Improving Bank-Level Parallelism for Irregular Applications

Xulong Tang[1], Mahmut Kandemir[1], Praveen Yedlapalli[2], Jagadish Kotra[1]
Pennsylvania State University[1]
University Park, PA, USA
Email: {xzt102, kandemir, jbk5155}@cse.psu.edu
VMware, Inc.[2]
Palo Alto, CA, USA
Email: praveen.yadlapalli@gmail.com

***Abstract*—Observing that large multithreaded applications with irregular data access patterns exhibit very low memory bank-level parallelism (BLP) during their execution, we propose a novel loop iteration scheduling strategy built upon the inspector-executor paradigm. A unique characteristic of this strategy is that it considers both bank-level parallelism (from an inter-core perspective) and bank reuse (from an intra-core perspective) in a unified framework. Its primary goal is to improve bank-level parallelism, and bank reuse is taken into account only if doing so does not hurt bank-level parallelism. Our experiments with this strategy using eight application programs on both a simulator and a real multicore system show an average BLP improvement of 46.8% and an average execution time reduction of 18.3%.**

I. Introduction

To maximize the performance of multithreaded applications mapped to multicores/manycores, one needs to consider *end-to-end data access performance*, not just the cache performance. In fact, trying to maximize LLC (Last-Level Cache) hit rates (which is the main goal of many compiler schemes) does not guarantee good, let alone being optimal, end-to-end data access performance [1], [2], [3]. This is because off-chip accesses can consume a lot of cycles, but more importantly, latencies they experience are not uniform, being dependent on several factors such as bank-level parallelism, row-buffer locality, memory scheduling policy, etc. Therefore, an end-to-end data access optimization strategy should consider cache performance as well as performance of the LLC misses. Unfortunately, while there are some recent hardware-based works targeting off-chip accesses [4], [5], [6], software works targeting off-chip accesses are still in their infancy.

One of the important factors that influence the performance of LLC misses is ”bank-level parallelism” (BLP), which refers to the number of concurrently-served memory accesses by different memory banks in the system. Note that BLP is a measure of memory-level parallelism since in the ideal case one would want all the banks to be busy in serving memory requests (LLC misses). Note also that, in order to have high BLP, one needs (1) a large number of concurrent LLC misses and (2) a balanced distribution of these misses over the available memory banks. To achieve (1), LLC misses need to be clustered and, to achieve (2), misses should be reorganized either through code transformations or data layout transformations. While these tasks are not trivial and have not received much attention so far from the compiler and runtime system communities, they are even harder in the context of *irregular applications,* i.e., applications whose data access patterns cannot be completely analyzed at compile-time (e.g., index array-based calculations in scientific codes).

978-1-5090-3508-3/16/$31.00 © 2016 IEEE

This paper presents a novel strategy to optimize BLP of index array-based irregular programs. Our strategy, built upon the inspector/executor paradigm [7], reorganizes LLC misses at runtime to maximize BLP. To our knowledge, this is the first compiler work that targets improving BLP in irregular applications. The contributions of this work can be summarized as follows:

- It presents experimental evidence, using eight multithreaded irregular applications, showing that (1) the BLP of the original versions of these irregular applications are very poor in general, (2) simply maximizing memory-level parallelism (by clustering misses) does not bring significant improvements, and (3) maximizing bank-level parallelism on the other hand can bring significant performance benefits.
- Drawing insights from this motivational data, it next proposes a compiler/runtime based *loop iteration scheduling strategy* to maximize BLP. A unique characteristic of this strategy is that it considers both bank-level parallelism (from an inter-core perspective) and bank reuse (from an intra-core perspective) in a unified framework. Its primary goal is to improve bank-level parallelism, and bank reuse is taken into account only if doing so does not hurt bank-level parallelism.
- It gives experimental evidence showing the effectiveness of the proposed strategy. We evaluate the proposed approach in both simulator (to collect detailed off-chip statistics and compare it against hardware-based memory schedulers) and real multicore hardware. Our results indicate that the proposed strategy reduces execution time by 18.3% on average.

II. Background on DRAM and BLP

DRAM in modern systems is composed of various components like ranks, banks, and sub-arrays. Cores in a multicore access the data from off-chip DRAM through a component called Memory Controller (MC). Upon a last-level cache (LLC) miss, read/write requests are mapped to a specific MC based on the address mapping which we describe below. Requests to a DRAM are queued in a buffer at the MC, and are issued to the DRAM by MC. Figure 1 shows the

basic organization of a DRAM and how it is connected to a multicore. Each MC manages a DRAM module also referred to as DIMM by issuing commands over address/data buses, also referred to as channel. Internally, each module is organized hierarchically as ranks, banks, and sub-arrays. We do not consider sub-arrays in our hierarchy as they are less common. Each DIMM is made up of multiple ranks. Each rank consists of multiple banks and all the banks in a rank share the timing circuitry. Each bank consists of a set of sense-amplifiers, referred to as *row-buffer,* where the memory row is loaded to before the data corresponding to the request is sent back over the channel. In an *open-row policy,* the row previously accessed is left open in the row-buffer. Consequently, if there is a request to the same row in the row-buffer, it need not be activated again, and hence incurs low latency resulting in a row-buffer hit. If there is a request to a different row, the current row in the row-buffer needs to be precharged and the new row has to be activated before the data is accessed and such a scenario is widely referred to as *row-buffer conflict.* Many works in the past proposed hardware-based schedulers that take advantage of the open-row policy. One such scheduler, FR-FCFS [8], [9], prioritizes accesses that target the current row in the row-buffer over other accesses.

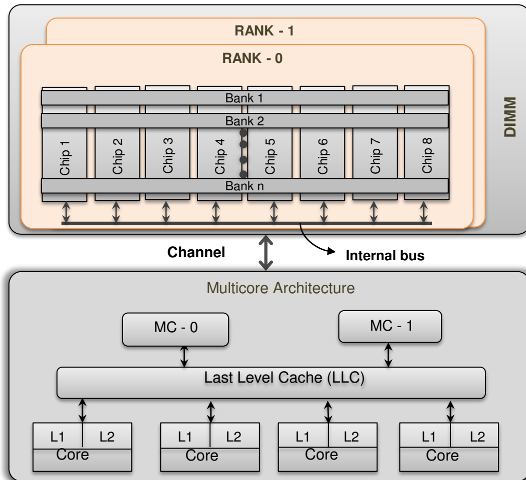


Fig. 1: DRAM organization and DRAM-multicore interfacing.

Address mapping, also referred to as *interleaving,* governs how data are distributed across various components in the DRAM. This mapping (from physical addresses to memory banks) is decided statically (at hardware design time), and depending on the mapping scheme employed by the hardware, a request to a physical address can result in an access to a channel/rank and bank. Various interleavings are possible at each level in the memory hierarchy like caches, channels and banks. Two widely-employed interleavings are *cache line level* and *page level.* Address mapping plays an important role in determining the performance of the system as it effects both the locality and parallelism in the memory hierarchy. Figure 2 shows how a physical address is mapped to a channel/rank and bank based on page-level interleaving. The least significant 12-bits represent the page offset for a 4KB page. Assuming there are 4 MCs, the next 2 bits (bits 12 and 13) represent the channel id where this physical address is mapped. In a corresponding channel, assuming there are 4 ranks, the next 2 bits (bits 14 and 15) represent the rank where this physical address is mapped. Once the rank is determined, assuming there are 8 banks in a rank, the next 3 bits (bits 16, 17 and 18) are the *bank bits* and determine which bank this physical address is mapped to.

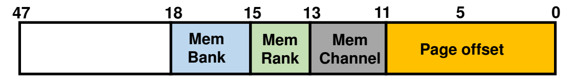


Fig. 2: Page interleaved address mapping.

Ideally, consecutive requests to different pages in time domain should be mapped to different banks such that these independent requests are served in parallel. This is commonly referred to as **bank-level parallelism** (**BLP**). In this paper, we define BLP as

> *the average number of requests being served in parallel by all the banks in the DRAM when at least one request is being served by any bank.*

This definition for BLP is same as the one used in [6]. There exist various hardware-based schemes to improve BLP, and we compare our work to a few of them in this paper.

