Dynamic Data Migration to Eliminate Bank-level Interference for Stencil Applications in Multicore Systems

Wei Hen Lo, Yen Hao Chen, and TingTing Hwang
Department of Computer Science, National Tsing Hua University, R.O.C
turtleevil_1@hotmail.com, tingting@cs.nthu.edu.tw

ABSTRACT

A stencil computation repeatedly updates each point of a d-dimensional grid as a function of itself and its near neighbors. Modern automatic transformation compiler framework can generate efficient tiling parallel stencil codes. Dynamically scheduling parallel stencils significantly improves system performance. However, memory contention problem exacerbates because of less idling cores and more memory requests sent to the DRAM memory. Traditional OS page coloring method which partitions the memory pages in advance can not alleviate the memory contention in dynamic scheduling parallel stencils. To address this issue, we provide a new software/hardware cooperative dynamic data migration method by exploiting the update-and-reuse *property of stencils. We notice that the OS page allocation needs to be aware of the flexibility for dynamic data migration in memory to eliminate the memory interference. Experimental evaluation in a* 8*-core x86 system shows that our method can improve the system performance by* 7*% as compared with dynamic scheduling stencils in* 8*-cores* 4*-memory banks system.*

I. INTRODUCTION

Stencil is one of the most fundamental computational patterns in numerical algorithms. These codes generally achieve low fraction of peak performance. The computational domains involved in stencils include medical and life science, petroleum reservoir simulations, weather and climate modeling, and physics simulations. Stencils may execute tens of thousands of iterations over spatial domain in order to resolve the time-dependent solution accurately. Generally, it takes hours running stencils on supercomputers. Therefore, any performance improvement may reduce the total runtime tremendously. Figure 1 shows an example of stencil application named *Heat 3D* over a *3*-dimensional data space where a grid point of array *B* in next time step *t + 1* is updated using neighboring indices of array *A* in time step *t*. After the computation is finished in each iteration, array *A* and array *B* are swapped and the output data will be reused as the input in next time step.

```
for (t=0; t<timesteps; t++)   {   // time step loop
  for (k=1; k<nz-1; k++) {
    for (j=1; j<ny-1; j++) {
      for (i=1; i<nx-1; i++) {
        // 3-d 7-point stencil
        B[i][j][k] = A[i][j][k+1] + A[i][j][k-1] +
             A[i][j+1][k] + A[i][j-1][k] + A[i+1][j][k] +
             A[i-1][j][k] – 6.0 * A[i][j][k] / (fac*fac);
      }
    }
  }
  temp_ptr = A;
  A = B;
  B = temp_ptr;
}
```

Fig. 1. Stencil : Heat 3D Equation

Over the last two decades, there has been significant improvement in the development of powerful compiler frameworks for dependency analysis and transformation of loop computations with affine bounds and affine array access functions [6]–[12]. These techniques often focus on the most computation-intensive components of scientific and engineering applications such as stencil applications. For stencil applications, compile-time optimization approaches have been developed using a polyhedral abstraction of programs and dependencies [13]. They provide an automatic transformation framework to optimize loop sequences with affine dependencies for parallelism and locality simultaneously which is also called loop tiling. Recent studies of loop tiling techniques for stencil applications also provide opportunities for concurrent start for parallelism [14]. Concurrent start allows parallel programs not to suffer from the pipeline start-up overhead. However, these generated parallel codes contain barriers that will lead to constrained inter-task synchronization. To avoid load imbalance and resource under-utilization, M. M. Baskaran et al. proposed a compiler-assisted dynamic scheduling method to schedule these tiles during run time [1]. The dynamic scheduling method extracts the inter-task dependencies and generates a directed acyclic graph (DAG) of tasks. Thus, the tasks can be scheduled dynamically on the cores without barriers, which improves system efficiency and scalability. Take *Heat 3D* as an example. When the parallel code is generated using dynamic scheduling model, the performance is improved by *23%* as compared with simple synchronization code with barriers.

Since the working sets of stencils are often much larger than the last level cache, memory contention is still a problem, especially when the number of processors scales up. In our experiments, dynamic scheduling stencils (ex: *Heat 3D*) suffer even more (*446%*) from the memory interference (memory bank conflicts) than barrier-synchronization stencils. Because the tasks are scheduled during run time, it is hard to alleviate memory contention by mapping data to memory banks statically. Once tasks in different DAG level being executed in parallel, pre-partitioned memory regions of these tasks may contend with each other in DRAM memory.

Many previous work has focused on managing memory contention problem in multi-programmed workloads [2]–[4]. For example, ATLAS [3] designs a memory controller to prioritize those memory requests from memory non-intensive application. Thus, those requests will be served first and will not contend with requests from memory intensive application. TCM [4], which classifies threads into memory intensive group and memory non-intensive group, improves not only the overall system performance but also the system fairness. Lei Liu et al. propose a software memory partition approach to eliminate bank-level interference, which applies page coloring method in OS to map the data of threads to specific banks [2]. Since OS partitions memory space in advance, interleaving effect will be propagated to the main memory and alleviate memory interference.

