Performance Evaluation of Memory Allocation Schemes on CC-NUMA Multiprocessors *

Trung N. Nguyen Zhiyuan Li Jian Huang Guohua Jin Dongsoo Kim

Contact email: li@cs.umn.edu

Abstract – Cache Coherent Non-Uniform Memory Access (CC-NUMA) architectures have received strong interests from both academia and industries. This paper studies the performance impact of design choices at different levels of address and memory mapping on CC-NUMA architectures. Through execution-driven simulations of five numerical programs, we find close interactions between data allocation, global address translation and cache set-addressing. Our results strongly discourage the use of direct-mapped caches in CC-NUMA machines. Results also show that data allocation often makes a great impact on memory miss ratio and execution time. A compiler scheme which allocates data and parallel tasks simultaneously is shown to perform quite well consistently.

Keywords – CC-NUMA, Data Allocation, Cache Mapping, Memory Management, Parallel Task Allocation

1 Introduction

In recent years, we see a trend toward the Cache Coherent Non-Uniform Memory Access (CC-NUMA) architecture in multiprocessors. Systems based on this architecture includes research prototypes such as the Stanford DASH[8] and FLASH[7], MIT Alewife[2], University of Toronto NUMAchine[19], and Sun's

*This work was supported in part by NSF CAREER Award CCR-9502541, by the Army High Performance Computing Research Center, under the auspices of Army Research Office contract number DAAL03-89-C-0038 with the University of Minnesota, and by the U.S. Army Intelligence Center and Fort Huachuca under Contract DABT63-95-C-0127 and ARPA order no. D346. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U. S. Army Intelligence Center and Fort Huachuca, or the U.S. Government. Additional support is provided by a donation from Cray Research Inc. and by an instrumentation grant from NSF, CDA 9414015.

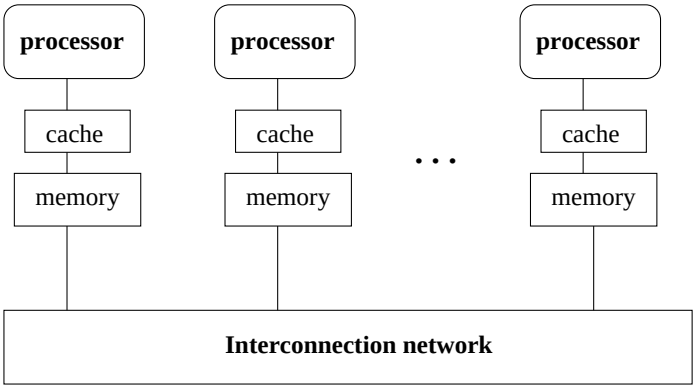


Figure 1: Example of a CC-NUMA system.

S3.mp[15], as well as commercial products including the Sequent STiNG[12] and Hewlett-Packard SPP[6]. On a CC-NUMA machine (*c.f.* Figure 1), an interconnection network connects a number of nodes, each of which consists of one or a few processors, a cache private to the node, and a local memory accessible remotely by other nodes. The ensemble of the local memories at all nodes forms a global address space. Cache coherence is maintained by a directory-based protocol. While a CC-NUMA machine may scale to several thousands of nodes, systems containing a small number of nodes have also appeared in the market place. The Sequent STiNG, e.g., has four nodes, each consisting of four processors. With the processor speed, and hence the memory reference rate, rapidly increasing (we are soon to see processors clocked at 500 MHZ or higher), one may see even smaller CC-NUMA machines replacing some of the bus-based ones due to the potentially heavy bus collision.

Memory latency is a well-known issue for CC-NUMA machines. This paper studies the impact of memory allocation on the cache hit ratio, the local memory hit ratio and the execution time. We regard memory allocation on CC-NUMA as a process which takes the following steps:

1. *Cache mapping.* Cache mapping refers to the mapping of a physical address to a location in a private cache. A design choice made at this level is the most difficult to change, especially if commodity microprocessors are used in the machine and the cache is on the processor chip.

2. *Memory module mapping.* This step determines which of the p memory modules corresponds to a given physical address. This is usually accomplished by designating a set of $n = log_2(p)$ bits in the physical address to identify the node which contains the memory module. For convenience, we assume in the

rest of the paper that each node contains one processor. We call those n identifying bits the *processor ID (pid) bits*. Compared to cache mapping, this step of memory mapping is easier to change as the modification is at the system level, involving only the memory management hardware and possibly the network routing.

3. *Virtual to physical address mapping.* This mapping is typically done by the paging subsystem of the operating system which maps a virtual page to a physical page.

4. *Program data to virtual address mapping.* When a compiler generates an executable object, it assigns virtual addresses to the data referenced in the program and also makes sure that the instructions use the correct addresses for data references. This step of mapping is the most flexible.

As we shall see later in this paper, the above steps closely interact with each other. The pid bits play a key role here. Their designation in the physical address can greatly affect the cache hit ratio. Once the pid bits are designated, the compiler should assign virtual addresses and manipulate the data references in such a way as to increase the local memory hit ratio. Since the design choice is more flexible at the higher level, it is advisable to make the design choices in a bottom-up order. Nonetheless, a few choices are quite obvious. For example, when a virtual address is translated to a physical address, the pid bits should be kept the same. Otherwise, it is impossible for the compiler to gear a data reference to the desired memory module. The focus of this paper is therefore on the other three mapping steps.

In this paper, we are interested mainly in parallel numerical programs. Through execution-driven simulations of five well-known numerical programs, we evaluate several design choices regarding the pid bits designation, data allocation and parallel task allocation. We discuss the interaction of these choices. The most important result of this paper is the strong evidence against the use of direct-mapped caches in CC-NUMA. Another important result is the data showing significant performance differences among data allocation schemes. A compiler scheme which allocates data and parallel tasks simultaneously is shown to perform quite well consistently.

The rest of this paper is organized as follows. In Section 2, we discuss the different mapping steps in more detail and present several design alternatives. In Section 3, we describe our experiments and discuss the results. Finally, we summarize our findings in Section 4.

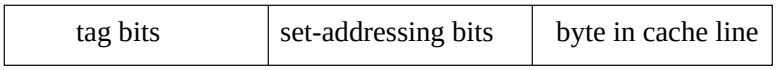


Figure 2: Typical mapping of the physical address to the cache in a uniprocessor or an SMP system.

2 Issues and alternatives

2.1 Cache mapping and pid bits designation

The current method to map the physical memory to a set-associative cache in a uniprocessor or a symmetric multiprocessor (SMP) system is shown in Figure 2. The lowest bits of the physical memory address differentiate the bytes in a cache line. The *set-addressing (SA) bits* are used to determine which set in the cache to search for or to store a particular line. In a direct-mapped cache, each set contains only one line, whereas in an n-way set associative cache, each set contains n lines. Hence, the number of bits needed to determine the set number depend on the cache associativity as well as its size. The highest bits in the physical address are used as a tag to identify the exact line stored in the cache.

Since the global address space in a CC-NUMA machine covers memory modules from different nodes, we see the need of associating a physical address with a memory module. Typically, this is achieved by designating certain bits in the physical address as the pid bits. Although this level of mapping is independent of cache mapping, the exact designation of these bits can affect how the cache is utilized. Obviously, the pid bits should not overlap with any bits that differentiate the bytes in a line. Otherwise, bytes in the same cache line will be from different memory modules, which is extremely inefficient. The next important decision concerns where to place the pid bits. Figure 3a shows one possible designation which overlaps all pid bits with the SA bits. With this designation, data to virtual address assignment does not have to deviate from that of a typical compiler for uniprocessors. When a virtual page is mapped to a physical page, data in this page is automatically partitioned into chunks of cache line size and the chunks are distributed to processors by interleaving due to the position of the pid bits (*c.f.* Section 2.2).

With the pid bits designation and the cache mapping scheme as shown in Figure 3a, a processor can only cache its local memory in the sets whose lower part of the SA bits matches the pid bits as shown in Figure 3b. This effectively reduces the cache capacity for caching local memory locations to $1/p$ of the cache

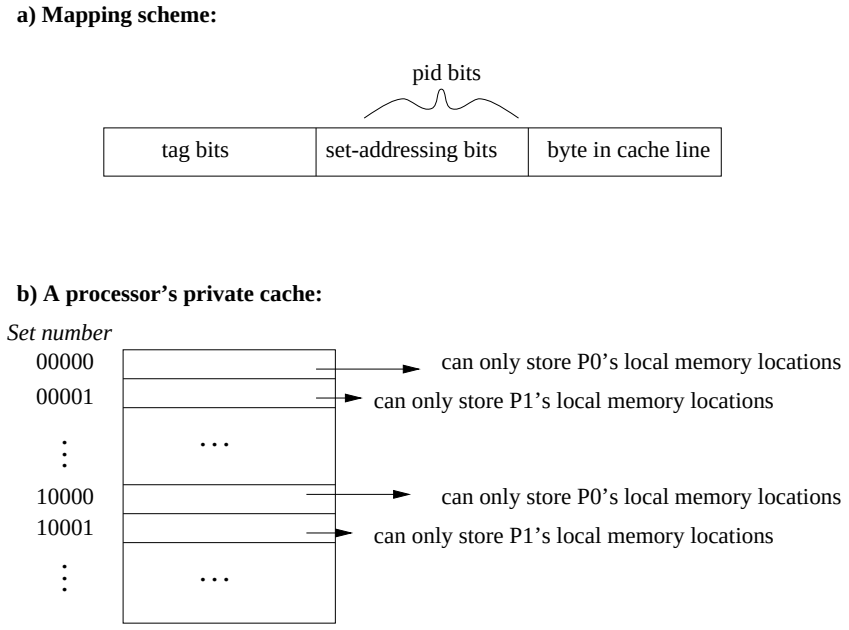


Figure 3: One possible CC-NUMA cache mapping and pid bits designation scheme and how it affects what can be cached to each set in the cache.

size where p is the number of processors. The rest of the cache can only be used to cache remote memory locations. This example assumes that the number of processors is smaller than the number of sets in the cache. Otherwise, the number of usable sets for caching local memory locations is reduced to just one. If a processor accesses many local memory locations, this pid bits designation will cause a high cache miss rate due to the much smaller usable cache size for caching local memory.

