Efficient Thread/Page/Parallelism Autotuning for NUMA Systems

Mihail Popov
Uppsala University
Uppsala, Sweden
mihail.popov@it.uu.se

Alexandra Jimborean
Uppsala University
Uppsala, Sweden
alexandra.jimborean@it.uu.se

David Black-Schaffer
Uppsala University
Uppsala, Sweden
david.black-schaffer@it.uu.se

ABSTRACT

Current multi-socket systems have complex memory hierarchies with significant Non-Uniform Memory Access (NUMA) effects: memory performance depends on the location of the data and the thread. This complexity means that thread- and data-mappings have a significant impact on performance. However, it is hard to find efficient data mappings and thread configurations due to the complex interactions between applications and systems.

In this paper we explore the combined search space of thread mappings, data mappings, number of NUMA nodes, and degree-of-parallelism, per application phase, and across multiple systems. We show that there are significant performance benefits from optimizing this wide range of parameters together. However, such an optimization presents two challenges: accurately modeling the performance impact of configurations across applications and systems, and exploring the vast space of configurations. To overcome the modeling challenge, we use native execution of small, representative *codelets*, which reproduce the system and application interactions. To make the search practical, we build a search space by combining a range of state of the art thread- and data-mapping policies.

Combining these two approaches results in a tractable search space that can be quickly and accurately evaluated without sacrificing significant performance. This search finds non-intuitive configurations that perform significantly better than previous works. With this approach we are able to achieve an average speedup of 1.97× on a four node NUMA system.

CCS CONCEPTS

•**General and reference** → *Measurement; Performance;* •**Computer systems organization** → *Multicore architectures;* •**Software and its engineering** → *Memory management; Scheduling;*

KEYWORDS

NUMA, autotunning, thread placement, page placement, code isolation, OpenMP, performance optimization

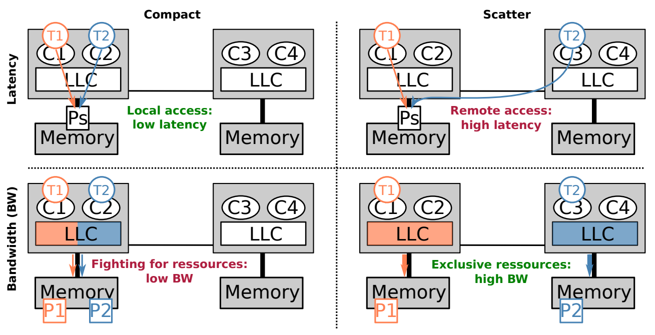
Compact
Scatter
Latency
Bandwidth (BW)
T1
T2
C1
C2
C3
C4
LLC
Ps
Memory
Local access: low latency
Remote access: high latency
Fighting for ressources: low BW
Exclusive ressources: high BW
P1
P2

Figure 1: Scatter and compact thread mappings on a two NUMA nodes system. Scatter increases bandwidth by spreading threads across memory controllers, but hurts latency to shared data by forcing such accesses to go over the node-to-node communications link. Compact does the opposite by grouping threads on the same node. The optimal configuration depends on both the application and system.

ACM Reference format:
Mihail Popov, Alexandra Jimborean, and David Black-Schaffer. 2019. Efficient Thread/Page/Parallelism Autotuning
for NUMA Systems. In *Proceedings of 2019 International Conference on Supercomputing, Phoenix, AZ, USA, June 26–28, 2019 (ICS '19),* 12 pages.
DOI: 10.1145/3330345.3330376

1 INTRODUCTION

Large multi-core systems enable increased core counts by tying together multiple processor dies (nodes) through node-to-node communications links. These links provide shared access to the entire physical memory space and increase overall bandwidth with separate memory controllers per node, but result in non-uniform memory latency and bandwidth depending on which core is accessing which node's memory. The resulting Non-Uniform Memory Access (NUMA) effects can cause significant performance problems if memory pages and threads are not appropriately *mapped* to the correct nodes at the correct times. Unfortunately, choosing the best mapping is a complex problem as it depends on both the application's behavior, which changes across application phases, and the details of the NUMA system.

Figure 1 illustrates the complexity of mapping just two threads (T1 and T2) with two private pages (P1 and P2) and one shared page (Ps) on a simple two-node NUMA system. In this example, T1 accesses the shared page Ps before T2. With the standard Linux *first-touch* (FT) page mapping policy, this will result in Ps being placed on whichever NUMA node T1 runs on. Even for such a simple example, there is a wide range of mapping choices.

Mapping: Threads. The left column of Figure 1 shows the resulting thread- and page-mappings for a *compact* mapping policy that places all threads on the same NUMA node. In this case, both threads have *local* access to the shared page Ps through the node's memory controller. The compact mapping thereby provides low-latency to the shared data, as the threads share the LLC and use the direct DRAM link to access the page. However, this mapping also delivers lower bandwidth for the private pages P1 and P2 as they share the same DRAM link and LLC.

An alternative is to use a *scatter* thread mapping (Figure 1, right), which places the threads on different NUMA nodes. As a result, accesses from T2 to Ps are *remote*, since they must cross the node-to-node interconnect link, thereby increasing latency compared to the compact mapping's local accesses. However, the scatter mapping increases overall bandwidth as each thread accesses its private pages P1 and P2 in its local DRAM via its own memory controller and LLC. In general applications have a mixture of private and shared pages, making the choice of thread mapping non-trivial.

Mapping: Pages. Typical applications have millions of shared and private pages active at any given time, which drastically complicates their mapping. To simplify this, many page mapping policies have been proposed. They target known NUMA bottlenecks such as latency, by mapping pages to threads that most frequently access them (*page locality*), or congestion, by distributing pages across nodes to increase overall bandwidth (*page balancing*). Applications that are sensitive to a mixture of latency and congestion require a *mixed* [10] policy that places pages to balance latency and bandwidth. Unfortunately, it is difficult to determine an application's sensitivity, which makes page placement challenging.

Mapping: NUMA Degree. In Figure 1, only one node was used with the compact mapping. It has been observed that some application/system combinations perform better when run on fewer NUMA nodes due to improved sharing through the LLC and reductions in remote DRAM accesses. This indicates that the *NUMA degree*, the number of NUMA nodes used, can be an important parameter. Previous work has included the NUMA degree as an implicit part of the thread mapping [9]. However, this parameter can be independently adjusted for both threads and pages, thereby exposing a further range of interesting configurations.

Mapping: Degree of Parallelism. Applications further benefit from different degrees of parallelism on different systems [34]. For example, the optimal mapping for a system may under-subscribe cores in order to increase the per-core shared cache or bandwidth.

Challenge: Search Space. The choices for thread mapping, page mapping, NUMA degree, and degree of parallelism are not independent [11], requiring a coupled search to identify the best option. In addition, applications have distinct phases with varying behavior. This means that not only may each application phase require a different optimal policy, but that the choices made for earlier phases may affect the choices for later phases, due to the need to change configurations between phases [23]. Furthermore, each NUMA system has different characteristics, which means that the best mapping for an application on one particular system may not be the same for another system.

The combination of the interdependence of thread mapping, page mapping, NUMA degree, and degree of parallelism, with per-phase and inter-phase application behavior and per-system differences leads to a very large search space: optimizing an application with two phases on a four NUMA nodes system naively, requires evaluating over 600 configurations. (We parameterize this search space in Table 1 to demonstrate how we explore it.)

Challenge: Performance Modeling. Choosing the best mapping requires understanding the combined effect of the optimizations on the application and target system. Unfortunately, systems are complex and diverse: they have different NUMA factors (ratio of local to remote access latency), topologies, bandwidth/latency, caching, prefetching, etc. Application behaviors are also complex and diverse: they have different bandwidth/latency sensitivities, arithmetic intensity, memory/cache footprints, etc., all of which may vary over time. These characteristics result in different performance bottlenecks depending on the combination of application, system, and mapping. To choose the best mapping, these complex interactions must be accurately evaluated without incurring prohibitively high overhead.

