1}}, ..., A_{span_{i_m}}^{stride_{i_m}} + O)$$

The original access region theory has been developed to simplify data access analysis such as array dependence and privatization analysis. It also offers a set of manipulations on regions such as region coalescing and intersection to allow privatization analysis. It also offers a set of manipulations that determine the memory access pattern of the threads. We call these code structures Thread Identifying (TI) Structures.

4.2 Multi-threaded Array Analysis

The original array access region cannot represent data accesses in multi-threaded applications because the same array access in a loop nest results in multiple instances of accesses by different threads. Additionally, the bounds of the loop nest and array subscripts may be different for each thread. Thus, each thread requires a corresponding number of unique regions to represent its data accesses. To address these issues, we developed the following techniques to extend the region concept.

4.2.1 Thread-Identifying Variables

Our study of multi-threaded benchmarks shows that a large portion of data-parallel applications tend to exhibit regularity in their data access patterns. In these benchmarks, each thread derives its own set of local variables with thread dependent values to specify which regions of each array to access. We call these local variables Thread Identifying (TI) variables. The compiler needs to identify these TI variables to determine how each thread accesses different portions of memory.

One of the commonly used methods for specifying TI variables is to pass different values to parallel threads as function arguments. As shown in Figure 2, the fourth argument of `pthread_create()` passes the addresses from `my_arg[0]` to `my_arg[num_threads]` from within a for loop. The passed addresses serve as local variables in forked function `SlaveProcedure` where each instance has a local variable `my` with a unique value. Thus, `my` is a TI variable that can be detected by the compiler. Another common way to specify TI variables in multi-threaded applications is illustrated in Figure 3. Multiple threads try to access and modify a global variable under the protection of a mutex lock. This type of code is much more difficult for a compiler to analyze. Thus, if the code uses this or any other methods to specify TI variables we require the user to include a directive. Thus from Figure 3, `#pragma TIV pid` specifies pid as a TI variable.

```c
for(i=0; i<num_threads; i++) {
    my_arg[i] = i;
    pthread_create(&p_ththreads[i], &attr,
                   SlaveProcedure, (void*)&my_arg[i]);
}
SlaveProcedure(void *my)
......
```

4.2.2 Thread-Identifying Structures

For the compiler to correctly discover the memory accesses of each thread it is necessary to interpret how the code structures use the TI variables. The study of multi-threaded code from a variety of program domains including scientific computing, multimedia, image processing and financial processing reveals that there are particular programming structures, such as loops and conditionals, that determine the memory access pattern of the threads. We call these code structures Thread Identifying (TI) Structures.

TI structures place constraints between allocated blocks of memory and the threads who can access them. Discovering all possible types of TI structures is not possible. However, we can identify important TI structures that are widely used. We summarize these TI structures into three categories as follows:

**TI Variables in Loop Bounds:** One method to partition data access during a particular phase of the application is to use a function of TI variables as loop bounds within the loop nest. As the different loop iterations typically access different array indices, in this method, TI variables place a constraint on which iterations of the nested loop can be executed by each thread. As a result, different threads access different, and often independent, array indices.

TI variables in loop bounds usually imply block access pattern or nested access pattern, as shown in Figures 4 and 5,
respectively. When both lower and upper bounds are functions of TI variables this often implies a block pattern. For example in Figure 4, the loop bounds myfirst and mylast are functions of TI variables and a blocksize. The access pattern is shown for nprocs=4 and contains four blocks of blocksize. As mylast is the only function of a TI variable in Figure 5, the pattern is nested. Thread with pid=0 accesses a single block of blocksize, which is nested inside of partition one, which in turn is nested in partition two, etc.