To avoid severely reducing the usable cache size for caching local memory, we should not overlap all of the pid bits with the SA bits. Figures 4a and 4b show two ways to separate the pid bits completely from the SA bits:

1. Move the SA bits as shown in Figure 4a but keep the current pid bits designation unchanged.

2. Designate the pid bits as shown in Figure 4b but keep the cache mapping unchanged.

The first approach requires a change in the conventional cache mapping hardware to reflect the movement of the SA bits, which is difficult to do if commodity microprocessors are used in the system. The second approach, in contrast, does not require any modification to the cache mapping hardware. Instead, it makes a change at the level of memory module mapping, which is much easier. In addition, by using a modifiable mask to specify the pid bits (similar to the mask used in the Cray T3D Block Transfer Engine [4]), the

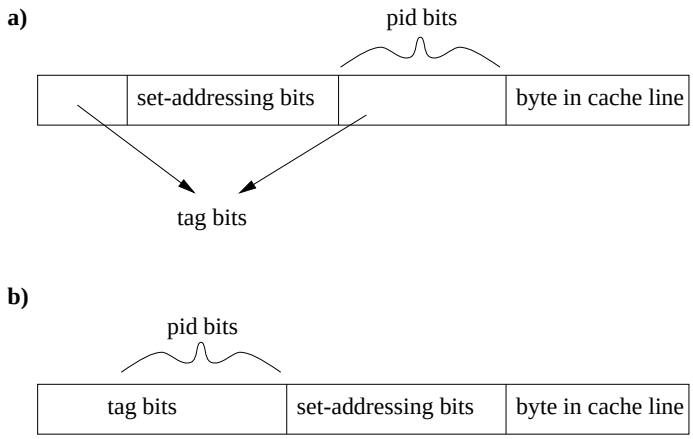


Figure 4: Two ways of designating the pid bits in the physical address such that they do not overlap with the set-addressing bits.

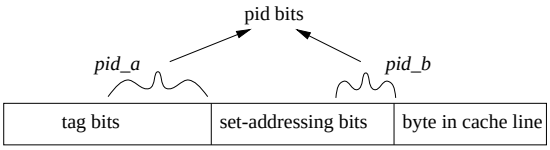
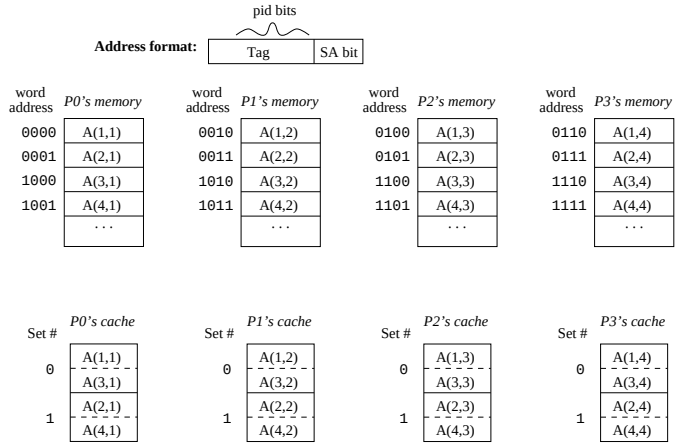


Figure 5: A pid bit designation which partially overlaps the pid bits with the set-addressing bits.

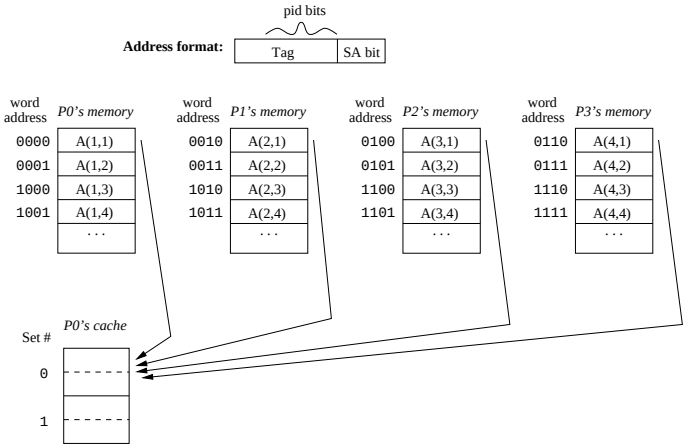
second approach can even allow changes to the overlap between the pid bits and the SA bits for different programs. As will be shown later in this paper, the optimal overlap can differ for different programs. Hence, we choose this approach of changing pid bits designation for the rest of our discussion.

Instead of separating the pid bits completely from the SA bits, one can choose to partially overlap the pid bits with the SA bits. As Figure 5 shows, the lower pid bits labeled pid_b can vary from zero up to n where n is $log_2(p)$ and p is the number of processors. In fact, the examples in Figures 3a and 4b are two extreme cases where all n or zero pid bits are in *pid_b* respectively. With only one overlapping bit in *pid_b*, each cache is effectively divided by two, each capable of caching half of the memory modules. As a result, a processor can only use half of its cache to store the data in its own memory module. For each additional overlapping bit in *pid_b*, the capacity of the cache which can be used to cache a processor's local memory is further reduced by half. The choice of the overlapping involves the tradeoff between having a smaller portion of a cache for local data and a possibility of more cache set conflicts due to references to different memories. Figure 6 illustrates this tradeoff.

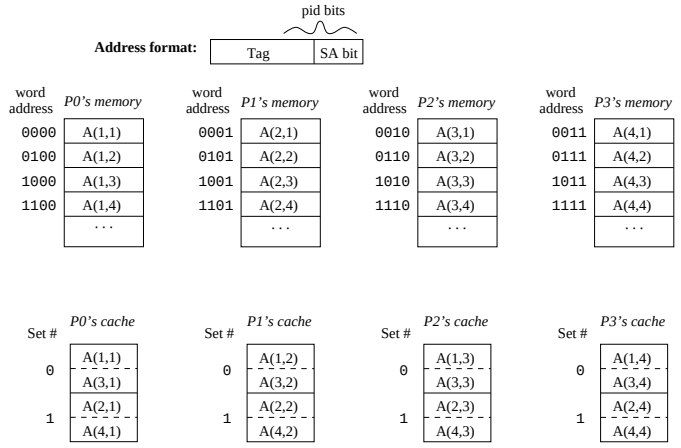
In Figure 6, we assume a system of four processors whose caches are two-way set associative. To make



(a) Good data allocation; the pid bits and the SA bit do not overlap.



(b) Bad data allocation; the pid bits and the SA bit do not overlap.



(c) Bad data allocation; the pid bits and the SA bit overlap.

Figure 6: An example of how overlapping the pid and the SA bits can reduce set conflicts. The byte address within the cache line is omitted.

the illustration simple, suppose each cache has four lines only and the line size is one word. Suppose four parallel tasks execute on the system. Each task repeatedly references one column of a 4 by 4 array A. We assume each array element is also one word so the byte address within the cache line is omitted in Figure 6. We examine two different address formats. One does not overlap the pid bits with the SA bit (*c.f.* Figures 6a and 6b) and the other has one overlapping bit (*c.f.* Figures 6c). The first format works perfectly with good data allocation which distributes the columns to the memory modules (*c.f.* Figure 6a). The sufficient cache size eliminates any set conflicts. In practice, remote memory references are often inevitable, either due to poor program characteristics or due to the limitations of the compiler. With poor data allocation which distributes the rows to the memory modules (*c.f.* Figure 6b), each processor tries to cache data from all memories to the same set, resulting in set conflicts. The second format, which has an overlapping bit, works well with poor data allocation, because data from different memories is now mapped to two sets in the cache, eliminating the set conflicts (*c.f.* Figure 6c).

2.2 Data allocation

The existence of local and remote memory necessitates the allocation of data to different processors. Because of the possibly large difference in latency between local and remote accesses, the distribution of data can greatly affect a program's performance. Since scalar variables are usually shared by all processors and most scalar references tend to be register references, their placement is not as important as array variables. Therefore, in this paper, we only consider array data allocation. In the following, we discuss some possible data allocation schemes. In these schemes, the compiler manipulates the pid bits in various ways to allocate array elements to memory modules. Therefore, it is important that the compiler knows how the pid bits are designated in the physical address and that virtual to physical address mapping does not alter the pid bits.

Flat-space interleave

In the *flat-space interleave* scheme, the entire address space is partitioned into chunks of cache line size. These chunks are then distributed to the processors by interleaving. This scheme is the simplest to achieve in a compiler, since it requires only permuting certain bits in the virtual address.

The remaining allocation schemes require the following modifications to the compiler. First, the compiler

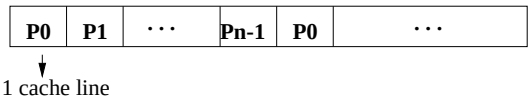


Figure 7: Examples of an array being distributed to n processors using the *per-array interleave* allocation scheme.

must pad each array so that the starting address is a multiple of the cache line size times the number of processors. Second, for each array reference, the address is computed by using a new mapping function as opposed to say, the conventional column-major mapping in Fortran. Since these involve compiler technical details, we leave the discussion elsewhere [13]. To illustrate the remaining allocation schemes, we show how a two-dimensional array is distributed to processors by each scheme in the following.

Per-array interleave

Per-array interleave partitions the linear address space of the array into chunks of the cache line size. The chunks are then distributed across the processors by interleaving. Figure 7 shows the linear address space of an array as it is distributed to processors using this scheme.