Efficient and Effective NUMA Optimization. Optimizing NUMA mappings can lead to substantial (up to 2×) performance improvements [9]. Although previous studies identified specific NUMA bottlenecks and provided policies to address them, there is currently no method to select the most efficient overall mapping for a given application and system for two reasons: First, the search space of thread- and page-mappings, NUMA degree, and degree of parallelism, combined with application phases and system-specific behavior is so large that previous works have had to decouple them and search each one independently. A typical approach has been to first optimize thread mappings for a fixed degree of parallelism, and then choose the best page mapping given the found thread mapping. This simplification vastly reduces the search space, but at a loss of optimization opportunities. Second, there are no straight-forward ways to both quickly and accurately model the complex performance effects of these mappings on applications and systems. This limitation combines with the large search space to make it either hard to find the best configuration (due to inaccurate performance predictions of fast models) or impractical (due to slow performance evaluations of full application execution).

In this paper, we address both of these challenges and deliver an approach that can automatically, and rapidly, explore the *coupled* search space of thread- and page-mappings, NUMA degree, degree of parallelism, and application phases for any system. To provide fast and accurate performance modeling, we turn to native execution of the applications on the target system. While this will faithfully reproduce the performance effects, naive native execution is prohibitively slow. To reduce this overhead, we extract and run short *codelets* (Section 3.2), which provide representative behavior of the application in a tiny fraction of the runtime. This allows us to quickly, and accurately, evaluate the performance of a given thread and page mapping.

While codelet-based native execution allows us to evaluate configurations much more quickly, the search space of configurations is still impractically large. To address this, we parameterize the search space based on existing NUMA policies (Section 3.3.2). This parameterization is built by combining state of the art thread, page,

Thread Mappings		
Thread NUMA Degree (TND)	**Thread Placement Policy (TPP)**	Note:
Sandy Bridge:1,2,4	Scatter	If TND >= NT, Scatter and Contiguous are the same
Broadwell:1,2	Contiguous[2]	If TND = 1, Scatter and Contiguous are the same
Page Mappings		
Page NUMA Degree (PND)	**Page Placement Policy (PPP)**	Note:
Sandy Bridge:1,2,4	First-touch, Locality,	Locality and First-touch require that TND matches
Broadwell:1,2	Balance, Mix, Dataset	the PND to be applied
Degree of Parallelism		
Number of Threads (NT)		Note:
Sandy Bridge:2,4,8,16,32		NT >= TND
Broadwell:2,4,8,10,20		NT <= TND * Cores per NUMA node
Application Regions		
First, find the best mapping for each region separately		
Second, evaluate each region's mapping on the other regions to find the best mapping for the full application		

Table 1: Parameterized NUMA search space. We evaluate different thread and page placement policies (TPP and PPP) with different NUMA degrees for the threads (TND) and the pages (PND) across different numbers of threads (NT). Unlike previous works, we explore all combinations of this search space.

and parallelism policies that address different NUMA bottlenecks, and applying them to each codelet to find the best overall mapping. When optimized together, these policies cover a large portion of the total search space and are able to deliver significantly better performance than applying the policies individually.

The contributions of this paper are:

- A codelet-based infrastructure to quickly and accurately profile page mappings via native execution. (Section 3.2).
- A parameterization of the thread/page/parallelism search space that implements a combined optimization of state-of-the-art NUMA mappings for a target application and system. (Section 3.3).
- The explicit evaluation of the number of NUMA nodes used independently for thread- and page-mappings, thereby finding optimizations that have not been previously explored.
- A whole-application optimization that evaluates mappings per- and across OpenMP regions. (Section 3.4).
- The validation of the resulting autotuning strategy over different benchmarks including NAS and RODINIA. (Section 4).
- A case study of potential performance instability caused by the interaction between the OpenMP runtime and the Operating System (OS). (Section 4.2).

2 MOTIVATION

To show the importance of optimizing across thread- and page-mappings, NUMA degree, and parallelism for each system, we first look at two benchmarks on two systems. This evaluation highlights the performance benefits (Section 2.2) of our work which combines these optimizations over previous work which uses a system-independent approach that optimizes for configurations independently. At the same time, this example identifies the significant cost (Section 2.3) of exploring such a large space.

2.1 Evaluation Setup

For this exploration we evaluate the benchmarks BT (NAS C OpenMP [16, 28]) and Streamcluster (Rodinia [4]) on Intel Sandy Bridge (8 core/4 node) and Broadwell (10 core/2 node) systems. For each benchmark we consider the dominant OpenMP region (`ZSolve` for BT, which represents 30% of the execution time, and `Pgain` for Streamcluster, which which covers 70% the time) and execute each application three times to reduce noise[1].

For page mappings we consider: *first-touch* (page placed on the node that first accesses it), *locality* (pages placed on the node that most frequently access them to minimize latency), *balanced* (pages spread across nodes to maximize bandwidth), and *mix* (a combination of locality and balanced). For thread mapping we consider *scatter* (threads placed round-robin across NUMA nodes) and *contiguous*[2] (threads placed sequentially across NUMA nodes). For parallelism, we consider two configurations: enough threads to fill one NUMA node and enough to fill all NUMA nodes. For the NUMA degree, we consider 1 or 2 nodes for the Broadwell system and 1, 2, or 4 nodes for Sandy Bridge, for both threads and pages separately.

By explicitly searching the NUMA degree for both threads and pages, we evaluate counter-intuitive configurations that use different numbers of nodes for thread- and page-mappings. For example,

[1]All performance measurements presented in this section are normalized to the standard baseline [25] of the default scatter thread placement with Linux's first-touch page placement strategy and one thread for each core. We include more degrees of parallelism and an additional page mapping policy in our final exploration in Section 4. The complete search space is presented in Table 1. To implement these policies, we map threads using *KMP_AFFINITY* and pages with the Linux *move_pages* function. Section 3.3 further details the exploration process.

[2] We explored contiguous instead of the previously defined compact mapping. Both policies sequentially map threads across the NUMA nodes. However, for a fixed TND, compact allocates threads to a node only when all the previous nodes have been saturated while contiguous evenly distributes the threads across the TND nodes. Therefore, compact may cause significantly unbalanced thread mappings (some nodes are fully saturated while others are empty). We selected contiguous because coupling it with TND allows us to reproduce the sequential compact mapping while preserving a balanced thread allocation.

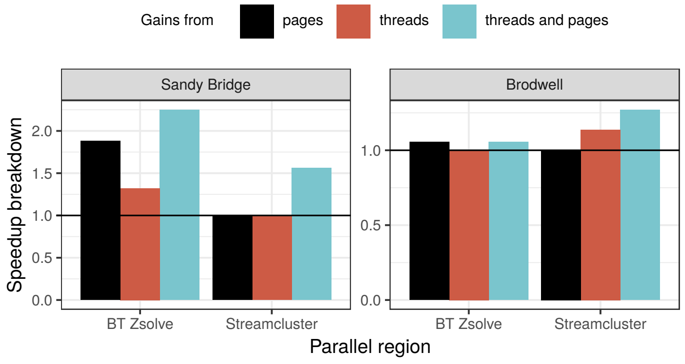

Figure 2: (Higher is better) Speedups for BT (Zsolve) and Streamcluster (Pgain) on Sandy Bridge and Broadwell. While optimizing only for page- or thread-mappings provides performance gains, coupling both parameters leads to significantly better results. These results include choosing the best NUMA degree and degree of parallelism.

a configuration may map threads to two NUMA nodes using scatter and pages to only one node, or map threads to only one node and use a balanced page mapping across all nodes. As a result of this flexibility, we consider 96 configurations on the Sandy Bridge system (the full search space is explored in Section 4). We are able to reduce the search space to 58 configurations by avoiding duplicate configurations, e.g., when all threads are mapped to the same node, then both locality and first-touch result in the same page mappings and scatter and contiguous in the same thread mappings.

2.2 Potential of Co-optimization

Optimizing across both threads and pages is critical for performance. Figure 2 shows the benefits for the two applications on the two systems, normalized to the default configuration[3]. Optimizing for both threads and pages together delivers an average of 1.54× improvement across the BT and Streamcluster regions. The improvements are larger for the Sandy Bridge system as it has more NUMA nodes, which makes it more sensitive to NUMA optimizations. If the page and thread optimizations are applied separately the gains are smaller: no gain for Streamcluster on Sandy Bridge, while on Broadwell thread optimization alone provides only half the benefit. We note that most of existing NUMA optimization techniques either target thread [5, 34] or page [2, 6, 26, 32] mappings but do not jointly consider both due to the huge search space size.