With loop nests, the TI structures can be more complex. Figure 6 shows a grid-like access pattern and its corresponding TI structure where bsm and bsn are the block size partitioned along two dimensions.

\[
\begin{align*}
bs &= \text{datasize}/\text{nprocs}; \\
\text{myfirst} &= \text{bs} \times \text{pid}; \\
\text{mylast} &= \text{bs} \times (\text{pid} + 1); \\
&\text{for}(j=\text{myfirst}; j<\text{mylast}; j++) \\
&A[j]......
\end{align*}
\]

Figure 4: Example TI structure and the corresponding block pattern (nprocs = 4)

\[
\begin{align*}
bs &= \text{datasize}/\text{nprocs}; \\
\text{mylast} &= \text{bs} \times (\text{pid} + 1); \\
&\text{for}(j=\text{myfirst}; j<\text{mylast}; j++) \\
&A[j]......
\end{align*}
\]

Figure 5: Example TI structure and the corresponding nested pattern (nprocs = 4)

**TI Variables in Array Subscripts:** Another way to distribute array accesses among multiple processors is to have TI variables serve as array subscripts. The statement \(A[\text{pid}+j] = x\) is an example of a TI structure of this style. For a TI structure of this kind, knowing the value of the function of TI variables used in the array subscripts is essential to understand how the data is accessed. For four processors, the access pattern from this TI structure is similar to the one shown in Figure 4.

**TI Variables in If Statements:** Another common TI structure includes a conditional statement that impacts the array access. In this case the TI structures are most difficult to analyze because the flexibility in the forms that can be employed in this manner is high. For example, we cannot determine a pattern from the TI structure: if (foo(i) == pid) unless we know the structure of function foo. Our method for analyzing TI structures from this category is to select regular structures that use conditionals, which are extensively used in many programs. For example, the TI structure if((i%4) == pid) in Figure 7 implies an interleaved access pattern as shown in that figure.

\[
\begin{align*}
&\text{ps} = \sqrt{\text{nprocs}}; \\
&\text{bsm} = \text{m}/\text{ps}; \text{bsn} = \text{n}/\text{ps}; \\
&\text{for}(i=(\text{pid}/\text{ps}) \times \text{bsm}; i<(\text{pid}/\text{ps}+1) \times \text{bsm}; i++) \\
&\text{for}(j=(\text{pid}/\text{ps}) \times \text{bsn}; j<(\text{pid}/\text{ps}+1) \times \text{bsn}; j++) \\
&A[i][j]......
\end{align*}
\]

Figure 6: Example TI structure and the corresponding grid pattern (A is a m × n array and nprocs = 4)

\[
\begin{align*}
&\text{for}(i=0; i<16; i++) \\
&\text{if}(i\%4 == \text{pid}) \\
&A[i]......
\end{align*}
\]

Figure 7: Example TI structure and corresponding interleaved pattern (0 ≤ pid ≤ 3)

### 4.2.3 Multi-threaded Memory Access Patterns

The purpose of TI structure analysis is to discover patterns of memory access for each thread in the system. Thus, we define the MMAP in a formal representation to describe the way multiple threads access blocks of data. Given a phase of parallel code for \(n\) threads, a list of \(n\) regions can be created for each array access by replacing TI variables with the actual values for that thread. Then we can define the MMAP for an array access in a parallel program phase as:

\[
\text{MMAP} = \{R(0), ..., R(n-1)\}
\]

In the above equation, region \(R(x)\) follows the notation in Eq. (3) and corresponds to the thread with the id \(x\). Access weight is also defined for each region in the MMAP to reflect the number of times the thread accesses its region. It can be calculated based on the information from loops...
associated with this region. Assume that an array access appears \(N\) times in a \(m\)-deep nested loop, and the lower bounds, upper bounds and steps of loop indices for thread \(x\) are \(l_1(x), \ldots, l_m(x), u_1(x), \ldots, u_m(x)\) and \(s_1(x), \ldots, s_m(x)\), respectively. Hence, the formula to calculate access weight for \(R(x)\) is:

\[
W(R(x)) = N \cdot \prod_{k=1}^{m} \frac{u_k(x) - l_k(x)}{s_k(x)}
\]  

(5)

For further analysis convenience, some properties for MMAP are defined as follows:

Definition 4. A MMAP, \(M\), is access uniform when \(W(R(0)) = W(R(1)) = \ldots = W(R(n-1))\).

Definition 5. A MMAP, \(M\), is size uniform when \(|R(0)| = \ldots = |R(n-1)|\) where \(|R(x)|\) is the number of elements in \(R(x)\).

The patterns shown in Figures 4, 6, and 7 are all size and access uniform MMAPs.