The above work mainly focuses on the memory interference in multi-programmed workloads. When parallel multi-threaded applications are addressed, Eiman Ebrahimi et al. propose a memory controller scheduling policy to improve parallel application performance [5]. They first estimate likely-critical threads based on lock contention information and progress of threads in parallel loops. Then, the memory controller prioritizes likely critical threads and shuffles priorities of non-likely-critical threads to reduce memory contention. This method works well while the structure of the parallel multi-threaded application is a series of simple parallel loops. However, for modern multi-threaded applications, parallel structures may be complex such as a pipeline or a group of tasks which are scheduled dynamically. It is hard to detect critical tasks during run time. In addition, applications such as parallel stencils have no lock at all. Hence, their method is not so effective for parallel stencils.

To solve the memory contention of various kinds of automatic parallel stencils, we develop a dynamic data migration method to migrate data

dynamically. The goal of our dynamic data migration method is to move data dynamically so that accesses to memory banks are interleaved and the memory contention caused by different cores is reduced. It includes the cooperation between OS and memory controller to make sure the migration process runs correctly and efficiently.

II. Motivation

In Section I, we introduced the compiler-assisted dynamic scheduling method for parallelization of stencil applications. The dynamic scheduling method is able to decrease the inefficiency caused by inter-task barrier synchronization and increase load balance. It is especially suitable for stencils because there may be heavy load imbalance when the dimension of stencils is high. Figure 2 shows the execution time of many stencil applications by dynamic scheduling programming model and barrier-synchronization programming model. The experimental environment setting is a *8*-cores CMP with *16*KB private L1 caches, *1*MB L2 cache, and *4* memory banks. We normalize the execution time to that of barrier-synchronization. From this figure, we can see that the execution time of the dynamic scheduling stencils are *28.1%* faster than that of barrier-synchronized stencils in average.

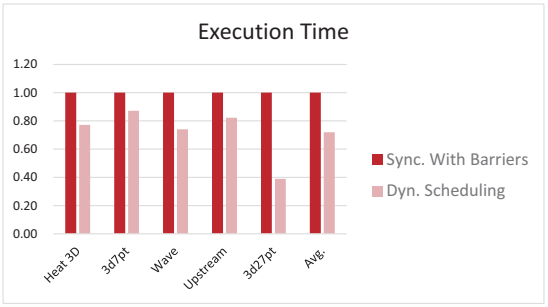


Fig. 2. Execution Time of Stencils in Different Program Models

Although dynamic scheduling programming model improves system performance, it also exacerbates the memory contention problem due to two reasons. First, the working sets of stencils are usually larger than the last level cache. Second, the dynamic scheduling allows more number of concurrent executing cores. Therefore, there are more memory requests sent to the memory controller per memory cycle. To further improve system performance, the problem of memory contention must be solved.

After carefully analyzing the type of contention, we notice that a large portion of memory bank conflicts are caused by only limited number of memory pages because stencils keep updating and reusing data. Considerable portion of bank conflicts is caused by stencils updating blocks of one task and reusing the same blocks in the the following iterations. We call it *update-and-reuse* memory access. If many requests from different tiles running on different cores collide in the same memory bank the first time, it is very possible that the subsequent requests accessing the same blocks will collide again in the future.

In order to verify this observation, we profile some stencil applications. Figure 3 shows the percentage of the bank conflicts caused by *update-and-reuse* blocks. In average, *67%* of bank conflicts are caused by *update-and-reuse* blocks. If we can migrate contending *update-and-reuse* blocks into other bank, a large amount of bank conflicts will be eliminated.

We use the following examples to illustrate our observation. Figure 4 is a directed acyclic graph (DAG) of tasks in a stencil. It represents the tiling tasks and their dependency. A child task can not be executed on processors until its parents tasks finish. In this example, we target the access of memory block *C1* of task *T3* and the access of memory block *C0* of task *T8*.

When the parallel model is barrier-synchronization, tasks at each DAG level need to wait for a barrier as shown in Figure 5(a). It means that *T8* must not execute with *T3* concurrently because of the barrier. Supposed that *C0* is mapped to bank *A* when *T1* first accesses *C0*, *C1* is also mapped to bank *A* when *T3* first accesses *C1*, and *T8* accesses *C0* after *T1* and *T3*

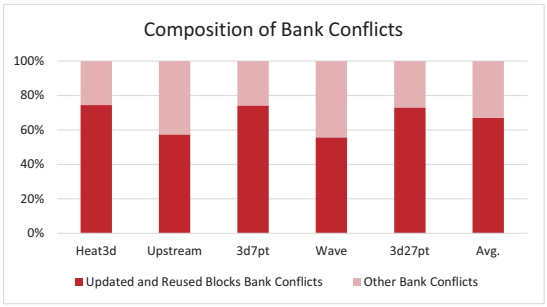


Fig. 3. Composition of Bank Conflicts in Dynamic Scheduling Stencils

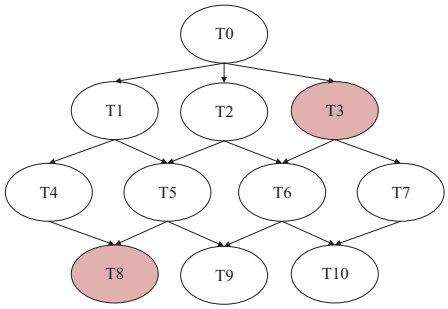


Fig. 4. DAG of a Stencil

finished as shown in Figure 5(a). There is no bank conflicts between the accesses of *C0* and *C1* and their subsequent accesses as shown in Figure 5(b), since the tasks *T3* and *T8* executes sequentially. The memory accesses denoted as *R* and *W* represents the request is a read request or write request.

