HYBRID CACHING FOR CHIP MULTIPROCESSORS USING COMPILER-BASED DATA CLASSIFICATION

by

Yong Li

B.S. Telecommunication Engineering, Chongqing University, 2005

Submitted to the Graduate Faculty of
the Swanson School of Engineering in partial fulfillment
of the requirements for the degree of
Master of Science

University of Pittsburgh

2010

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This thesis was presented

by

Yong Li

It was defended on

Nov. 8th 2010

and approved by

Alex K. Jones, Associate Professor, Department of Electrical and Computer Engineering

Rami Melhem, Professor, Department of Computer Science

Youtao Zhang, Assistant Professor, Department of Computer Science

Sangyeun Paul Cho, Associate Professor, Department of Computer Science

Thesis Advisors: Alex K. Jones, Associate Professor, Department of Electrical and Computer Engineering,

Co-Advisor: Rami Melhem, Professor, Department of Computer Science

HYBRID CACHING FOR CHIP MULTIPROCESSORS USING COMPILER-BASED DATA CLASSIFICATION

Yong Li, M.S.

University of Pittsburgh, 2010

The high performance delivered by modern computer system keeps scaling with an increasing number of processors connected using distributed network on-chip. As a result, memory access latency, largely dominated by remote data cache access and inter-processor communication, is becoming a critical performance bottleneck. To release this problem, it is necessary to localize data access as much as possible while keep efficient on-chip cache memory utilization. Achieving this however, is application dependent and needs a keen insight into the memory access characteristics of the applications. This thesis demonstrates how using fairly simple thus inexpensive compiler analysis memory accesses can be classified into private data access and shared data access. In addition, we introduce a third classification named *probably private* access and demonstrate the impact of this category compared to traditional private and shared memory classification. The memory access classification information from the compiler analysis is then provided to the runtime system through a modified memory allocator and page table to facilitate a hybrid private-shared caching technique. The hybrid cache mechanism is aware of different data access classification and adopts appropriate placement and search policies accordingly to improve performance. Our analysis demonstrates that many applications have a significant amount of both private and shared data and that compiler analysis can identify the private data effectively for many applications. Experiments results show that the implemented hybrid caching scheme achieves 4.03% performance improvement over state of the art NUCA-base caching.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE

Among many people who helped me with this thesis work, I first thank my advisor, Dr. Alex Jones for his relentless support during my graduate work. It was him who invited me to this excellent research group where I was able to get a start. His instructive advice helped me to build my research experiences from ground up and follow the right direction since then. His strong enthusiasm inspired me with great motivation on my research work. Without his help, I could have never done this work.

Second, I would like to thank Dr. Rami Melhem, who co-advises my research throughout my graduate study. His encouragement at an early stage of my work made me feel confident. His patient guidance and direction not only led me through the hardship I have experienced during my research work but also equipped me with all necessary capabilities for academic work. I also benefited a lot from his outstanding reasoning skills and rich research experiences.

I also thank Professor Youtao Zhang and Professor Sangyeun Paul Cho for being on my committee and giving me constructive advice on my thesis. Thanks are also given to Ahmed Abousamra and Colin Ihrig, my research group mates, for their suggestions and collaborative work.

Finally, I owe a thank to my wife, Ruoxin Zhang for her constant care and support, both physically and spiritually.

1.0 INTRODUCTION

As parallel processing is becoming mainstream technology to drive the ever-scaling computing power computer systems can deliver, modern computer architecture, including its major components such as caches and interconnects, are undergoing significant changes. Unlike traditional computer in which a few number of cores are integrated closely with a bus, current and future computers tend to have larger number of integrated cores connected using distributed network on-chip . Figure 1 shows a typical structure of a Chip Multiprocessor (CMP) that consists of 16 processor nodes. Each node in the CMP contains a processor core, 2 levels of cache, a coherence directory, a network adapter and a router, which connects the node to a $4 * 4$ mesh network.

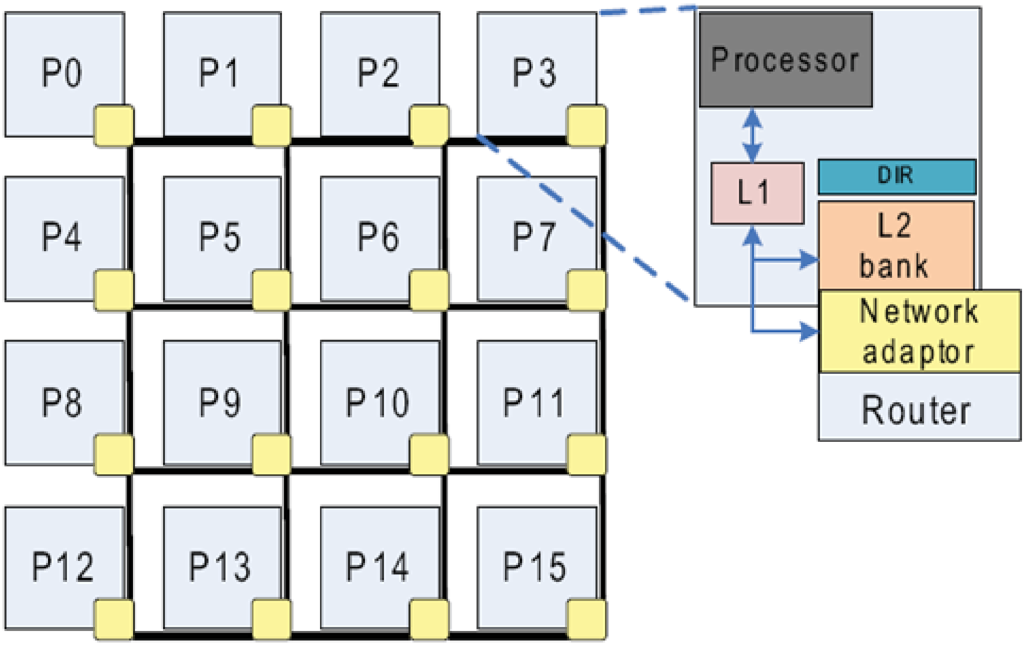


Figure 1: A Chip Multiprocessor connected using 4*4 mesh network on chip

Since the processing cores usually operate at a very high speed, the high performance such a chip multiprocessor can deliver is largely bottlenecked by memory access latency, which is highly dependent on the organizations of memory caches and on-chip interconnect. As the number of cores increases, the memory access latency in particular becomes an even greater bottleneck. Thus, designing an efficient on-chip cache memory organization is critical for the performance on CMPs and requires considerations of a variety of factors such as access distance, memory utilization, coherence problem, etc. Static non-uniform cache architectures (sometimes also called distributed shared) [10] and private [22] cache are two basic designs that have been widely used. In a distributed shared cache organization, a unique copy of data block is placed at a location determined by the physical address of the requested data. It keeps this simple location-address mapping for efficient placement and lookup of a certain data block at the expense of large amount of non-local data accesses, which incurs high access latency. On the other hand, private cache promotes locality by absorbing requested data to the local cache of the requesting processor. However, it consumes larger on-chip storage and results in cache coherence problem [29]. It would be desirable to achieve the benefits of both, namely, reduced remote data accesses as well as efficient cache capacity utilization. A number of architectural techniques have been proposed to achieve this goal [11, 42, 15]. In general, the proposed architectural techniques attempt to find a compromise between the two basic cache organizations. The goal is to keep data close to the processor cores that utilize it while efficiently utilizing the entire capacity of the cache.

Recently, some efforts [20, 21, 12] have been made to exploit the inherent distinction of the shared and private approach and use them to serve different data accesses. These proposed techniques classify data based on data access behaviors in the past, namely, the execution history, and then adopt placement or migration policy accordingly to achieve performance gain. In order to have a better insight of the data access behaviors of a certain application, it is necessary to utilize the information in the "future" by examining the application's characteristic before its actual execution. Many multi-threaded applications, such as those from the SPLASH-2 [4] and PARSEC [7], exhibit regular data access behaviors that can be exploited at compile time. In these benchmarks, accesses to the data in the stack segment are mainly local and the data in the text segment are largely shared. Even for heap objects allocated by memory allocator such as `malloc()`, the access patterns can be understood by the scope and usage analyses of the pointers that points to these objects. These patterns imply how multiple threads parallelize the data accesses and computations. Detecting these patterns is critical to gain an insight of how different data blocks should be placed, searched and accessed.

Consider the simple matrix multiplication problem as an example, as shown in Figure 2. When parallelized using shared memory programming mode *pthread*, matrix a is partitioned into different chunks which will be assigned to different working threads. Different threads work on their only share of data, specified by thread local variables (i.e. *mystart*, *myend* and *pid* in this example), as Figure 3 shows. Apparently as Figure 4 illustrates, different portions of data a and x are accessed exclusively by multiple threads (represented with different colors, black means accessed by all threads) while the data b is access by all threads. In other words, a and x is private versus b, which is shared. Treating a,b,x all in the same way, either using the private caching method or the shared approach, does not yield desirable performance.

```
float a[N][N],b[N],x[N];
    for(i=0; i<N; i++)
        for(j=0; j<N; j++)
           x[i] += a[i][j]*b[j];
```

Figure 2: Sequential Code for Matrix Multiplication

It is believed that the compiler is capable of gaining a global view of how data is used and detecting/classifying data with distinct characteristics. To leverage the potential benefits a compiler can offer, the underlying architecture, in particular the cache organization, must be cooperatively designed to take advantage of the analysis the compiler can provide. Additionally, there must be a mechanism to transfer data discovered by the compiler to the runtime system to utilize this information for reduced data access latency.

This thesis describes a hybrid S-NUCA and private cache architecture for CMPs driven by multi-threaded data analysis from the compiler. The compiler-assisted system we have developed will identify data that is locally accessed by a particular thread and use a private caching scheme for that data. This keeps that data in the local cache tile for faster access and does not harm overall

```
float a[N][N],b[N],x[N];
 row_per_thread = N/PROCS;
  ...fork multiple threads...
 my_start = pid*row_per_thread;
 my_end = (pid+1)*row_per_thread;
 for(i=my_start; i<my_end; i++)
     for(j=0; j<N; j++)
          x[i] += a[i][j]*b[j];
```

Figure 3: Multi-threaded Code for Matrix Multiplication

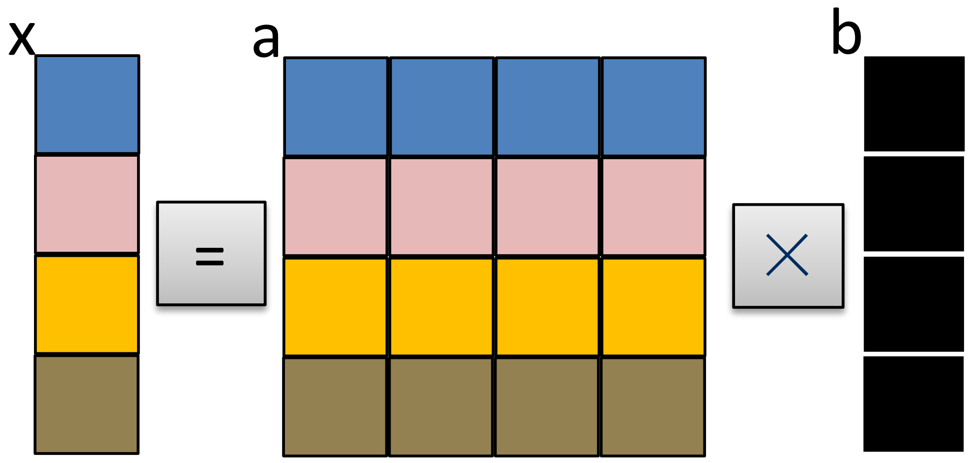


Figure 4: Matrix Multiplication

capacity (assuming a relatively even data distribution) because there will be no sharers. Data that is identified as shared by several threads by the compiler will utilize an S-NUCA caching scheme to maximize cache capacity.

This thesis demonstrates the compiler's capability to distinguish between private and shared style memory accesses. In addition, to relax the compiler analysis, a memory classification called *probably* private accesses is introduced. Probably private access implies a situation where the compiler cannot prove that the memory is only accessed by one processor but speculatively determines that private style access is likely. The data shows that a high percentage of these values are dominated by local access allowing the system to take advantage of access locality to reduce latency.

The rest of the thesis is organized as follows: Section 2 describes background knowledge and related efforts. Section 3 introduces our compiler analysis technique for data classification and identification. In Section 4, we introduce necessary system support and the cache organization in which the compiler analysis is utilized. We examine the effectiveness of the compiler for identifying private and shared data and demonstrate the performance gain brought by our compiler-architecture co-design approach in Section 5. Finally, we draw some conclusions and describe on-going and future work directions of this effort in Section 6 and Section 7.

2.0 BACKGROUND

2.1 COMPILER OPTIMIZATIONS

Compiler analyses and optimizations have been proven to be critical to improve code efficiency, reduce resource utilization, expose optimization opportunities, etc. The proposed data classification is also dependent on a series of compiler analyses including control and data flow analysis, symbolic analysis, etc.

A control flow graph (CFG) [3] is directed graph built on top of the intermediate code representation abstracting the control flow behavior of a function that is being compiled. In CFG, each vertex represents a basic block [1] and edges represent possible transfer of control flow from one basic block to another. Figure 5 shows an example CFG and its corresponding source code. Since CFG logically represents the relationship among different components of a program, it forms the basis for a large number of compiler analyses and optimizations ranging from pointer analysis, reaching definition, liveness analysis, dead code elimination, loop transformation, constant propagation, branch elimination, instruction scheduling, etc.

For data flow analyses, each basic block is further represented as a data flow graph (DFG) [24]. A DFG, which is also a directed graph, represents the data dependencies within the code between control points. In a DFG, each node represents an operator (e.g. addition, logical shift, etc.) or an operand (e.g. constant, variable, array element, etc.). Each directed edge represents a data dependency that denotes the transfer of a value.

Based on combination of basic compiler optimization approaches, one can perform various high-performance oriented compile-time optimizations including dependence analysis, data reuse analysis, data access analysis, loop transformation, auto-parallelization [16, 18, 17], as have done

[1] A basic block is a continuous sequence of code with only one entry point and only one exit

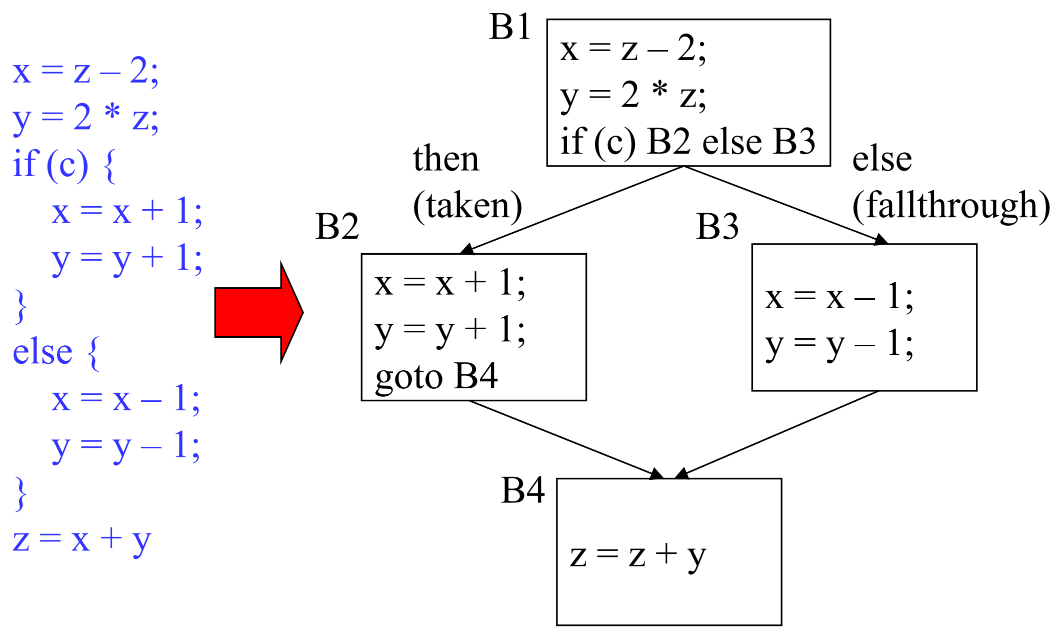


Figure 5: A control flow graph and its corresponding code

by many researchers. 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 [34] used matrix representation to formulate a data partitioning applied to multiprocessors without caches. Ju and Dietz [25]'s efforts focused on finding a data layout (row or column major) for memory block in a uniform memory access (UMA). In Wilson and Lam's work [39], data dependence was studied using a linear algebra approach. They distributed loop iterations and thus data that carry no dependence among multiple processors. Barua [6] developed a heuristic method to deal with data partitioning in a way that avoids NP-complete linear programming solutions.