There is a large performance diversity across NUMA optimizations. Figure 3 shows the performance for each configuration for the two benchmarks. In particular, there are significant gains (up to 2.2×) but there is not a unique optimal configuration. For a fixed system (Sandy Bridge) and a fixed degree of parallelism (32 threads) **(A)**, different codes benefit from different mappings: page locality or mix with contiguous threads are optimal for BT while Streamcluster benefits from page balance with scatter threads. For a fixed system (Sandy Bridge) and code (BT) **(B)**, different degrees of parallelism benefit from different mappings: contiguous and scatter threads have the same performance with 8 threads, but contiguous outperforms scatter with 32 threads. On a fixed code (Streamcluser) and with one thread per core **(C)**, different systems benefit from different mapping policies: for Broadwell, it is more efficient to map the threads to a single node (1.14× speedup) while a similar mapping delivers poor performance on Sandy Bridge (2.0× slowdown). While it is not surprising that we observe different optimal mappings, it is interesting to notice that even policies such as mix [10] that optimize trade-offs between locality and balance, and are supposed to optimize all configurations, cannot adapt to even this small set of applications and systems.

Adapting the NUMA degree provides additional performance gains. The optimal configuration for Streamcluster on Sandy Bridge counterintuitively uses 2 nodes for thread mapping and 4 nodes for page mapping. Indeed, balance page mapping provides 1.35× (2 nodes) and 1.56× (4 nodes) speedups. Naively mapping threads and pages to the same number of NUMA nodes, or across the full system, would miss such optimizations. Similarly, there are no significant performance differences when executing Streamcluster with 10 or 20 threads on Broadwell: the optimal solution in both cases uses a single node for thread mapping.

2.3 Search Cost

Finding the optimal mapping for each application requires a per application region exploration of thread- and page-mapping, NUMA degree, and degree of parallelism. This search is expensive due to the size of the search space and the time it takes to evaluate the performance of each configuration in the search space, as well as the overhead of profiling the applications' page access patterns for the first-touch and locality/balance page mappings[4].

For Streamcluster on Sandy Bridge system, we required two profiling runs with pin for the two degrees of parallelism considered, which took approximately 45 minutes each. For the evaluated OpenMP region and degree of parallelism, there are 29 possible page- and thread-mappings, taking into account the available NUMA nodes for each. For each combination of parallelism and per-region page/thread mapping, executing the application to measure performance takes approximately 4 minutes. This results in a total tuning time of 5 hours ($2 * 45$ minutes to profile the accesses + $2 * 29 * 4$ minutes to explore the mappings) for an application that executes in 4 minutes. This is the cost of the exploration without the codelets search speedup. While the overhead for this coupled optimization search is extreme, it delivers an average of 1.5× performance gain *over* optimizing each parameter individually. (For our final evaluation in Section 4, we consider 6 degrees of parallelism and all regions of the application.)

2.4 Sampling Applications

OpenMP applications tend to have regular phases dominated by repeated parallel regions calls [28] with consistent data access patterns [32]. As a result, we can reduce the search cost by sampling these parallel regions. Figure 4 shows the execution time of Zsolve from BT (left) and Pgain from Streamcluster (right) with a default thread scatter placement across 4 nodes with first touch page placement (blue) and with our optimized mapping (orange). For BT the

[3]When we only optimize for threads or pages, we used First-touch or Scatter-threads on all cores, respectively.

[4]Collecting page accesses with Pintool [22] incurs a 10× execution overhead.

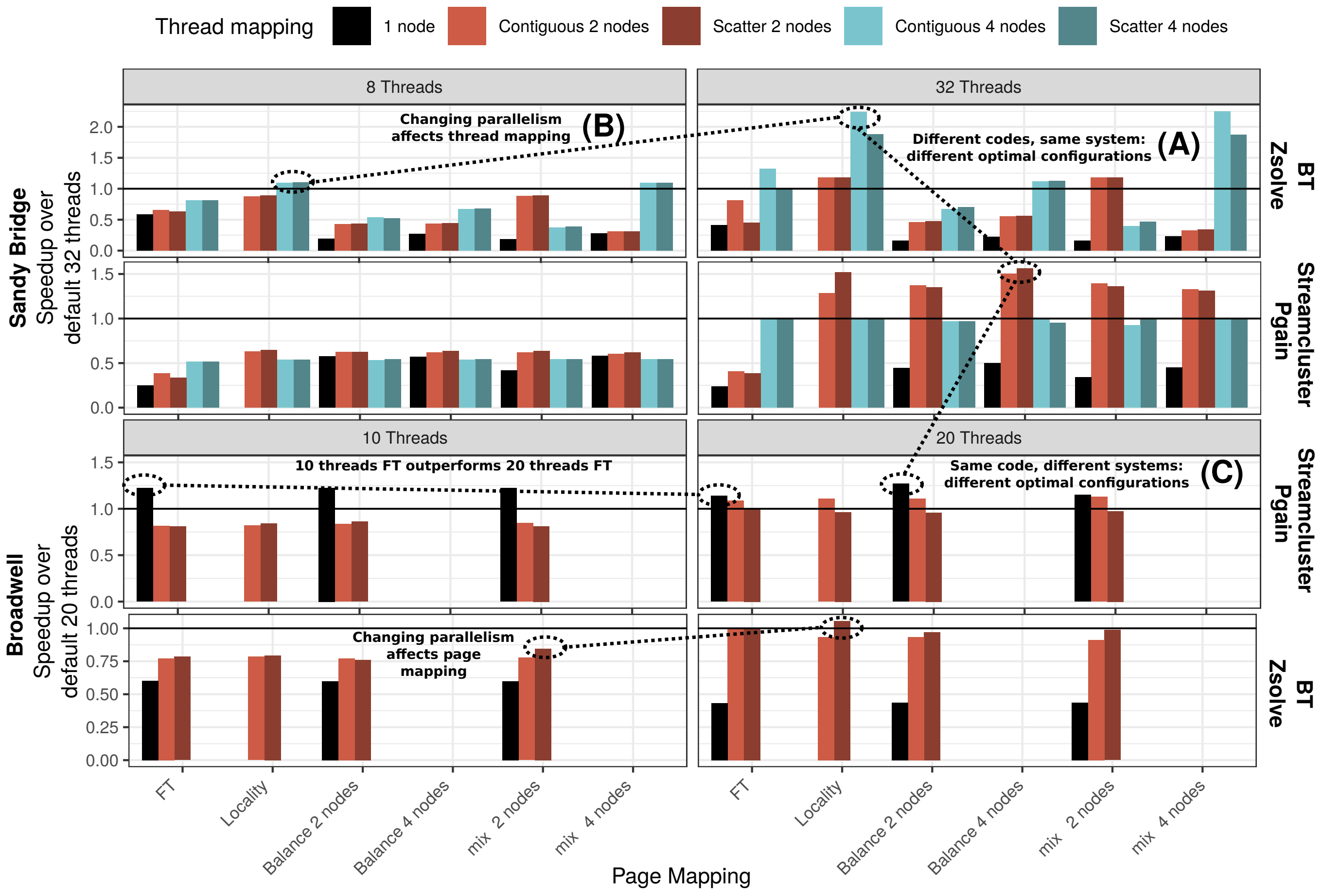

Figure 3: (Higher is better) Detailed performance gains provided by evaluating thread and page policies across different NUMA degree and degrees of parallelism. The baseline is First-touch with Scatter-threads on all cores. Thread mapping policies and thread NUMA degree are shown in colors while page mapping policies and page NUMA degree are shown horizontally. Broadwell has 2 NUMA nodes while Sandy Bridge has 4; the former results in a smaller search space.