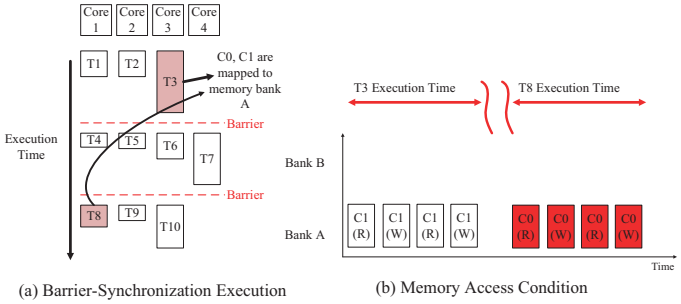


Fig. 5. Barrier-Synchronization and Memory Access Condition

Now, suppose the same example is executed by dynamic scheduling programming model as shown in Figure 6(a), where tasks *T3* and *T8* also execute sequentially. The requests of *C0* and *C1* will not contend with each other in the memory as shown in Figure 6(b), which is similar to Figure 5(b).

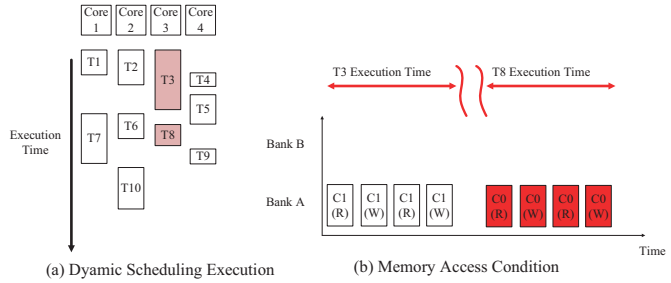
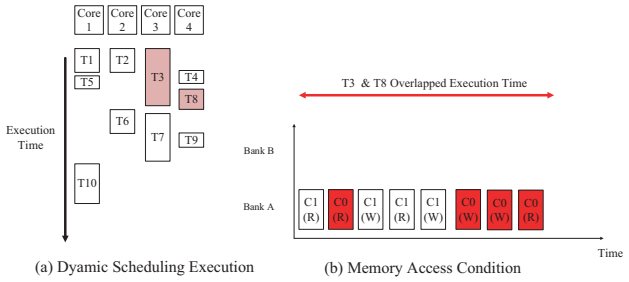


Fig. 6. Dynamic Scheduling and Memory Access Condition : No Conflicts

Suppose another possible execution condition of the dynamic scheduling stencil is shown in Figure 7(a). In this case, task *T3* and *T8* executes in parallel. Let the first requests accessing *C0* and *C1* contend with each other in memory bank *A*. Then, the subsequent memory accesses to *C0* and *C1*

cause more serious bank contention because data intensive applications such as stencils continuously update and reuse the same memory blocks as shown in Figure 7(b).



(a) Dyamic Scheduling Execution (b) Memory Access Condition

Fig. 7. Dynamic Scheduling and Memory Access Condition: Conflicts

The above two dynamic scheduling orders show that it is hard to predict the status of memory contention because we can not predict whether two tasks will execute in parallel or sequentially. Therefore, simply partitioning memory statically is not an effective solution to solve memory contention of dynamic scheduling stencils.

Figure 8 shows a solution to eliminate the memory contention of dynamic scheduling stencils. The symbols *R* and *W* represent that the request is a load request or a store request. If we find that block *C0* contends with block *C1* at time t_i at bank *A* and block *C0* is ready to be written to memory at time t_j, we write *C0* to bank *B*. Then, the following requests accessing *C0* will no longer collide with the requests accessing *C1*. We call this lazy migration. In general, if the updated blocks of concurrent executing tasks can be migrated to different memory banks, the memory contention problem will be alleviated.

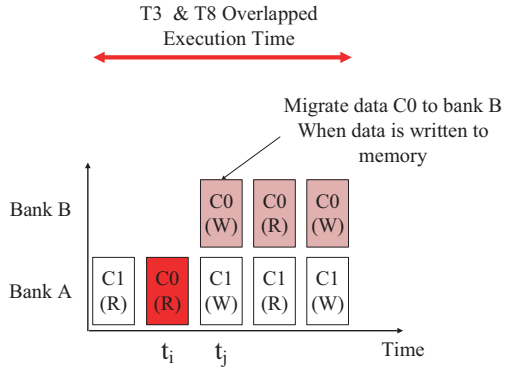


Fig. 8. Eliminate Memory Contention of Reusable Data

III. Methodology

In the following subsections, we first show the whole system flow from the automatic transform compiler framework of sequential stencil source code, the Operation System, the memory controller to DRAM memory in Section III-A. Next, a special function added to the operation system will be explained in Section III-B. In the end, we demonstrate how to design the memory controller to receive information from Operation System and allow us to migrate data between different memory banks.