More recently, Paek and Padua [33] 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. Michael Chu proposed a profiling based scheme [14] to determine a fine-grain data access partitioning. Another work [13] of his described an approach to partition data objects among multi-cluster processors. In the 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 [31], polyhedral model is used to perform localization analysis with the goal of finding a data layout transformation to promote the locality which would otherwise be destroyed by finely interleaving data among the tiled banks.

One of the uniqueness of this thesis work is that it targets analyzing multi-threaded applications and detecting useful informations (i.e. the data access classifications) that can be utilized by on-chip cache memory. In this work, particularly, control flow analyses, pointer analysis as well as other optimization approaches are used for data classification (see Section 3 for more details).

2.2 SUIF COMPILER INFRASTRUCTURE

We build our compiler analyses based on SUIF [40]. SUIF is a research-oriented compiler infrastructure that facilitates compiler optimization development. It contains a snoot-based C front end, an object-oriented intermediate representation of source code, a series of optimization passes and a library of useful compiler analyses. Compared to other publicly available compilers, SUIF is much easier to be modified and extended, making it an ideal platform for compiler technique development.

Figure 6 illustrates the class inheritance in SUIF structure. All major SUIF data structures are derived from the *suif_object* base class. Programming components in the source code are abstracted using these classes. For example, *sym_note* represents a symbol of a certain kind, such as a procedure symbol, a label symbol or a variable while *type_note* keeps track of the type information of a symbol and is attached with that symbol in the program.

A hierarchy of SUIF intermediate representation is shown is Figure 7. SUIF organizes source code file set at the top level, which contains a number of source files. In each source file, there are several tree procedure objects representing local functions or procedures. A tree node list is associated with tree procedure. The tree node list contains tree nodes of various kinds, including *tree block*, *tree instruction*, *tree if*, *tree for*, *tree loop*, etc. Each type of tree node represent a corresponding programming structures. For example, *tree for* represents a *for* loop in C source

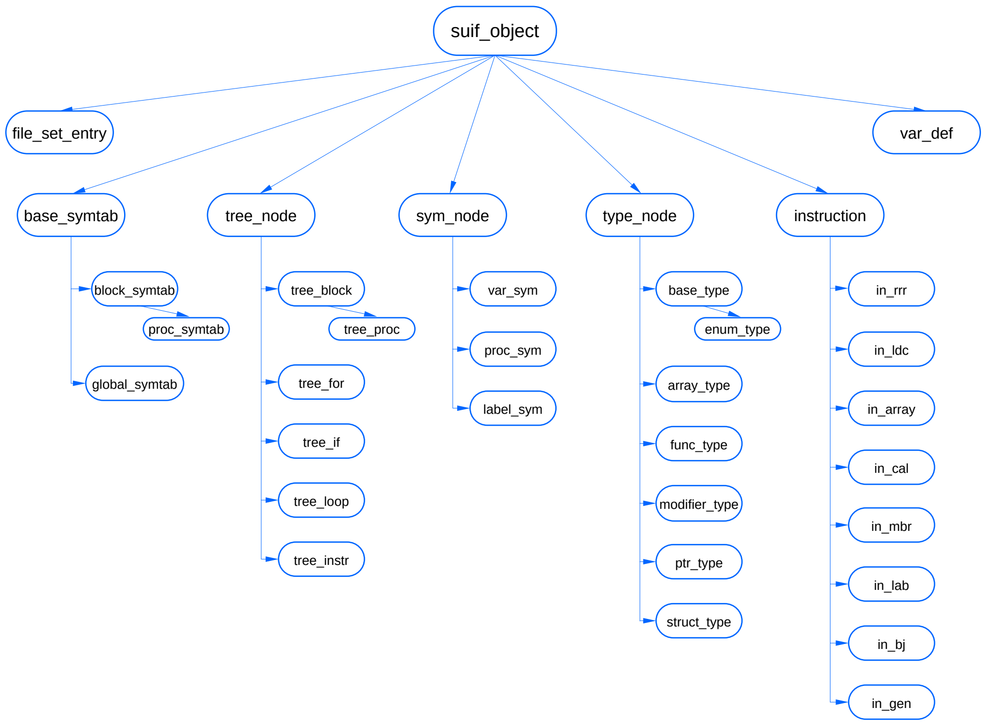


Figure 6: Class Inheritance of SUIF Structures

code. *Tree for* again consists of several components such as *lb*, *ub*, *step*, *body*, to capture the information of lower bound, upper bound, loop step and loop body, respectively. *tree instr* is another important class which stores the basic logical and computational instructions.

SUIF provides a whole set of methods to manipulate these objects in its IR hierarchy. This feature allows us to detect and modify certain programming structures, such as the arguments of a function call. Consider the following statements:

```
pthread_create(newThread, &attr, (void*)&(SlaveStart) , NULL);
```

Its corresponding SUIF IR is shown in Figure 8. The *mrk* is a nop instruction and has no actual meaning. The text in the [] is annotation used to store temporary information such as the line number in the source code. *cal* and *ldc* is *tree instr* objects which represent call and load constant instruction respectively. To retrieve a certain argument of the `pthread_create()` function, the method *argument(int n)* can be used. The following code obtains the third ar-

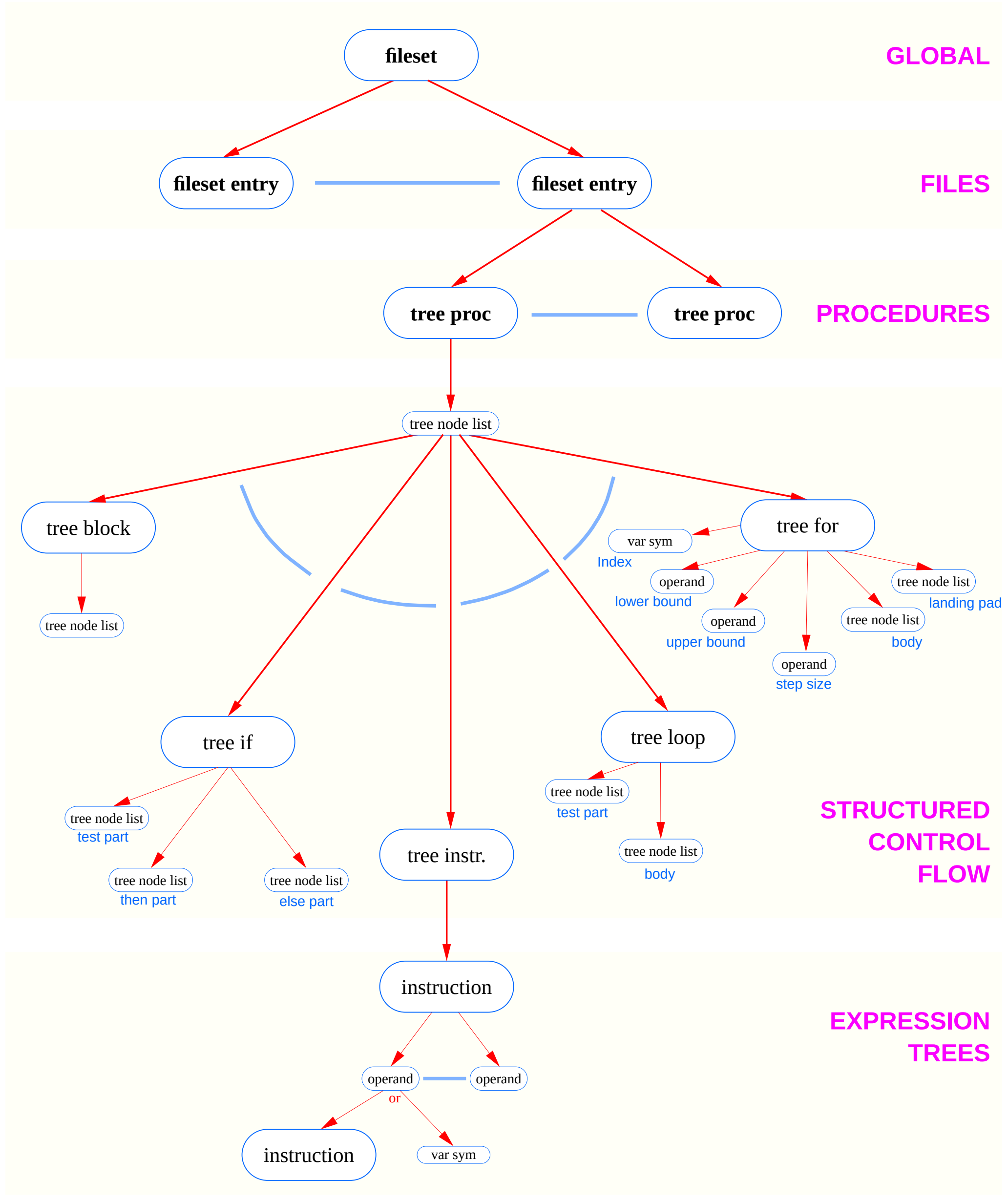


Figure 7: Hierarchy of SUIF intermediate representation

gument in the `pthread_create()` call (assume *the_call* points to the object that represents `pthread_create()`) and stores it in the variable *pfp* with a type of *operand*.

```
in_cal *the_call;

    operand pfp = the_call->argument(2);
```

For more detailed information please refer [1].

```
mrk
               ["line": 470 "fft.c"]
 822:        cal t:g4 (i.32) <nullop> = e1(.newThread, e2, e3, e4)
 823:          e1: ldc t:g412 (p.32) <P:.pthread_create,0>
1578:          e2: ldc t:g256 (p.32) <main/0.attr,0>
1579:          e3: ldc t:g258 (p.32) <P:.SlaveStart,0>
1580:          e4: ldc t:g30 (p.32) 0
 830:        mrk
```

Figure 8: SUIF IR for pthread_create() function call

2.3 BASELINE CACHE ORGANIZATIONS

This section introduces more details about two basic cache organizations, namely, the distributed shared and private cache, which are widely used in CMP architectures. The proposed hybrid caching scheme is built upon the two baseline cache designs.

2.3.1 Distributed Shared Cache Organization

Distributed shared cache organization, which is derived from S-NUCA, has been extensively used (e.g. Intel Core 2 Duo) as last level cache for its simplicity and efficiency. It provides a global single address space on top of physically distributed caches, thus creating an illusion that all processors in the system share one single large cache.

Figure 9 illustrates a typical structure for distributed shared based CMP. In such a configuration, L1 caches are usually private to their processors to ensure fast access. L2 cache is physically

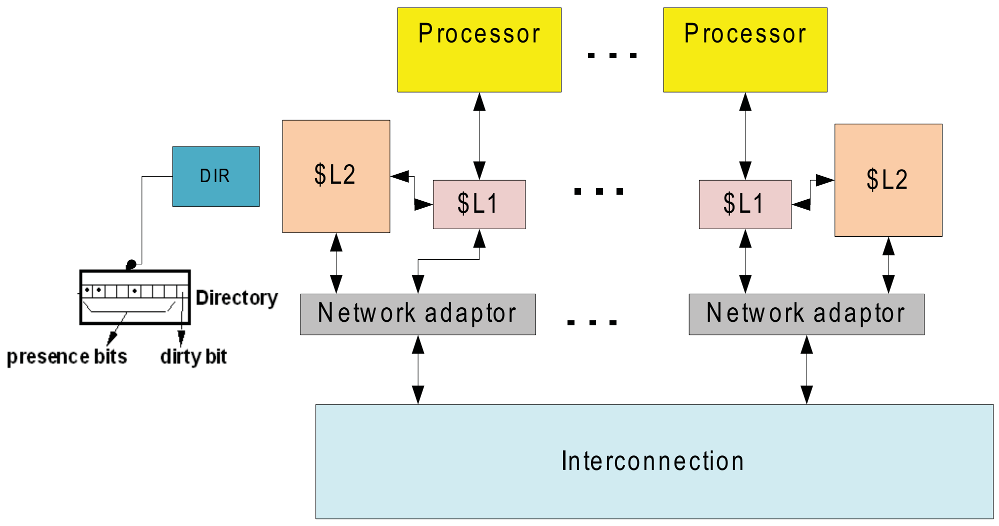


Figure 9: A typical distributed shared cache organization

distributed, logically shared by all processing nodes. As shown in Figure 9, each node contains one processor, a L1 cache (further divided into instruction and data cache) and a bank of L2 cache. The network adaptor is used when transactions such as cache misses and coherence messages need to be served remotely in another node.

In a distributed shared cache organization, data requests issued by CPU are first served in L1 cache if there is a hit on local L1. Upon a local L1 miss, cache controller searches a bank of L2 for the requested data. Which bank is searched is determined by a number of $\log n$ bits, where n is the number of processing nodes in the CMP, within the physical address of the requested data. Consider Figure 10 as an example. With a 32 bit memory address space, the cache line size is set to 64 bytes with 6 block offset bits. The 8 way associate set, 2 megabyte L2 makes 12 bit set index. The other 14 bits are for the tag area. For a multiprocessor system with 16 processing nodes, $\log 16 = 4$ bits are needed to determine the mapping. Utilizing bit-6 to bit-9, as shown in Figure 10, leads an interleave granularity at cache line size.

There is also an on-chip directory, and each entry of it keeps track of the L1 sharers of a corresponding cache block using a bit vector. Each bit in the bit vector indicates the presence status of the cache block in the corresponding L1 cache. The dirty bit shows whether the current

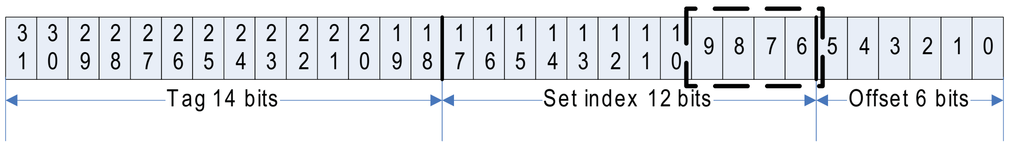


Figure 10: Structure of a cache block

cache block is valid. At L2 level, since there is only one unique copy for a particular cache block which is shared across all processors, it is not necessary to maintain coherence.

2.3.2 Directory Based Cache Coherence Protocol

Unlike distributed shared cache, private caching approach replicates data block to the local cache bank of every requesting processor. In other words, multiple copies of the same data exist at the same cache level simultaneously. This creates the coherence problem, which is need to be maintained by coherence protocols. MSI and MESI [36] are two popular protocols that have been widely used for their effectiveness and efficiency.

The MSI coherence protocol maintains three states for each cache line in the cache:

- **Modified:** data block is modified by one processor and all other copies is invalid.
- **Shared:** data block has one or more copies in different processorsćaches, and the copy in the next level memory hierarchy is also up-to-date.
- **Invalidate:** data block is invalid in any processor's cache.

The directory must trace each cache line's state and which processor has data when the cache line is in a Shared state. This is usually accomplished by keeping a bit vector. The vector's bit is set when a corresponding processor has the data. Another dirty bit is used to indicate if the cache line is in Modified state. Typically, three processors will be involved in one transaction:

- **Request node:** the node which issues a memory request on the target cache line.
- **Home node:** the node to which the target cache line belongs.
- **Remote node:** the node's cache contains the target cache line (modified or shared).

Specific coherence messages must be created to transfer the control information and actual data among the involved nodes in the coherence transactions. Table 1 gives a detailed description of the message types created to participate in the MSI coherence.

Table 1: Directory Protocol Messages List.