Per-array block

In the *Per-array block* allocation scheme, the linear address space of the array is evenly divided into p chunks and then assigned to the p processors. The size of the array is always rounded up to a multiple of the cache line size times p.

Per-dimension interleave allocation

In this allocation scheme, the rows or columns of the array are partitioned into chunks of the same size. The chunks are then distributed across the processors in an interleaving fashion. Figure 8 shows examples of such distribution. When the data are distributed by the rows, we label this distribution as *1st-dimension interleave*. Likewise, *2nd-dimension interleave* indicates that the columns are distributed to processors. If the array has more than two dimensions, we can have variations of this allocation scheme in the higher dimensions as well. Note that although the data chunk size may vary, the optimal size should agree with the chunk size of the employed task allocation method (*c.f.* Section 2.3).

Per-dimension block allocation

This allocation scheme is similar to the *per-dimension interleave* scheme discussed above. Here, however,

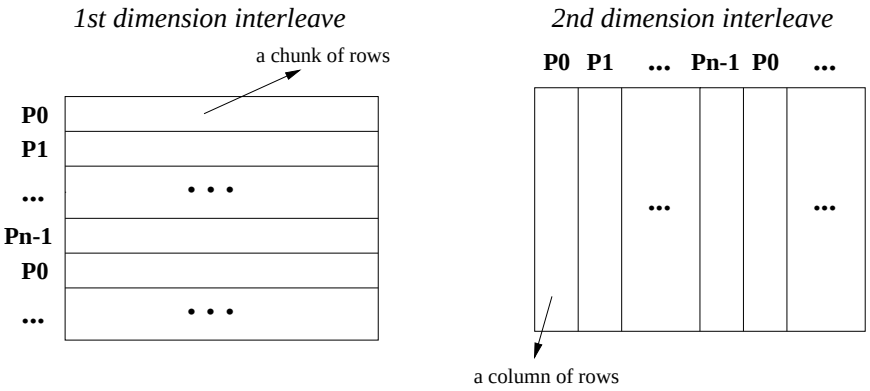


Figure 8: Examples of a 2-dimensional array being distributed using the *per-dimension interleave* allocation scheme.

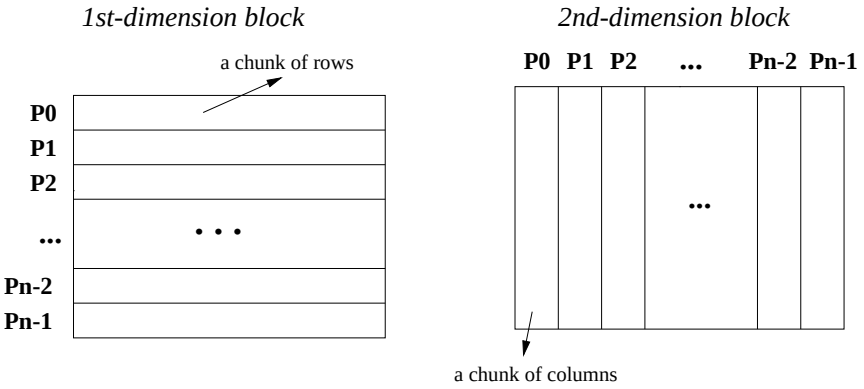


Figure 9: Examples of a 2-dimensional array being distributed using the *per-dimension block* allocation scheme.

the chunk size is fixed to $\lceil d/p \rceil$ where d is the size of the selected dimension to be distributed among the processors and p is the number of processors. Therefore, each of the first $p-1$ processors has exactly one chunk of rows or columns, while the last processor has the remaining rows or columns. A pictorial example of how the 2-dimensional array is distributed by rows and columns is shown in Figure 9.

2.3 Task allocation

As on a conventional SMP, task allocation can greatly affect the program performance on CC-NUMA multiprocessors. Here, we assume a fork-join model for parallel programs in which a fork occurs at the beginning of a parallel loop and a join follows at the termination of that loop. We consider parallel loops in the form of DOALL loops. A DOALL loop is similar to a Fortran DO loop except that it has no data dependences between different iterations. Hence, each iteration in the DOALL loop can be executed independently from

each other. Also, the DOALL loop terminates only when all iterations are completed. Synchronization must be done at the fork and the join of each DOALL loop to assure correct execution. In our simulations, synchronization is done by software through the use of synchronization variables.

If a loop nest has parallelizable loops at different levels, we need to pick one of them as a DOALL loop, because the machine code generator used in our experiments (*c.f.* Section 3) supports singly nested DOALL loops only. Our data-task co-allocation algorithm (*c.f.* Section 2.4) picks parallel loops in conjunction with data allocation. When any other data allocation scheme is used, we parallelize the outermost parallelizable loops unless the inner ones have better iteration counts.

Using the above programming model, task allocation involves partitioning a DOALL loop's iterations (or tasks) and assigning them to processors. Task allocation is therefore also referred to as loop scheduling. For a survey of loop scheduling methods, the reader is referred to [11]. Scheduling of loop iterations to processors can be determined at compile-time, called *static scheduling*. One commonly used method of static scheduling is known as *chunk-interleave scheduling*, in which loop iterations are partitioned into chunks of the same size. These chunks are then assigned to the processors by interleaving. The chunk size should be carefully selected to avoid false-sharing. For example, if all array data being referenced is in units of 8 bytes, then a chunk size of 4 works well for systems with a cache line size of 32 bytes. Another common method of static scheduling is known as *block scheduling*. With this method, the compiler divides the n iterations in a DOALL loop into p chunks such that the first $p-1$ chunks contain $\lceil n/p \rceil$ iterations each and the last chunk contains the remaining iterations. Each of these p chunks are then assigned to the p processors. The disadvantage of static scheduling is the potential imbalance of the finish time among different processors. The execution time of the workloads assigned to the processors may potentially vary because of branch conditions and other run-time effects such as cache hits and misses.

In contrast to static scheduling, *dynamic scheduling* makes each processor responsible for obtaining its tasks during the execution of the parallel loop. A well-known method of dynamic scheduling is *self-scheduling*, in which the processors compete for the tasks by accessing and updating a shared iteration counter variable. The number of iterations obtained by the processor, also called chunk size, are fixed for each parallel loop but can vary for different loops. A smaller chunk size yields a better balanced workload at the cost of higher

synchronization overhead. Another dynamic scheduling policy is *guided self-scheduling* or *GSS*, in which the iteration chunk size is calculated by $n = \lceil R/p \rceil$ where R is the number of remaining iterations and p is the number of processors executing the loop. By allocating larger chunks of iterations at the beginning, GSS potentially yields less overhead than self-scheduling. In the interest of load balance, the chunk size gets smaller as the remaining iterations are reduced. In all dynamic scheduling methods, the iterations to processors assignment cannot be determined and the data referenced may not align well from one loop to the next. Because of this, the compiler effort for good data allocation on CC-NUMA may be wasted when dynamic scheduling is employed.

2.4 Data-task co-allocation

The data-task co-allocation scheme uses the results of our interprocedural alignment algorithm. This section discusses an overview of the algorithm and some related works. More details on the algorithm and its implementation can be found in [14, 13]. The algorithm has been implemented in the Panorama compiler [5, 10, 14].

We assume that each iteration in a parallel loop is assigned to one unique *virtual processor* (VP). The VP's are eventually folded to the physical processors available for program execution. How the VP's are folded depends on which static loop scheduling scheme is used. Under the CC-NUMA model, each physical processor has a local memory module that is accessible by all processors. For our purpose, we regard each VP as having its own memory module as well. Concurrent with the assignment of parallel loop iterations, we allocate array data to memory modules. Our goal is to find heuristics that maximize the local memory accesses without sacrificing parallelism. The data-task co-allocation involves two optimization steps:

1. Phase I: Optimizing the *orientation* of the data allocation and the assignment of the parallel iterations. Suppose the VP's are numbered from 1 to N. For data allocation, orientation means whether we should allocate the elements of an array along with increasing or decreasing order of the VP's. Likewise, the orientation of the iteration assignment can be in the increasing or decreasing order. In this phase, we select one dimension for each multi-dimensional array to distribute among the VP's and also select which loop level remains a parallel loop in a multiply-nested parallel loop.

2. Phase II: Optimizing the *displacement* of the chosen orientations. Displacement means how far, if any, we should shift the data assignment or the iteration assignment to the left or to the right. If the optimal displacement is small, then its effect is not as important as choosing a good orientation because it can usually be covered by the effect of practical data cache block sizes which are greater than one word.

In each phase, an undirected acyclic graph is created with each node representing an orientation or displacement choice. We call the nodes representing array and loop orientation/displacement choices *array nodes* and *loop nodes* respectively. There is an edge between an array node to a loop node if the array choice matches the loop choice. A mismatch cost is associated with the edge to indicate the number of remote memory references that will occur if either the array or loop choice is selected but not both.

A simple heuristic is used to select array and loop orientation/displacement choices with the mismatch costs serve as a basis for choosing between two conflicting nodes. The complexity of the algorithm is on $O(n + m)$ where n is the number of nodes and m is the number of edges. In our current implementation, the algorithm only contributes 1 to 3 additional seconds to the compile time for all of our test programs.

2.4.1 Related works

Several researchers have proposed algorithms for allocating data and tasks on CC-NUMA architectures. Anderson and Lam model this problem in a linear algebra framework [3]. They assume that loop bounds and array subscripts are affine functions of the loop indices. Parallel loop iterations are mapped to a polytope in an l-dimensional iteration space, where l is the depth of a parallel loop nest. An n-dimensional array is mapped to an n-dimensional rectangular space. Likewise, processors in the systems are mapped to a processor space. By solving several potentially large systems of equations, array data and loop iterations are mapped to processors. During this process, some parallel loops may be sequentialized in order to eliminate or minimize remote memory references. Agarwal et al. also model this problem in a linear algebra framework [1]. Loop iterations and array data are mapped to a hyperplane. The loop iterations are then partitioned into optimal hyperparallelepiped tiles that minimize the interprocessor communication. This scheme, however, only considers a single loop nest. Li and Pingali take a different approach to the data and task allocation problem [9]. Access patterns of arrays in each loop nest are summarized in a *data access*

matrix. After calculating or estimating an invertible matrix of the data access matrix, they use it as a basis for transforming and normalizing the loop nest. This *access normalization* process transforms the loop nest into a new loop nest with better data locality.