A. Overview of System Flow

In this section, we first briefly introduce the flow of executing stencil codes shown in Figure 9. In the beginning, given a sequential stencil code, the compiler first uses a scanner and parser to construct an abstract syntax tree. Then, it uses polyhedral model to represent the data dependencies. After analyzing data dependencies, the compiler will perform tiling transformation to minimize the communication among tiles. The tiling transformation is to find a series of proper hyperplanes in the transformation space and uses these hyperplanes to partition the transformation space into rectangle tiles. In the end, all information, including dependencies, iteration spaces, transformations are all fed into a code generator tool such as Cloog [15] to generate the final parallel code.

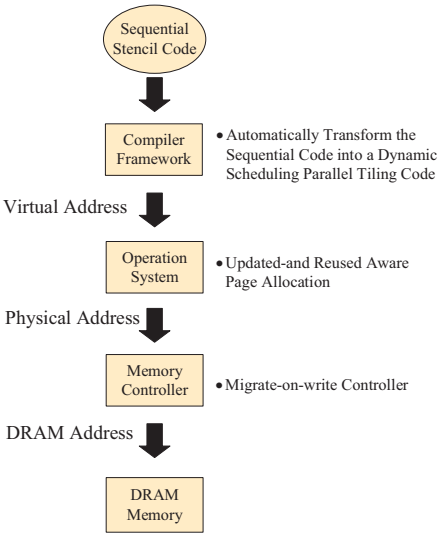


Fig. 9. Flow of Software-hardware Cooperation Data Migration Method

Figure 10 shows an example of stencil tiling technique, the *x*-axis represents the data array space and the *y*-axis represents the time frames. The red arrows represents the dependencies between iterations and the black arrows represents the inter-tile dependencies. Each tile can be seen as a task which will be executed on a core. Since the inter-tile dependencies are extracted by compiler, the directed acyclic graph (DAG) of tasks can be constructed and used during run time.

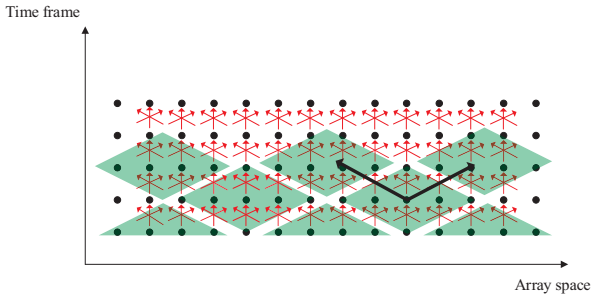


Fig. 10. Example of Stencil Tiling Technique

Based on the DAG, in the second step, the OS uses a task queue to dynamically schedule these tasks. Figure 4 illustrates the DAG of the tasks in the above stencil example. The arrows is the inter-tasks dependencies. The tasks are prioritized based on the length from current level to the bottom level.

During run time, when a task of the stencil program accesses the memory page in the data arrays for the first time, the virtual address of the page will be found not mapped to any physical frame yet. The memory management unit (MMU) will signal a page fault to OS. The Operating System (OS) then finds a free physical frame from the free-list and set up a new page table entry to map it to the requested virtual address.

Once the memory request is sent from core to memory hierarchy system, it first looks up the TLB and finds the mapped physical address. Then, the request will use this physical address to lookup the first level caches. If there is a cache miss, it will try to look up the next level cache. Once there is a last level cache miss, the request will be sent to the memory controller. In the third step, based on the address mapping scheme of the system, the actual physical DRAM memory address is translated by the memory controller. Assume that there is a system with *1* channel, *1* DIMM, *1* rank, *4* banks in the DRAM memory. Figure 11 shows an example of

one kind of address mapping scheme where the 17_{th} and 18_{th} bits of physical address decide which memory bank the physical frame is located at. We call 17_{th} and 18_{th} bits as *bank-bits*. If two addresses are mapped to different banks, the accesses to these locations are interleaving.

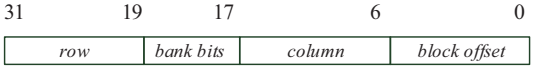


Fig. 11. Simple Address Mapping Scheme of 4 Memory Banks

To solve memory contention problem, we proposed a lazy migration technique. Three main problems related to the migration policy required to be solved are:

1) Where to migrate the data?
2) When to perform the lazy migration?
3) How to design a memory controller to support the lazy migration?

To solve these problems, we target on the modifications of step *2*, the OS level and step *3*, the memory controller in this flow.

B. Updated-and-Reused Aware Page Allocation Policy in OS

Ideally, we would like to have requests to memory bank to be balanced. Therefore, during the run time, the memory controller should migrate one contending *updated-and-reuse* data block to other memory bank. In other words, once a task starts to execute on a core, the memory controller gradually moves the *updated-and-reuse* data blocks to a specific memory bank for that core. As a result, the *updated-and-reuse* data blocks from the tasks executing in parallel will be migrated to different memory banks. An obvious next question is where the destination is for migration. One way is to select a destination located at the same relative location but different bank. That is, the address bits of the destination are the same as those of migrated data but *bank-bits*.