Definition 6. A MMAP, \(M\), is non-overlapping when \(\forall R(x), R(y) \in M, R(x) \cap R(y) = \emptyset\). Otherwise, the MMAP is overlapping.

In addition to the properties defined above, some other attributes can be explored such as the shape of a MMAP. For example, the MMAP that meets the condition \(R(1) \subset R(2) \subset \ldots R(n)\) has a nested shape, shown in Figure 5. Other shapes could be block (see Figure 4) or interleaved (see Figure 7), etc.

4.3 MMAP generation

Although MMAPs are primarily related to array accesses and TI-Structures, there are other programming components affecting the actual access patterns. For example, the accesses might appear in a conditional branch. Also, useful information such as loop bounds and array indices could be variables rather than constants at compile time. In this case, the regions and MMAPs are constructed using symbolic expressions, which are resolved at run time. To deal with relatively general situations and perform analyses in a systematic way, we adopt control flow and symbolic analysis, as used in [5] and [11], respectively. Our analysis approach is illustrated by an example shown in Figure 8.

The MMAP generation phase consists of both forward and backward passes. In the forward pass, the CFG is traversed from top to bottom. As the CFG is being traversed, malloc() routines and pointer assignments are detected. Each static malloc() call site is given a unique id and a reference list. Initially, there is only the return value of the malloc() (i.e. the pure pointer) in the reference list. For example, as the analysis reaches statement 2 in BB0 (\(x = (double*)malloc(bs)\)), it creates the reference list \(\{(malloc2, x)\}\) where 2 indicates the static malloc id and \(x\) is the pure pointer. Subsequent pointer assignments update the reference list as the analysis pass continues forward through the CFG. Thus, the assignment from statement three \((A = x)\) adds \(A\) to the reference list resulting in \(\{(malloc2, x, A)\}\). This pointer analysis is used to associate array accesses with the corresponding memory allocations, to which the MMAPs are eventually attached. In addition to pointer analyses, optimizations such as constant propagation and TI-variable recognition are also performed and used to annotate the corresponding nodes in CFG. However, for clarity we only show the results of the pointer analysis on the right-hand-side of Figure 8(C).

Whenever an array access is encountered, the backward pass is initiated at the current node in the CFG and traverses the nodes in reverse to search or calculate the enclosing TI-structures, weight, conditions, and memory allocation for this particular array access. Information found is then used in the construction of regions and MMAPs for the array access. Under most cases where input parameters are unknown, regions or MMAPs are calculated symbolically from the expressions available in the analyzing context rather than represented using constants. If there are conditional branches encountered during this process, the eventually generated MMAPs are appended with that condition. The MMAPs, their weights and the condition are thus denoted as a tuple: \(\{M, W, C\}\).

Figure 8(B) shows how the backward analysis is performed to produce the results for each stage of the analysis. For example, as the array access \(A[j]\) is detected at \(BB3\), the backward analysis generates region and weight (shown in the second cloud from bottom) using the loop bounds information. In the region expression \(A_{ML-1, MF} + MF\), 1 is the stride and \(ML - 1 - MF\) is the span. \(MF\) is the offset of the region. They are calculated using Eq. (1) and Eq. (2) based on loop and array access subscript information.

\[2 + (ML - MF)\] is the weight for the array access. As the backwards traversal continues, the condition (\(bs%bp == 0\)) is appended at \(BB2\). At \(BB1\), symbolic terms \(MF\) and \(ML\) are updated and represented in terms of \(bs\), \(pid\) and \(np\) (we use \(\alpha\) and \(\beta\) for short). Finally, the cloud on the top shows the generated MMAPs, each with several regions in it. The number of regions in each MMAP is equivalent to the number of threads \(np\).

Note that if there are different array accesses patterns to the same array in one phase, multiple MMAPs need to be attached to the corresponding malloc(). After constructing the MMAP(s) for the particular array access, forward analysis resumes to reach and analyze the next array access.

