Automatic Skeleton-Driven Memory Affinity for Transactional Worklist Applications

Noname manuscript No.
(will be inserted by the editor)

Automatic Skeleton-Driven Memory Affinity for Transactional Worklist Applications

Luís Fabrício Wanderley Góes · Christiane Pousa Ribeiro · Márcio Castro · Jean-François Méhaut · Murray Cole · Marcelo Cintra

Received: date / Accepted: date

Abstract Memory affinity has become a key element to achieve scalable performance on multi-core platforms. Mechanisms such as thread scheduling, page allocation and cache prefetching are commonly employed to enhance memory affinity which keeps data close to the cores that access it. In particular, Software Transactional Memory (STM) applications exhibit irregular memory access behavior that makes harder to determine which and when data will be needed by each core. Additionally, existing STM runtime systems are decoupled from issues such as thread and memory management. In this paper, we thus propose a skeleton-driven mechanism to improve memory affinity on STM applications that fit the worklist pattern employing a two-level approach. First, it addresses memory affinity in the DRAM level by automatic selecting page allocation policies. Then it employs data prefetching helper threads to improve affinity in the cache level. It relies on a skeleton framework to exploit

Luís Fabrício Wanderley Góes
PPGEE - GSDC Group, Pontifícia Universidade Católica de Minas Gerais, Brazil
E-mail: lfwgoes@pucminas.br

Christiane Pousa Ribeiro
INRIA - CEA - LIG Laboratory, Grenoble University, France
E-mail: pousa@imag.fr

Márcio Castro
INRIA - CEA - LIG Laboratory, Grenoble University, France
E-mail: bastosca@imag.fr

Jean-François Méhaut
INRIA - CEA - LIG Laboratory, Grenoble University, France
E-mail: mehaut@imag.fr

Murray Cole
School of Informatics - ICSA - CARD Group, University of Edinburgh, United Kingdom
E-mail: mic@staffmail.ed.ac.uk

Marcelo Cintra
School of Informatics - ICSA - CARD Group, University of Edinburgh, United Kingdom
E-mail: mc@staffmail.ed.ac.uk

the application pattern in order to provide automatic memory page allocation and cache prefetching. Our experimental results on the STAMP benchmark suite show that our proposed mechanism can achieve performance improvements of up to 46%, with an average of 11%, over a baseline version on two NUMA multi-core machines.

Keywords memory affinity and software transactional memory and parallel algorithmic skeleton and multi-core platforms

1 INTRODUCTION

Multi-core chips have recently become the predominant processor design [1]. However, as the number of cores per chip increases, memory access contention becomes a major bottleneck. A scalable solution to alleviate this problem is to build platforms with on-chip memory controllers, employing a Non-Uniform Memory Access (NUMA) design in order to keep the abstraction of a single shared memory. As a drawback, this non-uniformity can potentially increase memory access latency and degrade bandwidth usage.

The exploitation of memory affinity, which keeps data close to the cores which access it [2,4] thus becomes a key element in attaining scalable performance. In particular, enhanced affinity reduces the memory latency perceived by threads and memory contention. Mechanisms such as thread scheduling, page allocation and cache prefetching are common approaches to improve memory affinity. These have been predominantly studied in the context of regular parallel applications on NUMA multi-core platforms, in which the memory access behavior is stable and predictable [2,4,23,7,6].

In contrast, Software Transactional Memory (STM) [10,19] applications present an irregular behavior, in which data dependencies between threads are only known at runtime. Particularly, STM systems provide a simplified API that removes the burden of correctly synchronizing threads on data races. This programming model allows programmers to write parallel code as transactions, which are then guaranteed by the runtime system to execute atomically and in isolation regardless of eventual data races. This provides an efficient model for extracting coarse-grained parallelism from apparently unpromising irregular applications. However, typical STM systems are decoupled from issues of thread and memory management, leading to poor exploitation of memory affinity by the native operating system.

Parallel algorithmic skeletons [1,8,20] are a common solution to enhance thread and memory management based on the application behavior. Skeleton-based programming stems from the observation that many parallel algorithms fit into generic communication and computation patterns, such as pipeline, worklist and MapReduce [11]. Communication and computation patterns can be encapsulated in a common infrastructure, leaving the programmer with only the implementation of the particular operations required to solve the problem at hand. Thus, this programming approach eliminates some of the major challenges of parallel programming, namely thread communication, scheduling and

orchestration. As long as STM applications present a common algorithmic pattern, they can be encapsulated within a skeleton framework. This allows the implementation of efficient memory affinity mechanisms within the skeleton that take into account the STM nature and also the communication behavior of the application.