However, if there is some useful data already located in the destination of migrated data, the useful data will be overwritten by the migrated data. Once some tasks try to access that useful data, they will load the wrong data and lead to system crash. To solve this problem, we modify the OS to reserve spare free frames located in different memory banks for data migration when a page is loaded to memory for the first time. Since there is free spare space in other memory bank, the data migration will not destroy useful data.

In the following paragraphs, we will illustrate how we modify OS for the dynamic data migration method. Our modifications are mainly for the memory correctness after activating the data migration method.

The way to maintain memory correctness for dynamic data migration method is to reserve spare memory space in DRAM memory. Therefore, we target on the modification of the OS page allocating policy. Our goal is to migrate data blocks accessed by each task to reserved memory bank. Hence, when the OS receives a page fault, it tries to find a group of free frames whose physical address is only different in the *bank-bits*. The OS will map the virtual page to one of them, and reserves others as spare free pages for data migration. Therefore, memory controller is able to write data to any of the spare free frames in this group dynamically later and do not need to worry about the memory correctness. Note that modern DRAM memory allows OS to maintain multiple pools of free frames located at different memory banks [17].

Our new page allocation policy is illustrated in Figure 12. Figure 12(a) shows the original page allocation policy, where the memory management unit (MMU) maps the virtual page with a free physical frame. Assume there is *4* memory banks in the DRAM memory and the address mapping scheme is shown in Figure 11. Let a physical frame *Frame A* be located inside a certain memory bank as shown in Figure 12(a). Figure 12(b) illustrates our new page allocation method. Suppose we have *2* banks as a group, i. e., we reserve one frame in other bank of the same group for migration. Let 18_{th} bit be used to locate a reserved frame in the same group as shown in Figure 13. As shown in Figure 12(b), *Frame A* and *Frame B* are in the same group and their physical addresses are different only in the 18_{th} bit. These reserved frames play as a role of free space in the DRAM memory which allows the memory controller to migrate data from original frame to reserved frames.

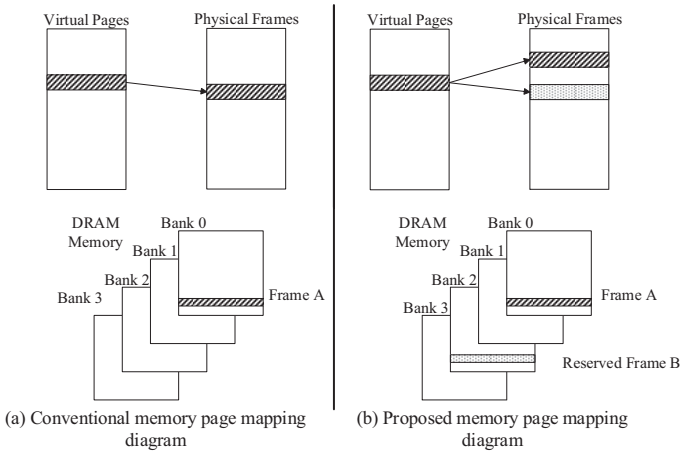


Fig. 12. Example of Our Spare Page Allocation Method

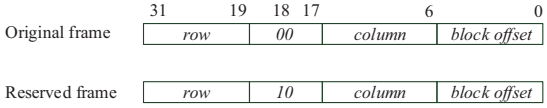


Fig. 13. Reserved Pages Scheme of 2 Memory Banks

C. Migrate-On-Write

In this section, we will introduce the key idea of our lazy migration policy, which is called *Migrate-On-Write*. *Migrate-On-Write* means that the memory controller migrates a data block to other bank only when it is a write-back request from last level cache. The main advantage of *Migrate-On-Write* is two folds. First, it is a lazy migration. The data is migrated only when it is written to memory. There is no extra memory requests for migration. Memory controller only needs to change the destination when the memory request is a write. Second, it migrates the data gradually (only migrate-on-write). Traditional data migration policy such as page migration requires to migrate a whole page to another memory bank. Therefore, the memory controller generates a lot of read and write requests to migrate data. These extra memory requests will cause more serious bank conflicts and fail to alleviate the memory contention. In our design, we migrate data at block level (i.e., a cache block) defined by write-back.

In order to access these data blocks in DRAM memory, we need to add extra hardware *mapping table* to record migrated locations of data blocks. Note that only bank locations are needed to record because all other bits remain the same. To access the *mapping table* in memory, we need an index to the table. Thus, *mapping table index* is designed which is appended to a physical address. When a page fault signals the OS to map a free frame, the OS also assign *mapping table index* in *page table*. The first bit, *movable*, of *mapping table index* is designed to record if the current frame has its reserved frame and can be migrated to other bank. The rest of bits are *frame index*. The size of *frame index* depends on the number of physical frames that are allowed to be migrated. For example, if *16K* physical frames have reserved frames for migration, the number of bits of *frame index* is *14* bits as shown in Figure 14.

D. Memory Controller for Bank-level Interference Elimination

As Figure 15 shows, we add an extra hardware *mapping table* in the memory controller to perform bank mapping. The *frame index* of *mapping table index* is an index to the *mapping table*. In the *mapping table*, each entry indexed by a *frame index* contains bank information of **all** blocks in the same frame.