4.4 Data Partitioning

One particular goal of the compiler analysis is to determine the data partition for each array and to use this to enforce data ownership. This has been motivated by the fact that for array accesses in most data-parallel benchmarks, each thread has a set of data on which the thread operates most. These sets of data imply a partition of the accessed array, which in many cases the compiler can discover. Thus, we describe a data partition based on regions and MMAPs for the array \(A\) as follows:

Definition 7. For a program with \(n\) threads numbered from 0 to \(n - 1\), a partition \(P\) is a set of non-overlapping regions: \(\{R_0(n), \ldots, R_{n-1}(n-1)\}\), as defined in Definition 6. Processor \(x\) is said to be the owner for region \(R_x(x)\) in the partition.

The above definition ensures that each element of array \(A\) appears in exactly one region, and hence, will be owned by one thread. This condition can be examined by region intersection manipulation [18].

In order to determine a good partitioning for an array,
double *x; double *A; double *B;  

Figure 8: MMAPs Generation Flowgraph.

all phases that contain accesses to this array in the program should be examined. This can be done by applying the MMAP representation to every phase and comparing these MMAPs to find the partition. To extend the MMAP representation across multiple program phases we add an additional subscript to denote the program phase number.

Thus, to determine a data partition for the entire application we select the dominating MMAP described as follows:

**Definition 8.** Given a set of MMAPs $M_0, ..., M_{k-1}$ corresponding to $k$ program phases, a MMAP $M_i$ is said to be the dominating MMAP if it has the largest total access weight: $\sum_{x=1}^{n} W(R_i(x))$ among all $k$ program phases.

To determine the dominating MMAP, weights of all MMAPs associated with the same memory block are compared. The one with largest weight is chosen as the dominating MMAP.

From the dominating MMAP, $M_D$, the partition can be created. First if by Definition (6) $M_D$ is non-overlapping, then $M_D$ is a partition. If $M_D$ is overlapping, non-overlapping portions of $M_D$ are constructed by region manipulation such as intersection and subtraction. The weight of the non-overlapping portions is then compared with the dominating non-overlapping MMAP. The one with larger weight becomes the partition. Based on our observation, almost all data-parallel benchmarks we studied exhibit non-overlapping dominating MMAPs. This reduces the probability of performing expensive region manipulations.

**Partitioning example:** Consider the parallel program for 4 processors shown in Figure 9. The compiler detects that the program has three nested loops or phases. From the analysis of each TI structure, we can determine the multi-threaded access patterns for each phase as follows: The first phase has an interleaved uniform access pattern, the second is block uniform and the third is a nested access pattern. According to relevant definitions, the second program phase will produce the dominating MMAP because it has largest access weight $(4 * (pid + 1) - 4 * pid) * 4 * 4 * 64 * 2 = 8192$.

To generate regions in the dominating MMAP $M_2$, thread-
specific values are used to replace the TI variables within the TI structure. In this example, thread ID pid ranges from 0 to 3, so the region list for phase 2 would be \{ R_2(0), R_2(1), R_2(2), R_2(3) \}. First we generate R_2(0) for thread 0 by replacing pid with 0. The array subscript 4i + j and k can be linearized to (4i + j) * σ + k [18] where σ is the size of second dimension of array A. In this example σ = 64. Then the array subscript becomes (4i + j) * 64 + k = 256i + 64j + k. Using Eq. (1) and Eq. (2), the span of 256i + 64j + k by changing i from its upper bound 3 to lower bound 0 is (256 * 3 + 64j + k) − (256 * 0 + 64j + k) = 256 * (3 − 0) = 768, and the corresponding stride is 256 * (1 − 0) = 256. The second span-stride pair, resulting from analyzing variable j, is a span of 64 * (3 − 0) = 192 and a stride of 64 * (1 − 0) = 64. The third pair from k is 1 * (63 − 0) = 63 and 1. The starting offset of the R_2(0) is 256 * 0 + 64 * 0 + 0 = 0. So the access region descriptor for R_2(0) can be expressed as A_{256,64,1}.

The access weight can be easily calculated using Eq. (5): 4 * 4 * 64 * 2 = 2048. Regions for other threads can be built in a similarly way: R_2(1): A_{768,192,63} + 1024; R_2(2): A_{256,64,1} + 2048; R_2(3): A_{768,192,63} + 3072.

Since these regions do not intersect, the MMAP they form is non-overlapping. Thus, the data partition of array A can be represented using this MMAP: P = \{ R_2(0), R_2(1), R_2(2), R_2(3) \}.