3 Experiments and results

We wish to find, through experiments, answers to several questions which arise in the earlier discussion:

- Determining the number of bits to overlap between the pid bits and the set addressing bits is a tradeoff between set conflicts due to local memory references and those due to remote memory references. How does cache associativity affect the tradeoff? How does the quality of data allocation affect the tradeoff? Is the tradeoff easy for practical programs?
- Do data allocation schemes make any difference to the number of memory misses and do memory misses make any difference to the program execution time?
- How does our data-task co-allocation perform when compared to the other simpler schemes which we have discussed? Which, if any, of the simpler schemes perform better than the others and thus can be used as a default scheme when data-task co-allocation is not applied?
- How do the different loop scheduling methods perform? How do they interact with data allocation?

In this study, we perform execution driven simulations as our experiments. The simulation methodology is described next.

3.1 Simulation methodology

Panorama, an interprocedural parallelizing compiler, is used to parallelize and instrument the benchmarking programs for this study. It also provides the framework for the implementation of the data-task co-allocation algorithm [5, 10, 14]. After a program is parallelized and instrumented by Panorama, it is compiled by SGI's `f77` compiler to a parallel object code executable on the SGI Challenge multiprocessor. The SGI Challenge provides run-time libraries which implement the loop scheduling methods mentioned in the last section. Block scheduling, however, is named *simple scheduling*.

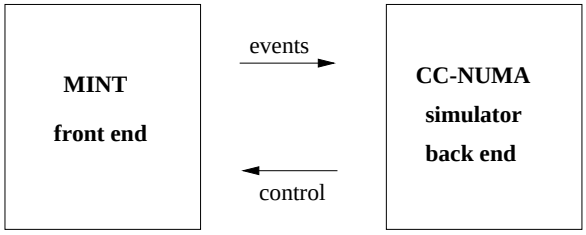


Figure 10: Overview of the CC-NUMA simulator.

The executable object code is then simulated by *NUMAsim*, an execution-driven CC-NUMA simulator. NUMAsim consists of two main parts: the MINT event generator front-end [18] and a CC-NUMA simulation back-end. Figure 10 illustrates how the components of NUMAsim interact with each other. The MINT front-end interprets each instruction in the executable object code and schedules events such as loads and stores for the back-end to simulate. MINT adopts a simple processor model in which each non-load/store instruction takes one cycle and the processor stalls at a cache miss. An event originating from the instructions may trigger additional events internal to the memory hierarchy. Such events are also scheduled by MINT upon the request from the back-end. When all events triggered by the load or the store are processed, the back-end sends a control signal back to MINT so it can continue to interpret the next instruction. This process continues until the interpreted code runs to stop.

Since Panorama is a source-to-source compiler, it does not generate virtual addresses for the machine code. Instead, in the NUMAsim simulator, we remap the virtual addresses generated by *f77* to reflect the selected data allocation scheme. Panorama inserts directives to allow the simulator to identify the starting address and the size of each array and what data allocation scheme to be used for that array. Using this information, the simulator changes the virtual addresses of array data on the fly. Because the simulator remaps virtual addresses before the instruction is interpreted, we can verify the correctness of the mapping processes by comparing the program's output before and after remapping. Most errors in the mapping process will cause a core dump or incorrect execution results.

All experiments performed in this paper use the following default settings for NUMAsim. The simulated system contains 16 processors, each with a 16 Kbytes cache. Each cache line consists of 32 bytes or 8 words. A bandwidth of one word per processor cycle is assumed for each memory module and for the network. Table 1 shows the duration time for different types of events that are associated with cache and memory

Table 1: Duration time for various events. *lsize* is the line size in words (which is 8 in our experiments).

Event type	*Latency in cycles*
Cache access	1
Memory access	7 + *lsize* - 1
Directory access	1
1-way network delay	20

accesses. A load or a store may trigger one or more of these events. A cache read miss that does not require a remote memory access contains 2 cache accesses (the second access is for filling the cache), a directory access, and a memory access for a total delay of 41 cycles. A remote read miss requires either an additional 2-way network delay for a non-dirty line or an additional 3-way network delay for a dirty line [17]. The actual delay for cache and memory misses may vary depending on the availability of the cache ports and the memory ports. Our simulator assumes a single port for both each cache and each memory module.

3.2 Data collected and test programs

For each simulated run, we collect many statistics including the execution time in processor cycles, the number of cache misses, and the number of memory misses. In this paper, we present program speedup, cache miss ratio and memory miss ratio.

The cache miss ratio is the ratio of the total number of cache misses over the total number of references in a program. This metric generally indicates how well the caches are used. However, there is a caveat. The total number of references are not the same for different runs with different parameters such as cache settings, loop scheduling methods, and data allocation schemes, even when using identical input data. This arises due to the number of references to synchronization variables which varies depending on when each processor finishes its tasks in a parallel loop. (Unfortunately, synchronization variables in the machine code appear as ordinary data variables and we are not yet able to filter them out.) In a few runs for one of the test programs, namely `mgrid`, we see less cache misses but a higher cache miss ratio. We could alternatively use the total number of references in a sequential run as a base. However, since some cache misses are due to references to synchronization variables, we may see some other abnormality.

The memory miss ratio is defined as the ratio of total remote memory accesses over the total number of accesses to any memory modules. Access to a memory module is necessary during a cache miss or a cache

write-back. A write-back to a remote memory module is considered a remote memory access.

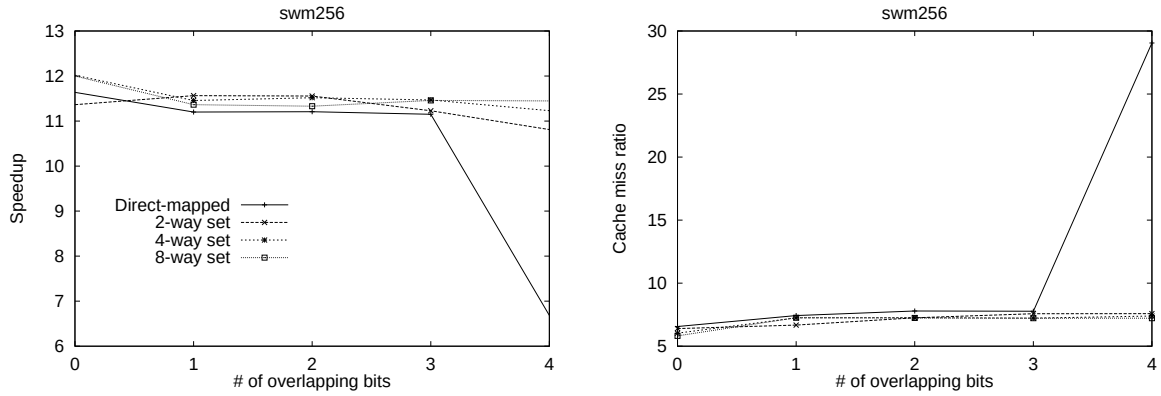
The program speedup of parallel machine code, under a particular simulation setting, is the execution time of the optimized code of the original sequential program over the parallel execution time. We use the `f77 -O2` command to produce the optimized sequential codes. The `-O2` flag is used for producing the parallel code as well, in conjunction with other flags specific to parallel code generation. We interchange DOALL loops with the outer sequential loops whenever possible, to reduce scheduling overhead. Loops are not interchanged in those sequential codes, since interchange did not improve the sequential execution.

We have collected data for the following five numerical Fortran programs: three programs, namely `swm256`, `tomcatv` and `hydro2d`, from the SPEC92 benchmarks, `mgrid` from the SPEC95 benchmarks, and `adi` [16]. Most programs use all 16 processors with the exception of `hydro2d` which uses only 8 processors because of its relatively low degree of parallelism. Our compiler scheme for data-task co-allocation is applied to all programs but `mgrid` whose dynamic array re-shaping is beyond the capability of our current program analysis.

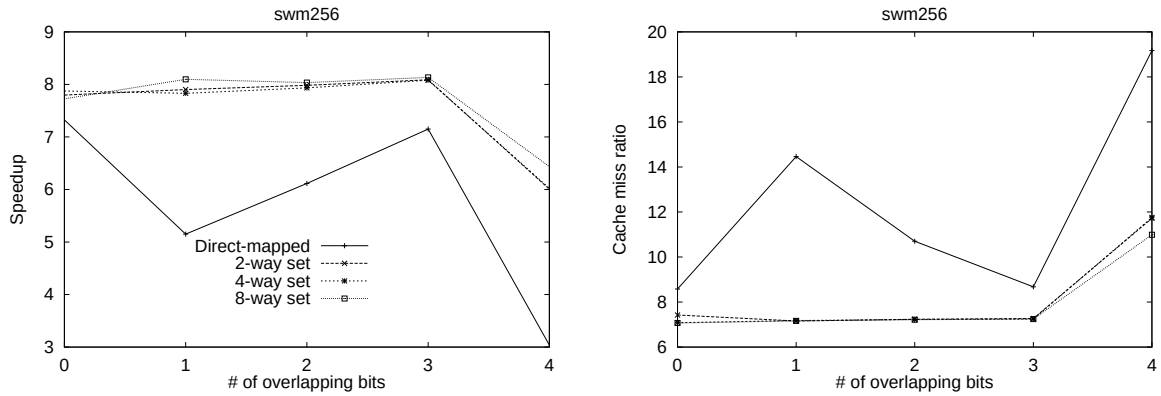
In order to shorten the simulation time, we reduce the iterative time steps in some programs. The number of steps is reduced from 1200 to 60 in `swm256` and from 100 to 5 in `tomcatv`. For `hydro2d`, we set the time steps to 400. For `mgrid`, the time steps are reduced from 40 to 2. We eliminate the outermost loop which does nothing but repeatedly exercising the machine with the same input producing the same result. We believe the above settings repesent a reasonably long execution history for each program. Any given data sizes remain intact in the test programs. In `swm256`, the main data consists of 14 arrays of 257 x 257 words each. In `tomcatv`, the main data consists of seven arrays of 257 x 257 double words each. In `adi`, the main data is nine 256 x 256 arrays of double words each. In `mgrid`, the main data consists of three arrays, one of which has 287,496 double words and the rest have 333,944 double words. In hydro2d, arrays of various sizes are declared, with the total memory requirement of about $134 \cdot MP \cdot NP = 134 \cdot 402 \cdot 160 \approx 8.6$ Mbytes.