In this paper, we propose an automatic skeleton-driven mechanism to improve memory affinity on a significant subset of STM applications that fits in the worklist pattern. First, this mechanism selects and enables NUMA-aware page allocation policies depending on the behavior captured by the skeleton framework called OpenSkel [16]. Then, it automatically enables skeleton-driven helper threads to prefetch data to the last level shared cache. Our experimental results on the STAMP benchmark suite show that our mechanism can achieve significant performance improvements over a baseline version on two NUMA platforms.

To the best of our knowledge, this is the first paper to make the following contributions:

- We propose a novel automatic mechanism to improve memory affinity in the DRAM and cache levels for STM applications;
- We provide a detailed analysis of a new skeleton-driven approach to implement software helper threads for data prefetching.

The rest of this paper is organized as follows. Section 2 shows the memory access behavior and some insights on how to improve memory affinity on transactional memory applications. Section 3 describes the transactional skeleton framework used to implement our memory affinity mechanism. Section 4 describes in detail our proposed mechanism and its implementation. Section 5 outlines our experimental setup while Section 6 presents results. Finally, Section 7 discusses related work and Section 8 concludes this paper.

2 MOTIVATION

On STM applications, data dependencies cannot be determined at compile time. In principle, no specific memory access pattern can be assumed since threads may potentially access memory addresses in an uniform fashion.

Figure 1 supports this observation by showing the memory page accesses footprint of four STAMP applications during their execution. In particular, we can observe that each thread ends up traversing most of the memory page space, frequently conflicting with other threads. This fact makes harder to enhance memory affinity by applying pure static approaches. At the same time, dynamic approaches usually require frequent data migrations that can be prohibitively expensive at a lower granularity level (e.g., page level).

To tackle this problem, we propose an automatic skeleton-driven mechanism to enhance memory affinity for transactional memory applications. It is a hybrid solution that exploits static and dynamic information to provide affinity in the memory and cache levels. In the memory level, it takes into account

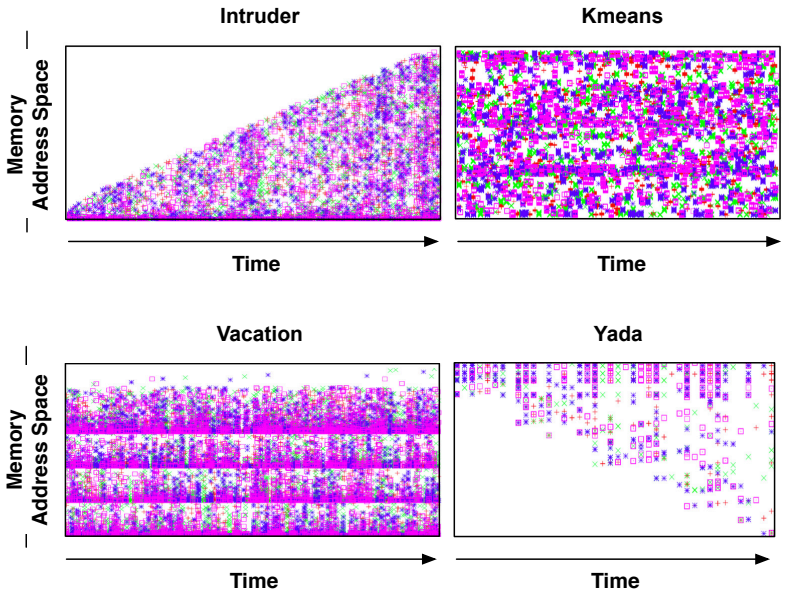

Fig. 1 Sampled memory footprint of pages accesses for transactional memory applications executing with four threads. Each data point represents a thread accessing a certain memory page at a specific time. Each symbol represents a different thread.

this irregular memory access pattern to apply a more suitable page allocation policy. Such policy is used to specify how memory pages are distributed over the physical memory banks of a machine. Additionally, based in the skeleton information, we apply a heuristic that switches to a different strategy if profitable. We do not employ page migration since it would be to costly as threads access different memory pages in a short period of time. Instead, we employ dynamically created *helper threads* (HT) in the cache level for data prefetching. These HTs make use of idle cores to bring potentially useful data into the last level shared caches.