When a block is accessed, *frame index* is used to index the entry in *mapping table* and the *block address* of its physical address is used to get

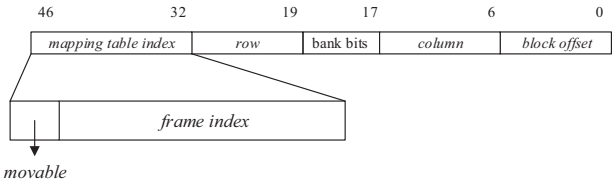


Fig. 14. Address Mapping Scheme for Data Migration

the destination bank. For example, let there be two banks in one group, where the 18_{th} bit decides destination bank, *64* blocks in one frame and an address *a* as shown in Figure 15. First, the *movable* is checked. Since it is *1*, the mapping table is accessed by using *frame index 00000000000000* as index. Next, since the block number is *000001*, the second block entry is accessed and it is *1*. Therefore, the translated bank address is *11* and its translated address *ta* is as shown. As to the delay of mapping table look-up, since it happens when a request is added to the memory controller queue, this check-up will not extend the critical path in the common case because queueing delays at the memory controller are substantial.

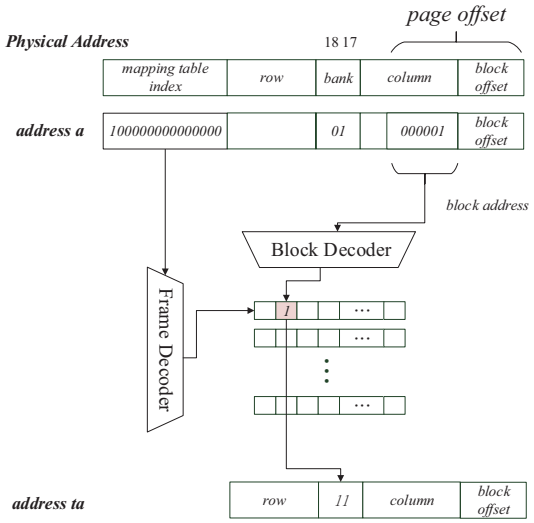


Fig. 15. DRAM Address Translation Scheme for Data Migration

As mentioned in previous section, the key of our data migration policy is *Migrate-On-Write*. When there is a write-back request coming from last level cache, the memory controller checks the *movable* bit to see if this block has reserved space for migrating. If the first bit of mapping table index is not set, the memory controller writes the data according to the bank bits in physical address. Otherwise, the memory controller finds a bank in a group for writing. Then, the next problem is which bank to write? Should the memory controller write the data to the current bank or migrate it to any other bank in a group?

To decide which bank to write, the key idea is to balance the loading of each memory bank. In OS, frames are reserved as spare frames in a group, we want to balance the use of frames in the group. For a task running on a core, we use the *core ID* to select the bank in a group. Assume there are *8* cores, whose *core ID* are *b000* to *b111*, *4* memory banks and the number of frames in a group is *2*. Suppose two frames in banks *01* and *11* are assigned as a group. Their frame addresses are the same except the 18_{th} bit. Since there are two frames in a group, we need only one bit from the *core ID*. Let the Least Significant Bit of *core ID* be used to select a bank in the group. Now, a write-bank request is sent to memory controller and its bank-bits are *01*. If the memory request is from *core ID 00*, then frame in bank *01* is written (no migration). If the request is from core *11*, then frame in bank *11* is written (data is migrated). Note that since *0* and *1* appearing in each bit of *core ID* are equal, we are able to balance the requests to each bank.

To decide when to turn on/off the dynamic data migration mechanism, we set a default threshold for the miss rate of last level cache. The state-of-the-art tools such as *LiMit* [16] are already able to correctly monitor per-thread behavior (e.g.,cache miss rate, memory bandwidth etc.) with negligible overhead. Therefore, the OS can dynamically change its policies from these tools. If the miss rate grows higher than a threshold, the OS will turn on the data migration. Otherwise, it will turn off the data migration. In our experimental setting, we set this default value as *0.2* misses per thousand instructions (MPKI).

IV. Experimental Results

This section presents the experimental results of our proposed method. In section IV-A, we will introduce our simulation environment. Section IV-B shows the overall result of our experiments.

A. Simulation Environment

Environment: We evaluate our method using SIMICS [18] + GEMS simulator [19] to generate a cycle-accurate x86 system running Linux *2.6.15*. The processor model is a simple in-order pipeline model, leaving each core with a fixed throughput of *1* instruction per cycle. The setting in detail is described in Table I.

Compiler Framework: Different branches of Pluto [13] are integrated and barrier-synchronization and dynamic scheduling parallel stencils codes are generated.

Workloads: We pick five high-dimensions stencils for evaluation because these programs have higher last level cache miss rate. Hence, these programs may suffer more from the memory contention in DRAM memory. These programs are *3D-Laplacian (3d7pt)*, *Heat 3D*, *27 points stencils (3d27pt)*, *Wave* and *Upstream* [13], [20]. The visualization of the stencil structures are shown in Figure 16. The stencil program *Wave* is a little bit different from other benchmarks since it depends on two previous time steps.