Message type	**Source**	**Destination**
Read	Local cache	Home directory
Read directory to lookup the status of a particular cache line		
Write	Local cache	Home directory
Write directory to update the status of a particular cache line		
Invalidate	Home directory	Remote caches
Home directory invalidate a shared copy at a remote cache bank		
Invalidate ack	Remote caches	Home directory
After invalidation, remote cache sends an ack back to home directory		
Fetch	Home directory	Remote cache
Home directory fetches a particular cache block		
Fetch-Invalidate	Home directory	Remote cache
Home directory fetch a particular cache block, then invalidate it afterwards		
Read reply	Home directory	Local cache
Read miss response, act as an acknowledgement, make the requesting processor a sharer		
Write reply	Home directory	Local cache
Write miss response, act as an acknowledgement, make the requesting processor a modified owner		
Write back	Remote cache	Home directory
Remote cache writes back a data value to home directory		

Figure 11 and Figure 12 illustrate the state transition from the CPU and directory point of view, respectively. Take the directory transition as an example. If a block is in the invalid state, which means the copy in memory is the current value, possible requests for that block are read miss or write miss. Read miss causes the requesting processor to be sent data from the memory, the requesting node to be added as a sharing node and the state of block to be made shared. Write miss makes the requesting node and block state modified. If the block is shared, which means the memory value is up-to-date, a read miss has the same behavior as in invalid block case. A write

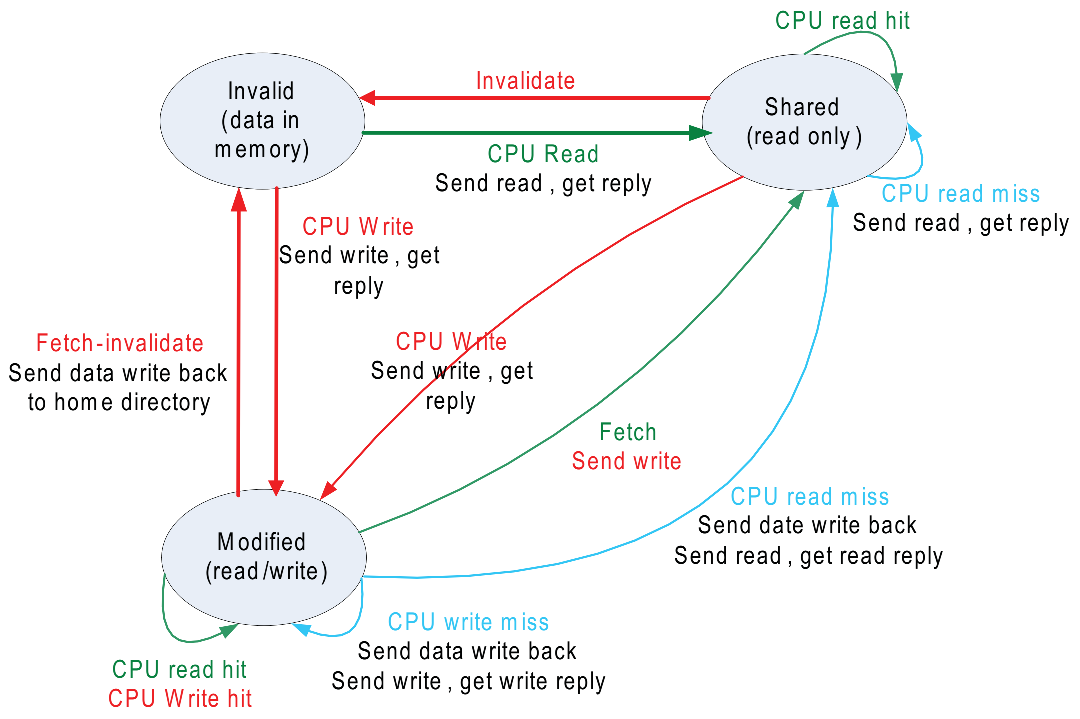


Figure 11: State machine for CPU requests for each memory block.

miss will invalidate all processors in the set sharers and the state of the block is made modified. If the block is modified, which means current value of the block is held in the cache of the processor identified by the set sharers (the owner), in a read miss, the owner processor gets a data fetch message, changes its state to shared and sends data to directory, where it's written to memory and sent to the request processor. The requesting processor is added to the set sharers, which still contains the owner (since it still has a readable copy). In a data write-back, the owner processor is replacing the block, and hence must write it back, making the memory copy up-to-date (the home directory essentially becomes the owner); the block is now invalid, and the sharer set is empty. In a write miss which means the block will have a new owner, a message is sent to the old owner causing the cache to send the value of the block to the directory from which it was sent to the requesting processor, which becomes the new owner. sharers is set only to the new owner, and the state of block is made modified.

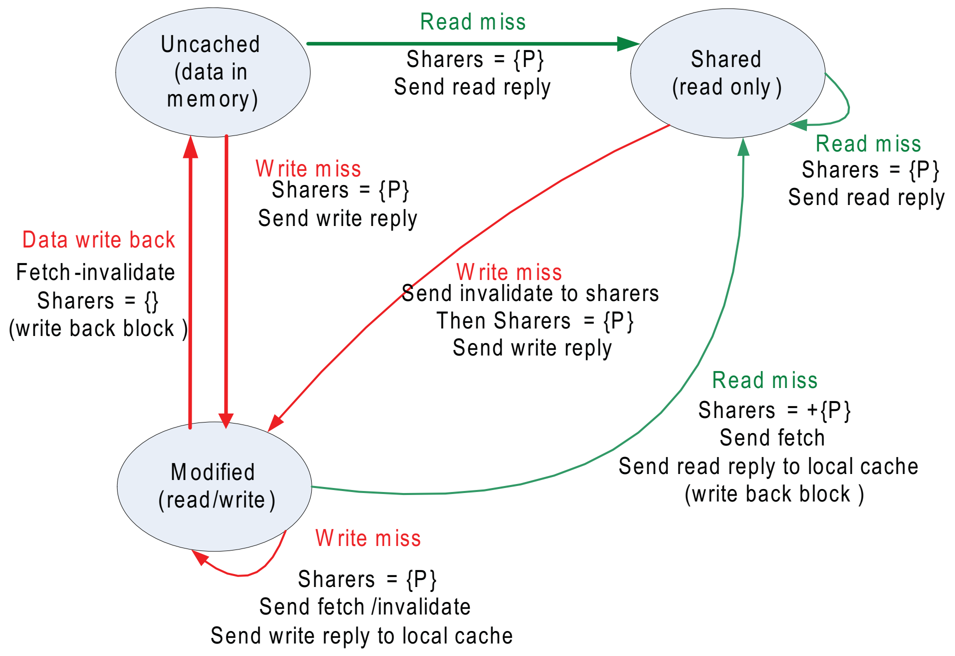


Figure 12: State machine for Directory requests for each memory block.

MESI protocol was originally developed based on MSI for bus transactions, and later introduced to distributed network on chip, as shown in Figure 13. The MESI protocol consists of four states: modified(M) or dirty, exclusive-clean (E), shared (S), and invalid (I). I means the cached block is now invalid. M indicates the current cache block is the only one valid block because it has been written/modified without writing back to main memory. E, the exclusive-clean or just exclusive state, means that only one cache (this cache) has a copy of the block, and it has not been modified (i.e., the main memory is up-to-date). S means that potentially two or more processors have this block in their cache in an unmodified state. When the block is first read by a processor, if a valid copy exists in another cache (indicated by the *NetQuery(S)* transaction) then it enters the processor's cache in the S state as usual. However, if no other cache has a copy at the time (indicated by the *NetQuery(S)* transaction), it enters the cache in the E state. When that block is written by the same processor, it can directly transition from E to M state without generating another network transaction, since no other cache has a copy. On the other hand, if there is a write on a line with state S or I, the network transaction *NetInv* is generated to invalidate the line in all other caches. Local read on E M or S lines neither change the line's states nor generate network transactions, as illustrated in Figure 13. Upon a read request from network, a node that has a corresponding line in M or E state will change that line to S state and acknowledge the requester with *NetAck* transaction. If the transaction *NetInv* is encountered, the corresponding line in any cache bank need to be set to I.

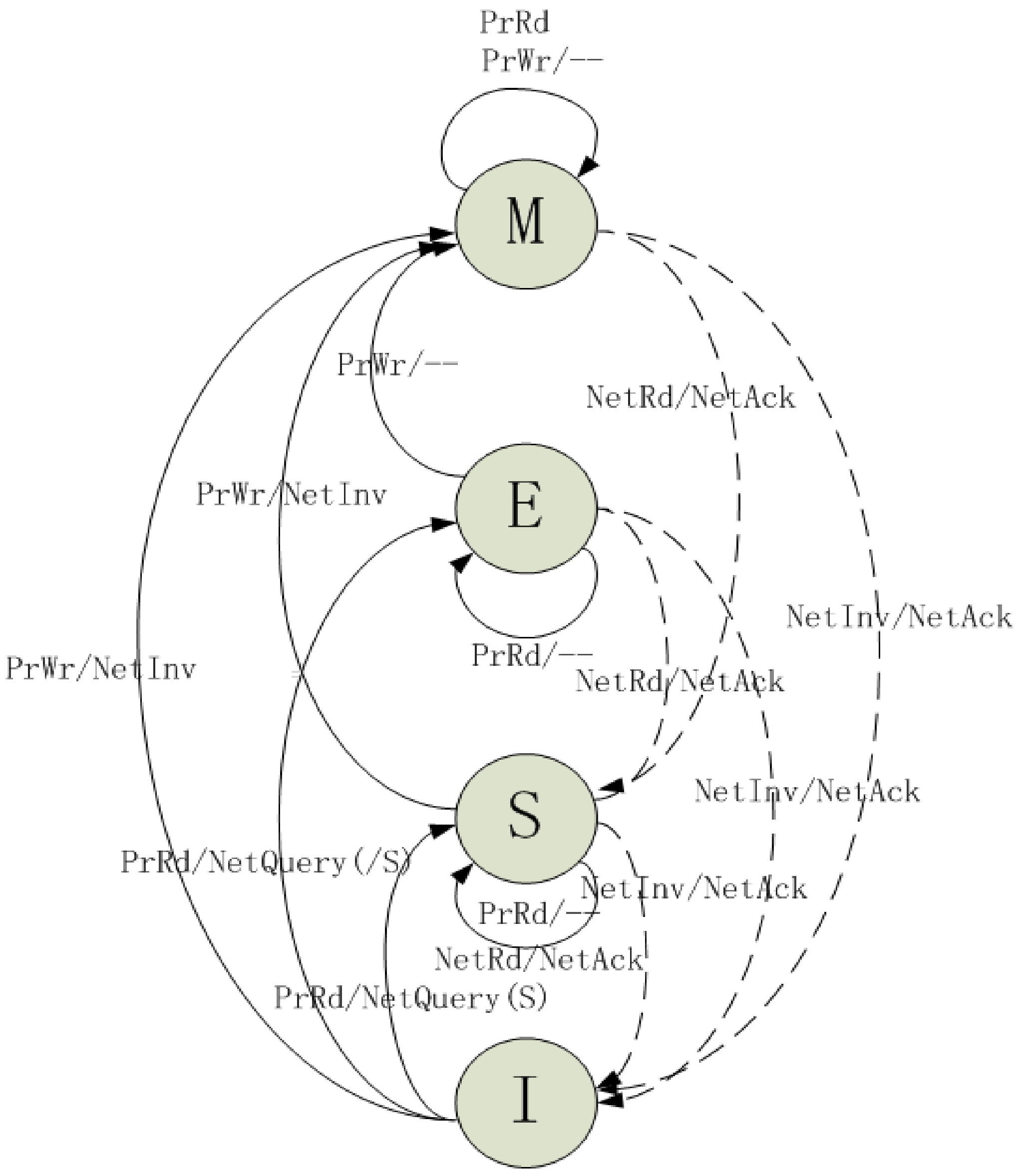


Figure 13: State transition diagram for MESI protocol

2.4 ADVANCED CACHING SCHEME

Since the S-NUCA scheme [26] was proposed, extensive research has been conducted to mitigate the impact of poor data proximity that is possible with this scheme often by adopting something like private caching [9]. However, private caches, due to the replication of shared data, do not as effectively utilize cache capacity as a distributed shared cache. Several variations have been proposed starting either with shared or private cache organizations to achieve an improvement [41, 28]. Specifically, Zhang and Asanovic [42] proposed the victim replication cache scheme based on a shared L2 cache structure. In their work, the local L2 cache slice is used as a place to hold victim cache lines from local as well as remote L1s. Chang and Sohi [11] designed cooperative caching based on a private caches. They enlarged the effective cache capacity by evicting cache lines which have multiple copies prior those with only a single copy. They also studied optimizations such as cache-to-cache transfer of clean data, replication-aware data replacement, and global replacement of inactive data. Dybdahl [15] tried to combine the advantages of both shared and private caches by dividing cache banks into shared versus private partitions. The size of each partition changes dynamically depending on the miss rate of the corresponding bank.

In recently proposed reactive NUCA (R-NUCA) [19], cache accesses are classified as private, shared, and read-only (e.g. instructions). Their cache scheme recognizes and classifies the cache access patterns at the page granularity. Data is assumed to be private until a second processor accesses the data (signaled by a translation look-aside buffer (TLB) miss). Data from private pages are cached privately to improve latency while elements from shared pages are cached using an S-NUCA style approach to improve capacity.

Another relevant work is Jin and Cho's software oriented shared (SOS) cache management [23]. They classify data accesses to a range of memory locations (returned by a memory allocation function such as `malloc()`) into several categories such as Even Partition, Scattered, Dominant Owner, Small-Entity, and Shared. These memory accesses are profiled and matched to one of the above categories. Then hints are attached to the memory allocation functions such as `malloc()` to show the related access patterns. Pages within the memory ranges are placed to cache tiles indicated by the hints. The paper states that this classification could be completed by a compiler but no such analysis is included.

Both R-NUCA and SOS work at the page granularity. As a result for these schemes, one data access pattern is in danger of being polluted by another. Additionally, the cache pressure may not be balanced. In this paper, we present a compiler-assisted hybrid cache organization. In particular, our compiler techniques assist in data access classification. This avoids the need for running and profiling programs prior to data classification, as profiling is imperfect and can be biased by the particular dataset used. Additionally, we use a modified version of `malloc()` to avoid pollution due to page granularity from traces or runtime classification.

3.0 COMPILER ANALYSIS METHODOLOGY

3.1 OVERVIEW

A study of multi-threaded codes from a variety of program domains such as scientific computing, multimedia, image processing and financial processing reveals that data structures can be used in a few common ways. Parallel applications utilize a few different styles, which may be classified as data-parallel, master-worker, and pipelined. Applications within a particular class typically exhibit distinct data access characteristics. We also observe that the data access patterns usually can be implied by information such as where in the virtual memory space the data is allocated and how the references of data are handled. Figure 14 shows the virtual space structure for a single process running on 32-bit Linux operating system. Other than the kernel space on the top and the shared library space in the middle, data allocated in other segments are process-specific. If the process has only one thread, all the data in these segments (read-only, data, bss, heap and stack segments) is private to this thread unless there is explicitly inter-process communication. If on the other hand, the process is multi-threaded, as Figure 15 shows, multiple threads share the read-only segment, read/write data segment and the heap. Stack area is partitioned among threads thus the data on it is implicitly local(or private). However, stack data is not guaranteed private as multiple threads essentially share the same process space and potential derive address pointer pointing to other threadsśtack data.

Generally speaking, data access on other segments exhibit more flexible patterns. For example, instructions and globally allocated data such as synchronization structures on the read-write segment are shared by all cores and are entirely read-only. Data on the boundaries of memory block partitions allocated on the heap by calls to `malloc()` (e.g. particle simulations) have a small number of sharers. Heap data blocks allocated within a thread are usually private to that thread.

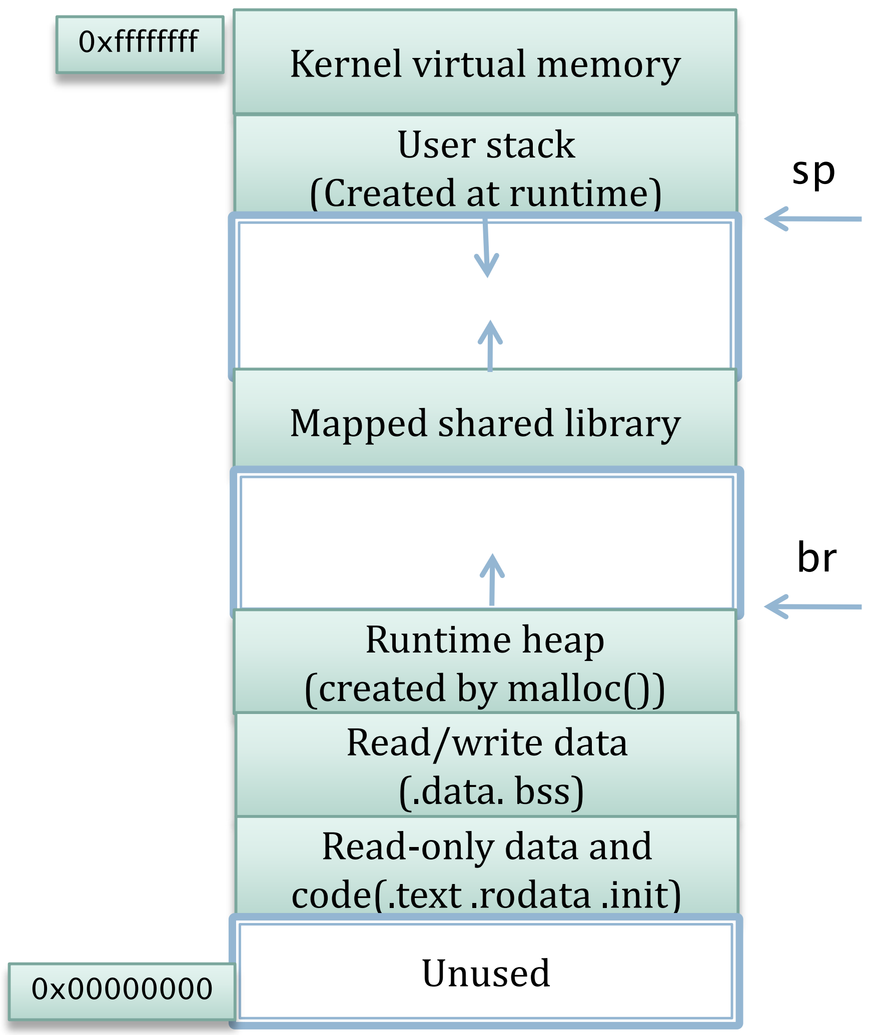


Figure 14: Single process virtual memory space for 32-bit Linux OS

However, as shown in Section 5, applications in different domains and with different workloads often have considerably different proportions of private versus shared data blocks with different degrees of sharing.