The skeleton framework allows the automatic creation and synchronization of HTs and also hints the next task to be processed. This enables our mechanism to enhance memory affinity on STM applications in a very dynamic and fine-grained level. In the next section, we describe the OpenSkel framework [16] that supports transactional worklist applications in which our mechanism is implemented.

3 THE OPENSKEL FRAMEWORK

Many TM applications exhibit the *worklist* pattern. Such applications are characterized by the existence of a single operation: process an item of work known

as a *work-unit* from a dynamically managed collection of work-unit instances, the *worklist*.

```
add n seed work-units into the worklist
foreach worker do
   while worklist ≠ empty do
      remove a work-unit w_i from the work-
      list
      process w_i
      [add new work-units w'_i]
   end while
end foreach
```

Fig. 2 Generic behavior of the worklist skeleton.

The algorithm in Figure 2 sketches the generic behavior of worklist algorithms. The worklist is seeded with an initial collection of work-units. The worker threads then iterate, grabbing and executing work-units until the worklist is empty. As a side effect of work-unit execution, a worker may add new work-units to the worklist.

In this paper, we use the OpenSkel framework, proposed in [16], to implement our memory affinity mechanism [25]. It is a C runtime system library that enables the use of the *transactional worklist skeleton*. It provides an API to handle transactional worklists. OpenSkel also exploits existing word-based STM systems to deal with transactions. As shown in Figure 3, the programmer is provided with three basic primitives so as to allocate, run and free a worklist. Additionally, the API provides a function, namely *oskel_wl_addWorkUnit()*, with which the programmer can dynamically add work-units to the worklist.

As shown in Figure 3, the programmer has to implement all four functions required to describe a transactional worklist. In order to initialize and terminate local variables used by each worker thread, the programmer has to use the *oskel_wl_initWorker()* and *oskel_wl_destroyWorker()* functions respectively. The programmer thus implements the kernel to process a work-unit in the *oskel_wl_processWorkUnit()* main function. Lastly, the *oskel_wl_update()* implements any kind of operation to update the global data when a worker thread is just about to finish. Additionally, the programmer has to declare two structures as part of the interface specification. The *oskel_wl_shared_t* structure contains all shared global variables, and the *oskel_wl_private_t* data structure specifies all private local variables of each thread.

Once an *oskel_wl_shared_t* instance is initialized and the worklist is loaded with work-units, the programmer has just to call *oskel_wl_run()*. The *oskel_wl_run()* function starts all worker threads and waits in a barrier.

Figure 4 shows OpenSkel's internal implementation of each worker. Each worker thread coordinates the execution of the aforementioned user functions. After initialization, each worker grabs work-units with *oskel_wl_getWorkUnit()* and calls the *oskel_wl_processWorkUnit()* function until the worklist is empty.

```
1: int main(void *args)
2: {
3:   oskel_wl_t* oskelPtr =
4:        oskel_wl_alloc(&oskel_wl_initWorker,
5:               &oskel_wl_processWorkUnit,
6:               &oskel_wl_update,
7:               &oskel_wl_destroyWorker);
8:
9:   file f = open(args[1]);
10:  oskel_wl_shared_t global = malloc();
11:  global -> data = malloc();
12:
13:  while (!feof(f))
14:    oskel_wl_addWorkUnit(oskelPtr,read(f));
15:
16:  oskel_wl_run(oskelPtr,global);
17:  oskel_wl_free(oskelPtr);
18: }
```

Fig. 3 User pseudocode.

```
1: void oskel_wl_worker(oskel_wl_t* oskelPtr,
2:                 oskel_wl_shared_t* global)
3: {
4:   void* workUnit;
5:   int tid = getThreadId();
6:   oskel_wl_private_t* local
7:                = oskelPtr -> locals[tid];
8:
9:   oskel_wl_initWorker(local,global);
10:
11:  do {
12:    atomic {
13:      if((workUnit = oskel_wl_getWorkUnit()))
14:        oskel_wl_processWorkUnit(workUnit,
15:                                 local,global);
16:    }
17:  } while(workUnit);
18:
19:  atomic {
20:    oskel_wl_update(local,global);
21:  }
22:
23:  oskel_wl_destroyWorker(local,global);
24: }
```

Fig. 4 OpenSkel Internal Worker Thread pseudocode.