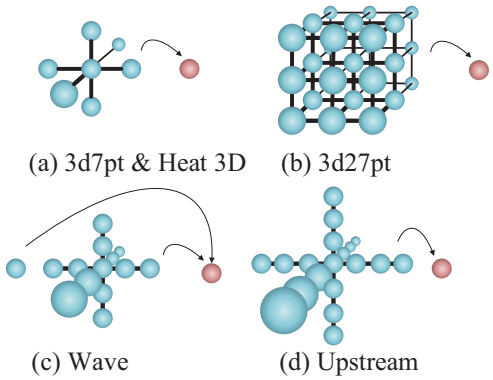


Fig. 16. Structures of Stencil Workloads

B. Comparison Results

In the first experiment, we execute the parallel stencils with different programming model in a *8*-cores, *1* memory channel, *4* memory banks multi-core system. In each stencil application, we test the following schemes: (i) This first scheme, denoted as *Sync.*, represents that the programming model of the stencils is barrier-synchronized, the OS applies default page allocation, and the memory controller is not modified. (ii) The second scheme is denoted as *Sync. + OSC.*. In this scheme, the programming model of the stencils is barrier-synchronized, and the OS applies OS page coloring method to statically partition the memory space. (iii) The third scheme is denoted as *Sync. + OSC + Ours*. This scheme represents that the programming model is barrier-synchronized, OS page coloring method is still applied, but the data can be migrated dynamically through our data migration method. (iv) The fourth scheme is denoted as *Dyn.*. In this scheme, the programming model is dynamic scheduling, the OS applies default page allocation, and the memory controller is not modified. (v) The fifth scheme is denoted as *Dyn. + OSC.*. This scheme represents the programming model is dynamic scheduling and the OS applies OS page coloring method. (vi) The sixth scheme is denoted as

TABLE I
SIMULATED SYSTEM PARAMETERS.

Processor Pipeline	2GHz processors, single pipeline
L1 Cache	16KB, 64B lines, 8-way assoc., Hit latency 1 cycles
L2 Cache	1MB, 64B lines, 16-way assoc., Hit latency 10 cycles
Main Memory	1GB, FR-FCFS, Micron DDR2-800 timing parameters, 1 Channel, 1 Rank, 4 Banks
Interconnect	MESH, dimensional routing, NOC routers
OS	Linux 2.6.15
Extra Hardware Entries	$16K \times 64$ $entries$, Access latency 10 cycles

Dyn. + OSC. + Ours. This scheme represents the programming model is dynamic scheduling and the system performs OS page coloring method and our data migration method.

Figure 17 compares the execution time for all stencil applications under the schemes described above. For each application, all bars are normalized to *Sync.*. We first compare the results of *Sync.*, *Sync. + OSC.* and *Sync. + OSC + Ours*. In average, compared to *Sync.*, *Sync. + OSC.* improves the performance by *3%* and *Sync. + OSC + Ours* improves the performance by *3%*. The result shows adding dynamic migration is not so useful because statically assigning pages to banks is good enough under static scheduling of task. However, when we compare the performance of scheme *Dyn.*, *Dyn. + OSC.*, *Dyn. + OSC. + Ours*, *Dyn. + OSC.* degrades the performance by *4.8%*. The reason is because partitioning memory space statically may cause more memory contention if tasks in different DAG level executing in parallel. When we compared *Dyn. + OSC. + Ours* to *Dyn.*, *Dyn. + OSC. + Ours* improves the performance by *7%* in average. Our data migration method can migrate the data evenly to different memory banks. Therefore, it successfully reduces memory contention.

In summary, *Dyn. + OSC.* exacerbates the memory contention of dynamic scheduling stencils while *Dyn. + OSC. + Ours* can successfully alleviate the memory contention of dynamic scheduling stencils. In average, our data migration method improves the performance of all stencils compared to the default setting by *3%* when the programming model is barrier-synchronized and improves the performance by *7%* when the programming model is dynamic scheduling. Compared to the system which only applies OS page coloring method, our method improves the performance of all stencils *11.3%* when the programming model is dynamic scheduling in average.

Figure 18 shows the comparison result of total bank conflicts under all schemes. It shows that the total bank conflicts of *Dyn.* is *415%* more than the total bank conflicts of *Sync.* in average. The *Sync. + OSC.* and *Sync. + OSC + Ours* reduces the total bank conflicts by *13%* and *14%* compared to *Sync.*, respectively. Compared to *Dyn.*, the *Dyn. + OSC.* increases the total bank conflicts by *24.3%* and *Dyn. + OSC. + Ours* reduces the total bank conflicts by *28.3%* in average.