3.3 Results and discussions

The simulation results are shown in Figures 11 through 20. For each test program, we present two figures. One figure, e.g. Figure 11, shows how the speedup and the cache miss ratio are affected by the number of



(a) Data-task co-allocation

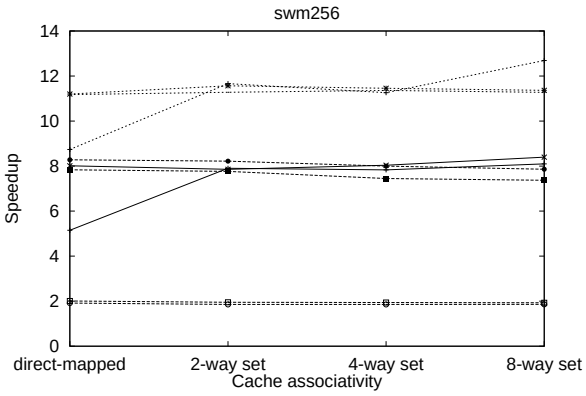


(b) Flat-space interleave allocation

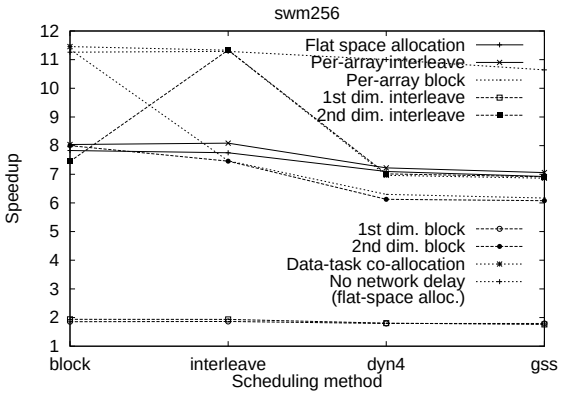
Figure 11: Speedup and cache miss ratio for different overlapping between the pid bits and the SA bits for `swm256`.

the bits overlapping between the pid and the set addressing bits. The other figure, e.g. Figure 12, shows how the speedup, the cache miss ratio and the memory miss ratio are affected by data allocation schemes and loop scheduling methods.

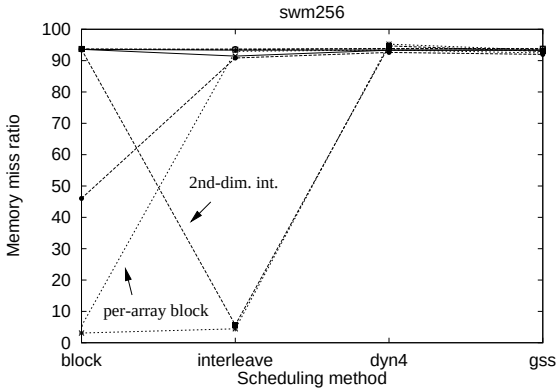
In the first figure for each program, two sets of subfigures depict results using two different data allocation schemes. For all programs except `mgrid`, we show results using the data-task co-allocation scheme, which achieves relatively low memory miss ratio consistently, and flat-space interleave, which produces relatively high memory miss ratio. As mentioned previously, the co-allocation scheme is not applied to `mgrid` because of its complicated array re-shaping. For this program, all allocation schemes using interleave achieve almost identical results and all other schemes which use blocking achieve almost identical results. Therefore, we



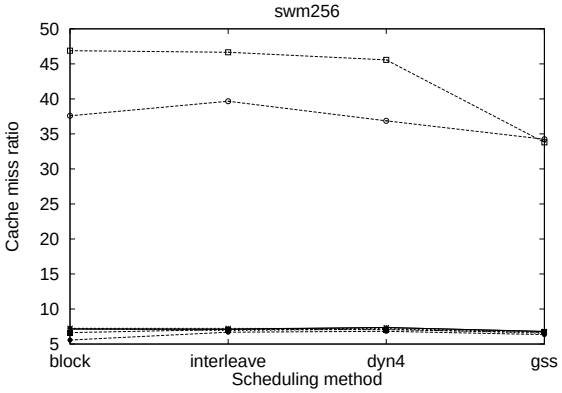
(a) Speedup for various set-associativity.



(b) Speedup for different scheduling methods.



(c) Memory miss ratio for different scheduling methods.



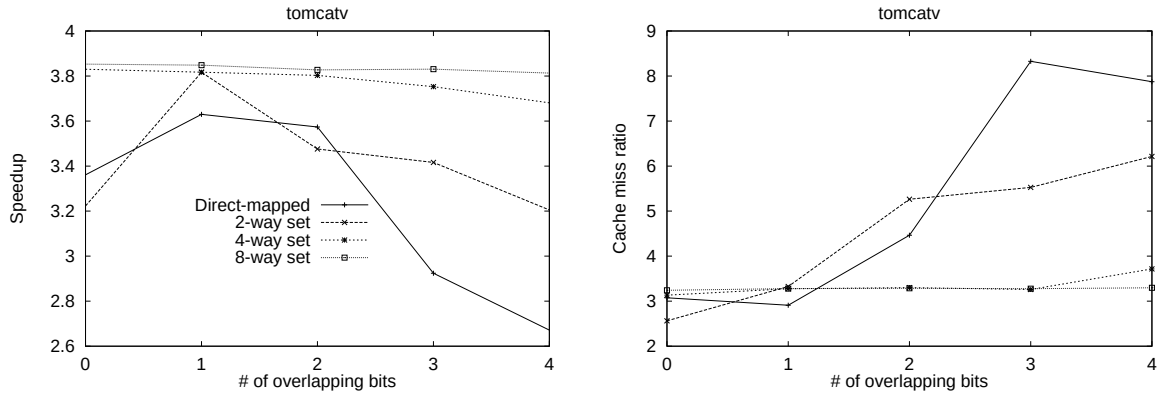
(d) Cache miss ratio for different scheduling methods.

Figure 12: The effect of data allocation on system performance for `swm256`.

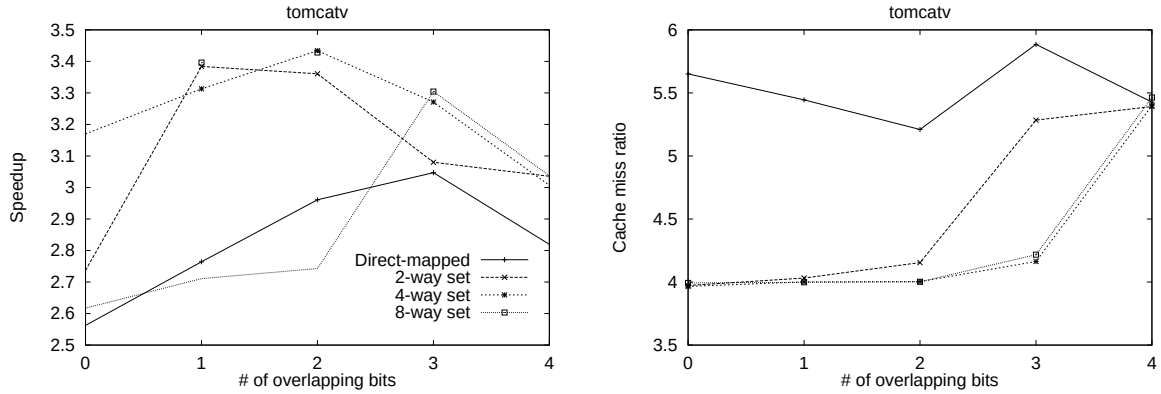
show one set of subfigures for interleave, which produces a higher memory miss ratio for this program, and the other set for blocking which produces a lower memory miss ratio. The loop scheduling method is fixed as block scheduling. In each subfigure, one performance curve is drawn for each different cache associativity.

From Figures 11, 13, 15, 17 and 19, we observe the following:

- Good data allocation favors fewer overlapping bits, while relatively poor data allocation favors more overlapping bits.
- Higher cache associativity tends to favor fewer overlapping bits, while lower cache associativity tends to favor more overlapping bits.