III. Motivational Results

The curves marked as "Original" in Figure 3 give BLP values for a period of 2 billion cycles for our applications on a 12-core, 64-bank system. One can make two critical observations from these results. First, most of these applications do not perform well from a BLP angle. In fact, the average BLP values for applications HPCG and GMR are 19.7 and 16.6, respectively, as given in Figure 4. Second, as far as BLP is concerned, each of these applications exhibits a quite repetitive pattern. This is primarily because these index array-based irregular applications have an outermost "timing" loop that iterates either for a fixed number of iterations or until a convergence criterion is met. In fact, this repetitiveness (not just in terms of BLP, but also in terms of access patterns and cache statistics) is the main reason why the inspector/execution paradigm (explained later) works well for irregular applications.

To illustrate the influence of these low BLP values on performance, we present in Figure 5, the execution times collected using our simulator. The first bar for a benchmark in this bar-graph plots the execution time of the original application in seconds. We see that, without performing anything special regarding BLP, the execution times of our applications vary between 55.6 seconds and 124.4 seconds. At this point, one may suggest that optimizing memory level parallelism (MLP), that is, simply increasing the *burstiness* of off-chip memory requests can help us improve BLP and ultimately reduce the overall application execution times. To check the validity of this, we implemented in our simulator a strategy where the off-chip accesses originating from each core have been clustered as much as possible, subject to data dependences. Note that clustering memory requests does not necessarily

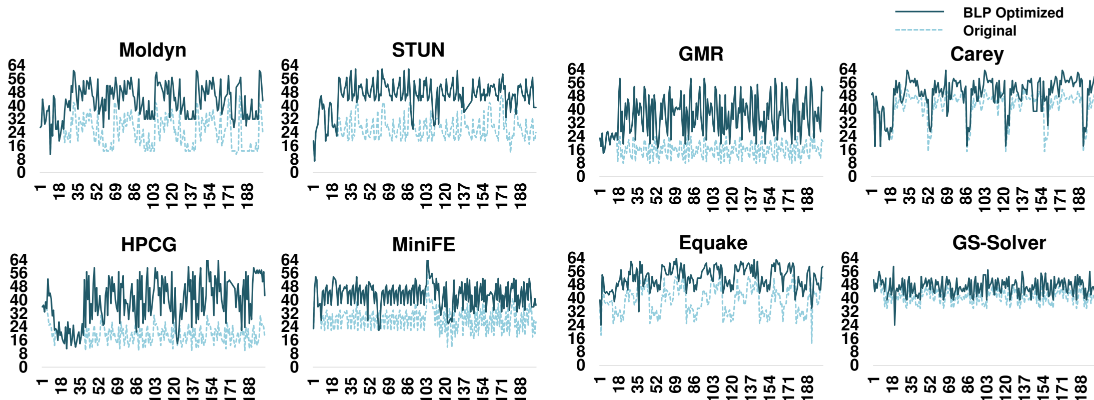


Fig. 3: BLP values (y-axis) for our applications over a period of 2 billion cycles in a 12-core, 64-bank system.

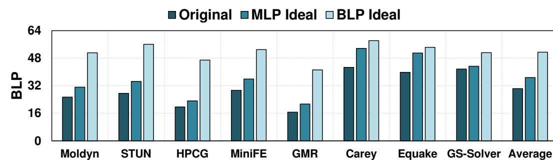


Fig. 4: BLP results with different schemes.

mean delaying all of them. It is true that some memory requests are delayed due to clustering but also some other requests are moved to an earlier point. Actually, what this strategy (called MLP Ideal) implements in the simulator is the hardware-equivalent of the compiler technique proposed by Pai and Adve [10] that aims to increase memory-level parallelism. MLP Ideal incurs around the same number of LLC misses as the original case (within 1% in our experiments), but incurs them at different points in execution. Clearly, MLP Ideal can increase BLP, depending on the target banks of the misses clustered. The second bar for an application in Figures 4 and 5 give the resulting BLP values and execution times with MLP Ideal. On an average, maximizing MLP (instead of BLP) improves BLP by 21.2%, and reduces application execution time by 6.5%, both compared to the original case. In other words, while optimizing for MLP brings some BLP benefits, it is not very effective, and leaves a lot of performance on the table.

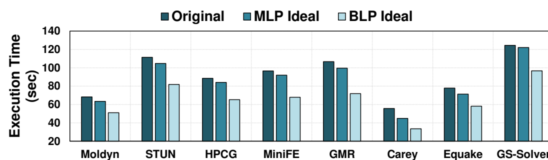


Fig. 5: Execution time results with different schemes.

To show what the potential of an *ideal scheme* that maximizes BLP (as opposed to MLP) would be, we performed another set of experiments. It needs to be observed that, at any given period of time, there could be two reasons why an application can experience less than maximum BLP. First, there may not be enough number of off-chip memory references (e.g., if we have only 16 outstanding memory references in a period of execution, we can have a maximum BLP value of 16). Second, even if we have enough off-chip accesses, those accesses may not get distributed evenly across available memory banks. In our implementation of the ideal scheme, we ensured that, if there are sufficient number of off-chip accesses, they are always distributed across the banks evenly. Therefore, the only reason this ideal scheme could not achieve maximum BLP is the lack of sufficient number of memory accesses. The results with this ideal scheme (called BLP Ideal) are given as the last bar for each application, in Figures 4 and 5. As compared to the original execution, this ideal scheme brings an average BLP improvement of 69.8%, resulting in an average execution time saving of 27.8%.

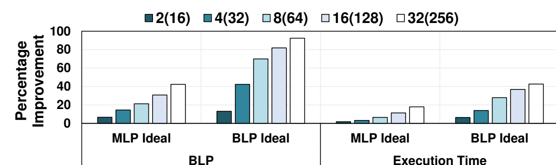


Fig. 6: BLP and execution time improvements brought by BLP Ideal, with different bank counts on a system with 4 MCs and 8 ranks per channel.

Our last set of experiments in this section quantifies the potential of this ideal scheme in a system with a large number of banks. In Figure 6, a bar marked "a(b)" indicates that the system has "a" banks per rank, giving a total of b=2x4xa banks, (8(64) is the default configuration used so far). The y-axis represents the average value across all applications. We see from these results that the effectiveness of the BLP-optimal scheme increases as we increase the number of banks, which is the current trend in system design.

Overall, the results plotted in Figures 3, 4, 5, and 6 clearly show that simply maximizing MLP does not bring significant BLP improvements, and instead, maximizing BLP can bring significant performance benefits, especially with larger configurations. However, BLP Ideal sets an *upper bound* for potential execution time improvements and cannot be directly implemented. Thus, we propose a practical BLP optimization strategy that *approximates* BLP Ideal.

IV. Technical Details

A. High Level View of Our Approach

The high-level view of our approach is illustrated in Figure 7 for a system with 4 cores and 4 banks. Each circle in this figure

represents a *slab*, a set of loop iterations, which is the unit for scheduling computations in our framework. In Figure 7(a), the default execution order is shown, where accesses from different cores are clustered into the same bank (at a given period of time), resulting in a BLP of 1. For example, in the first period, requests from all cores access the first bank. Figure 7(b) depicts the execution order after our approach is applied. In this case, at any given time, all banks are accessed, giving a BLP of 4, which is much better than the default case.

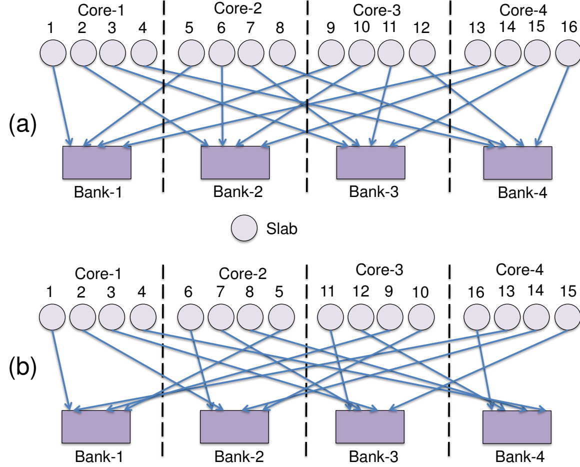


Fig. 7: High level view of a system with 4 cores and 4 banks. Each core executes its slabs from left to right.