4.5 Granularity of Data Ownership

For each thread, the partition from the above analyses exposes the data ownership information specifying which portion of memory, in terms of region, it accesses frequently. Depending on situations, regions in the partitions provide the flexibility to represent access patterns at as fine a grain as the array element level. At run-time, however, data is usually organized at either the cache-line or page granularity. A fine granularity such as the cache-line provides the most accurate ownership information at the cost of expensive computation and storage overhead. Conversely, large granularities reduces overhead but increases the danger of a data block owned by one processor being polluted by others (see Section 1). Another option is to divide the page into sub-pages and assign each sub-page an owner based on the partition from the compiler analyses. Regardless of the granularity used, the requested addresses need to be compared with the address ranges of regions in the partition to determine the ownership. In our work, we utilize the partition information at sub-page granularity as described in the following section.

5. CACHING SCHEME AND ARCHITECTURE SUPPORT

R-NUCA and first touch have recently been proposed as alternatives to distributed shared S-NUCA cache. In this section, we describe these cache policies and compare them with our compiler based approach. Additionally, we illustrate how we use the data partition information from compiler analysis to guide data placement.

5.1 Relevant run-time schemes

As introduced in Section 2, R-NUCA’s goal is to reduce data access latency based on page classification. To classify a page, the operating system (OS) must keep track of the requesters and within each page maintain an additional field which is used to store the core ID of the previous requester for that page. Data pages are initially classified as private. Private pages are cached in the local tile to the core that “owns” that page. A data page is reclassified to be shared when the OS detects a requester with a different core ID from the one that is recorded. The shared page is then stored in the standard distributed shared location based on the memory address. Naturally, there is an overhead for evicting the appropriate lines from the local tile and either migrating the data or reloading the data from main memory. However, like cold start misses, this penalty would only occur once. R-NUCA also proposes a technique to replicate instruction pages across clusters of cores to be shared by neighboring cores within a cluster. This technique is orthogonal to the data classification and is compatible with the other caching schemes including our own.

Another relevant scheme is first touch, as briefly described in Section 2. It is similar to private caching except that it does not allow replication. It preserves the locality and reduces remote accesses by placing the data in the local cache bank of the core that touches the data first. To adapt to dynamic access behavior and enhance performance, data is allowed to migrate to another processor’s local cache bank if that processor accesses the data more frequently than the current owner. Counters are used to keep track of the access to each cache line by different processors. First touch can be implemented at different granularities. If implemented at cache block granularity, a particular cache block can be placed in any location within the aggregate sets of all cache banks. In such an implementation, a directory composed of a tag and a tile number is needed to keep track of the location of each cache block. Any requester needs to consult the directory for the location of the requested block.

In contrast to R-NUCA and first touch, our proposed caching scheme utilizes compiler-assisted partitioning (CAP) to guide the data distribution. For the data that can not be partitioned by the compiler analysis (no ownership information), we use the first touch policy at the cache-line granularity as described above.

5.2 System Support

In order to leverage the partition information from compiler analyses, a facility is required to communicate the information from the compiler to the run-time system. Specifically, for CAP we must pass information about memory regions and their owners, which represents the partition for a particular block of memory. Our mechanism requires the usage of memory allocation hooks, which are used to modify the behavior of malloc(). The mechanism is illustrated in Figure 10. Each time the malloc() function is called, a customized hook function is invoked to record the return value, the size of the allocated memory block and the partition information into a memory partition table (MP-Table in Figure 10). The hook function for the memory allocator can be specified through the hook variable __malloc_hook defined in malloc.h. The hook driver is responsible for processing the partition information and determining the owner for pages/sub-pages in the page table. A page/sub-page that cannot be classified by the compiler is not added to the MP-Table and the runtime caching policy (e.g. first touch) is applied during these accesses.