Although the *oskel_wl_getWorkUnit()* is within a transaction, its variables are not protected by transactional barriers. Instead, this function internally uses locks to access OpenSkel's worklist and internal state. This is essential to decouple the worklist management from the transactional memory system, avoiding extra transaction conflicts and contention.

The *oskel_wl_processWorkUnit()* procedure is executed within transactional barriers placed by the skeleton library. This function is then translated to transactional code at compile time by any existing TM compiler such as Dresden TM [13]. This process is transparent and completely relieves the application programmer of the burden of having to handle transactions explicitly. The OpenSkel runtime also enables the implementation of skeleton-driven op-

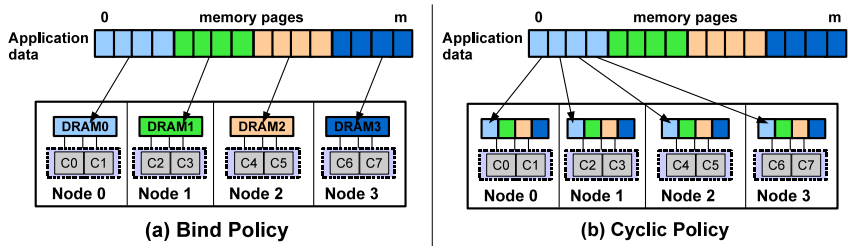

Fig. 5 Bind and cyclic memory page allocation policies.

timizations. This feature is essential to implement our memory affinity mechanism as we describe in the next section.

4 SKELETON-DRIVEN MEMORY AFFINITY

The skeleton framework knows about the application communication pattern and provides dynamic information on the application behavior. This information combined with the knowledge about the nature of STM applications allows us to implement efficient performance optimizations. In particular, our memory affinity mechanism, called *SkelAff*, implements memory page allocation policies and cache prefetching exploiting the information provided by the worklist pattern. For example, the skeleton knows which is the next work-unit to be executed. This work-unit can thus be processed in advance by a helper thread in order to prefetch potentially useful data that will be required in the near future. Additionally, it can also allocate pages close to a specific thread if it knows that a work-unit can generate new work-units that are memory related. In this section, we first present how our mechanism deals with memory page allocation. Then, we describe how it prefetches data using helper threads. Finally, we propose a heuristic to automate our mechanism.

4.1 Enabling Page Allocation Policies

In order to implement our mechanism in transactional applications, we exploit two page allocation policies named *bind* and *cyclic* [23].

The *bind* page allocation policy aims at reducing access latency by binding data of a thread to a single memory bank. In transactional worklist applications, the *bind* policy can lead to faster conflict resolution as long as memory related work-units are placed in the same node. Whenever a thread allocates a memory page, its corresponding virtual page is placed on a physical memory bank based on information about the machine topology provided by the skeleton framework. Figure 5a depicts the *bind* policy in a NUMA machine with four nodes. The application data is composed of m pages which are divided into four contiguous groups, each assigned to a different memory bank.

The *bind* policy can be profitable to transactional applications in which each individual thread generates and consumes its own new shared data.

A side-effect of binding data to restricted memory banks is that it may cause more memory contention when different transactional threads share the same memory range. In order to avoid such behavior, our mechanism also exploits the *cyclic* page allocation policy that spreads memory pages over all memory banks of the machine following a round-robin distribution. A page i is placed in the memory bank $i \ mod \ M$, where M is the number of memory banks of the machine. The *cyclic* policy aims at balancing memory banks usage, because it allows more memory banks to be accessed in parallel, providing more bandwidth to cores. Figure 5b shows a schema that represents the *cyclic* page allocation policy in a NUMA machine with four nodes.

In order to support page allocation policies in the OpenSkel framework, we used the following tools: *hwloc* [5] and *libnuma* [18]. The first one provides information about the machine topology (e.g., number of nodes and memory banks hierarchy) that is used by OpenSkel to map threads to cores. This allows OpenSkel to guarantee that threads do not migrate at runtime, avoiding data migration. The latter provides an API to set specific page allocation policies such as *bind* and *cyclic* which are encapsulated by the OpenSkel framework. As a consequence, our skeleton-driven mechanism avoids page allocation policies being exposed to the application programmer. Particularly, it can fully automate the process of selecting page allocation policies.