(a) Data-task co-allocation

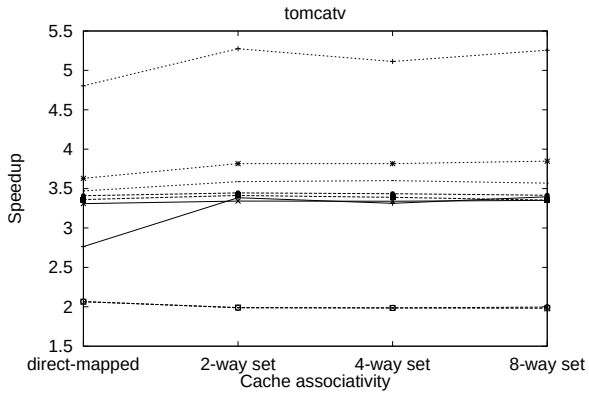


(b) Flat-space interleave allocation

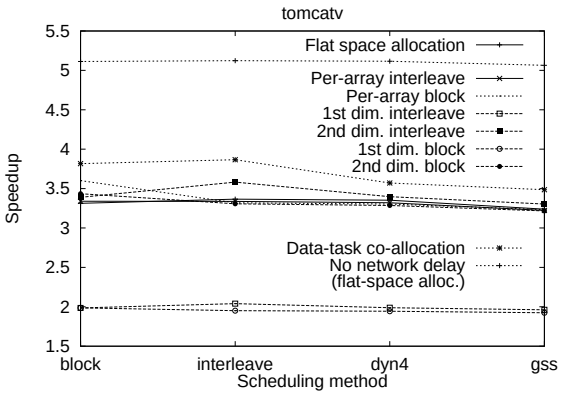
Figure 13: Speedup and cache miss ratio for different overlapping between the pid bits and the SA bits for `tomcatv`.

- With 4-way and 8-way set associativity, the performance by using zero overlapping bits or one overlapping bit is consistently good, although it may not be the best. The performance by using direct mapping is the most unstable, under which the number of optimal overlapping bits varies dramatically.

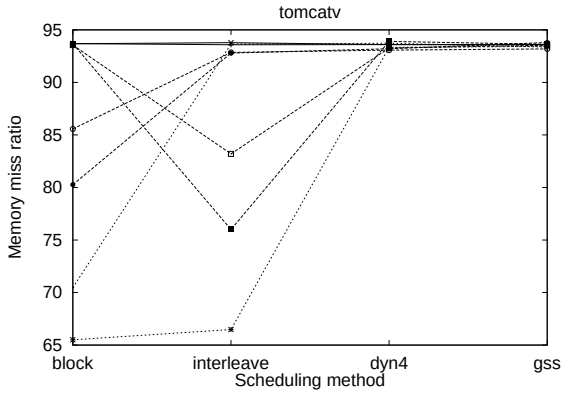
In the second figure for each program, we show four subfigures. One subfigure, e.g. Figure 12a, shows how different data allocation schemes compare in terms of speedup, and how stable the performance is when the cache associativity changes. The other three subfigures, with cache associativity fixed at four-way, show how the data allocation schemes compare, how the loop scheduling methods compare and how data allocation and loop scheduling interact. The "interleave" marking represents chunk interleave scheduling with the iteration chunk size determined at compile time to agree with the data allocation schemes. The



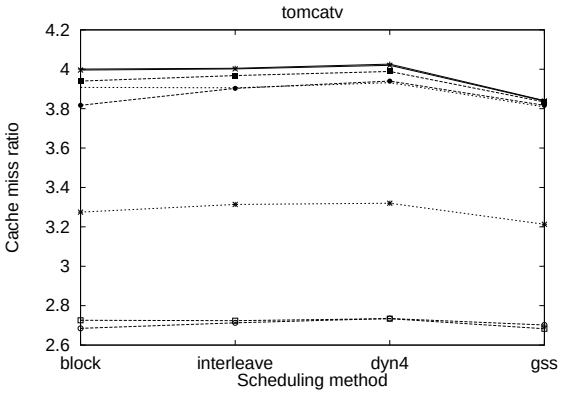
(a) Speedup for various set-associativity.



(b) Speedup for different scheduling methods.



(c) Memory miss ratio for different scheduling methods.



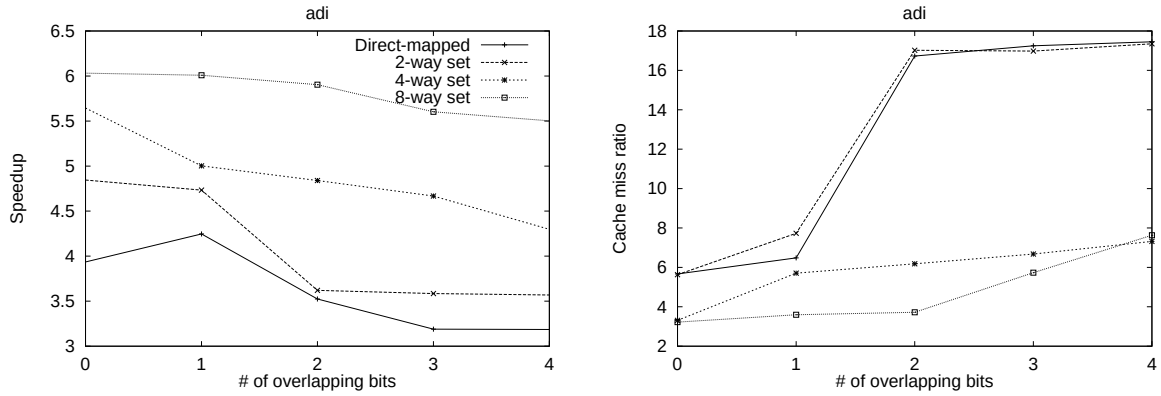
(d) Cache miss ratio for different scheduling methods.

Figure 14: The effect of data allocation on system performance for `tomcatv`.

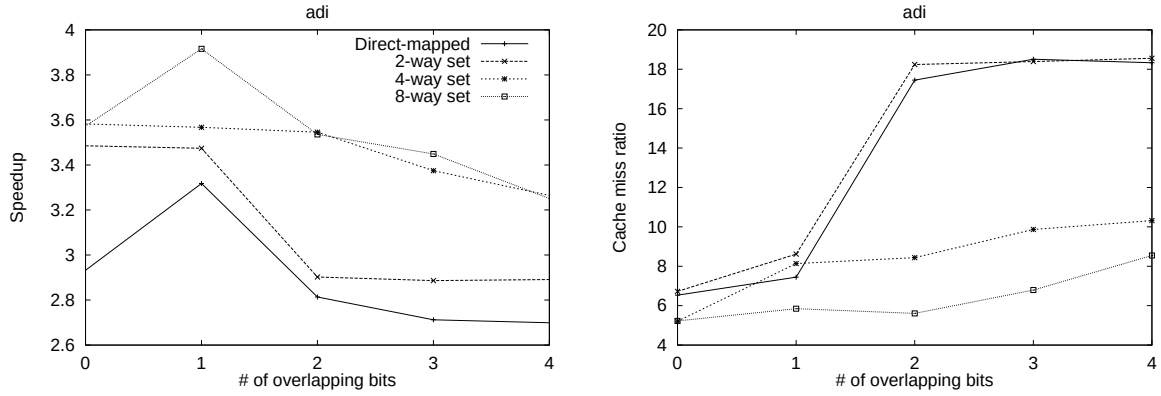
"dyn4" marking represents self-scheduling with an iteration chunk size of four. Each subfigure shows one of three performance metrics: the speedup, the cache miss ratio and the memory miss ratio. In all these subfigures, one overlapping bit (between the pid and set-addressing bits) is used. As a reference point, we show a speedup curve in the appropriate subfigures for the result of setting the network latency to zero, assuming the simplest flat-space interleave allocation scheme.

From Figures 12, 14, 16, 18 and 20, we observe the following:

- Our data-task co-allocation scheme consistently delivers the best speedup and the lowest memory miss ratio for all four programs on which it is applied. Data allocation can have a striking impact on the memory miss ratio, which ranges from less than 4% to over 90% in `swm256`, for example.



(a) Data-task co-allocation



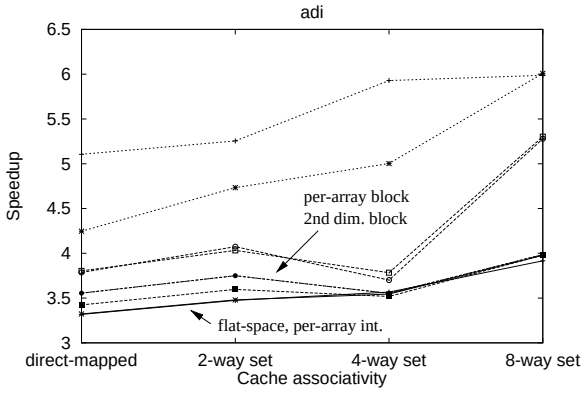
(b) Flat-space interleave allocation

Figure 15: Speedup and cache miss ratio for different overlapping between the pid bits and the SA bits for `adi`.

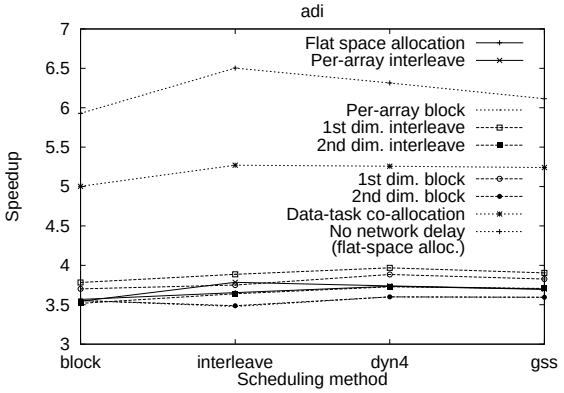
- Besides the co-allocation scheme, there seems to be no clear winner among the other simpler schemes. Nonetheless, per-array block seems to be a reasonable candidate as a simple default scheme, as it performs relatively well in many cases.

- For all programs but `swm256`, there is a quite significant gap between the no-network-delay speedup curve and the best curve among the rest, which suggests that there is some room for improvement either by a better data allocation scheme or by pre-fetching.

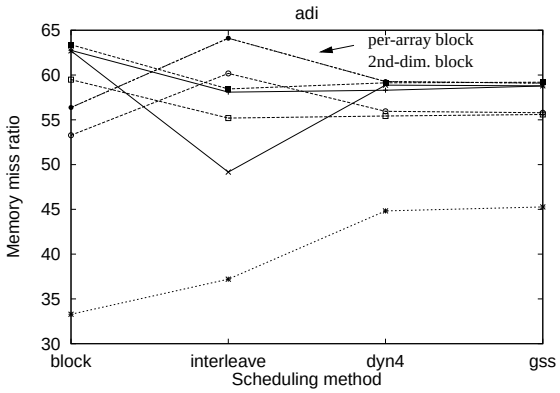
- The no-network-delay speedup curves suggest that loop scheduling overhead is quite considerable for hydro2d but is not a big concern for the other programs.