In our architecture, a page is logically partitioned into several sub-pages. Sub-page granularity was selected in order
to be sufficiently fine grain to avoid data pollution among multiple owners yet coarse grain enough to reduce storage and owner calculation overhead. Our study (Section 6.4) shows that a number of sub-pages from 2 to 8 yields the best results. To store the owner information for \( p \) cores, each sub-page is associated with \( \log_2 p \) bits indicating its owner ID. For the 16 core architecture used in our evaluation, 4 bits are needed to store owner information for each of 4 sub-pages, resulting a \( 4^2 \times 16 \) bits for a page table entry (PTE). This incurs only \( 16/64 = 25\% \) overhead on a typical 64-bits PTE structure used in various paging modes.

At memory allocation time (when a memory allocator is being called), the hook driver consults the information stored in the MP-Table. To determine an owner for a sub-page, the hook driver calculates the median address \((S + P/2)\) where \( S \) is the starting address of the sub-page and \( P \) is the sub-page size. If the median address belongs to a particular region, the corresponding sub-page is assigned the same owner of that region. Although this scheme is approximate, it is effective because usually regions owned by different threads in the partition do not interleave at finer granularity.

During virtual address translation using the TLB, the owner of a given address can be retrieved using the information stored in the page-table, thus avoiding the overhead of directory lookup. This is illustrated in Figure 11. As shown in the graph, the physical address needs to be obtained from either the TLB or the page table whenever a processor issues a request. The ownership information stored in the page table entry is then consulted during this process and used to locate or lookup a datum upon a L1 miss.

### 6. EVALUATION

Since our scheme requires data ownership information from compiler analyses, we provide the results to show the compiler’s capability of discovering data partitions. We also study cache miss rate and average memory access latency to show the advantage of using the data ownership information in our proposed cache organization. Finally, overall performance gains are reported for a set of benchmarks.

#### 6.1 Simulation Environment

We use Simics [16] as our simulation environment and implemented the relevant caching schemes (i.e. those discussed in Section 5.1) within it. As previously stated, for R-NUCA we only implement the features relevant to the comparison, namely the classification of data as shared or private at the page level. We do not simulate the effect of replicating application code pages or clustering as described in [9] because this is orthogonal to data page classification and is applicable to many caching schemes including ours. We configured Simics to simulate a tiled CMP consisting of 16 SPARC 2-way in-order processors, each clocked at 2 GHz, running the Solaris 10 operating system, and sharing a 4 GB main memory with 55 ns (110 cycles) access latency. The processors are laid out in a \( 4 \times 4 \) mesh. Each processor has a 32 KB private 4-way L1 cache (divided equally between instruction and data with access latency of 1 cycle).

For first touch and CAP, we use 16-way 1 MB banks (access latency: six cycles) for a total of 16 MB L2 cache. The distributed shared cache and R-NUCA use a similar configuration but are provided with an extra 4M capacity as compensation for the directory and storage overhead incurred by the first touch and CAP schemes (see Section 5). For all of our cache configurations we assume a 2-D mesh-based packet switching network. Our simulations are cycle accurate and use a 16 byte link width that assumes all control and data messages are one flit long. Each input port has four virtual channel buffers and each buffer is capable of storing five flits.

For routers, we simulate a three-stage pipeline: buffer write; virtual channel allocation and switch allocation; and switch and link traversal.

In the simulations of CAP, the analysis results from the compiler are passed into the system using Simics magic instructions and mimic the process of passing the same data using the page table. We selected a classification granularity of four sub-pages (blocks of 1K bytes). To evaluate the various systems we use benchmarks from the SPLASH-2 and PARSEC 2.0. Benchmark input parameters are listed in Table 1. We use both larger and smaller working sets to study the scalability of the proposed partition analysis (see Section 6.2). The larger working set is used for performance simulation.
6.2 Partition Percentage

Our current data ownership analyses are applied to heap memory blocks, which are only part of the overall data space used by programs. In some sense this is conservative since data on the stack is typically private. Data for which ownership cannot be determined by the compiler is placed using the first touch policy (see Section 5). For a particular application, the percentage of the data that can be analyzed limits the optimization potential of CAP. Figure 12 shows the proportions of sub-pages that are assigned ownership by the compiler versus those that compiler did not specify an ownership. The proportion of data with ownership is expected to increase as the input working set grows since benchmarks with larger working sets tend to allocate more memory on the heap. On average, 11.13% of all sub-pages used by the tested benchmarks with smaller working set are assigned ownership by the compiler. By contrast, this percentage increases to more than 60% when using the larger working sets from Table 1, which corroborates our theory. Since the percentage of data with ownership is usually sensitive to only the problem size not the actual number of processors, increasing number of cores does not reduce the overall proportion of data with an owner (although it may reduce the ownership per core). This makes the partition technique scalable to future systems with larger number of processors.