These tools integrated with OpenSkel enables *SkelAff* to set a page allocation policy as soon as the *oskel_wl_alloc()* is called. After this, all allocated data is placed on memory banks following that specific policy including the worklist and work-units. Additionally, OpenSkel provides several entry points in which the page allocation policy can be switched such as the *oskel_wl_addWorkUnit()* and *oskel_wl_getWorkUnit()* calls. Our memory affinity mechanism can also exploit the fact that OpenSkel provides information about the application behavior and algorithmic pattern to switch to an appropriate page allocation policy at runtime.

4.2 Enabling Cache Prefetching

Cache affinity can be enhanced by performing data prefetching using automatically created *helper threads* (HT) [24,9]. They are auxiliary threads that run concurrently with a main thread. Their purpose is not to directly contribute to the actual program computation, which is still performed in full by the main thread, but to facilitate the execution of the main thread indirectly. Typically modern multi-cores have at least one shared level of cache among the cores, so that HTs may try to bring data that will be required by the main thread into this shared cache ahead of time.

TM applications have a number of characteristics that render the use of HTs appealing. First of all, some transactional applications do not scale up to a large number of cores because the number of aborts and restarts increases.

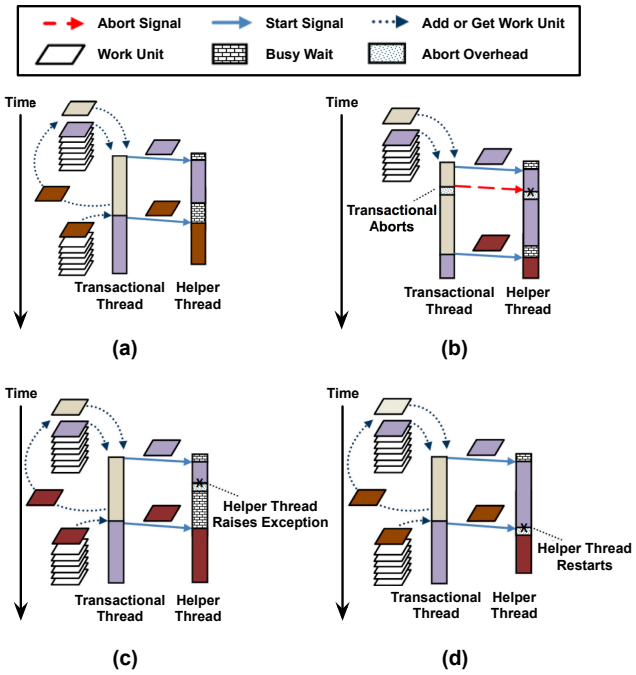

Fig. 6 Interaction between worker and helper threads.

If more cores are available, they can be used to run HTs instead of more TM threads, improving the performance of the applications. Another characteristic of STM applications is the high overhead and cache miss ratio of transactional loads and stores. This suggests that HT can more easily stay ahead of the main thread while effectively prefetching data for it.

Unfortunately, a STM does not have the required information to implement HT on its own. The worklist skeleton, on the other hand, provides two key information to make HTs feasible: when to start a HT and which data to prefetch.

Figure 6 shows how helper threads interact with transactional threads. In order to guarantee that the HT is always executing the correct next work-unit, the transactional or main thread always keeps two work-units at the same time. While it processes the former work-unit, it signals the HT with the latter one. Thus, when the transactional thread calls *oskel_wl_getWorkUnit()* for the first time, it grabs two work-units. For all the following calls to *oskel_wl_getWorkUnit()*, it grabs just one work-unit, sends it to the HT and starts executing the previous one as shown in Figure 6a. Meanwhile, the transactional thread can insert new work-units into the worklist with an *oskel_wl_addWorkUnit()* call. However, if a transaction modifies shared data, the HT may go down the wrong path, possibly prefetching wrong data or even worse, raising exceptions that could crash the whole application. *SkelAff* thus

implement a transparent mechanism to deal with exceptions within OpenSkel. If an exception is raised, our mechanism aborts the helper thread and restarts it in a wait barrier. Figure 6b, 6c and 6d show how our mechanism deals with transaction aborts as well as incorrect executions of helper threads.