Our approach works at the granularity of a *parallel region.* For the purposes of this paper, a parallel region represents a region that starts with an assignment to index arrays and ends with another assignment to them, as shown on the left side of Figure 8, for an example extracted from one of our application programs. The total set of iterations that will be executed by all cores in parallel region i is denoted using $C_i(1 \leq i \leq N)$, where N is the number of parallel regions. After the parallelization of C_i, the set of iterations assigned to core j is referred to as $L_{i,j}$, with $1 \leq j \leq P$, where P is the total number of cores. Each $L_{i,j}$ is divided into nearly equal sized slabs, S_{i,j,k_t}, where we execute S_{i,j,k_t} (from $L_{i,j}$) at scheduling slot (time) t.

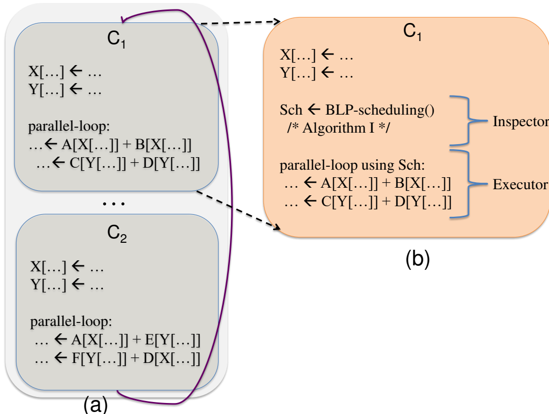


Fig. 8: (a) A code fragment with two parallel regions and (b) Modified version of the first parallel region in (a) based on the inspector-executor paradigm.

B. Optimization Goal

The main goal behind our loop iteration scheduling algorithm is to optimize BLP. In mathematical terms, at each scheduling slot t, we need to select and schedule a slab for each core (i.e., S_{i,j,k_t} from $L_{i,j}$) such that *we cover as many memory banks as possible.* Clearly, to figure out the bank(s) accessed by a given slab, we first need to be able to predict the LLC misses, which is quite hard in the case of irregular applications. Therefore, in our default implementation, we conservatively assume that all data accesses in the parallel region will miss in the LLC, and schedule loop iterations based on this assumption. Later, we also explain how one can relax this assumption. To determine the bank to be accessed by an LLC miss, we also need support from the architecture and the OS. Operating systems have APIs that allocate a physical page for a given virtual address using page-coloring algorithm. There is also an API called page-create-va(.) in Solaris (and similar calls in other operating systems) that can accept hints from the user such that the physical pages allocated by the OS honor these hints. We modified this call to allocate physical addresses such that the kernel uses the same bank bits from the virtual address for the physical address. As a result, the bits specifying the bank (e.g., bits 16, 17 and 18 in Figure 2) are *not* changed during the virtual-to-physical address translation, and for a given slab, we can determine the set of bank(s) that hold the data that slab will access, and this allows the compiler to optimize for BLP using virtual addresses and expect the corresponding improvements when the hardware uses physical addresses. We observed during our experiments that the number of page faults did not increase after our optimization. That is, while our approach changes the virtual address-to-physical address mapping, doing so does not lead to any observable change in the virtual memory performance.

Each slab S_{i,j,k_t} can be associated with a bitmap, called *bank-map* Δ_{i,j,k_t}, of the form:

$$< B_1, B_2, \cdots, B_Q >,$$

where B_z $(1 \leq z \leq Q)$ is set to 1 if S_{i,j,k_t} accesses memory bank z, and 0 if it does not (Q is the total number of banks in the system). Consequently, Δ_{i,j,k_t} in a sense represents the "bank access pattern" of S_{i,j,k_t} in a compact fashion. Now, one can try to maximize the value of the following expression to optimize BLP at scheduling slot t:

$$\odot\{ \bigvee_{1 \leq j \leq P} \Delta_{i,j,k_t} \},$$

where $\vee$ denotes "bitwise OR" operation and $\odot$ is an operator that returns the number of 1s in a bit-map.

While the objective function given above can be used to maximize BLP, it does not consider row-buffer locality at all. One option to take into account row-buffer locality would be defining another type of bitmap (row-map) where each entry (position) captures whether we access a certain memory row or not. These vectors, which represent data access patterns at a memory row granularity, can then be used to develop a scheduler that can account for row-buffer locality. However, the sheer number of rows makes this approach infeasible to be

implemented in practice as a part of dynamic scheme. Instead, we propose a strategy that works with the bank-maps defined earlier.

Our strategy is to maximize the value of the following target function, if doing so does not create a conflict with the BLP optimization goal discussed above:

$$\sum_{1\leq j\leq P}\sum_{1\leq t\leq T}\odot\{\Delta_{i,j,k_{t-1}}\otimes\Delta_{i,j,k_t}\},$$

where $\otimes$ refers to the "bitwise Exclusive-NOR" operation. It is important to note that what this function tries to capture is to ensure that the bank-maps of two successively scheduled slabs from the same core ($\Delta_{i,j,k_{t-1}}$ and Δ_{i,j,k_t}) have the same bit values (0 or 1) in as many positions as possible. That is, this function is oriented towards achieving *bank reuse* across the successively-scheduled slabs from the same core. It is also to be noted that, while bank reuse does not necessarily guarantee memory row reuse, it increases the chances for the latter (in our experiments, we quantify the impact of our approach on row-buffer hit rate).

Overall, our approach tries to optimize BLP across the cores in a given scheduling step (horizontal dimension), while considering row-buffer locality, for each core, across successive scheduling slots (vertical dimension). The rationale behind this can be explained as follows. First, given sufficiently large slabs, careful selection of slabs from different cores (at the same scheduling slot) can be expected, in most cases, to cover all the memory banks in the system. If, for some reason, one wants to work with small slabs (each with fewer iterations) however, one needs to consider not just a single scheduling slot but multiple neighboring slots to make sure that all banks are covered. This generalized formulation will be given in the next subsection. On the other hand, the reason why we consider only intra-core bank reuse instead of inter-core bank reuse is the observation that sharing (at a memory row granularity) across cores is not as frequent as sharing within a core (especially in carefully-parallelized scientific codes where inter-core data sharing is minimized).

C. Generalization

There are two generalizations that we discuss. First, in optimizing BLP, we can consider multiple scheduling steps, and second, in considering row-buffer locality, we can consider inter-core bank reuse, in addition to intra-core bank reuse. The target function to maximize for BLP when considering q successive schedule slots instead of only the current slot t can be expressed as follows:

$$\odot\{\bigvee_{1\leq j\leq P;\ t-q\leq r\leq t}\Delta_{i,j,k_r}\}.$$

Clearly, q is a parameter that can be tuned to strike a balance between BLP and runtime overheads (due to working with small-sized slabs). The objective function that considers both intra-core and inter-core bank reuse (row-buffer locality) can be expressed as:

$$\sum_{1\leq j\leq P}\ \sum_{1\leq v\leq P;\ 1\leq t\leq T}\odot\{\Delta_{i,v,k_{t-1}}\otimes\Delta_{i,j,k_t}\}.$$

This function can be further enhanced to capture the bank reuse across multiple scheduling steps as well. Our experiments with this generalized scheme revealed that, considering 2 steps (instead of 1) in making scheduling decisions brought an additional 1.8% improvement (over the 1 step case) but increasing it to 3 or 4 steps did not bring any additional improvement. Consequently, in this paper, we focus exclusively on the case where 1 scheduling step at a time is considered.

D. Algorithm and Example

To implement the objective function discussed in Section IV-B, our algorithm employs an iterative strategy. More specifically, to select the entries S_{i,j,k_t} in scheduling step t, our approach considers each core in turn, starting with the first one. For the first core, it selects a slab (as will be discussed shortly, bank reuse is taken into account for this). For the second core, it selects a slab such that this new slab covers as many banks as possible that have not been covered by the first slab. Similarly, for the third core, it picks up a slab that covers (if possible) the banks that have not been covered by the first two slabs, and so on. The row-buffer locality aspect on the other hand is taken into account as follows. Whenever we have multiple candidates (for a given core) to select from (i.e., candidates that cover exactly the same set of additional banks), we give priority to the one that maximizes bank reuse with the slab that has been scheduled on the same core in the *previous step.* In this way, row-buffer locality (actually, bank reuse) is considered only if doing so does not prevent us from reaching the best candidate from a BLP viewpoint.