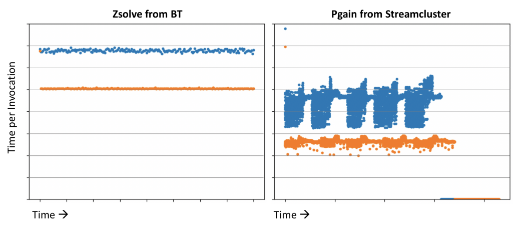

Figure 4: (Lower is better) Execution time for instances of the parallel regions Zsolve from BT (left) and Pgain from Streamcluster (right). The default NUMA configuration is in blue and the optimized configuration found from profiling only the first region instance is shown in orange.

performance of the regions is very consistent over time, indicating that sampling will be accurate. For Streamcluster there is significant performance variation, which will result in lower accuracy predictions from sampling. To evaluate this effect, we randomly selected five instances from the region to profile and used them to optimize the application. The resulting final execution times were similar, indicating that this variation did not affect the overall optimization. We further investigate accuracy in Section 4.4.

2.5 The Need for NUMA Optimization

As seen in the above examples, there is a wide diversity of application/optimization/system interactions that make the optimal choice non-obvious in many cases. We have shown examples where one optimization may yield the best results for a given application on a given system, but not for the same application on another system, and where the optimal number of threads or NUMA degree assigned to an application is not the maximum the system supports. We have further seen how these optimizations cannot be done independently to obtain the best results, necessitating a *coupled* approach wherein they are optimized together. However, a brute-force search here is prohibitively expensive due to the number of configurations and the time needed to profile page access and evaluate the resulting performance via native execution. This

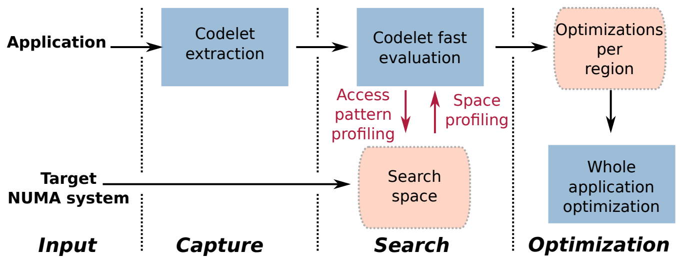

Figure 5: Codelet based mapping search strategy workflow.

overhead can be dramatically reduced if we can evaluate individual phases of applications instead of the whole execution, which we have shown provides sufficiently accurate results for optimization across OpenMP applications.

3 OPTIMIZATION

3.1 Workflow

Figure 5 presents our optimization workflow. The process starts by extracting codelets of the key parallel regions (including the data needed to execute them, see Section 3.2). The codelets are then executed for each degree of parallelism to capture the page access patterns using Numalize [11]. We combine the resulting information with the target system specification to produce a search space of thread- and page-mappings, NUMA degrees, and degrees-of-parallelism. The best configuration for each region is then found by replaying the codelets for each configuration and measuring the resulting performance (Section 3.3). Finally, the inter-region effects are evaluated by looking at how each region's optimization choice affects the other regions in the application given their execution order. From this we choose the best optimization for the application as a whole (Section 3.4).

3.2 Codelet Capture

To create a codelet for a region of an application we need to capture the input working set (data) required to replay the codelet and the default NUMA page mapping so that the codelet can reproduce them (e.g., to evaluate first-touch).

We use the Codelet Extractor and REplayer (CERE [7]) framework to capture and replay codelets. CERE uses Ptrace [27] to capture the memory by spawning a process which controls and monitors the application execution. To capture the default NUMA page mapping, CERE protects the application's memory space and starts its execution. When a protected page is accessed by the application, CERE intercepts the fault and records the thread of this first access to the page. CERE then unprotects the page and continues executing the application.

When the application reaches the region to capture, CERE again protects the full memory space and captures the content (data) and address of any page accessed during the region to allow them to be used later for replay. This provides both the information on which thread first touched each page (for replaying the first-touch policy) as well as the actual data needed to replay the codelet. We start the codelet capture just before the parallel region starts at *kmpc_fork*, which results in a codelet that can execute with different numbers of threads by adjusting the variable *OMP_NUM_THREADS* [28]. In this work, we used 4KB pages but we can explore different sizes if size is kept constant across capture and replay.

We extend CERE to record the Instruction Pointer (IP) of the access and track dynamically allocated memory segments coming from libraries such as malloc, realloc, and memalign. To do so, CERE records the allocation's addresses and collects the order of the memory function calls by augmenting the standard allocator via *LD_PRELOAD* and protecting pages to detect accesses to them. Some special memory sections must not be protected to avoid deadlocks, such as the pages containing the code of the tracing library, the OpenMP runtime, and the segmentation fault handler.

Recording the allocated segments' addresses allows CERE to identify the allocation site of all the memory accesses that touch dynamically allocated data. This information is useful when coupled with the instruction that accesses the data to identify groups of pages that are allocated and used together, and apply dataset-based mapping policies over them (see Section 3.5). Collecting the memory function calls' order allows us to optimize page mappings in the original application despite Address Space Layout Randomization [31] (ASLR), which changes the absolute addresses of dynamically allocated pages across different executions (see Section 3.4). These updates will be released as part of CERE.

3.3 Search Space

3.3.1 Fast Codelet Exploration. Once codelets are captured they can be replayed on any system using different page- and thread-mappings, NUMA degrees, and degrees of parallelism to quickly evaluate optimizations. To measure a codelet's performance, CERE restores the data accessed by the codelet to the appropriate locations in memory, warms the cache state by executing the codelet once, and then executes the codelet 10 times and reports the median execution time. This allows rapid evaluation of multiple configurations compared to executing the full application, while still including the complex application and system-level interactions that affect performance. Section 4.4 compares the performance gains predicted by the codelets to measurements on the applications themselves and shows that a codelet based on the first instance of a parallel region is accurate enough to enable effective optimization.

3.3.2 Generating the Search Space. We consider the previously described (Table 1) mapping policies in section 2.1 (thread scatter/contiguous, page first touch/local/balance/mix and dataset) and evaluate them across the available number of NUMA nodes and parallelism of the target system. The additional dataset policy is described in Section 3.5.

3.3.3 Collecting memory information. To optimize pages for locality or balance, we need to know how pages are accessed by threads over time. For locality, pages are mapped to the node of the thread that accesses them most frequently, while balance spreads them across all threads that access them. This requires a costly profiling of page accesses per thread for each degree of parallelism. To reduce the profiling cost, we only execute and profile the first instance of each OpenMP region via codelet replay. We use the Pin-based Numalize [11] tool to capture the threads accessing each page.

The time saved by only capturing memory information for a codelet easily outweighs the overhead of codelet capture itself. For instance, on Streamcluster, capturing and profiling the codelet took 210 seconds, which is 10× faster than profiling the whole application once. However, only profiling the first instance of a region does not provide the information needed for first-touch accesses, which are likely to have happened earlier in the application's execution. Instead, we collect this information during the codelet capture.

3.4 Optimizing The Whole Application

While the above search finds the best configuration per application region, it does not consider how each region affects the other regions in the application. In particular, the NUMA optimization choices for one region may reduce the performance of a later region, due to where pages are placed, or introduce a significant overhead by requiring pages to be re-mapped between regions [23].

To address such behaviors, we perform an additional exploration step. We consider two regions A and B, for which we found the best region-specific optimizations *Ca* and *Cb*. To optimize the whole application:

(1) We statically check if *Ca* and *Cb* share pages/threads that are mapped differently. If not, there is no conflict and we can apply Ca to A and Cb to B. To perform this check, we compare the memory accesses that we collected during the codelet capture.
(2) Otherwise, we measure codelet B with *Ca* and A with *Cb* and select the most efficient solution for both.

For applications with multiple regions, we evaluate all best-per-region configurations. Across our benchmarks, only 3 (CG, BT, SP) were sensitive ($> 3\%$) to such region conflicts. Section 4.6 explains how we optimized BT and SP.

Another challenge in optimizing page mappings is that data addresses of dynamically allocated pages change across executions due to Address Space Layout Randomization (ASLR). As a result, for each page, we know which node to map it to and its address during profiling, but we do not know the corresponding address for future executions. To solve this, we can calculate the offset of the page right after the memory allocation call as we kept track of the allocation calls order, and then use the new base address for subsequent runs to identify the page and placement. For our evaluation, we simply disable ASLR.

3.5 Dataset Page Mapping Policy

In addition to the policies discussed earlier, we add a *dataset*-based page-mapping policy. The intuition is that a particular dataset is likely to be used uniformly across threads, and that we can map the dataset by dividing it up into as many chunks as there are threads, and mapping the chunks across the nodes [32][5].