Helper thread code is generated and instrumented in the same way compilers such as TM Dresden Compiler [14] and OpenTM [3] would do for STM systems. However, instead of function calls to the STM system, it uses modified functions for reading and writing shared variables. Every time a HT has to access a shared global variable, it has to use special functions to redirect accesses to the internal metadata structures managed by our runtime system. This is a somewhat similar approach to hardware HT, but we do not rely on specific hardware to perform buffering. As HTs do not change the state of the application, each write to a global variable is done in its local entry in a hash table rather than in the actual memory location. If the same variable is read after being written, the value will be extracted from the hash table instead of the actual memory location. This enables HTs to follow the correct path of control and hopefully prefetch the correct data.

4.3 Automatic Memory Affinity

SkelAff relies on a simple but efficient heuristic to automatically enable and select its affinity mechanisms. This heuristic presented in Figure 7 exploits the application algorithmic pattern and system information to choose which mechanisms should be applied. It firstly employs a thread mapping strategy. Since STMs do not manage threads, then they are left with the operating system default scheduling strategy. The Linux scheduling strategy tends to map threads initially following the scatter mapping strategy. Scatter distributes threads across different processors avoiding cache sharing between cores in order to reduce memory contention. However, at runtime the Linux scheduler migrates threads trying to reduce memory accesses and I/O costs. For this reason, our heuristic firstly employs a static scatter mapping strategy in which threads are not allowed to migrate at runtime, guaranteeing a more predictable performance. This also avoids, when possible, that HTs compete with each other for the same cache. Thread mapping in STM applications has been extensively studied in [6, 7] and it is not the focus of this paper.

As aforementioned in Section 2, STM applications exhibit very irregular memory access footprint tending to an uniform distribution. Based on this observation, *SkelAff* then set *cyclic* as the default page allocation policy as it distributes memory pages equally across nodes. This can increase the average performance of STM applications since each thread will potentially access the same amount of data on each node. However, if the skeleton informs that threads are generating new work-units, *SkelAff* optimistically assumes that these work-units are memory related. Then it switches the page allocation policy to *bind*.

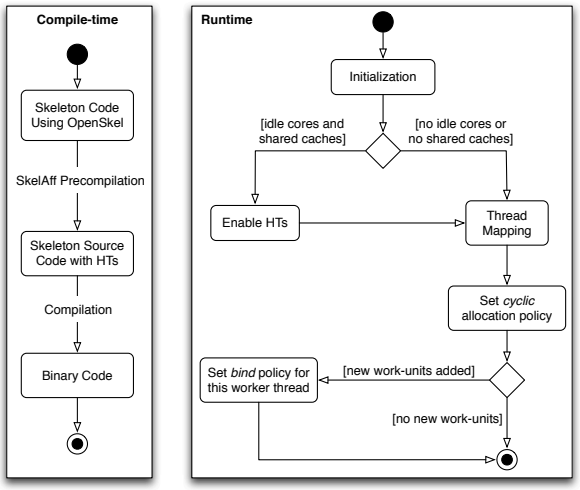

Fig. 7 Heuristic to automate the selection and enabling of the memory affinity mechanisms.

Our mechanism has also an optimistic approach to enable helper threads. It always activates HTs if there are idle cores that share the last level cache. If so, it schedules each pair of worker thread and helper thread to cores that share the same level of cache. To reduce cache pollution due to inefficient prefetching, a lifespan parameter (i.e., number of words prefetched per work-unit) and a limit to the hash table size are employed. By employing this heuristic, our proposed mechanism delivers automatic skeleton-driven memory affinity to STM worklist applications.

5 EXPERIMENTAL SETUP

In this section we present our experimental setup to evaluate the presented memory affinity mechanism. We selected two representative multi-core platforms with NUMA characteristics:

- **NUMA16**: a multi-core machine based on eight Dual Core AMD Opteron Processor 875. Cores have private L1 (64KB) and L2 (1MB) caches and do not share any cache memory;
- **NUMA32**: four eight-core Intel Xeon X7560 processors. Each core has a private L1 (32KB) and L2 (256KB) caches and all cores on the same socket share a L3 cache (24MB).

Table 1 summarizes the hardware characteristics of these machines. NUMA factors[1] are shown in intervals, meaning the minimum and maximum penalties to access a remote DRAM in comparison to a local DRAM.

[1] Remote read latency divided by local read latency (obtained from BenchIT).

Table 1 Overview of the NUMA multi-core platforms.