From the system performance point of view, data that has few sharers, such as thread-local and stack data, is largely private and thus, has good locality. While data with many sharers, such as global synchronization structures, have poor locality and occupy a large amount of cache storage space when using a private style caching scheme. However, traditional distributed shared caching

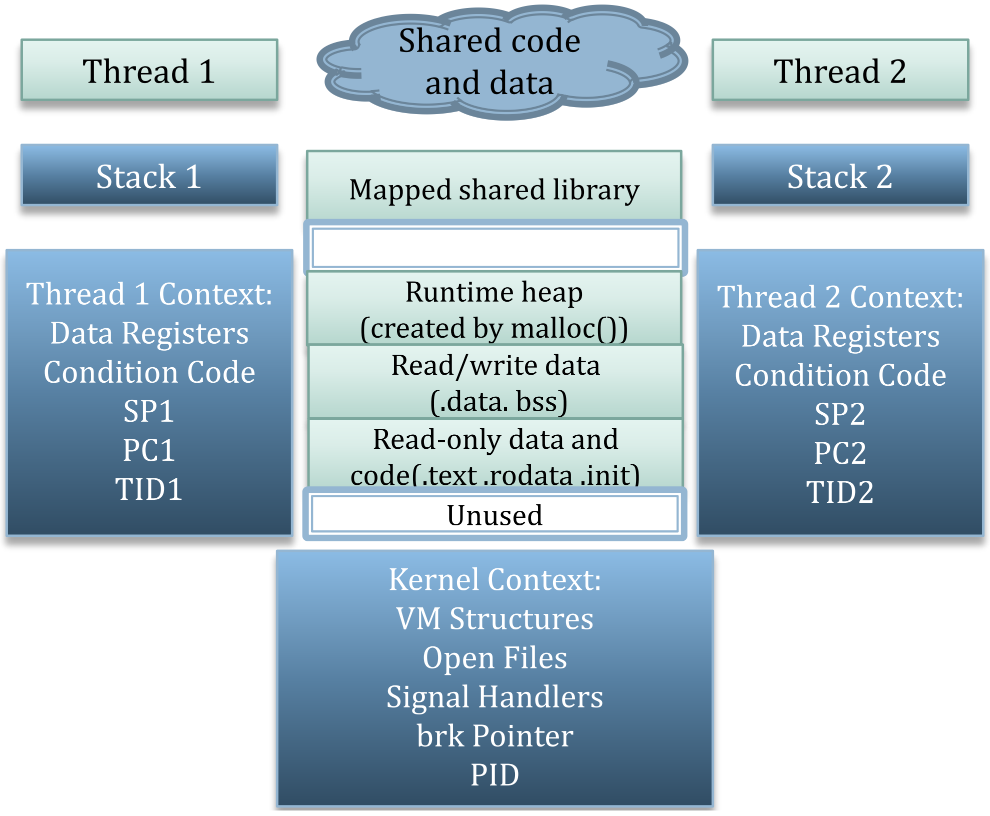

Thread 1
Shared code and data
Thread 2
Stack 1
Mapped shared library
Stack 2
Thread 1 Context: Data Registers Condition Code SP1 PC1 TID1
Runtime heap (created by malloc())
Read/write data (.data. bss)
Read-only data and code(.text .rodata .init)
Unused
Thread 2 Context: Data Registers Condition Code SP2 PC2 TID2
Kernel Context: VM Structures Open Files Signal Handlers brk Pointer PID

Figure 15: Multi-threaded process virtual memory layout

(e.g. S-NUCA) destroys locality of all data access for the benefit of better cache capacity utilization. Thus, we attempt to discover in the compiler when to employ private caching and when to employ S-NUCA caching.

Data access latency can be reduced by distinguishing data blocks with distinct access characteristics and treating them differently in the cache. This classification can be done at run time, as introduced in Section 2. Alternatively, we propose a compiler-architecture co-design approach to identify and classify the data access information at compile time and utilize this information in the architecture during application execution. Since a variety of multi-threaded benchmarks feature extensive usage of dynamic memory allocation for managing the computed data, we concentrate on analyzing data blocks allocated through memory allocators such as `malloc()`. The analyzing approach can be extended to other memory allocation routines such as `calloc()` and `new()`.

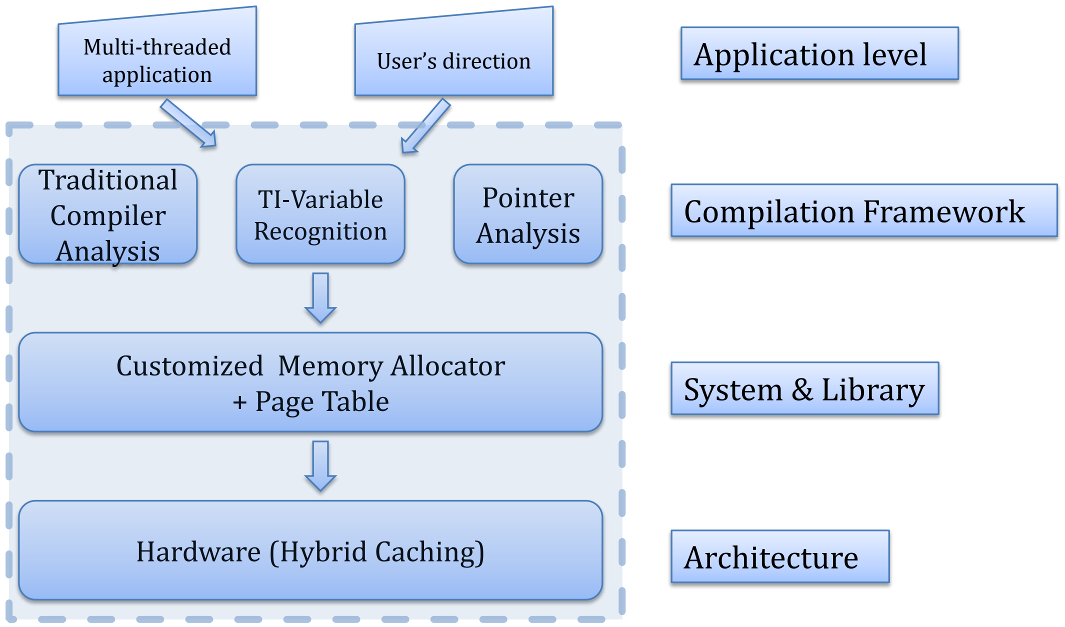


Figure 16: Compiler framework for hybrid cache organization

An overview of the proposed compiler framework is presented in Figure 16. The experimental compiler framework accepts multi-threaded applications as input, on which the traditional compiler analyses, TI-variable recognition and pointer analysis are performed. The analyzed results(i.e. data classification of memory blocks) are then passed to the runtime system from a customized memory allocator. Page table structure is instrumented with additional bits to store these classification information. Finally, the underneath hybrid cache organization utilizes the data classification information to optimize data access, placement and lookup on last level cache. The following sections introduce more detailed about the proposed compilation techniques.

3.2 DATA ACCESS PATTERN CLASSIFICATION

Understanding the data access behaviors in multi-threaded programs is essential to determine an optimal data placement. Parallel applications tend to exhibit flexible and diverse data access patterns. This poses a challenge for the compiler to detect and describe these patterns. However, our study shows that there are several representative patterns existing in a variety of parallel applications and they dominate the entire program execution. This section introduces the method to classify data access patterns.

The classification of data access patterns must meet certain criteria for our technique of compiler analysis to improve data access latency and ultimately performance, in particular:

1. Different patterns within an application exhibit remarkable disparity in terms of locality, storage requirement, and access behavior and as such must be treated differently
2. Patterns are practical for the compiler to identify and
3. Patterns are representative for a wide spectrum of parallel applications.

A straightforward method is to classify data blocks into two distinct access categories: private versus shared, as done in [19]. Private data is accessed by only one processor and thus is suitable to be placed locally to reduce access latency and promote locality. Conversely, shared data is accessed by more than one processor and is suitable to be placed at a fixed location which can be located as a function of address. Another possibility is to place the data at the "center of gravity" of its requesters if replication is not allowed [5]. However, determining an optimum location for a shared data requires more information such as the number of sharers, the sharing patterns among multiple sharers, the distribution of actual sharers, etc, which may be possible to be computed by a compiler but is beyond the scope of this paper.

In order to remain correct, our compiler analysis must remain conservative to identify private data. Unfortunately, the compiler analysis we employ, such as pointer analysis introduced later in this section, is often not able to prove that data which is likely private is guaranteed to be private. Consider Figure 17 where the entire memory range is sub-divided into blocks that are *guaranteed* to be private by the compiler and those that are not. Only for three of our benchmarks has any noticeable amount of private data been detected and that data only represents a low percentage of all the addresses utilized by the applications. Other benchmarks exhibit negligible access to private data compared with the total number of accesses. Thus, to relax the burden of the compiler, we extend our classification to include a third category of *probably private* data. Thus our classification is described as follows:

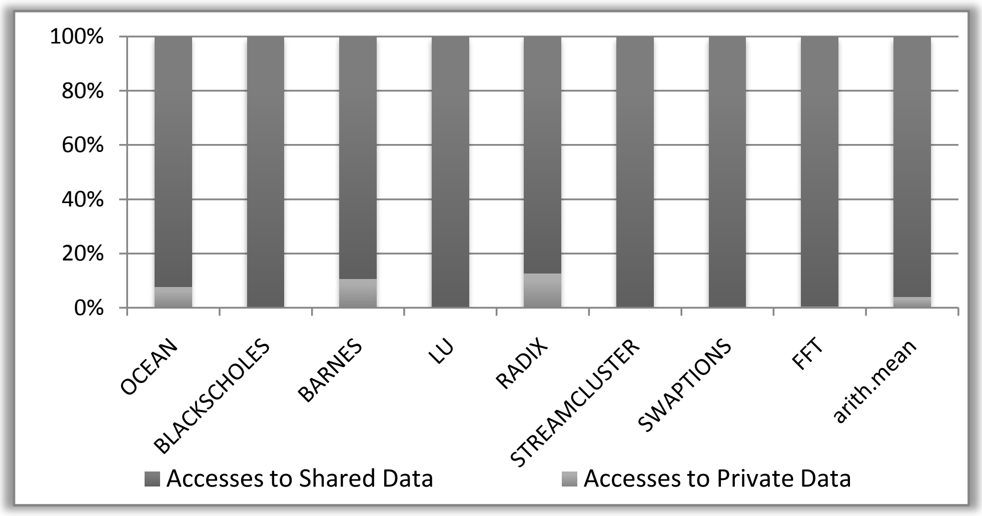


Figure 17: The percentage of private vs shared data blocks identified by compiler analysis

1. **Private:** In multi-threaded applications, a data block (returned by `malloc()`) is said to be *private* if every element in it is accessed by only one thread in the parallel program section. This is also the case when multiple threads in the program partition a data block evenly without overlap.

2. **Probably Private:** The compiler can not always detect all private data in the program for a variety of reasons: The granularity of analysis (e.g. data-allocation level) may not be small enough, or the analysis is too complicated for the compiler to identify data access pattern safely. When the compiler detects features of the program that imply private data access but cannot guarantee this condition, the data is classified as *probably private*. Essentially, probably private data represents memory blocks "owned" by one thread but possibly with some amount of sharing or data exchange with one or more other threads.

3. **Shared:** Some data blocks, such as synchronization and global counters, are accessible by all threads and there is no thread specific variables involved in the accesses to these blocks. These data blocks are defined to be *shared*. This is the default classification when memory blocks can not be classified as utilizing some form of private access.

Table 2: Statements and their effects

Statement	Gen	Kill
$a =$ `malloc()`	create a reference list with pure pointer a in it	remove a
$b = a$	add b to the reference list that contains a	remove b
$b = \&c$	add b to the reference list associated with c	remove b
$b = c[i]$	if $c[i]$ is in a reference list, add b to it, otherwise no effect	remove b

3.3 DATA CLASSIFICATION IDENTIFICATION

The data access classifications defined above reveal different opportunities to optimize the cache access performance with *private* having the lowest access latency and requiring the fewest coherence messages. Determining a data block as *shared* is safe, but has higher access latency with significant coherence traffic. Alternatively, classifying a block as *probably private* promotes data locality and proximity, but potentially requires coherence traffic and reduced overall cache capacity if there are in fact sharers. To achieve a desirable result, it is essential for the compiler to carry out data access pattern analyses that detect private data conservatively and probably private data accurately.

To simplify the illustration of identifying data access patterns, we define some notions as follows:

Definition 1. *A* pure pointer *is a pointer that has been assigned the return value of a memory allocator such as* `malloc()`*.*

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

Definition 3. *A* reference list *is associated with a heap object such as a memory block allocated by* `malloc()` *and contains a series of pure or/and derived pointers that point to it.*

Initially, reference lists are created at the procedure `malloc()`'s call sites. The pure pointer, which stores the return address of `malloc()` is added into the corresponding reference list.

Whenever a pointer in a reference list appears on the right hand side of a pointer assignment, we record the left hand side as an alias of this pointer, which might be used in later accesses, and add this alias to the reference list. When a pointer in the reference list appears on the left hand side, we move the pointer to the new reference list based on the right hand side of that pointer assignment.

The above procedure is performed on the CFG (control flow graph) of the analyzed program. Let s be a statement,
$succ(s)$ = { immediate successor statements of s },
$pred(s)$ = { immediate predecessor statements of s},
$In(s)$ = { reference lists before executing s}, and
$Out(s)$ = { reference lists after executing s}.
Each statement s in the program has two effects on $In(s)$ and $Out(s)$: Gen and $Kill$. Gen generates a new reference list or a pointer for an existing reference list, depending on the value of s. $Kill$ removes a reference list or a pointer within it. Table 2 gives the description of the effects of Gen and $Kill$ for same example statements.

The data flow equation for updating the reference list can be derived based on the above notions:

$$Out(s) = Gen(s) \bigcup (In(s) - Kill(s)) \tag{3.1}$$

$$In(s) = \bigcup_{s\prime \epsilon pred(s)} Out(s\prime) \tag{3.2}$$

As the analyses traverse the CFG, the reference lists are updated to accommodate all the pointers that *may* be used to reference a particular memory block. The access pattern of a memory block can be determined by examining the pointers in the updated reference list associated with that block along with certain code constructs common to multi-threaded applications. One of the important features for the compiler to discover are *Thread-Identifying Variables* [30] described as follows:

Thread-Identifying variables (TI variables) are thread-local variables which have unique values for different threads of execution. Typically, these variables are used to determine which memory blocks and in some cases which portion of the memory block the thread will access. The compiler must identify which variables in the program are TI variables in order to determine how the pointers in the reference lists are dereferenced and used by each of the threads.

One common method to assign values to TI variables is to pass different values to each of the parallel threads as function arguments during thread creation. As shown in Figure 18, 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 the 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 populate TI variables in multi-threaded applications is illustrated in Figure 19. Multiple threads try to access and modify a global variable under the protection of a mutex. This type of code is much more difficult for a compiler to analyze. If the code uses this or any other subtle methods to specify TI variables we require the user to include a directive to assist the compiler in determining TI variables for analysis. In the example from Figure 19, `#pragma TIV pid` specifies `pid` as a TI variable.

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

void SlaveProcedure(void *my)
  ......
```

Figure 18: Detecting TI variables passed as parameters.

Given the discovery of TI variables, we introduce the rules to perform the defined data classification as follows:

1. **Determine Private Data:** Private data can be assured if the data block is allocated after the parallel threads are created and the relevant pointers (pure/derived pointers associated with this block) are never assigned to any global pointers anywhere throughout the program.