We use two criteria to group pages into datasets: the call that allocates the data and the instruction that first accesses the page. If two pages share the same allocation call, they are considered to belong to the same dataset. The first-access instruction is additionally used to group data that are statically allocated and therefore do not have an explicit allocation site. Allocation calls and first-access instructions are collected during the codelet capture.

[5]We have the ability to apply different mapping strategies to each dataset but we were unable to find any benefit to this, despite Trahay et. al. reporting benefits for doing so on Streamcluster [32]. We suspect this is due to the system differences.

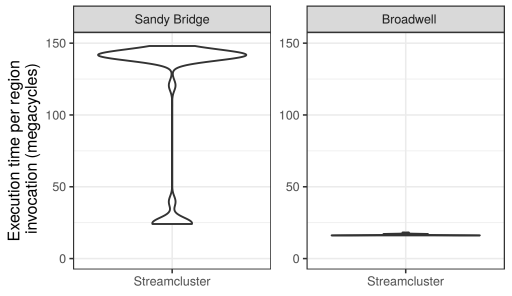

Figure 6: Violin plot with probability density on the X axis demonstrating the observed Streamcluster performance instability. Streamcluster has two distinct behaviors when executed on Sandy Bridge due to initialization (first-touch placement) happening on the core that the OS launches the application on instead of the core chosen by OpenMP after its first parallel region.

We can also change the size of the blocks within a dataset. We can reduce the block size to 1 to mimic *page interleaving*, which evenly distributes the pages according to their addresses. In section 4, we use datasets with the largest available block size (number of pages within the data set divided by number of threads).

4 EXPERIMENTAL RESULTS

4.1 Experimental Setup

We evaluate our optimizations over 17 benchmarks from the NAS 3.0 C OpenMP benchmarks [16, 28] using the Class A working set size and the Rodinia benchmarks [4] with the largest provided datasets (native-input). Applications were compiled using LLVM 3.8 [20] for 64-bit x86 and linked against the Intel/LLVM Runtime [1]. Threads were explicitly pinned. Performance numbers were collected by selecting the median across 3 runs. We extracted and optimized codelets for all benchmark regions that accounted for more than 5% of the application parallel execution time. The target system architectures are: Sandy Bridge (Intel E5-4605, 2.7GHz, 20MB LLC, 128GB RAM, 4 nodes/32 cores) and Broadwell (Intel E5-2630 v4, 2.2GHz, 25MB LLC, 128GB RAM, 2 nodes/20 cores).

4.2 Streamcluster Performance Instability

During our experiments, we observed significant performance instability in Streamcluster on Sandy Bridge: a minority of the executions were 5× faster than the median runs. Figure 6 shows the runtime of 31 executions of Streamcluster on Sandy Bridge versus Broadwell, which show little variation. This is particularly significant as Streamcluster is often identified as one of the most interesting applications for NUMA optimizations as it is sensitive to interconnect congestion. Previous works have suggested diverse – and inconsistent – optimizations for it (e.g., using multiple NUMA nodes [6, 32, 35] or a single one [34] for thread mapping).

We examined the local- and remote-bandwidth of the bimodal Sandy Bridge behavior and saw that the fast executions have nearly exclusively local DRAM accesses, while the slow ones exclusively access remote DRAM. For these experiments all OpenMP threads were mapped to the same, single node in the 4-node system and the standard first-touch page policy was used. This indicates that the performance instability of this benchmark is due to how the data is placed relative to how the OpenMP runtime places threads.

The reason for this is that Streamcluster initializes (first-touches) its data before its first OpenMP parallel region. As a result, the placement of the thread that does this first-touch, and, hence, the data's placement, is based on the *OS thread mapping* at the start of the application and *not* the OpenMP thread mapping, which takes effect later. This instability is less significant on Broadwell due to its having fewer NUMA nodes for the OS to choose among. Note that this behavior is legal under the OpenMP specification [8], as OpenMP only promises that threads are bound to cores when the application reaches the first parallel region.

To address this, we implemented a compiler pass to insert a parallel OpenMP call before initialization, which forces OpenMP to bind the threads before the data is first touched. This produces the desired, and expected, first-touch mapping. For the rest of the paper, we use this approach to remove the performance instability.

4.3 Region Optimization

To estimate the performance potential of our search space and better understand which configurations improve the performance, we profiled each codelet across diverse subsets of the search space. Figure 7 shows the performance of each region for six subsets of our full search space, representing the optimizations proposed in previous work, as well as the full search space. If an application has more than one region, we display the application name (BT) and the region (xsolve), e.g. *bt_xsolve*.

From Figure 7 we can see that none of the subsets are able to achieve the best results on all benchmarks, and that there are several benchmarks where our full search space is required to achieve the best results. For example, the regions from btree, ft, and bfs perform best with a PPP (Page Placement Policy) exploration. However, PPP provides no benefit for nn, which instead requires an exploration of TND. For cg and needle, one region in each (needle_176 and cg_405) can obtain the optimal performance with PPP/PND, but their other regions (needle_116 and cg_551) require the full search space to achieve the best results. Overall, our full search is able to achieve an average speedup of 2.48× (median 2.23×) compared to 1.84× (median 1.27×) for the best subset of existing TPP/NT/TND searches [12, 17, 27, 29, 30, 34].

4.4 Codelet Prediction Accuracy

Figure 8 (top) shows the predicted speedups from optimizing the first instance of each region in the application via the codelet vs. the actual performance achieved by all instances of that region in the application with the optimization. The codelets predict average speedups of 2.5× on Sandy Bridge and 1.4× on Broadwell, while the actual application speedups are 2.0× and 1.2×, respectively.

The inaccuracy in codelet-predicted speedup comes from two sources: First, using a single codelet assumes that the each instance of a region operates over the same data with the same access patterns as the region instance used to extract the codelet. This is not necessarily true: different calls can have different behaviors. For instance, the region *ft_516* is called six times. Before optimization, the region's median execution time is 35M cycles and the total time (all instances) is 270M cycles. After optimization, the median region execution time is reduced to 17M cycles (2× speedup) and the total region time to 170M cycles (1.59× speedup). This difference is because the region instances do not uniformly benefit from the optimization. Indeed, in this case, one region instance even sees an increase in execution time to 50M cycles. A similar effect occurs in *needle_116* region from the benchmark NW, which is executed 128 times. Its first instance and total execution time before optimization are 34M cycles and 35M cycles (the first instance represents 97% of the region time), respectively, and 21M cycles (1.6× speedup) and 58M cycles (0.6× speedup) after optimization. A way to reduce these effects is to trade speed by sampling multiple instances. Second, we use a codelet warm-up that executes the full codelet before taking performance measurements. This approach can be overly-optimistic [27], leading to better performance than expected. We further discuss these limitations in Section 6.

The accuracy of codelet performance predictions is most important for the search process to determine the correct optimization. Figure 9 shows how this affects the search space for two regions. For Streamcluster (right), the inaccuracy in the codelet-predicted speedup will cause the search to pick the third-from-best configuration, but the net difference in actual performance is very small. For BT the codelet is accurate enough to pick the best configuration. The average codelet accuracy per region (Streamcluster/BT) is 94%/99% on Sandy Bridge and 96%/98% on Broadwell.

Despite these limitations, the predicted speedup from the codelets is accurate enough to find significant optimization opportunities beyond previous work both per-codelet (Section 3) and across the full application (Section 4.6). If more accuracy is required from the codelet performance estimation, more samples of the region can be selected and evaluated during the search process.

4.5 Codelet Search Speed

The speedup of searching via codelet evaluation, as compared to full-application evaluation, is shown at the bottom of Figure 8 (Acceleration). A few regions, such as needle_176, take longer to optimize with codelets due to the 10 codelet executions used to measure performance (we replay multiple times the first instance which represents most of the region execution time). Overall, using codelets results in an average search speedup of 66× (median 8×) by avoiding having to execute the full application for each configuration in the search space.