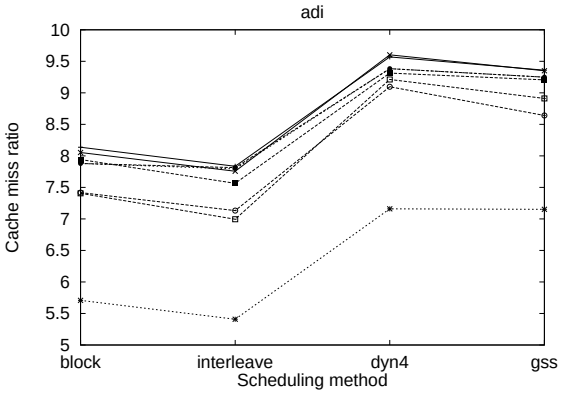
(a) Speedup for various set-associativity.



(b) Speedup for different scheduling methods.



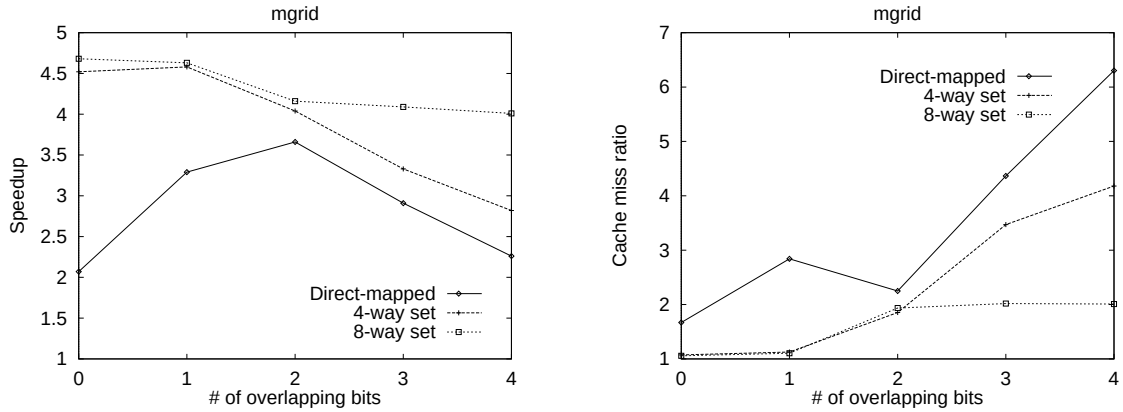
(c) Memory miss ratio for different scheduling methods.



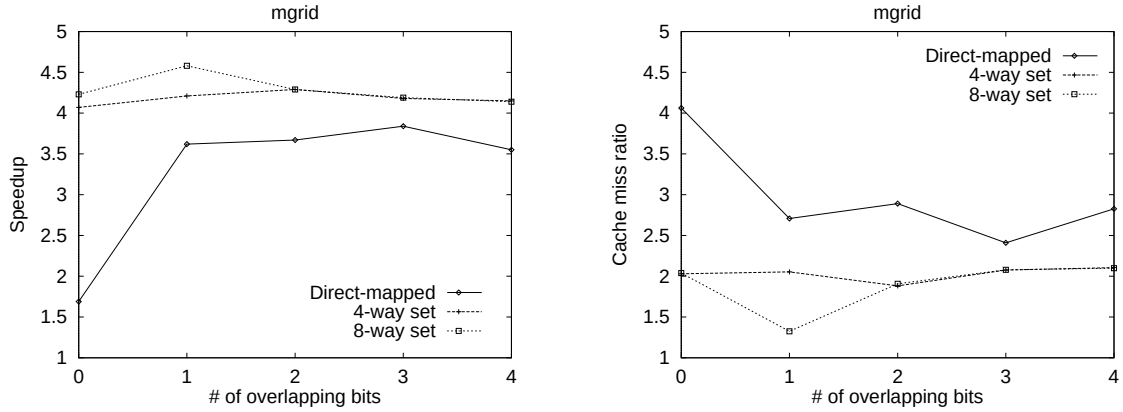
(d) Cache miss ratio for different scheduling methods.

Figure 16: The effect of data allocation on system performance for `adi`.

- The mismatch between data allocation and task assignment can result in high memory miss ratio and low speedup. For example in `swm256` (*c.f.* Figure 12b), dynamic scheduling causes data misalignment for the two good allocation schemes (data-task co-allocation and 2nd-dimension interleave), resulting in a sharp drop in speedup. However, dynamic scheduling does not seem to further worsen the performance when there is already a mismatch.



(a) Block allocation



(b) Interleave allocation

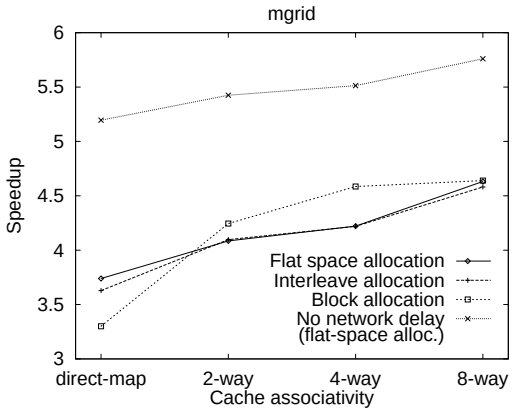
Figure 17: Speedup and cache miss ratio for different overlapping between the pid bits and the SA bits for `mgrid`.

4 Conclusion

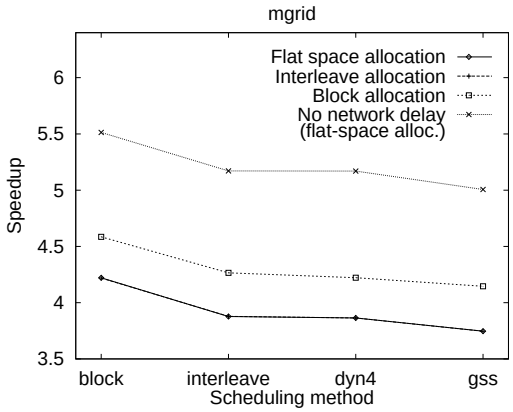
Based on our experimental results with five numerical programs, we draw the following preliminary conclusions about memory allocation on CC-NUMA.

First, we conclude that data allocation schemes often make a great impact on the memory miss ratio and the execution time. Our scheme which considers data allocation and task allocation simultaneously delivers the best results. However, it needs to be improved in order to handle complicated array re-shaping.

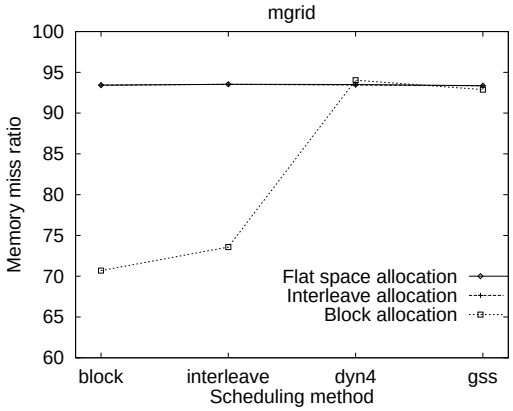
Second, whether the pid bits should overlap with the set-addressing (SA) bits for cache mapping is not a simple matter. It is clear that complete inclusion of the pid bits in the SA bits is always a bad choice,



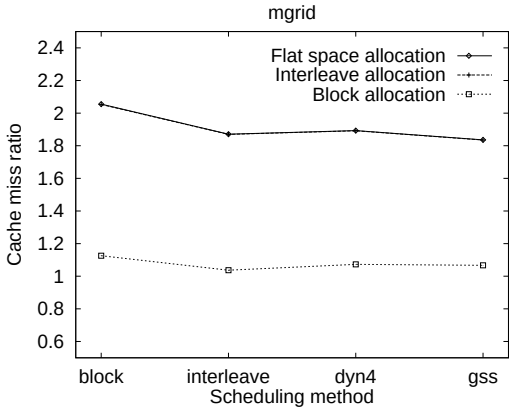
(a) Speedup for various set-associativity.



(b) Speedup for different scheduling methods.



(c) Memory miss ratio at different scheduling methods.

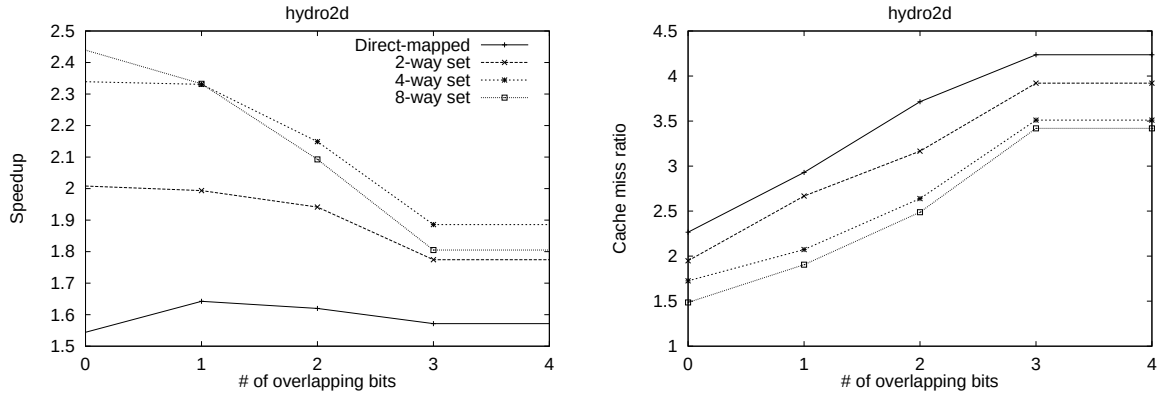


(d) Cache miss ratio at different scheduling methods.