Characteristic	NUMA16	NUMA32
Number of cores	16	32
Number of sockets	8	4
NUMA nodes	8	4
Clock (GHz)	2.22	2.27
Last level cache (MB)	1 (L2)	24 (L3)
DRAM capacity (GB)	32	64
Memory bandwidth (GB/s)	9.77	35.54
NUMA factor	[1.1; 1.5]	[1.36; 3.6]

Both machines run Linux (kernel 2.6.32) with GNU C Compiler 4.4.4. We selected TinySTM [13] as the STM library to carry out our experiments. Unlike other STM libraries, TinySTM can be configured with several locking and contention management strategies. We configured TinySTM with encounter-time locking, write-back memory update and a suicide contention strategy.

To evaluate the developed mechanism, we selected four applications from the STAMP benchmark suite [21] that matched the worklist model: *Intruder*, *Kmeans*, *Vacation* and *Yada*. In a previous work, these applications were ported to conform with the OpenSkel API and results showed that these applications present similar performance compared to the original ones. Finally, we executed these selected applications with the recommended input data sets. *Kmeans* and *Vacation* have two input data sets, high and low contention. As *Intruder* and *Yada* only have high contention input data sets, we chose the low contention inputs for *Kmeans* and *Vacation* to cover a wider range of behaviors. Table 2 summarizes the main characteristics of the selected applications.

Table 2 Summary of STAMP application runtime characteristics on TinySTM for the 32-core NUMA machine.

Application	Scalable up to # Cores	Transaction Abort Ratio	L3 Cache Miss Ratio	Transaction Length
Intruder	4	high	medium	short
Kmeans	8	high	high	medium
Vacation	16	low	low	short
Yada	16	high	medium	medium

6 EXPERIMENTAL RESULTS

In this section we evaluate the impact of the *SkelAff* mechanism on the selected benchmarks. For all applications, the input work-units were shuffled before their computation. This is done to avoid benefits from a particular input order. The baseline version of each application uses the default work sharing mechanism available in OpenSkel. All the results presented are based on an arithmetic mean of 10 runs.

In Section 6.1, we first evaluate the performance impact of individual page allocation policies and our skeleton-driven memory affinity mechanism. Such

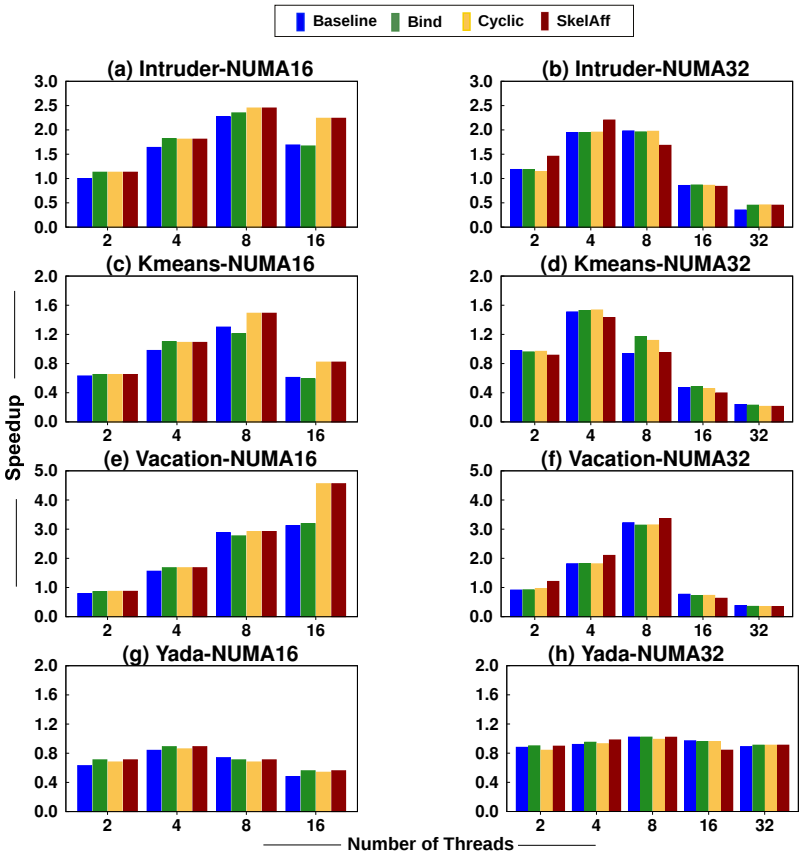

Fig. 8 Comparison between different page allocation policies (bind and cyclic) and our automatic skeleton-driven memory affinity (SkelAff) on NUMA16 and NUMA32 platforms.

performance impact is analyzed by executing the selected STAMP applications on both NUMA platforms. Then, in Section 6.2 we perform a deeper study on the benefits of using software HTs to prefetch data.