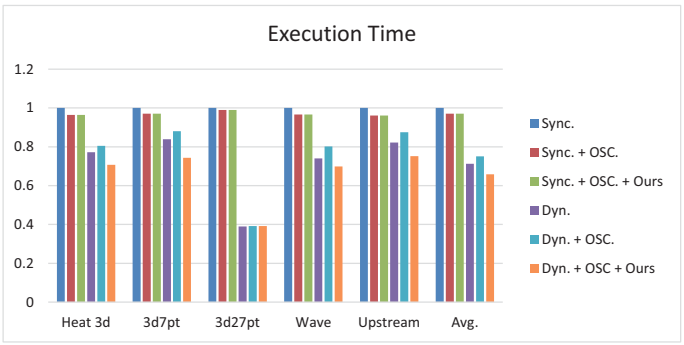


Fig. 17. Performance of Barrier-Synchronization Baseline, Barrier-Synchronization with Our Method, Dynamic Scheduling Baseline, and Dynamic Scheduling with Our Method

V. CONCLUSION

In this paper, We have designed and evaluated a new dynamic data migration method targeting dynamic scheduling parallel stencils. The

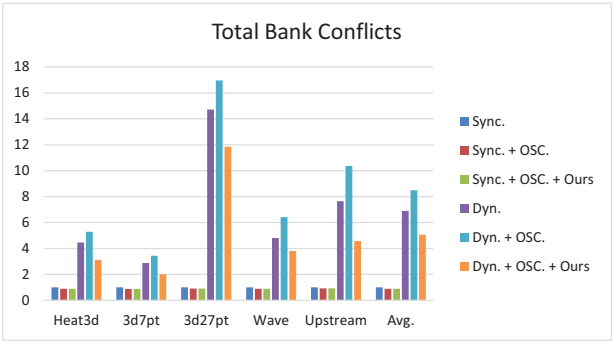


Fig. 18. Total Bank Conflicts of Dynamic Scheduling Baseline, and Dynamic Scheduling with Our Method

barrier-synchronized parallel code generated by automatic parallelization approaches suffers from the excessive synchronization in the form of barriers. On the other hand, the DAG-based dynamic scheduling parallel program breaks the barriers and improve load balance for effective parallel execution on multi-core systems. However, it also exacerbates the memory contention problem because there are more concurrent running cores generating memory requests. Traditional OS page coloring method which partitions the memory in advance fails to alleviate the memory contention because the scheduling order of tasks is decided during run time. Therefore, we have proposed a dynamic data migration method to solve this problem. The experimental results show that our method can improve the system performance by *7%* compared with default setting in a *8* cores and *4* memory banks system.

REFERENCES

[1] M. M. Baskaran, N. Vydyanathan, U. K. Bondhugula, J. Ramanujam, A. Rountev, P. Sadayappan, "The Compiler-Assisted Dynamic Scheduling for Effective Parallelization of Loop Nests on Multicore Processors," *PPoPP'09*, pp. 219-228, April 2009.
[2] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, C. Wu, "A software memory partition approach for eliminating bank-level interference in multicore," *PACT'12*, pp. 367-376, September 2012.
[3] Y. Kim, D. Han, O. Mutlu, M. Harchol-balter, "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers," *HPCA'12*, pp. 1-12, February 2012.
[4] Y. Kim, M. Papamichael, O. Mutlu, M. Harchol-Balter, "Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior," *MICRO'10*, pp. 65-76, December 2010.
[5] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, O. Mutlu, and Y. N. Patt, "Parallel Application Memory Scheduling," *MICRO'11*, pp. 362-373, December 2011.
[6] C. Ancourt and F. Irigoin. "Scanning polyhedra with do loops." *PPoPP'91*, pp. 39-50, February 1991.
[7] C. Bastoul. "Code generation in the polyhedral model is easier than you think," *PACT'04*, pp. 7-16, September 2004.
[8] P. Feautrier. "Dataflow analysis of array and scalar references," *IJPP*, 20(1):23-53, 1991.
[9] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, O. Temam. "Semi-automatic composition of loop transformations," *IJPP*, 34(3):261-317, June 2006.
[10] A. Lim. "Improving Parallelism And Data Locality With Affine Partitioning." PhD thesis, Stanford University, August 2001.
[11] W. Pugh, "The Omega test: a fast and practical integer programming algorithm for dependence analysis," *Communications of the ACM*, 8:102-114, August 1992.
[12] F. Quillere, S. V. Rajopadhye, and D. Wilde. "Generation of efficient nested loops from polyhedra." *IJPP*, 28(5):469-498, 2000.
[13] U. Bondhugula, J. Ramanujam, and P. Sadayappan. "Pluto: A practical and fully automatic polyhedral parallelizer and locality optimizer." Technical Report OSU-CISRC-10/07-TR70, The Ohio State University, Oct. 2007.
[14] V. Bandishti, I. Pananilath, and U. Bondhugula. "Tiling Stencil Computations to Maximize Parallelism," *SC'12*, pp. 1-11, November 2012.
[15] CLooG: The Chunky Loop Generator. http://www.cloog.org.
[16] J. Demme, S. Sethumadhavan, "Rapid Identification of Architectural Bottlenecks via Precise Event Counting." *ISCA'11*, pp. 353-364, June 2011.
[17] JEDEC. Standard No. 21-C. Annex K: Serial Presence Detect (SPD) for DDR3 SDRAM Modules, 2011.
[18] P. Magnusson et al. "Simics: A full system simulation platform." Computer, 35(2), Feb 2002.
[19] M. M. K. Martin et al. "Multifacets general execution-driven multiprocessor simulator (GEMS)"
[20] M. Christen, O. Schenk, H. Burkhart, "PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures," *IPDPS '11*, pp. 676-687, May 2011.