Our approach is implemented using the *inspector-executor paradigm.* Specifically, for each parallel region, the compiler inserts the scheduler code right after the values of the index arrays are known (the index array assignments and the scheduler code together constitute the inspector). The main parallel loop that follows (known as executor) uses the schedule determined by the scheduler (see the right side of Figure 8). The formal algorithm for the scheduler is given as Algorithm I. The asymptotic complexity of this algorithm is $O(N*M*P^2)$, where N, P, M are respectively the total number of parallel regions, number of cores and number of slabs per core. This algorithm goes over cores one by one, and for each core, selects a slab from the remaining ones. In selecting a slab for a core j at schedule slot t, two rules are observed. First, the slab that contributes to most 1s when it is ORed with the slabs selected for cores 1 through $j-1$ in the same schedule slot (t) is selected. Second, if there are multiple such candidates, we give priority to the one that reuses most banks with the slab scheduler in the previous slot ($t-1$) on the same core. We also implemented a slightly modified version of this slab selection strategy, where cores (at a given schedule slot) are not visited in order, but based on the flexibility they have at that point. For example, if a core has only 1 potential slab that can enhance the current BLP, it is given priority over the others. This is because the others are less constrained and we may still find suitable slab candidates for them when they are visited. However, we observed in our experiments that, the

Algorithm 1 BLP_scheduling

INPUT: Number of parallel regions (N); number of cores (P); number of slabs per core (M);

1: //Initialization
2: **for** i $from$ 1 to N **do**
3:   **for** j $from$ 1 to P **do**
4:     $L_{i,j} \leftarrow \{S_{i,j,1}, ..., S_{i,j,k}, ..., S_{i,j,M}\}$
5:   **end for**
6: **end for**
7: **for** C_i $from$ C_1 to C_N **do**
8:   **for** $L_{i,j}$ $from$ $L_{i,1}$ to $L_{i,P}$ **do**
9:     $t \leftarrow 0$
10:     **while** $L_{i,j} \not\equiv \varnothing$ **do**
11:       $schedule \leftarrow \varnothing$
12:       **if** $t \not\equiv 0$ **then**
13:         $Search\ S_{i,j,k}\ in\ L_{i,j}$
14:         $choose\ S_{i,j,k}\ makes$
15:         $\odot\{\Delta_{i,j,k_{t-1}} \otimes \Delta_{i,j,k_t}\} maximum$
16:       **else**
17:         $Random\ choose\ S_{i,j,k}\ from\ L_{i,j}$
18:       **end if**
19:       $delete\ S_{i,j,k}\ from\ L_{i,j}$
20:       $schedule \leftarrow schedule \cup S_{i,j,k}$
21:       //Use iterative method to search for candidates
22:       **for** l $from$ $j+1$ to P **do**
23:         $Candidate \leftarrow \varnothing$
24:         $Search\ S_{i,l,k}\ in\ L_{i,l}$
25:         $Candidate \leftarrow all\ S_{i,l,k}\ makes$
26:         $\odot\{\bigvee_{1 \le r \le l} \Delta_{i,r,k_t}\}\ maximum$
27:         **if** $t \not\equiv 0$ **then**
28:           $Search\ S_{i,l,k}\ in\ Candidate$
29:           $choose\ S_{i,l,k}\ makes$
30:           $\odot\{\Delta_{i,l,k_{t-1}} \otimes \Delta_{i,l,k_t}\} maximum$
31:         **else**
32:           $Random\ choose\ S_{i,l,k}\ from\ Candidate$
33:         **end if**
34:         $delete\ S_{i,l,k}\ from\ L_{i,l}$
35:         $schedule \leftarrow schedule \cup S_{i,l,k}$
36:       **end for**
37:       schedule set is the set of slabs to schedule
38:       $t \leftarrow t + 1$ $//Time\ increased$
39:     **end while**
40:   **end for**
41: **end for**

difference between these two alternate implementations is less than 1%.

We want to emphasize that our approach is capable of handling a wide variety of indexed array applications. This includes applications where each index array is assigned only once in the program, as well as the applications where an index array is updated in multiple points in the program. Since our compiler analysis detects index arrays automatically, if no index array is used in the program, our optimization is simply not applied. We also believe that our implementation can be extended to work with a set of pointer codes where, once the pointer-based data structure is built, it is visited multiple times (e.g., many decision tree algorithms fall into this category). In such cases, we can collect the bank access patterns (or conservatively assume that every data access will be an LLC miss and go to main memory) after the data structure (e.g., tree) is built, and use this information in scheduling the chunks of computations that go over the data structure.

We focus on a small system with 4 cores and 4 memory banks. Each core is assumed to have been assigned 4 slabs. Figure 9(a) depicts the initial state of the cores at the first scheduling slot ($t = 1$). We randomly pick one slab ($slab$ 1) from core 1. For core 2, to maximize the BLP, we have three choices ($slab$ 1, $slab$ 2 and $slab$ 3). Our selection is still random at this point since no bank reuse is possible for

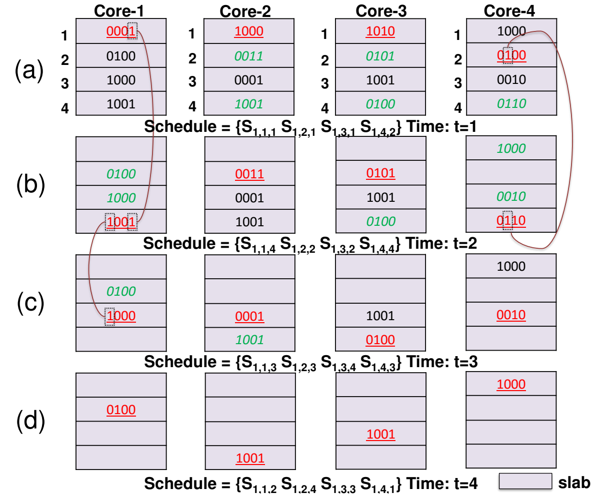


Fig. 9: An example of BLP optimized scheduling with 4 cores and 4 banks. Each core is assigned 4 slabs.

the first schedule slot. At the end, we have the schedule set ($\{S_{1,1,1}, S_{1,2,1}, S_{1,3,1}, S_{1,4,2}\}$), giving the maximum BLP of 4 for this schedule slot. Figure 9(b) illustrates the case when time moves to the next scheduling slot ($t = 2$). Now, we have three choices for core 1 ($slab$ 2, $slab$ 3 and $slab$ 4). Taking memory bank reuse into consideration, we pick $slab$ 4 as this slab reuses the memory bank 4 from the previous schedule slot ($0001 \otimes 1001$). Similarly, for core 4 at this schedule slot ($t = 2$), we pick $slab$ 4 to achieve memory bank reuse without hurting BLP. Therefore, the complete schedule for the second schedule slot is ($\{S_{1,1,4}, S_{1,2,2}, S_{1,3,2}, S_{1,4,4}\}$). Figure 9(c)and Figure 9(d) illustrate the results for the subsequent schedule slots ($t = 3$ and $t = 4$). Our algorithm ends when all the slabs are scheduled with maximum BLP while also exploiting bank reuse as much as possible.

E. Handling Regular Accesses

In determining the schedule (within the inspector code), the regular accesses are taken into account along with irregular accesses. Essentially, once the index arrays are assigned, we have all the information we need, and regular accesses along with irregular ones contribute to the determination of the schedule. Note also that the first job of the inspector is to determine the bank access pattern of a slab. As long as there is at least one regular or irregular reference from a slab to a bank, it is captured in the bank-map; repeated occurrences of the same irregular reference will not add anything more. However, if the same irregular reference is touched by two slabs, the bit of the corresponding bank is set in both the bank-maps. Also, if the same index array is updated multiple times, it is possible that the same reference in one place will point to a bank, and in another place to another bank. This is fully captured in our implementation.