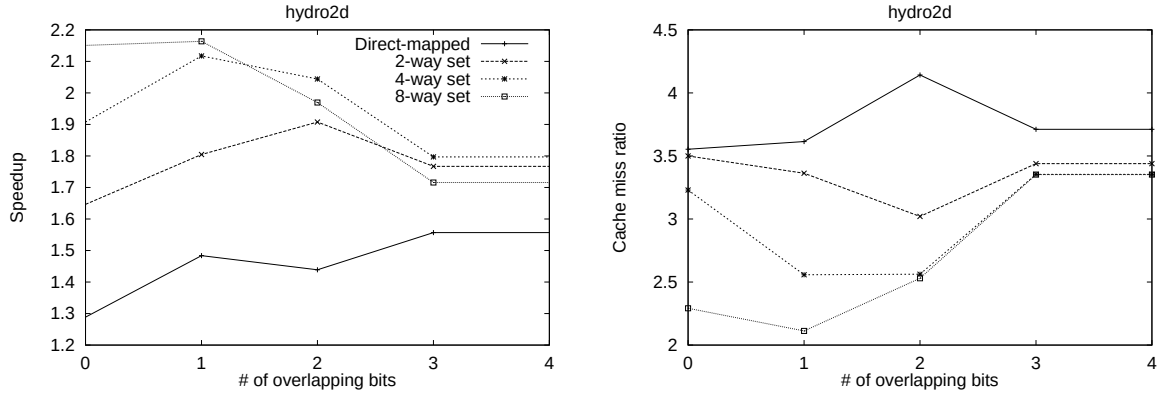
Figure 18: The effect of data allocation on system performance for `mgrid`. Note that the data-task co-allocation scheme is not applied.

because it drastically reduces the effective cache size for local memory references. On the other hand, complete separation of the pid bits from the SA bits does not always give the best results. The optimal number of bits shared by the pid and SA depends on two major factors, namely, cache associativity and the program's data-task affinity.

With four-way or eight-way set-associative caches, putting zero pid bits and one pid bit in the SA delivers comparable results – both are better than the other choices. With direct-mapped caches, however, the optimal number of bits shared by the pid and SA changes from one program to another, showing dramatically different performance. The optimal choice ranges from zero to three bits on our simulated 16-processor system. Therefore, our results strongly discourage the use of direct-mapped caches for CC-NUMA



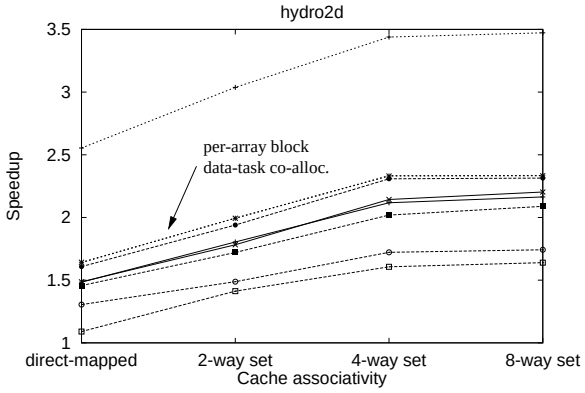
(a) Data-task co-allocation



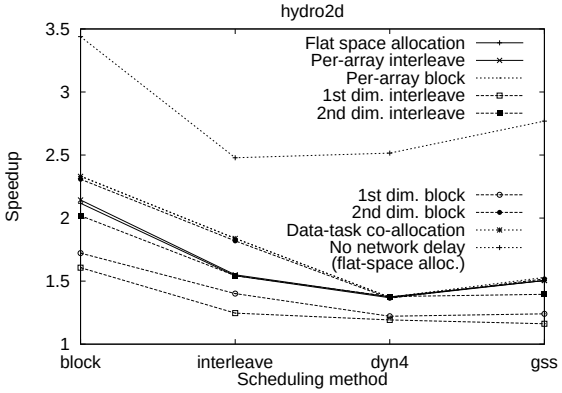
(b) Flat-space interleave allocation

Figure 19: Speedup and cache miss ratio for different overlapping between the pid bits and the SA bits for `hydro2d`.

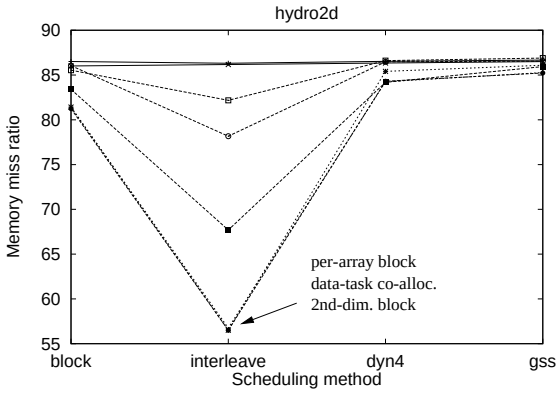
machines because of the difficulty of designating the pid bits. The behavior of direct-mapped cache can be affected by many factors. However, the machine code's data-task affinity often seems to be a decisive factor. The codes which show good data-task affinity seem to favor zero or one bit, while those exhibiting poor affinity seem to favor more bits. This has two implications. First, a good data allocation scheme makes the choice of the pid bits designation easier. Second, it is advisable to adopt an address translation hardware scheme which can dynamically designate pid bits in the data addresses. How to determine the pid bits designation according to program characteristics may be a topic for future work. This study opens up many new issues. The most unclear issue is how our conclusions scale to larger CC-NUMA systems running programs with larger data sets. We are currently conducting experiments towards this end.



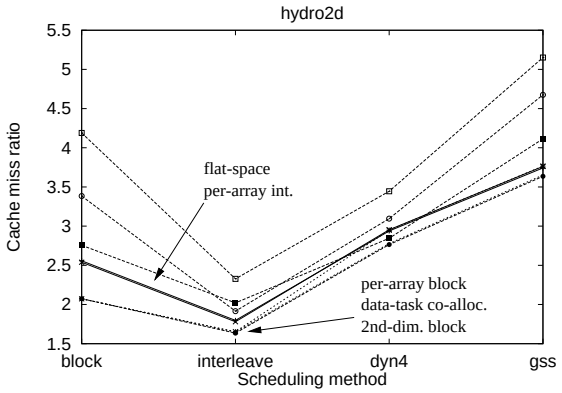
(a) Speedup for various set-associativity.



(b) Speedup for different scheduling methods.



(c) Memory miss ratio for different scheduling methods.



(d) Cache miss ratio for different scheduling methods.

Figure 20: The effect of data allocation on system performance for `hydro2d`.

Acknowledgements

We would like to thank to Jack Veenstra for providing MINT and taking the time to answer our questions and to Bob Glamm for proofreading this paper.

References

[1] A. Agarwal, D. Kranz, and V. Natarajan. Automatic partitioning of parallel loops and data arrays for distributed shared memory multiprocessors. In *Proc. International Conference on Parallel Processing*, volume I: Architecture, pages 2–11, St. Charles, IL, 1993.

[2] A. Agarwal et al. The MIT alewife machine: A large-scale distributed-memory multiprocessor. Technical Report 454, MIT/LCS, 1991.

[3] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proc. ACM SIGPLAN Conf. on Prog. Lang. Design and Imp.*, pages 112–125, June 1993.

[4] Cray Research Inc. *CrayT3D Technical Summary*, 1993.

[5] J. Gu, Z. Li, and G. Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Proc. Supercomputing '95*, December 1995.

[6] Hewlett-Packard Co. HP releases high-end technical-solutions road map with consistent architecture to beyond year 2000. http://www.convex.com/.

[7] J. Kuskin et al. The Stanford FLASH multiprocessor. In *Proc. Int. Sym. on Computer Architecture*, pages 302–313, 1994.

[8] D. Lenoski et al. The Stanford DASH multiprocessor. *Computer*, 25(3):63–79, March 1992.

[9] W. Li and K. Pingali. Access normalization: Loop restructuring for NUMA computers. *ACM Trans. on Computer Systems*, 11(4), November 1993.

[10] Z. Li. Propagating symbolic relations on an interprocedural and hierarchical control flow graph. Technical Report CSci-93-87, University of Minnesota, 1993.

[11] D. J. Lilja. Exploiting the parallelism available in loops. *IEEE Computer*, 27(2):13–26, 1994.

[12] T. Lovette and R. Clapp. STiNG: A CC-NUMA computer system for the commercial marketplace. In *Proc. Int. Sym. on Computer Architecture*, pages 308–317, 1996.

[13] T. N. Nguyen. *Interprocedural Compiler Analysis for Reducing Memory Latency and Network Traffic.* PhD thesis, University of Minnesota, to be compeleted, 1996.

[14] T. N. Nguyen, J. Gu, and Z. Li. An interprocedural parallelizing compiler and its support for hierarchical memory research. In C.-H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proc. 8th Int. Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science 1033*, pages 96–110, Columbus, Ohio, August 1995.

[15] A. Nowatzyk et al. The S3.mp scalable shared memory multiprocessor. In *Proc. Int. Sym. on Computer Architecture*, 1995.

[16] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes: The Art of Scientific Computing (Fortran Version).* Cambridge University Press, 1989.

[17] P. Stenstrom, J. Truman, and A. Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In *Proc. Int. Sym. on Comp. Arch.*, pages 80–91, 1992.

[18] J. E. Veenstra and R. J. Fowler. MINT tutorial and user manual. Technical Report 452, University of Rochester, June 1993. Dep. of Computer Science.

[19] Z. Vranesic et al. The NUMAchine multiprocessor. Technical report, University of Toronto, 1995.