```
#pragma TIV pid
pthread_mutex_lock(&(idlock));
 pid = Globalid;
 Globalid++;
pthread_mutex_unlock(&(idlock));
```

Figure 19: Identifying TI variables using directives.

2. **Determine Probably Private Data:** There are many types of data that are private but for some reason the compiler cannot guarantee this condition easily. For example, heap data blocks allocated after thread creation are identified as probably private if at least one pointer in the reference list is passed to a global pointer. Similarly, data on the stack is also classified into this category. We extend this definition to also include heap data blocks allocated before thread creation if pointers in the reference list are dereferenced with a TI variables. Typically, this indicates memory block that has been logically partitioned between the threads.

3. **Classify Shared Data:** Data that can not be identified as belonging to the above two categories is classified as shared.

To illustrate the operation of this compiler analysis, consider the example shown in Figure 20. The sample code on the left allocates data using `malloc()` and accesses the allocated data after multiple threads are forked. The analysis begins by constructing a CFG. As the CFG is being traversed, `malloc()` routines and pointer assignments are detected. Using the $Gen/Kill$ functions and data flow equations, reference lists are created and updated, as shown on the right hand side in Figure 20. Initially, the reference list set is empty. At the first `malloc()` call site (labeled as $malloc1()$), x is added into the reference list as a pure pointer. The next assignment $A = x$ adds A into the reference list that contains x, as indicated in Table 2. When x is reassigned by a second `malloc()` x is removed from the reference list with A and a new reference list is created to which B is added with the succeeding assignment statement.

When the conditional branch is encountered, C is added into both reference lists. After parallel threads are created, where pid has been discovered as a TI variable, C is used to reference one of the allocated locations with the subscript j. Since j, $first$ and $last$ are all TI variables, the memory blocks that C references according to the analysis (i.e. $malloc1()$ and $malloc2()$) from the statement $C[j] = j$ are classified as probably private, as described in data classification rule. The analysis is conservative, for a block to be labeled as private, all accesses to this block must be private, any probably private access overrides all private accesses and any shared access overrides any private or probably private accesses.

In the following section we describe how this data classification information can be passed to the runtime system and used with minimal intrusion and overhead to the runtime system and cache controllers.

3.4 DATA CLASSIFICATION IMPLEMENTATION

Data classification approach described in the previous section is implemented based on a series of compiler optimizations available in SUIF compiler infrastructure. As stated in Section 2.1, control and data flow analyses form a basis for many compiler optimizations. SUIF provides a framework called Sharlit [38] to facilitate the implementation of control and data flow analyses. Sharlit supports a style of optimization based on a model by Kildall, in which each phase consists of a data-flow analysis (DFA) of the program and an optimization function that transforms the program. To support the model, a set of abstractions or flow values, flow functions, path simplification rules, action routines are provided. Sharlit turns a DFA specification consisting of these abstractions into a solver for a DFA problem. At the heart of Sharlit is an algorithm called path simplification, an extension of Tarjan's fast path algorithm [37]. Path simplification unifies several powerful DFA solution techniques. By using path simplfication rules, one can construct a customized data-flow analyzers, from simple iterative ones, to solvers that use local analysis, interval analysis, or sparse data-flow evaluation by specifying data flow functions, actions before/after a statement and meet actions, which indicate the actions to be taken when multiple paths join. The *Kill* and *Gen* sets described in Section 3.3 are specified as data-flow functions and the Eq. (3.2) and Eq. (3.1) are specified as meet action.

Figure 21 is the definition of the control and data flow graph. It contains various member functions to manipulate on its nodes and obtain useful information. For example, *purepointer_kind_map* is a map data structure to store the information of pure pointers. The function *walk_for_purepointer_and_kind* collects all the pure pointers in the current graph and stores them in the data member *purepointer_kind_map*, which is an associate array.

The following code presents the implementation of a function to walk through all the SUIF IR structures recursively to collect pure pointers. To update reference list associated with memory allocation routines, it is also necessary to collect pointer assignments. In a similar manner the implemented compiler analyses also detect TI variables, which are used to determine the classification of memory accesses.

```
void fg::walk_for_purepointer_and_kind(tree_node *tn)
{
  switch(tn->kind())
    {
    case TREE_INSTR:
      {
instruction *i;
i = ((tree_instr *)tn)->instr();
        if(i->opcode() != io_cvt)
        return;
        in_rrr *the_cvt = (in_rrr *)i;
        operand cvtop = the_cvt->src_op();
        if(cvtop.kind()!=OPER_INSTR)
        return;
        instruction *thein = cvtop.instr();
        if(thein->opcode() == io_cal)
          {
            in_cal *ic = (in_cal *)thein;
            proc_sym *ps = proc_for_call(ic);
            if(!ps)
```

```
return;
if(strcmp(ps->name(),"malloc")==0)
 {
   operand dstop = ic->dst_op();
   switch(dstop.kind())
    {
      case OPER_NULL:
         return;
      case OPER_INSTR:

         return;
      case OPER_SYM:
        {
         sym_node *dstsn = dstop.symbol();
         switch(dstsn->kind())
   {
            case SYM_VAR:{
    var_sym *dstvs = (var_sym *)dstsn;
            purepointer_kind_map.associate(dstvs,PURE_HEAP);
            return;
                         }
    case SYM_PROC:
    case SYM_LABEL:
    return;
   }
         break;
        }
      default:
         return;
    }
```

```
            }
  }
       return;
    }



  case TREE_LOOP:
    {
tree_loop *loop = (tree_loop *)tn;
walk_for_purepointer_and_kind(loop->body());
walk_for_purepointer_and_kind(loop->test());
return;
    }

  case TREE_FOR:
    {
tree_for *loop = (tree_for *)tn;

/* treat these as being outside the loop */
walk_for_purepointer_and_kind(loop->lb_list());
walk_for_purepointer_and_kind(loop->ub_list());
walk_for_purepointer_and_kind(loop->step_list());
walk_for_purepointer_and_kind(loop->landing_pad());
   walk_for_purepointer_and_kind(loop->body());
return;
    }

  case TREE_IF:
    {
tree_if *if_node = (tree_if *)tn;
```

```
  walk_for_purepointer_and_kind(if_node->header());
  walk_for_purepointer_and_kind(if_node->then_part());
  walk_for_purepointer_and_kind(if_node->else_part());
  return;
      }

    case TREE_BLOCK:
      {
  tree_block *block = (tree_block *)tn;
  walk_for_purepointer_and_kind(block->body());
  return;
      }
    }
  assert(0);
}
```

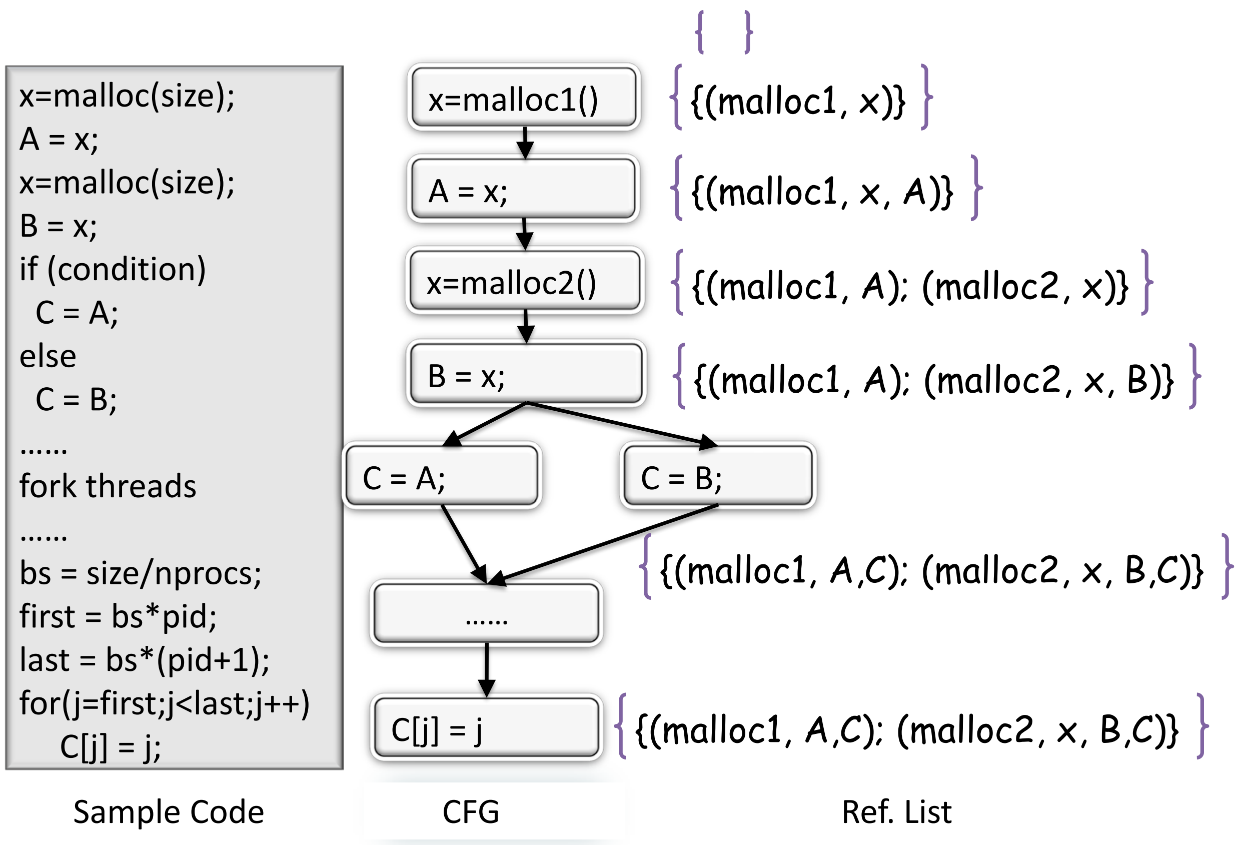

```
x=malloc(size);
A = x;
x=malloc(size);
B = x;
if (condition)
  C = A;
else
  C = B;
......
fork threads
......
bs = size/nprocs;
first = bs*pid;
last = bs*(pid+1);
for(j=first;j<last;j++)
    C[j] = j;
```
Sample Code
x=malloc1()
A = x;
x=malloc2()
B = x;
C = A;
C = B;
......
C[j] = j
CFG
{ }
{{(malloc1, x)}}
{{(malloc1, x, A)}}
{{(malloc1, A); (malloc2, x)}}
{{(malloc1, A); (malloc2, x, B)}}
{{(malloc1, A,C); (malloc2, x, B,C)}}
{{(malloc1, A,C); (malloc2, x, B,C)}}
Ref. List

Figure 20: Compiler analyses example for data classification