F. Discussion

We now want to discuss a couple of important points. First, there is the question of why we consider BLP as the main optimization target and consider row-buffer locality only if doing so does not hurt BLP. The reason for this is two-fold.

First, our framework operates with bank-maps and row-buffer locality optimization using them is a speculative one, i.e., its deemed benefits may or may not be realized at runtime. Second, memory schedulers employed by current architectures (e.g., FR-FCFS) compensate, to some extent, not-so-good row-buffer locality by prioritizing memory requests that hit in the row-buffers of DRAM banks over other requests, including older ones. So, even if the row-buffer locality is not optimized perfectly, the memory scheduler can still achieve some locality at runtime.

Second, while we assumed so far that the slabs in C_i can be executed in any order, our approach can be modified to work with the scenarios where we have inter-slab dependences (intra-slab dependences are taken naturally into account as the iterations in a slab are executed in their default order). To capture such dependences, we build a *dependence graph at the slab level*, where nodes correspond to slabs and an edge from one slab to another represents a data dependence between them. With this representation, our approach can be modified to consider (at each scheduling step) only the slabs that are "schedulable" and select, if dependences allow, one slab for each core to maximize BLP while considering bank-level reuse.

The third issue is regarding the validity of our conservative assumption which states that all data accesses will miss in the LLC. Clearly, this assumption does not hold in practice. By making this assumption however, capture a scenario where we put the maximum pressure on the main memory system. Clearly, depending on the actual misses at runtime, the relative success of our iteration scheduling strategy may vary. Also, in many irregular applications, a given slab is executed multiple times (once in each iteration of the outermost loop of the application). Consequently, it is possible to predict the banks that will be accessed by a given slab based on its previous executions. Current Intel processors like Xeon E5-E7 series [11] already provide uncore performance counters which can be read by a Unix performance tool like *perf* to understand the row-buffer hit rates in the DRAM banks. With all the physical address information already available in the MSHRs (Miss Status Handling Registers, which keep track of outstanding LLC misses), we assume that new ISA instructions which can capture bank accesses would be a logical extension to these performance counters. These ISA instructions could easily read the physical addresses in the MSHRs and identify the bank bits by performing a simple right shift bit-level operations and a modulo operation. With this hardware support, it would be possible to learn which bank(s) a given slab accessed in its previous executions, and accordingly, predict which banks it will access in its future executions.

Fourth, note that the scheduling problem we have is NP-hard, and our greedy algorithm may not work well in some cases because it makes a (local) slab selection decision at each step, and that decision binds the scope for future decisions (for other cores). To reach the optimal BLP, slabs for all cores should be selected considering *all cores at the same time* (instead of one-core-at-a-time). However, we decided against such a scheme because of two factors. First, this would increase the complexity of our algorithm. Second, we also formulated an ILP problem for the optimal slab selection and found that the additional execution time improvements it brought over our greedy scheme was only 2.5% on average (per parallel loop, after 7 hours of execution of the ILP solver).

V. Experimental Evaluation

A. Setup and Applications

We implemented our scheme using *LLVM 3.5.0* [12] as a source-to-source translator. The original source code and the resulting optimized code are then compiled for simulator and actual hardware using different node compilers with the *highest optimization level* available, thereby activating all cache optimizations as well. We observed the following increases in compilation time (over the compilation time of the original applications): Moldyn:41%, STUN:26%, HPCG:36%, miniFE:39%, GMR:19%, Carey:14%, Equake:51%, and GS-Solver:67%. The longest compilation time we observed when using our approach was about 57 seconds. We evaluated our approach over a set of eight application programs described in Table I. The third column shows the total input size (in MBs), and the last column gives the MPKI values of the original codes under our default simulation platform (described below). GMR, STUN and Carey are three codes written by our group.

TABLE I: Benchmarks used in evaluation.

App	Description	Input Size	MPKI
Moldyn [13]	Generalized program for the evaluation of molecular dynamics models	336.2MB	81.3
STUN	Parallel sparse direct solver	1.25GB	66.6
HPCG [14]	High performance precondit- -ioned CG solver benchmark	210.8MB	87.4
MiniFE [15]	Finite element mini application	654.1MB	26.5
GMR	Generalized minimal residual based iterative sparse solver	308.6MB	91.2
Carey	Epidemic diffusion simulation on large social networks	1.10GB	12.5
Equake [16]	Earthquake simulation	487.7MB	14.1
GS-Solver [17]	Gauss-Seidel based iterative sparse solver	390.2MB	9.1

We used both simulation-based evaluation (using GEM5 [18]) and commercial hardware-based evaluation in this work. The reason for the former is three-fold. First, we wanted to get detailed BLP statistics to measure the impact of our approach. Unfortunately, we are not aware of any way of measuring BLP in real systems. Second, we also wanted to conduct a sensitivity study where we change the values of critical system parameters/policies. Third, we wanted to compare our compiler-based scheme to existing hardware-based BLP optimization schemes and this comparison could only be done using a simulator. However, in addition to this simulation based study, we also collected execution time results on an Intel Ivy Bridge based multicore system. Note that, the default parameters used in our simulation closely follow those of the Intel architecture.

Table II gives the important features of the default system we modeled in our simulator. Note that the default memory scheduler in GEM5 is FR-FCFS, and later when comparing

TABLE II: Major platform parameters.

Parameter	Default value
Cores	12 Xeon E5 (3.4GHz)
Cache Line	64 bytes (for all caches)
L1 Cache	32KB per core (private), 8-way, 4 cycle-latency
L2 Cache	256KB per core (private), 8-way, 12 cycle-latency
L3 Cache	10MB shared, 32-way, 32 cycle-latency
Data TLB	Two-level; L1: 64 entries, 4-way, L2: 512 entries, 4-way
Main Memory	4 DDR4-2933 MCs, 32GB capacity (tCL, tRCD, tRP) = (20cycles,20cycles,20cycles) 8 ranks/DIMM, 2 banks/rank, 2KB row size FR-FCFS (64 max requests/MC)

our approach to other memory schedulers, we changed this default scheduler. Our results are collected when both hardware and software prefetchers are ON (in both the simulation-based experiments and real-hardware executions). In the simulator, we implemented a state-of-the-art stride-prefetcher. In all the experiments presented below, the slab size is set to 1/50th of the iterations assigned to a core. Our sensitivity analysis with different slab sizes generated similar results, as long as the slab size chosen is large enough.

We define a metric called "coverage ratio" which captures the percentage of original executor iterations that use the new schedule after the optimization (in the ideal case, this ratio would be 100%). The coverage ratios for our benchmarks varied between 73% and 91%, averaging on 86%, indicating that our compiler was able to optimize a very large fraction of each application.

B. Simulation Results

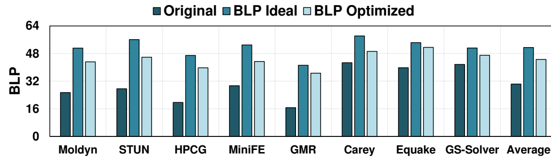


Fig. 10: BLP results with different schemes.

The curves marked "BLP Optimized" in Figure 3 give the BLP results when our scheme is applied. Comparing these results to those of the original executions shown in the same figure, one can see that our scheme brings significant improvements in BLP. The third bar for each benchmark in Figure 10 gives the average BLP value with our scheme. The first two bars of the same graph reproduce results from Figure 4 for ease of comparison. When averaged over all application programs in our experimental suite, the proposed approach achieves an average BLP of 44.5, which is much better than the average BLP of the original benchmarks (30.3), and not too far from the BLP Ideal case (51.5).

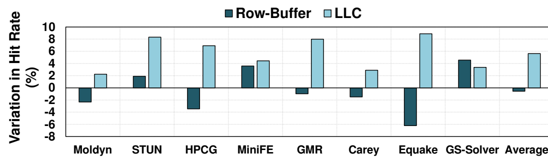


Fig. 11: Variations in LLC and row-buffer hit rates as a result of our approach (BLP Optimized).