4.6 Whole-Application Performance Gains

Figure 10 shows the effect of optimizing for one application region on the other application regions on Sandy Bridge. For the two applications (SP top and BT bottom), the plots show the performance obtained for each region (groups on the x-axis) if the optimization chosen for one region (color) is applied to it. For example, the top-left plot shows that choosing the optimization for the sp_rhs region (black) dramatically slows down the sp_xsolve, sp_ysolve, and sp_zsolve regions. Similarly, on the left side of the plot we

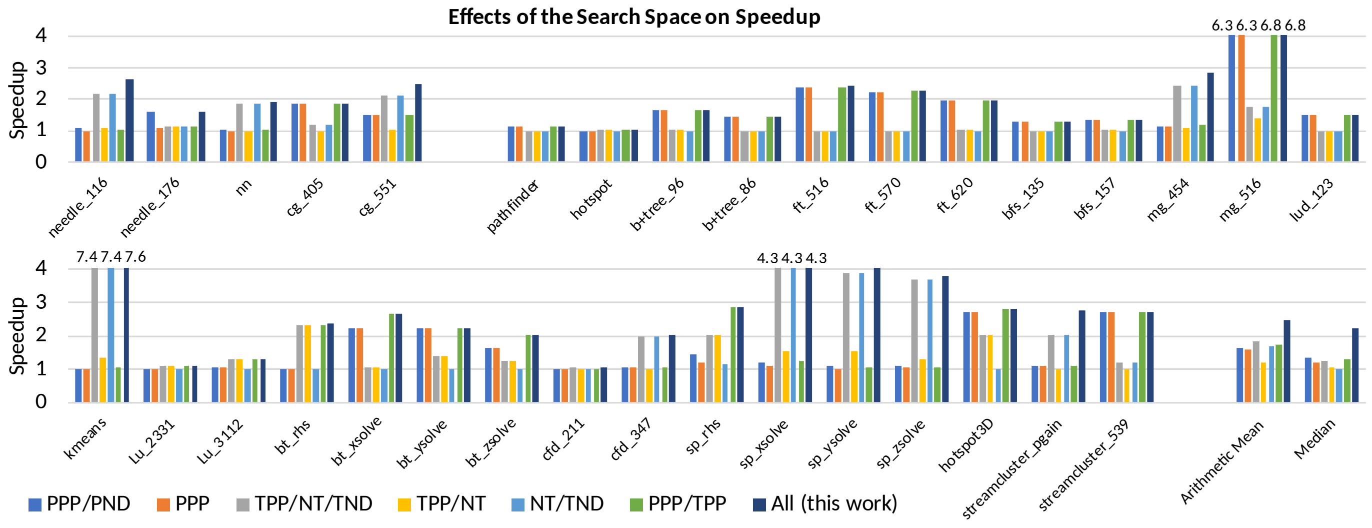

Figure 7: Comparing the limited search space results of previous work to our wider search space. Each of the previous works is able to achieve the best performance for some of the benchmarks, but none of them explores enough of the search space to achieve the best for all, as we do. (PPP=Page Placement Policy, PND=Page NUMA Degree, TPP=Thread Placement Policy, NT=Number of Threads, TND=Thread NUMA Degree) Previous work PPP: [6, 10, 23, 24, 26], TPP/NT/TND: [12, 17, 27, 29, 30, 34], PPP/TPP: [11]. Our work includes all optimizations and performs significantly better. Exploration performed on Sandy Bridge.

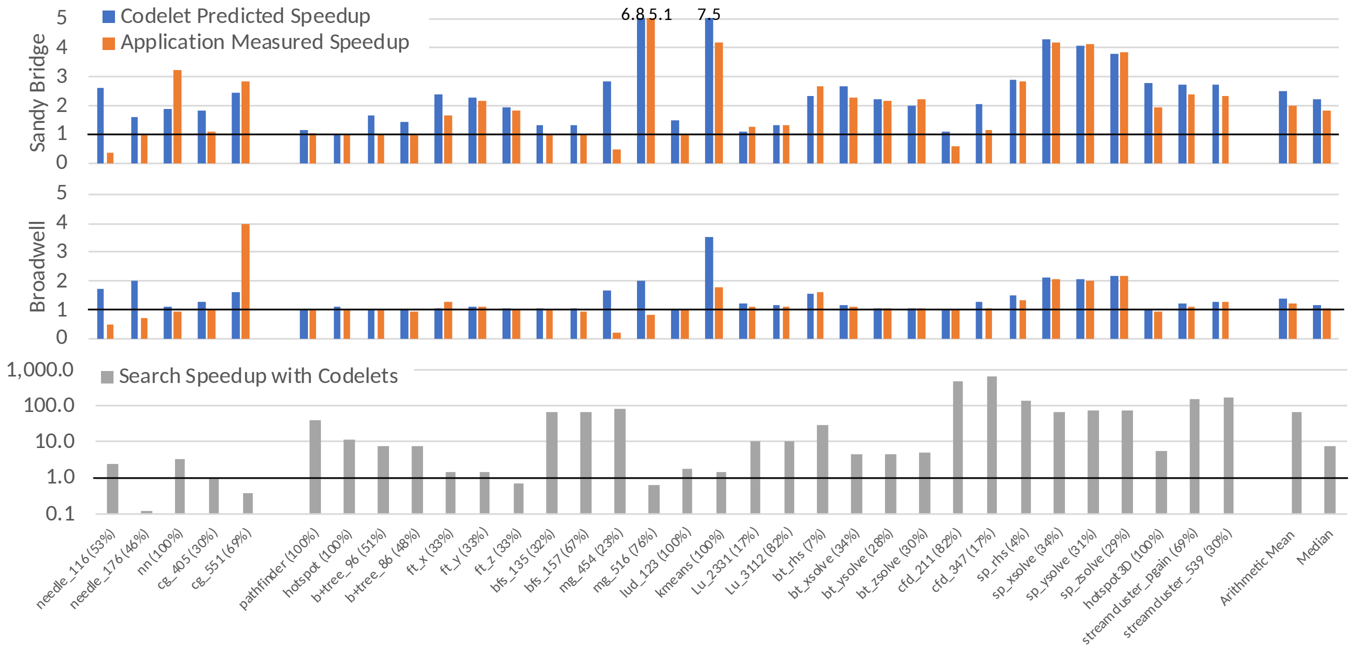

Figure 8: Top: Predicted region speedup from codelets (blue) and measured results (orange). Bottom: Speedup from using codelets for search vs. full application execution (gray). The percent of time each region accounts for in its application is shown after the region name. Regions on the left have execution times of less than 1 second.

can see that choosing the optimization for any of the solve regions (cyan/green) slows down the sp_rhs region.

To obtain the best speedup for each region independently, we would have to migrate pages and/or threads between region invocations. We were unable to find a case where migrating pages at this granularity compensated for the migration cost. Indeed, the

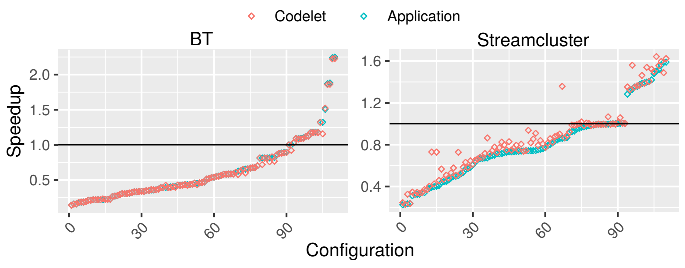

Figure 9: BT Zsolve and Streamcluster Pgain speedups predicted by the codelets and measured in the applications across our full search space. Codelets are accurate enough to pick optimization points very close to the best, even when choosing the first instance of a region as shown here.

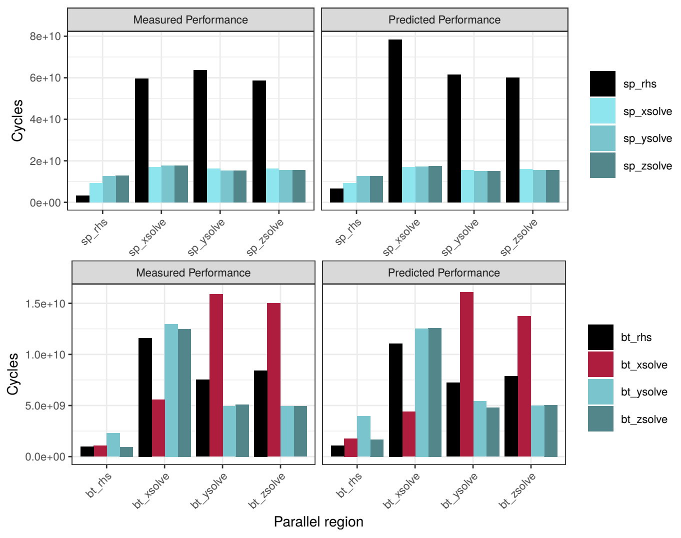

Figure 10: X axis: parallel region execution. Y axis: execution time. Colors: the target region for optimization. Optimizing a region impacts the performance of the other regions. SP regions are on top and BT regions are on bot.

page migration cost was at least 10× higher than the performance gains of dynamically migrating the pages for each region.