6.3 Impact on Cache Performance

To study the cache performance of the various schemes, we examined the cache miss rate and the average memory access latency. Figure 13 shows the cache miss rate normalized to the distributed shared cache. For most of the tested benchmarks, the miss rates are only nominally different. This result was expected since all the tested caching policies do not allow replication eliminating unnecessary capacity misses that might exist in a scheme like private caching. For some benchmarks, such as LU, WATER, and BARNES, first touch and CAP increase the miss rate slightly. This is due to the slightly smaller cache capacity of these schemes compared to distributed shared and R-NUCA resulting from the storage dedicated to maintaining the directory (see 6.1).

![Figure 12: Percentage of sub-pages with owners for (smaller working set on left)](image)

![Figure 13: Cache miss rate (normalized to dist. shared)](image)

The memory access latency plays an important role in the performance of CMPs (see Figure 14). Distributed shared caches destroy data locality by interleaving data blocks across the cache. Thus, most data is stored in a remote tile from the core that heavily accesses it resulting in the highest amount of remote accesses. R-NUCA also suffers from high access latency similar to distributed shared caches. With R-NUCA, even a single access to a page by a second core forces it to be stored in the shared location. We believe a disproportionate number of predominantly private pages are being classified as shared due to coarse granularity of access and infrequent non-local access.

In contrast, first touch can be more effective because the core which accesses a data block first is likely to access the same block frequently, especially in embarrassingly parallel applications. Additionally, by treating cache blocks independently, there is less false sharing. However, there are several cases where first touch can be fooled, for example if the data is initialized in the main thread but accessed in other working threads. This may mislead the system to store the data in a non-optimal tile and create a load imbalance problem [21]. Our compiler-assisted caching addresses this problem through pattern analyses. This is reflected in Figure 14. Compared with distributed shared, R-NUCA and first touch, CAP reduces the average memory access latency by 28.54%, 26.41% and 12.63%, respectively. Notice that BLACKSCHOLES and SWAPTIONS show a remarkable decrease in memory access latency. Our studies show that these two benchmarks are highly parallelized with extensive amount of local data accesses. The infrequent sharing and high degree of parallelism make these benchmarks perform well on CAP cache. OCEAN also receives noticeable decrease in memory access latency. This conforms to the fact that OCEAN has 2-D nearest-neighbor sharing and exhibits large amount of locality. Some decrease of memory access latency is also observed for LU, WATER and BARNES.

6.4 Overall Performance

From Figure 15 we can see that the distributed shared cache, on average, gives the worst performance due to the effects described in Sections 6.2 and 6.3. As can be observed in the reported results, BLACKSCHOLES achieves about a 40% improvement with CAP. SWAPTIONS, BARNES and OCEAN also show decent speedups. WATER and LU only see a nominal improvement due to large amount of shared
Figure 14: Average memory access latency (normalized to dist. shared)

Figure 15: Speedup (normalized to distributed shared)

Figure 16: Speedup over distributed shared for CAP for various block sizes (mean of all tested benchmarks).

7. CONCLUSION

In this paper we have presented compiler analysis techniques to detect data access patterns and partitions for multi-threaded applications. The proposed techniques reveal the available locality and data access behavior of different multi-threaded applications. The information discovered is then used to assign ownership to sub-pages, and this information is then used by the compiler-assisted partitioning (CAP) caching scheme for data distribution. We tested the proposed approach on various benchmarks from the SPLASH-2 and PARSEC benchmark suites and compared the results with other relevant schemes. The results demonstrate that the compiler-assisted caching scheme inherits the capacity benefit of distributed shared caches while reducing the memory access latency. More potential improvements are expected if the compiler can identify and analyze more complicated data structures and patterns. We plan to focus on this in our future work.

8. ACKNOWLEDGEMENT

This work is supported, in part, by NSF award CCF-0702452.

9. REFERENCES