We now quantify the impact of our approach on execution times of our applications. Note that BLP is only one part of the big performance equation, and there are at least two important factors to consider here, in addition to BLP values, which may influence execution times: LLC behavior and row-buffer hit rates. On the positive side, recall that our approach tries to improve bank-level reuse, if doing so does not conflict with the BLP optimization goal. We can expect this to have a positive effect on both LLC performance and row-buffer hit rates. On the negative side, we have two issues to consider. First, since our approach changes the execution order of loop iterations, this can negatively affect the cache behavior. However, we do not expect this to be a major issue, as our approach works at a slab granularity and, since once a slab is scheduled all its iterations are executed in their original order, the impact on cache behavior will be quite limited. The second issue is due to the inherent conflict between BLP optimization and row-buffer locality optimization. Since our approach is primarily driven by the former, it may negatively affect the latter, though we expect the bank-reuse optimization to compensate for it. Further, the FR-FCFS memory scheduler used by the hardware is also expected to help with the negative impact of our approach on row-buffer locality. Figure 11 gives the variations on LLC hit rate and row-buffer hit rate when our approach is applied. We see from this plot that overall BLP Optimized improves row-buffer locality in 3 of our 8 applications, and distorts it in the remaining ones. One can also observe that the improvements in the LLC hit rates brought by our approach vary between 2.2% and 8.8%, averaging on 5.6%. It is also important to note from Table I that our applications have relatively high MPKI values, that is, they are memory intensive. With such high MPKI values, even after the optimization (and the reduction in LLC misses), there are still enough misses that allow BLP to play an important role. Further, comparing the last column of Table I and Figure 10, we see that our approach achieves better BLP improvements with applications that have higher MPKI values, as there are more memory accesses to schedule for different banks.

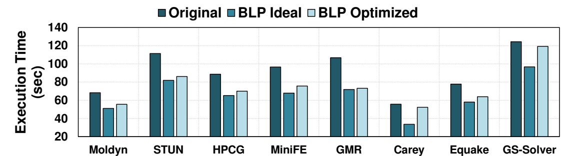


Fig. 12: Execution time results (simulator).

The execution time results with BLP Optimized are presented in Figure 12, as the third bar for each benchmark. *These results capture the impact of our approach on BLP, row-buffer locality and LLC misses.* We see that our approach improves the execution times of all the applications. These improvements range between 4.1% (GS-Solver) and 22.6% (STUN), averaging on 18.3%. The relative improvements are lower in applications Carey and GS-Solver, which align well with the relatively lower BLP improvements observed in Figure 10. We want to emphasize that these execution time results also include all the runtime overheads incurred by our approach, which includes the execution of the code that determines the new scheduling as well as any impact on caches and on-chip network. Figure 13 zooms in this overhead for each benchmark, and quantifies it as a fraction of the

total execution time. On an average, the contribution of the overheads amounts to 4% of the execution time. Actually, scheduling costs can be considered from two aspects. First, note that our optimization target is a parallel region, not loops. Therefore, if there is a large loop body, loop fission can be applied before our approach. Second, our approach actually uses a sliding window-based implementation. The reason is that searching all candidates to maximize BLP is not feasible in practice. Therefore, at any given time, our algorithm only considers the candidates in a window. Further, some portion of the overheads are also probably hidden during parallel execution. This is why in practice the overheads we observe are not very high.

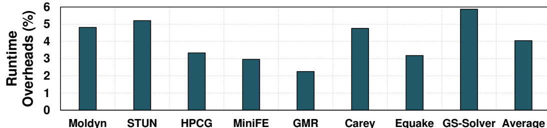


Fig. 13: Contribution of the runtime overheads to the total execution time.

C. Results with Intel Ivy Bridge

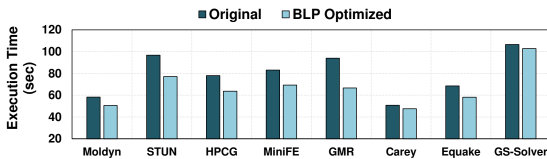


Fig. 14: Execution time results (Ivy Bridge).

Next, we quantify the execution time improvement on our Ivy Bridge based multicore platform, which is equipped with 4 DDR3-2133 memory controllers (14 cycles for each of tCL, tRCD, and tRP). It is important to note that in the Ivy Bridge platform, there is no way to measure BLP directly. However, we are able to measure LLC misses, row-buffer hits and conflicts, and execution time. Due to space concerns, we present only execution time results but want to mention that the impact of our approach on LLC and row-buffer statistics were similar to the simulator case. In particular, compared to the original execution, our approach distorted row-buffer locality by 1% on average (generating, though, better row-buffer hit rates in three applications – STUN, miniFE, and GS-Solver), and improved LLC hit rates by 6.6% on average (improvements range from 1.8% to 9.2%). The execution time results given in Figure 14 indicate similar trends to those plotted in Figure 12. The average improvement brought by BLP Optimized is 15.7%. Clearly, there may be other factors in the real system that influence the execution times but cannot be captured by the simulator; however, our results indicate that applying our BLP optimization improves execution time significantly in the real system as well. As a point of comparison, when we modeled the same DDR3-2133 system in our simulator, we observed BLP improvements (over the original version) ranging between 18.8% and 56.3%, averaging on 30.7%. As indicated above however, there is no way to collect such BLP statistics from the real hardware.

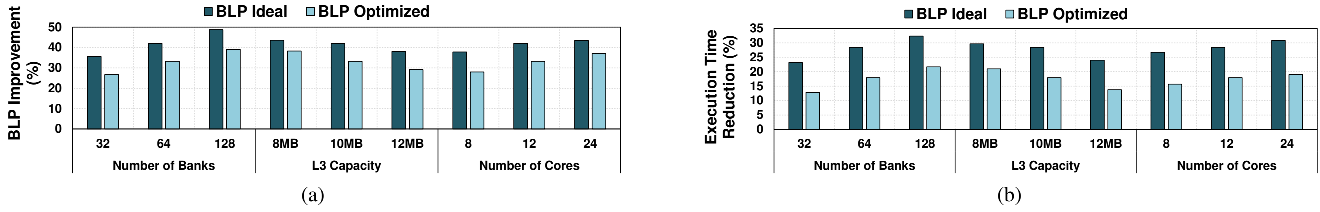
D. Sensitivity Experiments

We now report, with the help of our simulator, the BLP and execution time results under different values of the system parameters. Each group of bars in Figure 15 represents the *average values* from a single sensitivity experiment, that is, the value of only one system parameter is varied, and the values of the remaining parameters are kept at their default values shown in Table II. These results indicate that the effectiveness of our approach increases as we increase the number of banks, L3 (LLC) capacity, and number of cores. When we increase the number of banks (while keeping the number of memory requests the same), the number of idle banks in any given period of time increases, leading to a drop in the relative BLP (*not* in absolute BLP, as the absolute BLP increases with the increased bank count, but the relative BLP [the ratio between the observed BLP and maximum possible BLP] gets reduced). Consequently, there is more scope to optimize (more gap between the maximum BLP and observed BLP), and this in turn increases the potential impact of our BLP optimization.

One can also see that, although the effectiveness of our approach gets reduced with the increased L3 capacity, even with 12MB L3 cache it achieves 29% BLP improvement and 13.7% execution time improvement. On the other hand, it is not easy to predict the impact of increasing the number of cores on BLP. In the case of our benchmarks, we found that our optimization scheme generates better savings with the increasing core count, except for STUN and GMR. This is probably because increasing the number of cores creates more bank-level conflicts, which presents more opportunities to our approach. In addition to these three parameters, we also gauged the sensitivity of our approach to the row-buffer size, the number of memory controllers (keeping the total number of banks the same), slab size, and the size of the memory queue. Our results showed that the percentage performance improvements brought by our approach were less sensitive to these two parameters (within 2%). We also tested our irregular applications (in their default form) under both open-page and closed-page policies, and found that the former results in 4% better performance than the latter, due to primarily data locality/sharing across different threads (which may be more pronounced than in the case of commercial workloads). This observation plus the fact that we have more scope for optimization in the case of open-page policy motivated us to use the open-page policy as our baseline.