```
class fg
{
  Tree_to_fg_node tree_map;
  Variable_bit_index var_index_map;
  Pointerass_bit_index pointerass_map;
  Pointerass_list pointerass_list;
  Pure_pointer_to_kind purepointer_kind_map;
  void collect_all_pure(proc_sym *p);
  void collect_pointerass_info(instruction *);
  ......
  void walk_for_purepointer_and_kind(tree_node *);
  void collect_vars(instruction *in);
  void collect_referenced_vars(instruction *i, bit_set *, bit_set *);
  void walk_for_used_var_info(tree_node *, bit_set *, bit_set *);
public:
  proc_sym       *const procedure;
  fg(proc_sym *);
  /* These methods are for creating flow graphs from SUIF trees */
  int enter(fg_node *u) { return forward_graph->enter((CFGnode *)u); }
  fg_node *enter(fg_node *pred, fg_node *u);
  void analyze();
  inline void link(CFGnode *u, CFGnode *v)
  ......
  boolean is_pure(var_sym *);
  inline int pointerass_num() { return pointerass_list.hi; }
  int get_pointerass_index(instruction *);
  instruction *get_pointerass(int i) { return pointerass_list[i]; }
};
```

Figure 21: Source code that defines the control flow graph

4.0 SYSTEM SUPPORT

The basic mechanism to communicate the data classification from the compiler to the underlying cache architecture is through the page table in a similar manner as SOS [23]. The architecture accesses the data classification information through the TLB during address translation to minimize performance overhead. The following subsections describe this technique in details:

4.1 EXTENSIONS OF THE MEMORY ALLOCATOR AND PAGE TABLE

4.1.1 Customized Memory Allocator

Existing memory allocators such as `malloc()` obtain the starting address of the heap from the operating system (OS) for the requested data block and maintain a list to keep track of allocated as well as free memory blocks in virtual heap space. Whenever a block of memory with certain size is requested, the memory allocator traverses the list to find a free block of appropriate size using schemes such as best fit (block of closest size to the requested block) or first fit (first block on the list big enough to satisfy the request) depending on the size of the requested block. If there is not enough heap space available, the memory allocator calls the OS routine `mem_sbrk()` to expand the heap area.

To maintain the free/allocated block list, the allocator needs some data structure that allows it to distinguish block boundaries and distinguish between allocated and free blocks. Most allocators embed this information in the blocks themselves.

The newly allocated block is filled with some useful information in its first few bytes such as the size and pointer to the next block. This information is necessary to maintain the list. One

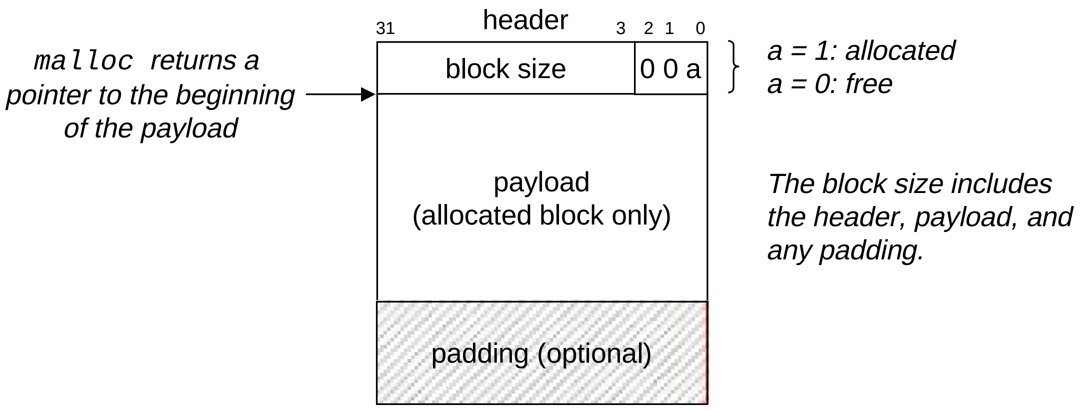

malloc returns a pointer to the beginning of the payload
header
31
3 2 1 0
block size
0 0 a
a = 1: allocated
a = 0: free
payload
(allocated block only)
The block size includes the header, payload, and any padding.
padding (optional)

Figure 22: Data blocks maintained by memory allocator

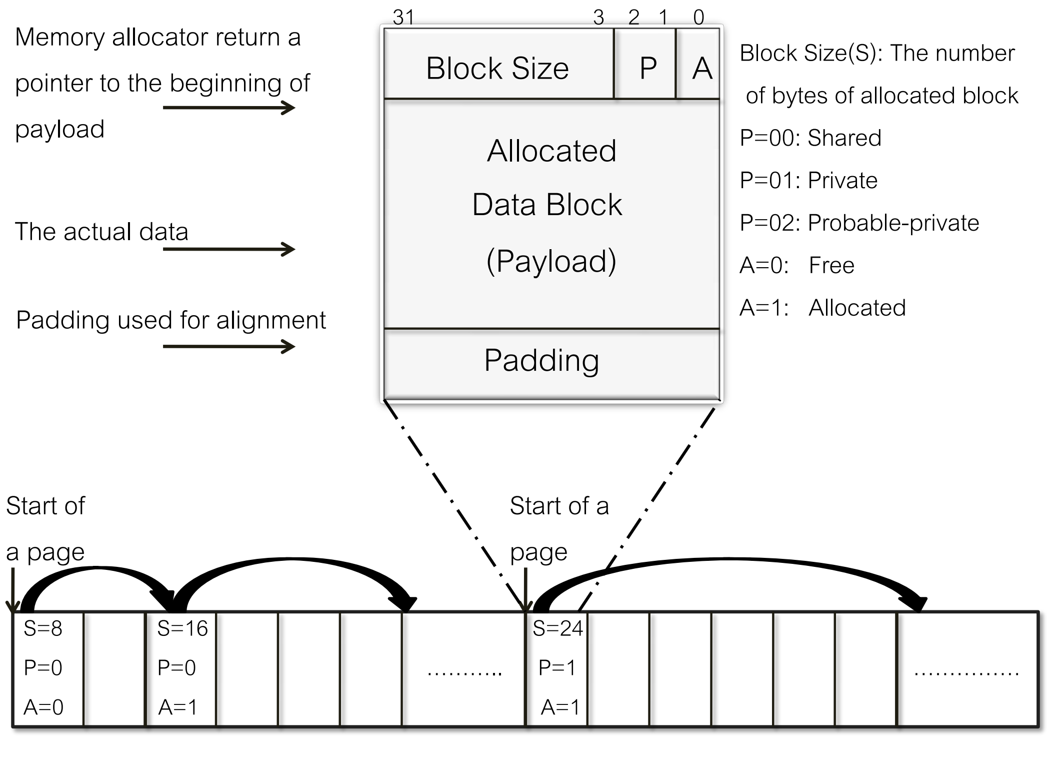


Figure 23: Data blocks maintained by modified memory allocator

simple approach is shown in Figure 22. In this case, a block consists of a one-word header, the payload, and possibly some additional padding. The header encodes the block size (including the header and any padding) as well as whether the block is allocated or free. If we impose a double-word alignment constraint, then the block size is always a multiple of eight and the three low-order bits of the block size are always zero. Thus, we need to store only the 29 high-order bits of the block size, freeing the remaining three bits to encode other information. In this case, we are using the least significant of these bits (the allocated bit) to indicate whether the block is allocated or free.

The memory allocator also optimizes the allocated blocks by reducing memory fragmentation through enforcing certain number of bytes alignment, splitting and coalescing blocks to improve space utilization, etc. A similar mechanism can be used to provide support for efficient sharing of information between the compiler and architecture. To inform the hardware of the data classifications identified by the compiler, we extend the prototype of current `malloc()` to: `void *malloc(size_t size, classification_t class);` The `size` parameter is retained from the original version of `malloc()` and is the size of the requested data block. The second parameter `pattern` is automatically filled by compiler with the data access pattern classification from Section 3.2. The memory allocator adds this classification parameter to the record associated with the newly allocated memory block in addition to the other parameters such as size, whether the block is actively allocated or has been freed, etc. In our modified `malloc()` we also extend the memory allocator to aggregate blocks with the same classification and to keep them page aligned. In other words, data blocks with different classifications are not permitted to be allocated within the same page. Figure 23 shows the structure maintained by our memory allocator. As illustrated in the figure, the first block starts at the beginning of a page and has $P = 0$ stored in the pattern field, indicating a shared classification. As stated before, the first block also contains a pointer pointing to the next block, which has a size of 16 and shared pattern. All blocks that follow these two blocks within the current page have the same value in its pattern field, namely, $P = 0$. The third block (with a size of 24) however, exhibits a private classification and thus is allocated in a new page.

4.1.2 Page Table Extension

4.1.2.1 A Standard Page Table/TLB Structure Page table is supported in almost all mainstream architecture and operating system. Take Intel Pentium and Linux as an example. Every Pentium system uses a two-level page table shown in Figure 24. The level-1 table, known as the page directory, contains 1024 32-bit page directory entries (PDEs), each of which points to one of 1024 level-2 page tables. Each page table contains 1024 32-bit page table entries (PTEs), each of which points to a page in physical memory or on disk. Each process has a unique page directory and set of page tables. When a Linux process is running, both the page directory and the page tables associated with allocated pages are all memory resident. A register called page directory base (PDBR) points to the beginning of the page directory. If the PTE is cached in the set indexed by the TLBI (a TLB hit), then the PPN is extracted from this cached PTE and concatenated with the VPO to form the physical address. If the PTE is not cached, but the PDE is cached (a partial TLB hit), then the MMU must fetch the appropriate PTE from memory before it can form the physical address. Finally, if neither the PDE or PTE is cached (a TLB miss), then the MMU must fetch both the PDE and the PTE from memory in order to form the physical address.

4.1.2.2 Modified Page Table/TLB To support data classification aware caching, the page table entries (and corresponding TLB entries) have been extended to store the data access classification information. When creating the page table entries, the OS consults the data access classification from the headers of the data block and fills each page table entry with two extra bits indicating the data access classification of the page. Since each PTE or PDE in a standard architecture already maintains a certain number of useful bits, as illustrated in Figure 25, the extra 2-bit classification information incurs almost no overhead (the reserved and unused bits in an existing configuration can be used).

When a memory access is initiated the virtual address is applied to the TLB to obtain the physical address. Once the physical address is obtained, returned with it is the data access classification associated with that page in memory. In regard to the data access classification information there is no difference between a TLB hit or a miss as on a miss the page fetched from memory will populate both the physical address in the tag as well as the data access pattern for that entry. This

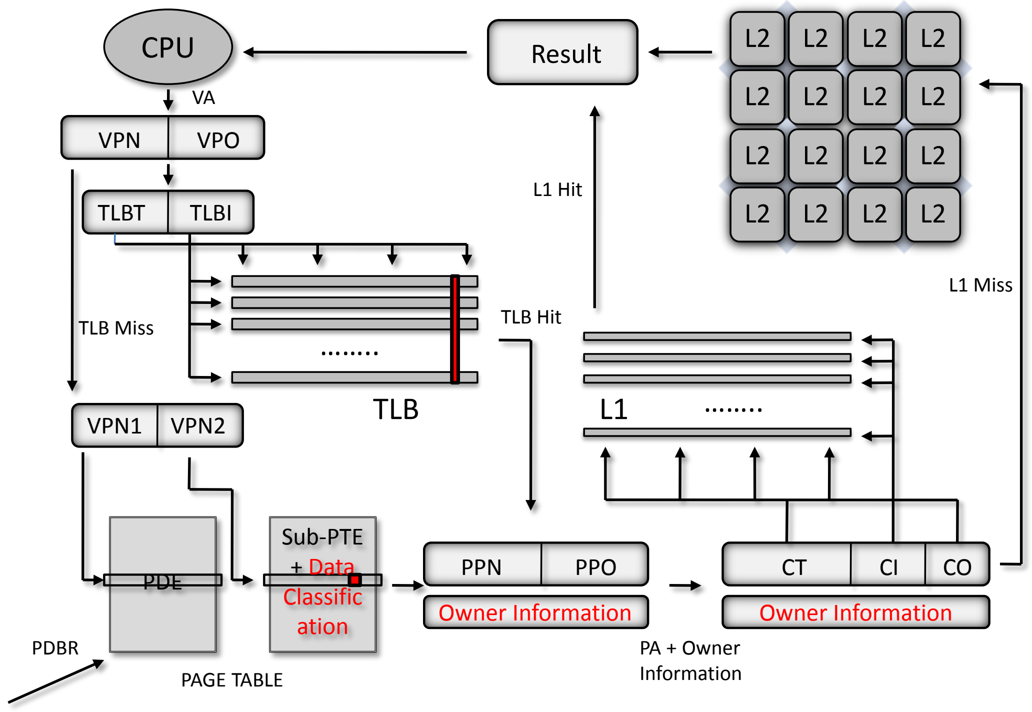


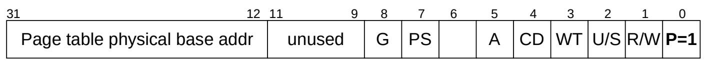
Figure 24: Page Table and TLB structure for Linux running on Intel Pentium

is illustrated in Figure 24 using red font.

In the following section we describe the cache architecture and the method for combining the private and shared access into the same cache.

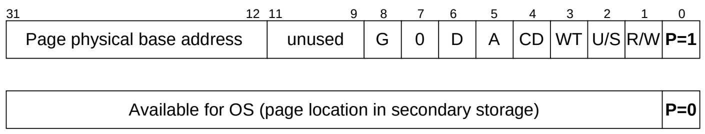
4.2 CACHE ARCHITECTURE

Unlike traditional private or shared caches, the proposed hybrid cache architecture provides mechanisms for caching both private and shared data. This can be achieved through a minimal set of changes to a baseline two-level cache organization. We assume the common case that L1 access is private and as such only on an L1 miss does the data access classification become relevant and the L2 access behavior depends on the data classification.



Field	Description
P	page table is present in physical memory (1) or not (0)
R/W	read-only or read-write access permission
U/S	user or supervisor mode (kernel mode) access permission
WT	write-through or write-back cache policy for this page table
CD	cache disabled (1) or enabled (0)
A	has the page been accessed? (set by MMU on reads and writes, cleared by software)
PS	page size 4K (0) or 4M (1)
G	global page (don't evict from TLB on task switch)
PT base addr	20 most significant bits of physical page table address

(a) Page Directory Entry (PDE).


31
12
11
9
8
7
6
5
4
3
2
1
0
Page physical base address
unused
G
0
D
A
CD
WT
U/S
R/W
P=1
Available for OS (page location in secondary storage)
P=0

Field	Description
P	page is present in physical memory (1) or not (0)
R/W	read-only or read/write access permission
U/S	user/supervisor mode (kernel mode) access permission
WT	write-through or write-back cache policy for this page
CD	cache disabled or enabled
A	reference bit (set by MMU on reads and writes, cleared by software)
D	dirty bit (set by MMU on writes, cleared by software)
G	global page (don't evict from TLB on task switch)
page base addr	20 most significant bits of physical page address

(b) Page Table Entry (PTE).

Figure 25: PTE/PDE structure

When a memory transaction is served, the cache controller consults the *pattern* field before communicating with the corresponding cache banks. This creates an illusion that both private and shared cache blocks are respected in their favored cache organizations, private and shared cache, respectively. A distributed directory is used to maintain the coherence among L1 private caches for shared data and among L2 hybrid caches for certain types of private data.

In particular, the placement and search policy after an L1 miss is described as follows:

- **Private:** For a private access, we directly check the local L2 cache tile. Upon a miss the data is directly obtained from main memory and added to the cache as if it were a private L2 scheme. Because the private data is only accessed locally, there is no need to maintain any cache coherence information, which saves considerable overhead.
- **Probably Private:** Probably private data is likely to be accessed as private data, thus the local core would retrieve the data from the local cache tile. However, because it is not guaranteed to be private it may be accessed by other cores. To ensure correctness and promote locality at the same time, we place data of this category within the local cache tile of the requester (e.g. first touch access), and adopt MESI protocol to maintain coherence, as performed in the traditional private cache organization. For data that is shared by two or more cores this can result in replication and reduced overall cache capacity. However, because the compiler has determined this is probably private we expect this type of sharing to be infrequent and not significantly harm overall capacity. In the best case latency would approach the speed of *private* access.
- **Shared:** Shared data is statically distributed throughout all the cache tiles as a function of its physical address. This type of data is very likely to be used by multiple processors. Thus, for an L2 cache access of shared data the cache bank to access is directly decoded from the physical address and a L2 miss results directly in access to main memory to retrieve the data. This keeps a unique copy at a fixed location to maximize effective cache capacity, simplifies data search and avoids the need to keep coherence at the L2 level (unlike private access) but still requires L1 coherence.

To support the hybrid caching scheme, each cache line (both in L1 and L2) is attached with an additional two-bit field, *pattern*. The field *pattern* is used to store the data access classification information obtained from the corresponding page table entry during address translation. While the classification can be directly obtained during address translation in the TLB for the current address being accessed if an entry must be evicted from the cache during an access the classification for the evicted block must be acquired from the cache directly. It may be possible to avoid the overhead of storing the access mode in the cache blocks with silent eviction and this is something we will study in our future work.

5.0 EVALUATION

In this section, we examine the effect of using the compiler to classify the data access modes and use this information to guide how data is handled in the cache. We first study the accuracy of our compiler-based data classification approach by comparing its results with classification based on run-time profiling. Second to evaluate the performance improvement obtained by the compiler directed caching, we compare our scheme with two baseline cache organizations, distributed shared and private, as well as the runtime page-level data classification mechanism from R-NUCA[1] [19] (see Section 2).

5.1 COMPILER-BASED DATA CLASSIFICATION

As a result of our proposed compiler-based data classification, memory access requests issued by applications can be either private, probably private, or shared[2]. To demonstrate the data-classification capability of the compiler analysis, we show in Figure 26, the the percentage of data accesses in each category during the application execution. Except for FFT, which is dominated by shared accesses, typically more than 50% of the accesses are some form of private access.

To determine the effectiveness of our classification we examined the accuracy of data classified as "probably private." Figure 27 shows the actual sharing behavior at run-time for the data that is identified as probably private by the compiler analysis. More than 63%, on average, of all probably private data was actually accessed privately. There is also a significant portion, ranging from 3.1%

[1]We simulate only the data access classification component of R-NUCA as the code page replication and clustering component is orthogonal to data classification and can be applied to many caching schemes including ours.

[2]We treat non-user space accesses such as OS-interventions, accesses to shared-libraries, and exceptions as shared accesses.

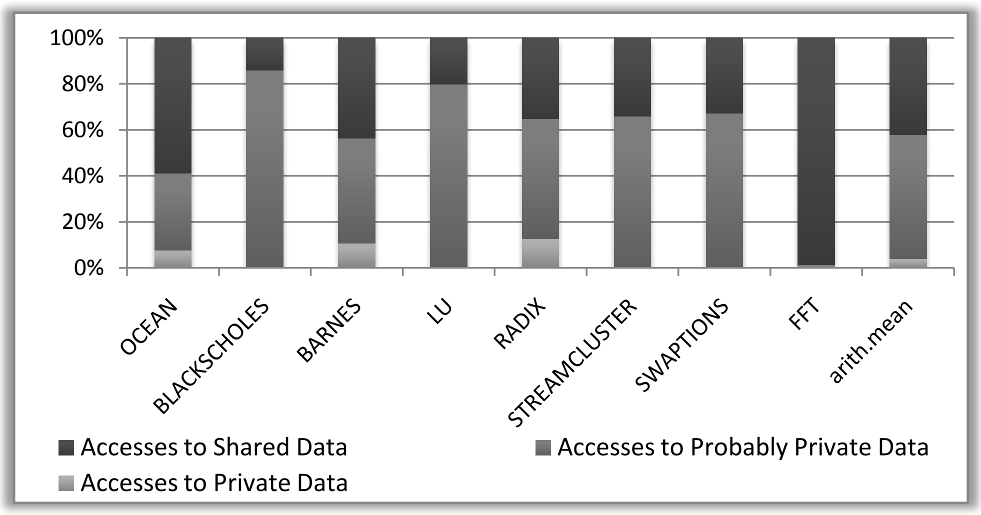


Figure 26: Percentage of accesses classified by the compiler as shared, probably private, and private.

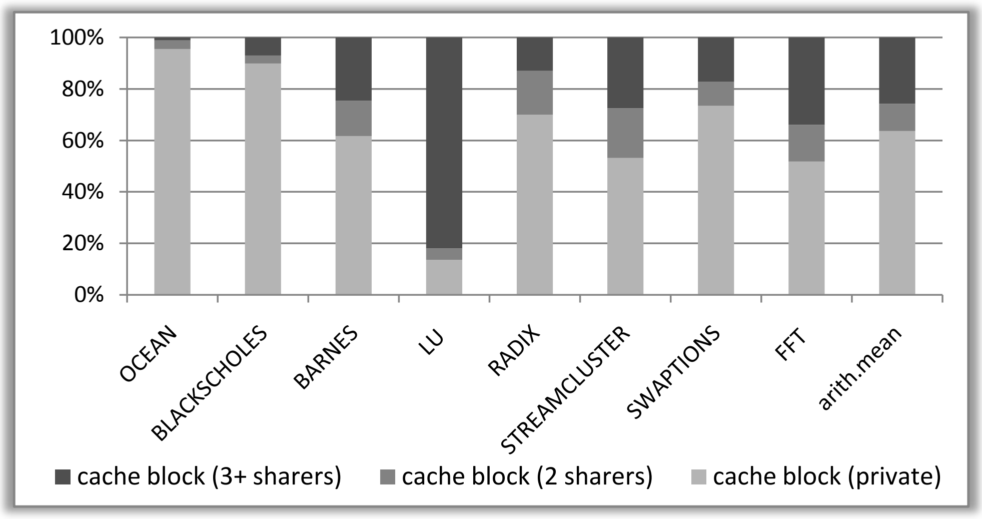