As a result, a region-based exploration predicts speedups of 2.3× on BT and 4× on SP, while the actual most efficient configuration speedups for the whole application are 1.6× and 3.3×, respectively, due to inter-region optimization conflicts. To identify such behaviors, we compare the thread and page mappings across the different regions. If there are differences (e.g. sp_rhs maps the threads across 4 nodes while the other regions map them to a single one), we measure the performance of each selected optimization on the codelets for the other regions in the application. Figure 10 right shows that the predicted performance provides a good estimate of the actual results.

To quantify how much such region conflicts reduce performance gains, we profiled all parallel regions with all selected optimizations for the whole application. Figure 11 compares the predicted gains if there are no region conflicts (region based exploration

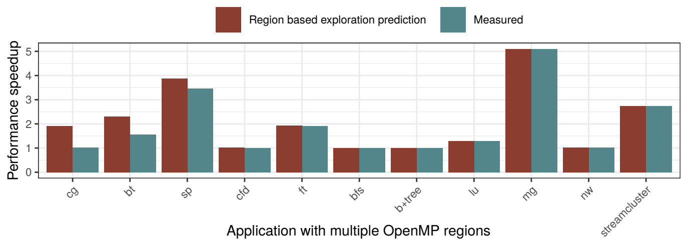

Figure 11: Performance gains for applications with multiple OpenMP regions on Sandy Bridge. Only three applications are significantly affected by region conflicts.

prediction) against the actual measurements across the 11 benchmarks that have multiple regions. Beyond BT and SP, CG is the third benchmark affected by region-conflicts: they cancel the performance gains. As a result, our method provides an arithmetic mean speedup of 1.97× instead of the 2.09× predicted by the region exploration on Sandy Bridge. While the performance difference caused by region conflicts is very small (0.12×), it does not mean that conflicts are negligible: most of the applications do not have region conflicts and as result, the averages are similar. However, for applications such as BT, SP, and CG with conflicts, the performance slowdown is significant (0.67× on average).

To further evaluate our performance gains across different input sets, we optimized BT, FT, and SP on CLASS B with the mapping policies found for CLASS A and respectively achieved 1.19×, 1.8×, and 4.2× speedups on Sandy Bridge. As expected, changing the input changes the optimization results: performance gains on BT have been reduced but remain constant on SP and FT. We also compared our results with a the Intel icc-18.3 and Clang-3.8 compilers for Streamculster on Broadwell before and after our thread tuning: icc code is 1.3× faster than Clang while icc+tuning is 1.5× faster than Clang+tuning. This shows that optimizing the code emphasizes the NUMA effects, further increasing our optimization gains.

5 RELATED WORK

Here we review methods for collecting memory information on NUMA systems, exploring NUMA optimizations, and analysing NUMA performance. For more details, Diener et al. [9] provide a full state of the art survey for thread and page mappings.

5.1 Collecting Memory Information

Many NUMA tools gather information on how pages are accessed by threads over time for analysis and subsequent placement decisions. Hardware Performance Monitoring Units (PMU) can be used during application execution [6, 21, 32] to sample a limited number of events, such as long-latency or DTLB misses. The virtual memory system can be used to lock pages to detect access to them on faults and record the thread and address accessed [7, 14, 19, 21]. This approach does not require sampling, and can collect more detailed information for first/next-touch policies [14, 19] but the overhead of locking/unlocking and trapping is greater than querying the PMU. Binary instrumentation [2, 11, 33] can also track page accesses by threads and avoid sampling, but the standard tool, Pin [22] incurs a 10-20× [2] overhead.

In this paper, we use a hybrid approach that benefits from both user space memory locking (CERE for first touch accesses during codelet capture) and instruction instrumentation (Numalize during codelet replay). We sample at the region level, which ensures that whichever region instance we capture is captured with full fidelity.

5.2 Optimization/Searching

Online optimization [3, 6] requires very low overhead, which typically limits the scope of the search. Offline optimizations [2, 11, 32], such as our work, require additional profiling steps and can be sensitive to the input data used when profiling.

There have been many proposals that address Thread Mapping [12, 17, 27, 29, 30, 34]. ForestGOMP [3] groups threads sharing data close to each other in the memory hierarchy. Wang et al. [34] further optimize thread placement and degree of parallelism via an integer programming model that quantifies bandwidth. Our work explores a similar configuration space but provides further gains by co-optimizing with the page-mapping, as seen in Streamcluster.

Page Mapping [6, 10, 23, 24, 26] has also been extensively explored. Carrfour [6] implements an online page migration policy based on PMU sampling, while Dashti et al. [6] map pages to balance remote access and congestion. Piccoli et al.[26] use compiler loop analysis and profile information to migrate pages for locality, but they cannot take into account page balance due to limited information. Majo and Gross [23, 24] address incompatible data access patterns within loops by providing developer primitives for manually distributing pages. Diener et al. [11] coupled both thread- and page-mapping, with the goal of optimizing for locality, while our search space targets a wider range of NUMA bottlenecks.

Figure 7 demonstrates that by combining a broad range of thread- and page-mapping policies with degree of parallelism, we are able to achieve significantly better speedups than any of them alone.

5.3 Performance Analysis

Many tools [13, 18] exist to assist developers to manually chose appropriate mapping by visualizing NUMA effects [2, 32], identifying bottlenecks [21], analyzing performance [15], or quantifying locality and balance sensitivity [10]. Our work automatically searches and applies the best mappings using a wide range of possible optimizations. The automation of the search is important as Figure 7 showed that no one of these approaches is optimal in all cases.

Automatic optimization requires a sufficiently accurate means of assessing the performance effects of an optimization to correctly explore the search space. Native execution is one way to avoid the difficulties of accurately modeling application/system interactions, but requires that the search be re-run for each system and requires significant execution time for each configuration evaluated. Native execution has been used for exploring thread mappings during execution [17], using random sampling to explore thread mappings per application and per phase [29, 30], and evaluating degree of parallelism [12]. These approaches were limited by the cost of native execution for exploring each configuration and the huge size of page mappings. In this work we use codelets [27] to reduce the overhead of native execution and a set of five page mappings to make the search space tractable. This makes it the first strategy to simultaneously optimize thread- and page-mappings, along with NUMA degree, and degree of parallelism.

6 LIMITATIONS AND FUTURE WORK

Our method accuracy is high enough to find the right optimizations and therefore provides significant gains (Figures 8 and 9). In this section we discuss some of its drawbacks and future work.

Code patterns constrain the codelet evaluation [28]. Codelet replay fails if the application allocates memory (accessed by the codelet) based on the running number of threads. We replay such codelets by setting a lower number of threads than the one used during capture (captured Kmeans with 32 threads). Moreover, the codelet warmup can affect the prediction accuracy and speedup. We warmup caches optimistically by replaying the codelet (the first region call) 10 times over-itself. This warmup is accurate when all calls touch similar data (true for most benchmarks), but may slow down the exploration and cause the codelet to miss-predict the execution time if the first call is more costly than the others (Needle). To address this issue, we can sample more calls or improve CERE warmup by keeping a trace of the recently accessed pages just before the parallel region for each thread [7].

As future work, we plan to prune policies with thread-access-patterns (e.g. use balance with many shared pages) and PMU (e.g. use bandwidth/latency to guide TND). We note that such models will be dependent on the target NUMA system.

7 CONCLUSION

This work shows that the coupled optimization of thread- and page-mapping, NUMA degree, degree of parallelism, and inter-region interactions delivers significantly better performance across a range of applications and systems than previous approaches of optimizing for only one or two particular NUMA bottleneck(s). While it is well-known that there is no one optimization that works optimally for all applications and systems, this work is the first to demonstrate a practical way to automatically explore this large space.