Our next experiment focuses on increasing the dataset sizes, and reports both simulation and Ivy Bridge results. In only two of our benchmarks (HPCG and GMR), we were able to change the input size safely. The results plotted in Figure 16[1] indicate that, as we increase the input size, the improvement brought by our approach increases but to a certain point. Beyond that point, the relative savings slightly reduce. This can be explained as follows. When the dataset size is increased, we incur more misses, which makes it even more important to

[1]On the x-axis, "x" corresponds to the default input size of the benchmark (210.8MB for HPCG and 308.6MB for GMR).



(a) (b)

Fig. 15: Results from the sensitivity experiments. (a) BLP improvements, and (b) execution time reduction. In each experiment, all versions use the same hardware configuration, specified by the x-axis.

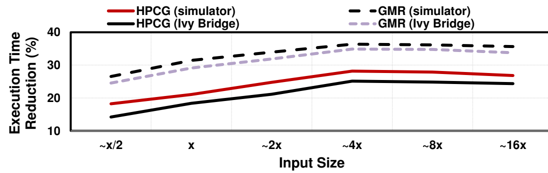


Fig. 16: Execution time reduction for HPCG and GMR with different input sizes.

exploit BLP. However, beyond a certain point, the outstanding misses start to fill all banks, and the original code starts to exhibit high levels of BLP. Since Figure 16 gives the relative improvements over the original version, we observe a reduction in savings.

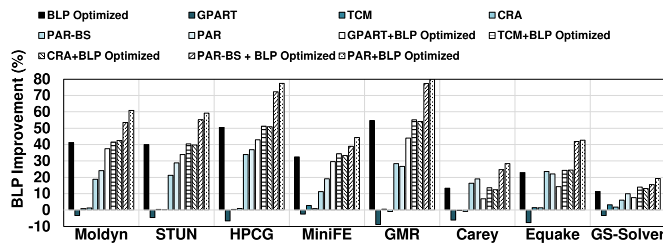
E. Comparison against Alternate Strategies

We are not aware of any compiler scheme that tries to optimize BLP for irregular applications. Pai and Adve [10] improve MLP in the context of single-core machines by clustering cache misses, and Ding et al [19] enhance BLP for multicore systems. However, both of these work with *regular loop structures with affine accesses* and do not have any runtime component, and consequently, they *cannot* handle irregular codes. In this subsection, we compare our approach, using GEM5, against five alternate schemes (one software based and four hardware based). The software scheme is GPART [20], which is a hierarchical graph clustering algorithm designed to re-organize data layout in irregular applications to improve cache performance (in our implementation, we adjusted the cluster sizes to maximize GPART's performance). The reason why we perform experiments with GPART is to gauge the impact a pure cache-locality oriented compiler scheme can have on BLP. The hardware schemes against which we compare are (1) PAR-BS [6], (2) TCM [21], (3) the critical region-aware parallel application memory scheduling scheme [22] (called CRA henceforth), and (4) the scheme described in [4] (called PAR henceforth). Note that CRA is a memory scheduling scheme designed exclusively for multithreaded workloads, whereas the remaining three hardware schemes are originally designed for multiprogrammed workloads. For each scheme we compare, we run that scheme alone as well as when it is coupled with our BLP Optimized.

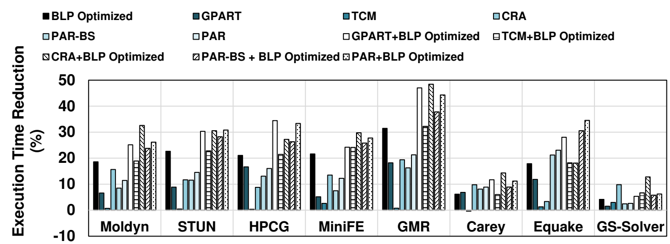
The results presented in Figures 17(a) and (b) are normalized with respect to FR-FCFS and use the default system parameters given in Table II. We see from Figure 17(a) that GPART degrades BLP (with respect to the original case) in all eight applications. This is mainly because it does not do anything special for BLP and clustering data accesses (for better cache performance) tends to distort BLP, compared to the original case. Although not presented here due to space concerns, our experiments also showed that, while GPART improves the cache performance by 9.6% on average, it has little impact on row-buffer performance. These cache/row-buffer results combined with the BLP results generated, on average, 9.4% execution time improvement for GPART, as plotted in Figure 17(b).

Regarding the hardware schemes, we start by observing that TCM does not improve much over the FR-FCFS, primarily because TCM exploits the differences in memory intensities of different cores to improve system throughput, which makes a lot of sense in multiprogrammed workloads where one runs, for example, one application in each core. In a multithreaded application however, all threads normally have *similar* memory intensities. Consequently, except for two applications (miniFE and GS-Solver), TCM does not improve performance, and BLP Optimized generates an average BLP (resp. execution time) improvement of 32% (resp. 16.8%) over it. CRA prioritizes the threads holding locks over the others to reduce serialization; it improves over the default scheduler (as expected) in terms of the execution time, but it is orthogonal to our scheme (as it does not do anything specific for BLP). Consequently, our approach improves further over CRA; specifically, CRA and CRA + BLP Optimized generate average execution time savings of 11.5% and 26.7%, respectively, over the original execution.

PAR-BS tries to process the requests from a thread as a batch. Our approach generates better BLP results than PAR-BS in all programs except two (Carey and Equake). This is because, while PAR-BS can only take advantage of the potential BLP in memory queues, BLP Optimized can perform BLP-aware scheduling at a much larger scope (parallel region level). These trends translate to execution times, and we generate 6.9% improvement, on average, over PAR-BS, when all benchmarks are considered. When the two schemes (PAR-BS and BLP Optimized) are combined, we observe further improvements in application performance (23.4% on average over the original execution). Finally, PAR is a scheme originally designed for exploiting the potential MLP of prefetch requests. It has two components: the first one issues prefetch requests to MSHRs in a BLP-aware fashion and, the second one tries to preserve BLP exhibited by individual cores by removing interferences. We observe that, while PAR improves over FR-FCFS, our approach generates much better savings. This is mainly because PAR in a sense tries to improve BLP



(a)



(b)

Fig. 17: Comparison of our approach against alternate strategies: (a) BLP improvement and (b) Reduction in execution time.

in a similar fashion to PAR-BS (in fact, as stated in [4], the benefits of the two schemes can partially overlap). The results in Figure 17(a) indicate that PAR and PAR+BLP Optimized generate average BLP (resp. execution time) improvements of 23.2% (resp. 13.7%) and 51.5% (resp. 26.7%), respectively.

To sum up, BLP Optimized outperforms, in most cases, all four hardware-based schemes tested and more importantly it can be used in conjunction with any BLP-aware scheduler (such as PAR-BS and PAR) to generate additional execution time savings. Also, it is orthogonal to schemes such as CRA (which improves aspects of execution other than BLP) and can be combined with them to obtain higher performance savings.

F. Scheduling with Dependences

Recall that, so far, if a code region has inter-slab dependences, we did not execute it in parallel. We also performed a set of experiments where such code regions are also executed in parallel, using the dependency graph discussed in Section IV-F. Although we do not present the detailed results due to lack of space, we want to say that 7 of our codes had at least one inter-slab dependence (miniFE did not have any), STUN having the largest number of such dependences (17 in total). Our approach generated an additional 3.3% (average) execution time improvement in this case, compared to the sequential execution of the code regions with inter-slab dependences (in STUN and Equake, the additional gains were 8.4% and 6.3%, respectively).

VI. Related Work

Software approaches to improve MLP: Liu et al. [23] proposed an OS based bank-level partitioning scheme, where OS allocates pages to cores (threads) from a particular bank, thereby reducing the interference from the other applications. Pai and Adve [10] introduced the concept of clustering cache misses to improve memory level parallelism. Ding et al. [24] proposed iteration space tile scheduling to improve BLP. Targeting regular codes with affine references, they predict last-level cache misses per tile in a loop nest in the first step, and in the second step, they identify which banks are accessed by the corresponding indices and schedule the tiles such that they increase the BLP.