Figure 27: Proportions of data blocks classified as probably private that are accessed by one core (private), two cores, or three or more cores.

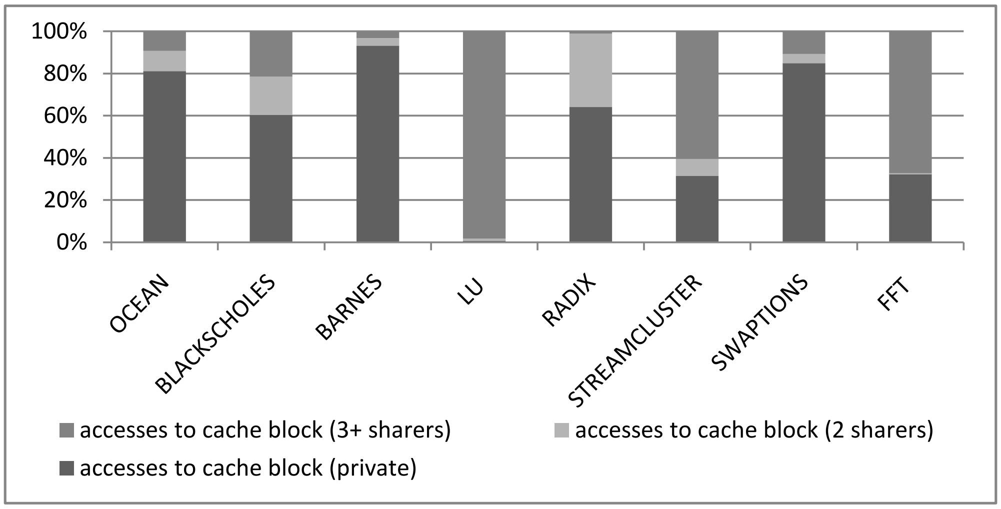


Figure 28: Proportions of the actual amount of accesses to private and shared data blocks for probably private data identified by the compiler

to 19.3%, of the data that has only has 2 sharers. If LU is removed for the equation as an outlier the average increases to almost 70% of probably private data being private. It is possible that LU and FFT would still benefit from the compiler analysis if the data shown as shared is heavily accessed by a single core. From our study of partitioning and sharing of data from [30], for LU in particular, we believe the probably private data from LU is heavily accessed by a single core and that private caching for this probably private data will still provide a benefit. Figure 28 shows the actual amount of accesses to data with different classifications. Similar conclusion can be drawn. In the next section we examine this in detail as we study the performance impact from this compiler classification used in the hybrid caching scheme.

5.2 SIMICS SIMULATOR

Architecture research relies largely on simulation because it is prohibitively expensive to build prototypes to demonstrate each architectural innovation. In order to produce reliable results when evaluating the experimental architectures, we adopt virtual Simics [32, 35], an execution driven full system simulation infrastructure which is capable of modeling a variety of mainstream architectures as well as devices. Supported architectures include PowerPC, Intel and AMD x86, SPARC, MIPS, ARM, Renesas, Texas Instruments, etc. Simics models the target system at the level of individual instructions, executing them one at a time It fully virtualizes the target computer, allowing simulation of multiprocessor systems as well as a cluster of independent systems, and even networks. The virtualization also allows Simics to be cross platform. For instance, Simics/SunFire can run on a Linux/x86 system, thus simulating a 64-bit big-endian system on a 32-bit little endian host. The analysis support includes code profiling and memory hierarchy simulation (i.e., cache hierarchies). Debugging support includes a wide variety of breakpoint types. The above features make Simics an ideal platform to perform study of experimental architecture modeling and simulation.

To model the experiment architecture, corresponding modules such as different levels of caches and network need to be added into Simics. In a typical modern micro-processor, there are several levels of caches. At the uppermost level are L1 caches, which usually have one I-cache for instruction and one D-cache for data. Simics use a splitter to partition requests between I-cache and D-cache. Below L1 is a larger, slower L2 cache. In many configurations, there is also an L3 cache either on-die or off-die.

The cache implementation in Simics uses many event-driven callbacks. This is necessary to model all the latencies and transactions involved in caches. When the processor needs to interact with the memory system, it creates a memory request. The type of memory request depends on the type of memory operation (e.g. read or write), the highest memory hierarchy (e.g. I-cache or D-cache), etc. The data itself is not actually modeled in the cache system, as we only care about the data flow and timing information. After the memory request is initialized, it is called to access the highest level cache object.

The cache object receives the requests from the upper level, and processes the requests according to their types. When the cache line is found and the requirements are satisfied, it schedules a callback to acknowledge the upper level object in the future, according to the cache port's availability. If a cache miss occurs, the memory request goes down to the next level cache to fetch the data.

One key issue in a shared cache multiprocessor is the cache consistency model which defines the cache coherence. As L1 caches are private, different processors may see different values of the same memory location. Therefore, cache coherence is mandatory in the system. Snoopy bus is a popular cache coherence policy in small scale multiprocessor systems. It has a global bus which connects all caches and memory banks. All memory addresses are broadcast on that global bus. All caches and memory banks "snoop" (listen to) that bus. The operations require broadcast and bus is a natural broadcast medium.

We adopt MESI based write-invalidate, write-back protocol. In a write-invalidate policy, the writing processor forces all other caches to invalidate their copies in a write operation. It produces less network traffic than a write-update policy, in which the writing processor forces all others to update their copies. In a write back protocol, the memory is updated only when the block in the cache is being replaced. It produces less bus traffic than a write-through policy, in which the memory is updated every time the cache is updated. The coherence state machine is similar to one shown in Figure 13.

5.3 PERFORMANCE EVALUATION

To evaluate the performance impact of our hybrid caching using compiler-based data classification (HCDC) we compared our approach with both distributed shared [27] and private [9] caches as well as the state of the art runtime caching method that leverages data access classification, R-NUCA [19]. In this section we examine and compare cache miss rate, average memory access latency, and speedup for these caching schemes.

Table 3: Simulation Configurations

Benchmark	Shared-averse configuration	Private-averse configuration
BLACKSCHOLES	20000 options; 16M L2	200000 options; 8M L2
SWAPTIONS	64 swaptions; 16M L2	512 swaptions; 8M L2
BARNES	524288 particles; 16M L2	1048576 particles; 8M L2
OCEAN	1026x1026 matrix; 16M L2	2050x2050 matrix; 8M L2
LU	1024x1024 matrix; 16M L2	4096x4096 matrix; 8M L2
RADIX	10485760 radix; 16M L2	104857600 radix; 8M L2
STREAMCLUSTER	1024 data points; 16M L2	1024 data points; 8M L2
FFT	$2^2$6 even integers; 16M L2	$2^2$6 even integers; 8M L2

To conduct our experiments, we use Simics [32] as our simulation environment and implemented the relevant caching schemes within it. As previously mentioned, for R-NUCA we focused on the features relevant to the comparison, namely, the classification of data as shared or private at the page level. 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 75 ns (150 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 with the hit latency of 1 cycle.

For the workload, we examined benchmarks from Splash-2 and Parsec-2.0. Splash-2 benchmark suite is a popular choice to with typical parallel applications that use a rich set of data access patterns. Following is a brief description of the benchmarks we use and their characteristic.

- FFT - The FFT kernel is a complex 1-D version of the radix- sixstep FFT algorithm, which is optimized to minimize inter-processor communication. Data sets are organized $\sqrt{N} \times \sqrt{N}$as matrices partitioned so that every processor is assigned a contiguous set of rows which are allocated in its local memory. Every processor transposes a contiguous submatrix of $\sqrt{N}/p \times \sqrt{N}/p$ from every other processor, and transposes one submatrix locally. Communication require all-to-all inter-processor communication.
- LU - The LU kernel factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The dense matrix A is divided into an $N \times N$ array of $B \times B$ blocks ($n = NB$) to exploit temporal locality on submatrix elements. To reduce communication, block ownership is assigned using a 2-D scatter decomposition, with blocks being updated by the processors that own them. Communications include one-to-many, many-to-one traffic.

- RADIX - The integer radix sort kernel, The algorithm is iterative, performing one iteration for each radix r digit of the keys. In each iteration, a processor passes over its assigned keys and generates a local histogram. The local histograms are then accumulated into a global histogram. Finally, each processor uses the global histogram to permute its keys into a new array for the next iteration. This permutation step requires all-to-all communication.
- OCEAN - The Ocean application studies large-scale ocean movements based on eddy and boundary currents. It partitions the grids into square-like subgrids to improve the communication to computation ratio. The grids are conceptually represented as 4-D arrays, with all subgrids allocated contiguously and locally in the nodes that own them. It needs nearest neighbor interactive communication.
- BARNES - This application simulates the evolution of a system of bodies under the influence of gravitational forces. It is a classical gravitational N-body simulation, in which every body is modeled as a point mass and exerts forces on all other bodies in the system. The simulation proceeds over time-steps, each step computing the net force on every body and thereby updating that bodys position and other attributes. By far the greatest fraction of the sequential execution time is spent in the force computation phase.
- BLACKSCHOLES - The BLACKSCHOLES application is an Intel RMS benchmark. It calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation [8]. There is no closed form expression for the Black-Scholes equation and as such it must be computed numerically. The blackscholes benchmark was chosen to represent the wide field of analytic PDE solvers in general and their application in computational finance in particular. The program is limited by the amount of floating-point calculations a processor can perform.
- STREAMCLUSTER For a stream of input points, it finds a predetermined number of medians so that each point is assigned to its nearest center. The quality of the clustering is measured by the sum of squared distances (SSQ) metric. Stream clustering is a common operation where large amounts or continuously produced data has to be organized under realtime conditions, for example network intrusion detection, pattern recognition and data mining. The program spends most of its time evaluating the gain of opening a new center. This operation uses a parallelization scheme which employs static partitioning of data points. The program is mem-

ory bound for low-dimensional data and becomes increasingly computationally intensive as the dimensionality increases. Due to its online character the working set size of the algorithm can be chosen independently from the input data. streamcluster was included in the PARSEC benchmark suite because of the importance of data mining algorithms and the prevalence of problems with streaming characteristics.

- SWAPTIONS - The swaptions application is an Intel RMS workload which uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions. The HJM framework describes how interest rates evolve for risk management and asset liability management for a class of models. Its central insight is that there is an explicit relationship between the drift and volatility parameters of the forward-rate dynamics in a noarbitrage market. Because HJM models are non-Markovian the analytic approach of solving the PDE to price a derivative cannot be used. Swaptions therefore employs Monte Carlo (MC) simulation to compute the prices. The workload was included in the benchmark suite because of the significance of PDEs and the wide use of Monte Carlo simulation.

As the size of the dataset can bias the results toward a particular type of cache, one that favors private (i.e. a small dataset/working that can easily fit into the cache even with significant replication) and one that favors shared (a larger dataset/working set where cache capacity is at a premium). However, the problem size of our simulations was limited due to the intractably long simulation times of large workloads. Thus, to simulate these two conditions, we use two configurations, shared-averse and private-averse. In shared-averse configuration, the aggregate L2 cache capacity is configured as 16M bytes which favors private caches for the size of data sets available in the benchmark suite. To simulate a private-averse system we reduced the cache size to 8M bytes to make the overall capacity a bigger factor. To further distinguish these configurations during our runs we also used different working set sizes for the tested benchmarks, using a larger size for private-averse configuration and a smaller size for shared-averse. These sizes are described in Table 3.

In the next several sections we examine the behavior HCDC compared to distributed shared, private, R-NUCA caches for cache miss rate, data access latency, and performance, respectively.

5.3.1 Miss Rate

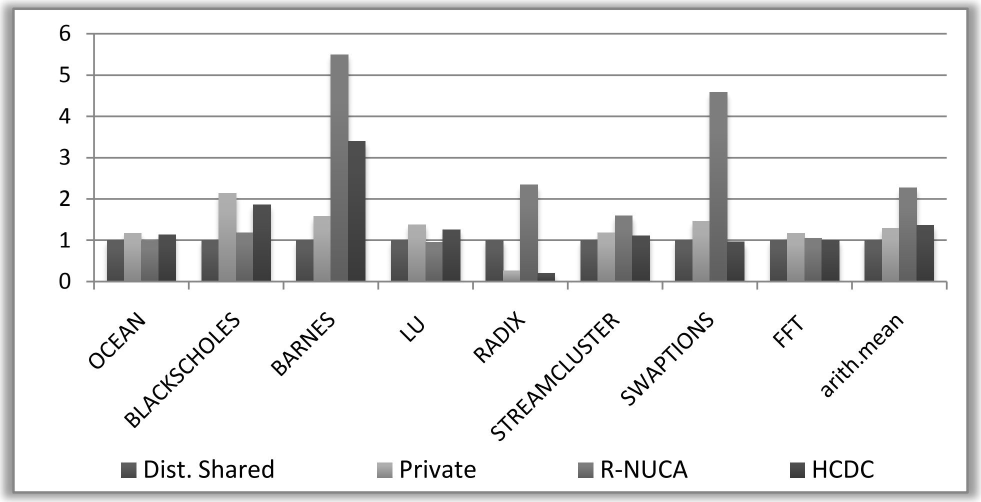


Figure 29: Cache miss rate for shared-averse configuration

Figure 29 and Figure 30 show the L2 cache miss rate for shared-averse and private-averse configurations, respectively (each normalized to the distributed shared cache). In general, distributed shared caches have the lowest miss rate because it does not allow replication and as such uses the cache capacity most effectively. Conversely, in the private cache organization, multiple cache blocks become replicated and consume more capacity, typically resulting in a higher miss rate than the shared cache. Even for the shared-averse configuration, private caches still suffer from a slightly higher miss rate, although this effect is obviously attenuated.

Somewhat surprisingly, R-NUCA has an undesirable performance in miss rate due to the page re-classification mechanism it adopts. When a page initially classified as private is re-classified to shared status, all the cache blocks within the page that have been cached must be invalidated, resulting in high miss rate.

The miss rate of the proposed HCDC is typically in the middle and often approaches the better of the two baseline caching schemes based on the conditions. For shared-averse, HCDC is better than shared and approaches the quality of private. For private averse, HCDC is better than private and approaches the quality of shared.

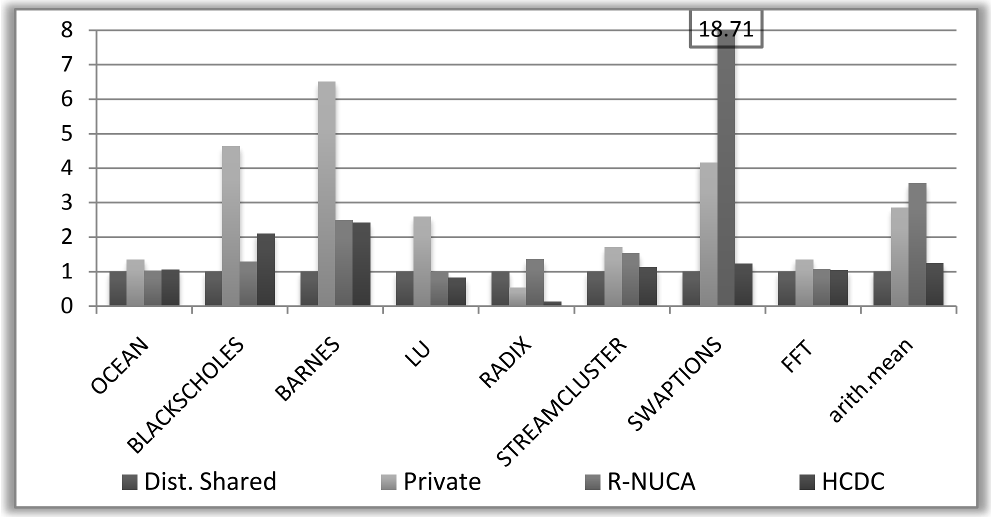


Figure 30: Cache miss rate for private-averse configuration

We believe this is due to a fundamental advantage of our HCDC scheme. We allow replication for the probably private data identified by the compiler which keeps data that is private but may be infrequently accessed by a sharer local to the core that accesses it. Moreover, for data that is heavily shared, we keep only copy of the data for all sharers which is good for effective use of available cache capacity. Additionally, the miss rate of HCDC is not as sensitive to working set size as private caches as seen in Figures 29 and 30 because private caches will replicate data regardless of the number of sharers and will not effectively utilize the cache when space is at a premium. Furthermore, in our experience, the more heap data utilized by the application, the higher proportion of data that is labeled as private or probably private, however, there is still some increase in the amount of shared data as well. Thus, as the working set increases, HCDC's efficiency allows more of the private/probably private data to be cached making it scale better than private caches, which waste a disproportional amount of cache capacity with shared data that has been highly replicated.

5.3.2 Latency