To accomplish this, we have addressed the challenges of parameterizing the search space and efficiently evaluating the effects of each optimization. We addressed the first challenge by using combinations of existing policies that target specific NUMA bottlenecks. The combination of them, when searched in a coupled manner, allows us to find a broad range of solutions. To quickly evaluate the performance impact of each configuration change, we extracted samples of the key parallel regions of each application as codelets, which we could then quickly re-run for each configuration. While the codelets do not perfectly reproduce the application behavior, they are close enough and fast enough to allow us to identify significant performance gains across our large configuration space.

This combination of a broad search space and an efficient and accurate search methodology allows us to make automatic optimization across a broad range of NUMA criteria practical for the first time.

ACKNOWLEDGMENTS

This work was supported by the Swedish Foundation for Strategic Research (FFL12-0051 and RIT15-0012), the Knut and Alice Wallenberg Foundation through the Wallenberg Academy Fellows Program, and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant No 715283).

REFERENCES

[1] Accessed: 2019-01-01. LLVM OpenMP runtime. https://www.openmprtl.org/. (Accessed: 2019-01-01).

[2] David Beniamine, Matthias Diener, Guillaume Huard, and Philippe OA Navaux. 2015. TABARNAC: visualizing and resolving memory access issues on NUMA architectures. In *Proceedings of the 2nd Workshop on Visual Performance Analysis*. ACM, 1.

[3] François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, and Raymond Namyst. 2010. ForestGOMP: an efficient OpenMP environment for NUMA architectures. *International Journal of Parallel Programming* 38, 5-6 (2010), 418–439.

[4] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee, 44–54.

[5] Eduardo HM Cruz, Matthias Diener, and Philippe OA Navaux. 2015. Communication-aware thread mapping using the translation lookaside buffer. *Concurrency and Computation: Practice and Experience* 27, 17 (2015), 4970–4992.

[6] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic management: a holistic approach to memory placement on NUMA systems. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 381–394.

[7] Pablo de Oliveira Castro, Chadi Akel, Eric Petit, Mihail Popov, and William Jalby. 2015. CERE: LLVM Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 1 (2015), 6. DOI: http://dx.doi.org/10.1145/2724717

[8] Bronis R de Supinski, Thomas RW Scogland, Alejandro Duran, Michael Klemm, Sergi Mateo Bellido, Stephen L Olivier, Christian Terboven, and Timothy G Mattson. 2018. The Ongoing Evolution of OpenMP. *Proc. IEEE* 99 (2018), 1–16.

[9] Matthias Diener, Eduardo HM Cruz, Marco AZ Alves, Philippe OA Navaux, and Israel Koren. 2017. Affinity-based thread and data mapping in shared memory systems. *ACM Computing Surveys (CSUR)* 49, 4 (2017), 64.

[10] Matthias Diener, Eduardo HM Cruz, and Philippe OA Navaux. 2015. Locality vs. Balance: Exploring data mapping policies on NUMA systems. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*. IEEE, 9–16.

[11] Matthias Diener, Eduardo HM Cruz, Laércio L Pilla, Fabrice Dupros, and Philippe OA Navaux. 2015. Characterizing communication and page usage of parallel applications for thread and data mapping. *Performance Evaluation* 88 (2015), 18–36.

[12] Juan J Durillo, Philipp Gschwandtner, Klaus Kofler, and Thomas Fahringer. 2018. Multi-Objective region-Aware optimization of parallel programs. *Parallel Comput.* (2018).

[13] Alfredo Giménez, Todd Gamblin, Barry Rountree, Abhinav Bhatele, Ilir Jusufi, Peer-Timo Bremer, and Bernd Hamann. 2014. Dissecting on-node memory access performance: a semantic approach. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*. IEEE, 166–176.

[14] Brice Goglin and Nathalie Furmento. 2009. Enabling high-performance memory migration for multithreaded applications on linux. (2009).

[15] William Jalby, David Kuck, Allen D Malony, Michel Masella, Abdelhafid Mazouz, and Mihail Popov. 2018. The Long and Winding Road Toward Efficient High-Performance Computing. *Proc. IEEE* 99 (2018), 1–19.

[16] Hao-Qiang Jin, Michael Frumkin, and Jerry Yan. 1999. The OpenMP implementation of NAS parallel benchmarks and its performance. (1999).

[17] Tobias Klug, Michael Ott, Josef Weidendorfer, and Carsten Trinitis. 2011. autopin–automated optimization of thread-to-core pinning on multicore systems. In *Transactions on high-performance embedded architectures and compilers III*. Springer, 219–235.

[18] Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. 2012. MemProf: a memory profiler for NUMA multicore systems. In *ATC-USENIX Annual Technical Conference*.

[19] Stefan Lankes, Boris Bierbaum, and Thomas Bemmerl. 2009. Affinity-on-next-touch: an extension to the Linux kernel for NUMA architectures. In *International Conference on Parallel Processing and Applied Mathematics*. Springer, 576–585.

[20] Chris Lattner and Vikram Adve. 2004. 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*. IEEE Computer Society, 75.

[21] Xu Liu and John Mellor-Crummey. 2014. A tool to analyze the performance of multithreaded programs on NUMA architectures. *ACM Sigplan Notices* 49, 8 (2014), 259–272.

[22] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, Vol. 40. ACM, 190–200.

[23] Zoltan Majo and Thomas R Gross. 2012. Matching memory access patterns and data placement for NUMA systems. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. ACM, 230–241.

[24] Zoltan Majo and Thomas R Gross. 2013. (Mis) understanding the NUMA memory system performance of multithreaded workloads. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*. IEEE, 11–22.

[25] Abdelhafid Mazouz, Denis Barthou, and others. 2011. Performance evaluation and analysis of thread pinning strategies on multi-core platforms: Case study of spec omp applications on intel architectures. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*. IEEE, 273–279.

[26] Guilherme Piccoli, Henrique N Santos, Raphael E Rodrigues, Christiane Pousa, Edson Borin, and Fernando M Quintão Pereira. 2014. Compiler support for selective page migration in NUMA architectures. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 369–380.

[27] Mihail Popov, Chadi Akel, Yohan Chatelain, William Jalby, and Pablo de Oliveira Castro. 2017. Piecewise holistic autotuning of parallel programs with CERE. *Concurrency and Computation: Practice and Experience* 29, 15 (2017), e4190.

[28] Mihail Popov, Chadi Akel, Florent Conti, William Jalby, and Pablo de Oliveira Castro. 2015. PCERE: Fine-grained Parallel Benchmark Decomposition for Scalability Prediction. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 1151–1160.

[29] Petar Radojković, Vladimir Čakarević, Miquel Moretó, Javier Verdú, Alex Pajuelo, Francisco J Cazorla, Mario Nemirovsky, and Mateo Valero. 2012. Optimal task assignment in multithreaded processors: a statistical approach. *ACM SIGARCH Computer Architecture News* 40, 1 (2012), 235–248.

[30] Petar Radojković, Paul M Carpenter, Miquel Moreto, Vladimir Čakarević, Javier Verdu, Alex Pajuelo, Francisco J Cazorla, Mario Nemirovsky, and Mateo Valero. 2016. Thread assignment in multicore/multithreaded processors: a statistical approach. *IEEE Trans. Comput.* 65, 1 (2016), 256–269.

[31] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*. ACM, 298–307.

[32] François Trahay, Manuel Selva, Lionel Morel, and Kevin Marquet. 2018. NumaMMA: NUMA MeMory Analyzer. In *International Conference on Parallel Processing*.

[33] Sébastien Valat and Othman Bouizi. 2018. NUMAPROF, A NUMA Memory Profiler. In *European Conference on Parallel Processing*. Springer, 159–170.

[34] Wei Wang, Jack W Davidson, and Mary Lou Soffa. 2016. Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale numa machines. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on*. IEEE, 419–431.

[35] Hao Xu, Shasha Wen, Alfredo Gimenez, Todd Gamblin, and Xu Liu. 2017. DR-BW: identifying bandwidth contention in NUMA architectures with supervised learning. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 367–376.