Hardware approaches to improve MLP: Several techniques [25], [26], [27], [28], [29], [30] are proposed to improve memory parallelism. Lee et al. [4] proposed two schemes (1) MSHR issuing policy which prioritizes prefetch requests to different banks ahead of prefetch requests to the same bank to increase the BLP. (2) a BLP preserving scheme that allocates the requests in to memory controller such that the BLP across an application is preserved minimizing the interference from the other applications running on different cores. Mutlu and Moscibroda [6] proposed a scheduler which provides fairness and higher MLP.

Irregular applications: There exist many compiler works that target irregular applications. Most of those works focus on data layout optimizations to improve cache locality [31], [32], [33], [34], [35], [36], [37]. The other body of work in this area focus on parallelizing irregular applications [38], [39], [40], [41]. [20] presented a hierarchical clustering method (GPART) to improve cache locality in irregular applications. Han and Tseng [42] employed graph partitioning to improve locality in irregular applications. Zhong et al. [43] described how an affinity-based hierarchical partitioning of data can improve cache locality. Das et al. [7] are the first one to propose inspector-executor model to identify parallelism in irregular applications. Ding et al. [19] proposed trading cache hit rate for memory performance to improve performance. All these studies aimed at either improving cache performance or parallelism. Our main focus on the other hand is on BLP.

VII. Concluding Remarks

This paper proposes and evaluates a novel loop iteration scheduling strategy to improve memory bank-level parallelism (BLP) of irregular application programs. The proposed scheduling strategy uses bank-maps to capture bank access patterns and reorganizes groups of iterations, called slabs, across cores to increase the number of concurrently-accessed banks. It also considers bank reuse, in an attempt to improve row-buffer locality, if it does not conflict with the BLP optimization. Our detailed evaluations of this scheduling strategy indicate significant improvements in terms of both BLP (46.8% on average) and execution times (18.3% on average).

Acknowledgment

We thank the anonymous reviewers for their feedback. This research is supported in part by NSF grants #1213052, #1302557, #1409095, #1439021, #1439057, #1526750, #1629915 and #1629129, and a grant from Intel.

References

[1] A. Pan and V. S. Pai, “Imbalanced Cache Partitioning for Balanced Data-parallel Programs,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013.

[2] S. W. Keckler, “Rethinking Caches for Throughput Processors: Technical Perspective,” *Commun. ACM*, 2014.

[3] K. S. McKinley, “Author Retrospective for Optimizing for Parallelism and Data Locality,” in *Proceedings of the 25th International Conference on Supercomputing*, 2014.

[4] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt, “Improving memory bank-level parallelism in the presence of prefetching,” in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[5] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu, “A Case for Exploiting Subarray-level Parallelism (SALP) in DRAM,” in *Proceedings of the 39th International Symposium on Computer Architecture*, 2012.

[6] O. Mutlu and T. Moscibroda, “Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems,” *SIGARCH Comput. Archit. News*, 2008.

[7] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang, “Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures,” *J. Parallel Distrib. Comput.*, 1994.

[8] S. Rixner, “Memory Controller Optimizations for Web Servers,” in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, 2004.

[9] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, “Memory Access Scheduling,” *SIGARCH Comput. Archit. News*, 2000.

[10] V. S. Pai and S. Adve, “Code Transformations to Improve Memory Parallelism,” in *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, 1999.

[11] I. Intel Uncore Performance Monitoring Guide, “Intel Xeon Processor E5 v2 and E7 v2 Product Families Uncore Performance Monitoring Reference Manual,” 2014. [Online]. Available: http://www.intel.com/content/www/us/en/processors/xeon/xeon-e5-2600-v2-uncore-manual.html

[12] C. Lattner and V. Adve, “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation,” in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 2004.

[13] D. J. Craik, A. Kumar, and G. C. Levy, “MOLDYN: a generalized program for the evaluation of molecular dynamics models using nuclear magnetic resonance spin-relaxation data,” *Journal of Chemical Information and Computer Sciences*, 1983.

[14] J. Dongarra and M. A. Heroux, “HPCG: Toward a New Metric for Ranking High Performance Computing Systems,” 2013. [Online]. Available: https://software.sandia.gov/hpcg/

[15] “Minife,” available at https://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks.

[16] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. Jones, and B. Parady, “SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance,” 2001. [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-44587-0_1

[17] H. Hhonghao, Y. Dongjin, H. Yi, and X. Jinqiu, “Preconditioned Gauss-Seidel Iterative Method for Linear Systems,” in *Information Technology and Applications. IFITA ’09*, 2009.

[18] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, “The Gem5 Simulator,” *SIGARCH Comput. Archit. News*, 2011.

[19] W. Ding, M. Kandemir, D. Guttman, A. Jog, C. R. Das, and P. Yedlapalli, “Trading Cache Hit Rate for Memory Performance,” in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, 2014.

[20] H. Han and C.-W. Tseng, “Exploiting Locality for Irregular Scientific Codes,” *IEEE Trans. Parallel Distrib. Syst.*, 2006.

[21] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, “Thread cluster memory scheduling: Exploiting differences in memory access behavior,” in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[22] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt, “Parallel application memory scheduling,” in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[23] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu, “A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems,” in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, 2012.

[24] W. Ding, D. Guttman, and M. Kandemir, “Compiler Support for Optimizing Memory Bank-Level Parallelism,” in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[25] Y. Chou, B. Fahs, and S. Abraham, “Microarchitecture optimizations for exploiting memory-level parallelism,” in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.

[26] S. Eyerman and L. Eeckhout, “A Memory-Level Parallelism Aware Fetch Policy for SMT Processors,” in *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, 2007.

[27] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, “A Case for MLP-Aware Cache Replacement,” in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, 2006.

[28] A. Jain and C. Lin, “Linearizing irregular memory accesses for improved correlated prefetching,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013.

[29] I. Hur and C. Lin, “Adaptive history-based memory schedulers,” in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, 2004.

[30] P. Yedlapalli, J. Kotra, E. Kultursay, M. Kandemir, C. R. Das, and A. Sivasubramaniam, “Meeting midway: Improving cmp performance with memory-side prefetching,” in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013.

[31] E. Petrank and D. Rawitz, “The Hardness of Cache Conscious Data Placement,” in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[32] C. Zhang, C. Ding, M. Ogihara, Y. Zhong, and Y. Wu, “A Hierarchical Model of Data Locality,” in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006.

[33] C. Ding and K. Kennedy, “Improving Cache Performance in Dynamic Applications Through Data and Computation Reorganization at Run Time,” in *Proceedings of the ACM 1999 Conference on Programming Language Design and Implementation*, 1999.

[34] J. Mellor, D. Whalley, and K. Kennedy, “Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings,” in *International Journal of Parallel Programming*, 2001.

[35] N. Mitchell, L. Carter, and J. Ferrante, “Localizing non-affine array references,” in *the Proceedings of Parallel Architectures and Compilation Techniques (PACT)*, 1999.

[36] M. M. Strout, L. Carter, and J. Ferrante, “Compile-time Composition of Run-time Data and Iteration Reorderings,” in *Proceedings of the ACM 2003 Conference on Programming Language Design and Implementation*, 2003.

[37] Y. Jo and M. Kulkarni, “Automatically enhancing locality for tree traversals with traversal splicing,” in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2012.

[38] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval, “How Much Parallelism is There in Irregular Applications?” in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2009.

[39] P. Yedlapalli, E. Kultursay, and M. T. Kandemir, “Cooperative Parallelization,” in *Proceedings of the International Conference on Computer-Aided Design*, 2011.

[40] L. Codrescu, D. Wills, and J. Meindl, “Architecture of the Atlas chip-multiprocessor: dynamically parallelizing irregular applications,” *Computers, IEEE Transactions on*, 2001.

[41] H. Yu and L. Rauchwerger, “Adaptive Reduction Parallelization Techniques,” in *Proceedings of the 14th International Conference on Supercomputing*, 2000.

[42] H. Han and C.-W. Tseng, “Improving Locality for Adaptive Irregular Scientific Codes,” in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, 2001.

[43] Y. Zhong, M. Orlovich, X. Shen, and C. Ding, “Array Regrouping and Structure Splitting Using Whole-program Reference Affinity,” in *Proceedings of the ACM 2004 Conference on Programming Language Design and Implementation*, 2004.