Average memory access latencies of all relevant schemes are reported in Figure 31 and Figure 32. Data access latency is affected by both miss rate and on a hit, the distance that must be traversed to retrieve the data from a potentially remote tile. In distributed shared caches, most data is stored in a remotes tile from the core that heavily accesses it, resulting in a higher latency, especially in a shared-averse configuration. In contrast, private caching absorbs all the data to the local tile and thus has a lower hit latency because off tile cache accesses are minimized. This is true especially when the working set size does not exceed the cache capacity, as shown in Figure 31 for a shared-averse configuration. As the cache capacity is pressured by increasing working load, the latency is dominated by off-chip misses and the performance begins to degrade, as demonstrated in Figure 32.

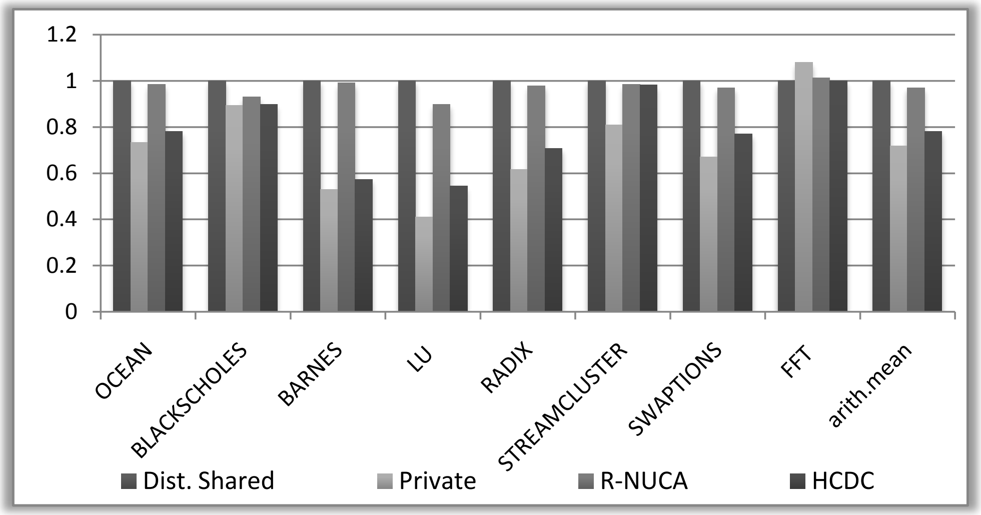


Figure 31: Average memory access latency for shared-averse configuration

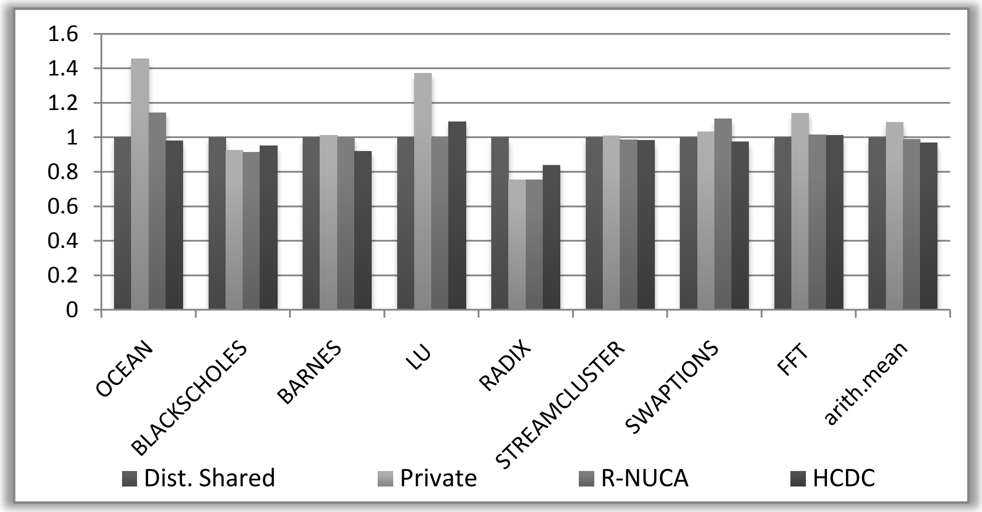


Figure 32: Average memory access latency for private-averse configuration

R-NUCA data classification also suffers partly from high access latency similar to distributed shared caches when the pages are classified as shared, although it reduces the access latency when pages are initially accessed locally or remain private to a particular processor. Another problem of R-NUCA data classification is the page granularity makes it impossible to optimize smaller memory blocks. One byte of shared access in a private page results in the whole page being reclassified as shared. Additionally, R-NUCA is an "all-or-nothing" approach. For even a single access by a second core the page is classified as shared even if this is a uncommon or one time occurrence and a private style scheme will still save considerable data access latency.

Our compiler-assisted caching addresses these problems through customized placement policies for classified data, packing a page with data of the same classification (see Figure 23), and for tolerating a stray shared access in a probably private configuration. For the shared-adverse configuration, HCDC performs better than distributed shared and R-NUCA and for private-adverse configuration, it outperforms distributed shared, private and R-NUCA by a factor of 3.02%, 8.43%, 1.24%, respectively.

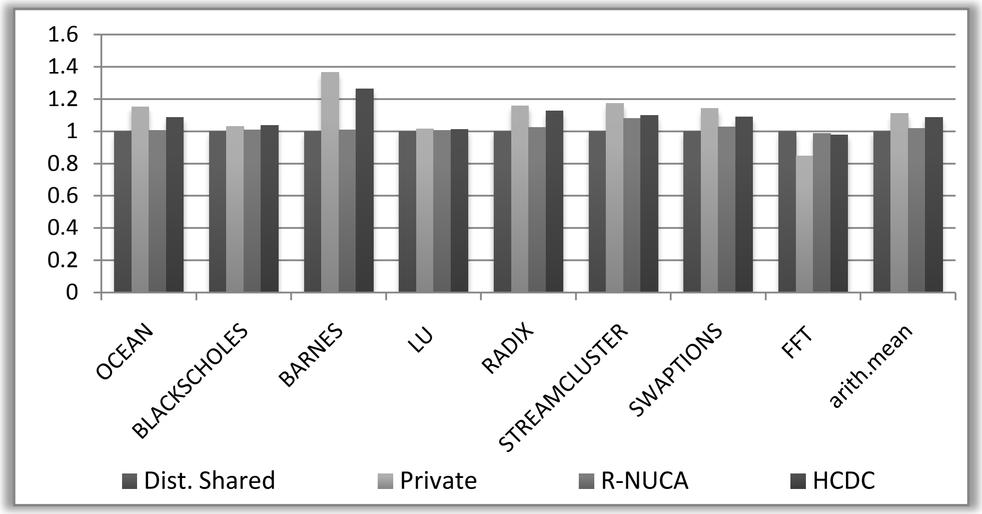


Figure 33: Speedup for shared-averse configuration

5.3.3 Performance Improvement

As data access latency is an important factor in overall application performance [2, 30] we expect the trends from the previous section to be an indicator of application performance for the various caching schemes. From Figure 33 we can see that the private gives the best performance result for smaller working set sizes with a large cache capacity (shared-adverse configuration). Our scheme HCDC closely follows the speedup for private cache. In contrast, Figure 34 indicates that for large working set sizes, the performance of private caching degrades significantly while the distributed shared caching exhibits a large improvement due to efficient utilization of cache capacity. HCDC has similar cache utilization efficiency as distributed shared, and thus performs as well as shared caching in many cases. For some benchmarks, such as OCEAN, BARNES and STREAMCLUSTER, HCDC yields even better results than shared caching because it attracts probably private data to its local tile. For both configurations on average, we achieve 5.75%, 6.61% and 4.30% speedups over distributed shared, private and R-NUCA organizations, respectively.

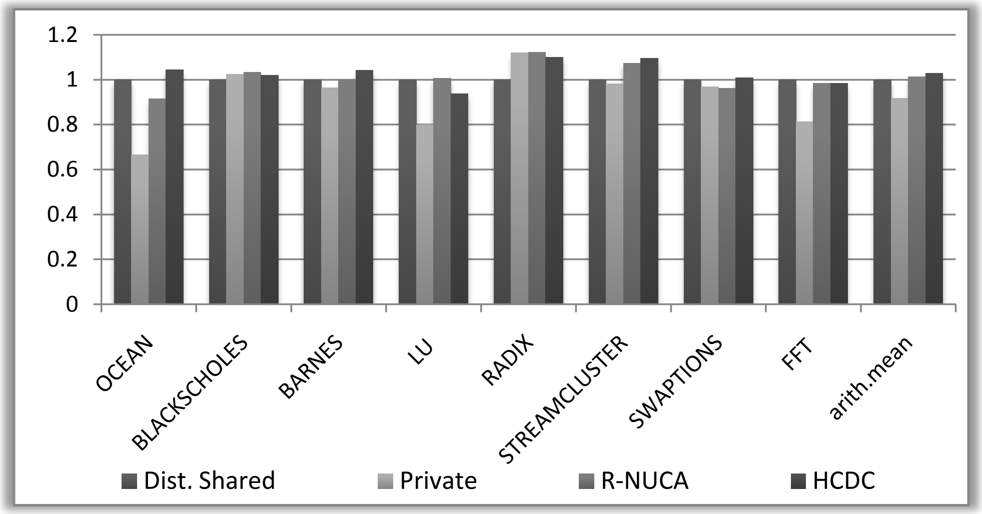


Figure 34: Speedup for private-averse configuration

6.0 CONCLUSIONS

In this thesis we have presented a compiler-assisted hybrid caching scheme which is aware of different data access classifications of private and shared. In addition, due to necessity of simplifying our compiler analysis, we introduce a data classification of “probably private” where the compiler identifies the data is private but cannot guarantee this condition. We have proposed a cache uses a hybrid of private and S-NUCA caching directed by the compiler classification to use distinct placement and search policies for the identified classifications so as to reduce the remote data access penalty (memory access latency) while increasing the efficiency of the cache capacity utilization. Comparing with other cache mechanisms, in particular R-NUCA in addition to traditional private and shared caches, our HCDC cache is unique as it utilizes the data pattern classification available at compile time, thus, eliminating the need for potentially expensive runtime profiling. Additionally, we also employ a modified memory allocator to group data with same access classifications into a page. This exposes more opportunities to perform optimizations on data of distinct access patterns and reduces inefficiencies of classification at such a large data granularity.

To pass the data classification from compiler to run-time system, a modified page table structure is adopted. Each entry of the page table is associated with additional bits to accommodate the data classification information, which is retrieved by the memory controller during the virtual-physical address translation process.

The described techniques have been implemented in the execution-driven full system simulator Simics. We also implemented the relevant scheme R-NUCA as well as two baseline cache organizations(i.e. private and distributed shared). To perform a fair evaluation without being biased towards either distributed shared or private caching scheme, two sets of experiments have been conducted to evaluate the miss rate, average memory access latency and speedup. Our results show that using the compiler analysis to classify the data and to guide the data placement in the cache results in better than 4% improvement over the state of the art runtime technique to classify data accesses as private and share and to leverage this in a hybrid cache, R-NUCA.

7.0 FUTURE DIRECTIONS

Some of the future directions include expanding our analysis to determine the weight of private access for shared or probably private data classifications. Because private access provides the potential for significant latency reduction, we plan to explore a more aggressive compiler analysis based on memory access pattern analysis of multi-threaded applications. We hope to move some of the probably private data into a guaranteed private data classification. This requires more aggressive compiler analyses and will further increase the data access locality of multi-threaded applications.

Another potential direction is to carry out a more precise data access pattern analysis, rather than just data classification. We have already studied some multi-threaded applications and found interesting patterns that are used commonly in data-parallel programs. Data access patterns provide more detailed information (e.g. how multiple threads/processor partition and utilize data) than data classification thus expose better opportunities for data placement and lookup in CMPs. Further more, it forms a basis for communication pattern extraction, which could also be a field for future study. Achieving this however, is especially challenging because the multiprocessor interactions are transparent to the programmer, which means the programmer is not aware of the underlying architecture. The communications among threads incurs implicitly, usually implied by applicationsḿemory access pattern, in shared memory programming model, which adds more complexities to the analyses. However, the initial research on this topic shows many of the memory access patterns can be still obtained using aggressive compiler analysis.

The scalability of the proposed technique to systems with larger number of processing cores could also be studied in the future. Since the remote data access latency is likely to increase with the scaling number of processors in CMPs, intelligent data placement and caching is expected to have higher impact.

Finally, we hope to explore some new hybrid caching schemes designed specifically to take advantage the compiler analysis to improve latency and capacity such as use of a center of gravity style placement or even a compiler assisted placement [30] for shared data and no-replication private for probably private data.

BIBLIOGRAPHY

[1] [Online]. Available: http://suif.stanford.edu/suif/suif1/docs/suif_toc.html

[2] A. Abousamra, R. Melhem, and A. K. Jones, "Winning with pinning in NoC," in *Proc. of Hot Interconnects (HOTI)*, 2009.

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*, 2nd ed. Addison Wesley, 2006.

[4] J. M. Arnold, D. A. Buell, and E. G. Davis, "Splash 2," in *SPAA '92: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 1992, pp. 316–322.

[5] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter, "Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches," in *IEEE International Symposium on High Performance Computer Architecture*, 2009.

[6] R. Barua, D. Kranz, and A. Agarwal, "Communication-minimal partitioning of parallel loops and data arrays for cache-coherent distributed-memory multiprocessors," in *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*, 1996.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," Princeton University, Tech. Rep. TR-811-08, January 2008.

[8] F. Black and M. S. Scholes, "The pricing of options and corporate liabilities," *Journal of Political Economy*, vol. 81, no. 3, pp. 637–54, May-June 1973. [Online]. Available: http://ideas.repec.org/a/ucp/jpolec/v81y1973i3p637-54.html

[9] J. A. Brown, R. Kumar, and D. M. Tullsen, "Proximity-aware directory-based coherence for multi-core processor architectures," in *SPAA*, 2007, pp. 126–134.

[10] D. B. C. Kim and S. W. Keckler, "Nonuniform cache architectures for wire-delay dominated on-chip caches," *IEEE Micro*, vol. 23, no. 6, pp. 99–107, 2003.

[11] J. Chang and G. S. Sohi, "Cooperative caching for chip multiprocessors," in *The 33rd International Symposium on Computer Architecture*, 2006.

[12] Z. Chishti, M. D. Powell, and T. N. Vijaykumar, "Optimizing replication, communication, and capacity allocation in cmps," in *ISCA*, 2005, pp. 357–368.

[13] M. Chu and S. Mahlke, "Compiler-directed data partitioning for multicluster processors," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2006.

[14] M. Chu, R. Ravindrany, and S. Mahlke, "Data access partitioning for fine-grain parallelism on multicore architectures," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[15] H. Dybdahl and P. Stenstrom, "An adaptive shared/private nuca cache partitioning scheme for chip multiprocessors," in *Proceedings of International Symposium on High Performance Computer Architecture*, 2007.

[16] M. Gupta and P. Banerjee, "Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, no. 2, pp. 179–193, 1992.

[17] M. R. Haghighat and C. D. Polychronopoulos, "Symbolic analysis for parallelizing compilers," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 4, pp. 477–518, 1996.

[18] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam, "Maximizing multiprocessor performance with the suif compiler," *Computer*, vol. 29, pp. 84–89, 1996.

[19] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "R-nuca data placement in distributed shared caches," in *Computer Architecture Lab at Carnegie Mellon Technical Report*, 2009.

[20] ——, "Reactive nuca: near-optimal block placement and replication in distributed caches," in *ISCA*, 2009, pp. 184–195.

[21] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler, "A nuca substrate for flexible cmp cache sharing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 8, pp. 1028–1040, 2007.

[22] R. K. J. A. Brown and D. M. Tullsen, "Proximity-aware directory based coherence for multicore processor architectures," *SPAA*, pp. 126–134, 2007.

[23] L. Jin and S. Cho, "Sos: A software oriented distributed shared cache management approach for chip multiprocessors," in *Intl Conference on Parallel Architectures and Compilation Techniques PACT*, 2009.

[24] A. K. Jones, S. Shao, Y. Zhang, and R. Melhem, "Symbolic expression analysis for compiled communication," *Parallel Processing Letters*, vol. 18, no. 4, pp. 567–587, December 2008.

[25] Y. Ju and H. Dietz, "Reduction of cache coherence overhead by compiler data layout and loop transformation," *In Languages and Compilers for Parallel Computing*, pp. 344–358, 1992.

[26] C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[27] ——, "Nonuniform cache architectures for wire-delay dominated on-chip caches," *IEEE Micro*, vol. 23, no. 6, pp. 99–107, 2003.

[28] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-way multithreaded sparc processor," *IEEE Micro*, vol. 2, no. 25, pp. 21–29, 2005.

[29] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the dash multiprocessor," in *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*. New York, NY, USA: ACM, 1990, pp. 148–159.

[30] Y. Li, A. Abousamra, R. Melhem, and A. K. Jones, "Compiler-assisted data distribution for chip multiprocessors," in *PACT '10: Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. New York, NY, USA: ACM, 2010, pp. 501–512.

[31] Q. Lu, C. Alias, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Y. Chen, H. Lin, and T.-f. Ngai, "Data layout transformation for enhancing data locality on nuca chip multiprocessors," in *PACT '09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 348–357.

[32] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, February 2002.

[33] Y. Paek, E. Z. A. Navarro, J. Hoeflinger, and D. Padua, "An advanced compiler framework for noncache-coherent multiprocessors," in *IEEE Trans. Parallel and Distributed Systems*, vol. 3, no. 3, Mar. 2002, pp. 241–259.

[34] Ramanujam and P. Sadayappan, "Compile-time techniques for data distributionin distributed memory machines," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 2, no. 4, pp. 472–482, 1991.

[35] "Simics", http://www.virtutech.com/.

[36] P. Sweazey and A. J. Smith, "A class of compatible cache consistency protocols and their support by the ieee futurebus," in *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1986, pp. 414–423.

[37] R. E. Tarjan, "Fast algorithms for solving path problems," *J. ACM*, vol. 28, no. 3, pp. 594–614, 1981.

[38] S. W. K. Tjiang and J. L. Hennessy, "Sharlit—a tool for building optimizers," in *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*. New York, NY, USA: ACM, 1992, pp. 82–93.

[39] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarsinghe, J. M. Anderson, S. W. K. Tjiang, S. W. Liao, C. W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "Suif: An infrastructure for research on parallelizing and optimizing compilers," *ACM SIGPLAN Notices*, vol. 29, no. 12, pp. 31–37, December 1994.

[40] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarsinghe, J. M. Anderson, S. W. K. Tjiang, S. W. Liao, C. W. Tseng, M. W. Hall, M. s. Lam, and J. L. Hennessy, "Suif: An infrastructure for research on parallelizing and optimizing compilers," in *SIGPLAN Notices*, 1994.

[41] Z.Chishti, M. D. Powell, and T. N. Vijaykumar, "Optimizing replication, communication, and capacity allocation in cmps," in *Intl Symp. Computer Arch.*, 2005.

[42] M. Zhang and K. Asanovic, "Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors," in *32nd Annual International Symposium on Computer Architecture*, 2005.