6.1 Evaluating Memory Affinity

Figure 8 reports the speedup of the selected benchmarks on both NUMA16 and the NUMA32 platforms, comparing the performance of the baseline version, individual page allocation policies (*bind* and *cyclic*) and our skeleton-driven memory affinity solution (*SkelAff*). There are three important details about this figure. Firstly, we varied the number of threads not only to observe the impact of the memory affinity mechanisms with different thread counts but also to see if those mechanisms can actually improve the scalability of the applications. Secondly, when HTs are enabled (*SkelAff*), there is one auxiliary

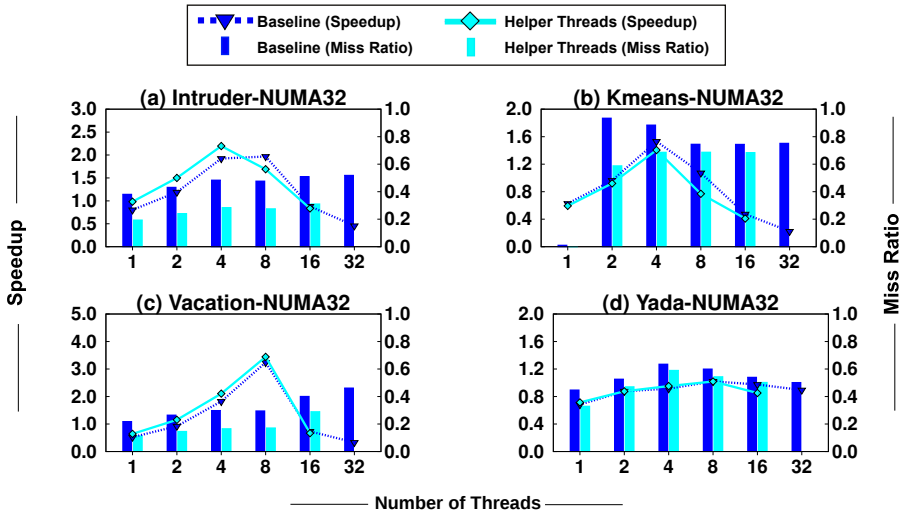

Fig. 9 Shared cache miss ratio between the baseline version and software HTs on NUMA32.

thread for each worker thread and both are placed on different cores as close as possible to profit from shared caches. For instance, the result with 4 threads in Figure 8b using our *SkelAff* mechanism represents the speedup obtained with 4 working threads plus 4 HTs (8 cores in total). This will be always the case of all applications running our *SkelAff* mechanism on the NUMA32 platform. Finally, the baseline version uses the Linux operating system policy called *first-touch*. This policy allocates memory preferably on the current node of the running thread [2].

Overall, memory affinity improves the performance of the applications and the average gains compared to the baseline are the following: *bind* (less than 1%), *cyclic* (8%) and our skeleton-driven memory affinity solution (11%). Our solution can surpass the gains of individual page allocation policies due to the use of software data prefetch (helper threads) when appropriate. However, the performance can also be degraded when the use of software data prefetch is not beneficial.

Individual page allocation polices presented more significant performance gains on NUMA16, resulting in improvements up to 46%. This was expected because such platform does not have shared cache memories, so the NUMA factor plays an important role on the overall performance. We also observed that our skeleton-driven memory affinity solution always selected the best page allocation policy to be applied automatically. However, our approach cannot benefit from using helper threads to prefetch data due to the absence of shared cache memories in this platform. Because of that, the performance of our solution does not surpass the performance gains of individual page allocation policies.

High memory contention benchmarks such as *Intruder* (Figure 8a), *Kmeans* (Figure 8b) and *Vacation* (Figure 8c) presented better performance gains with the *cyclic* page allocation policy on NUMA16. In particular, *cyclic* delivers significant performance improvements as we increase the number of threads because the contention on the worklist is also increased considerably. Such contention is then alleviated when we apply *cyclic*, since it makes more bandwidth available per core due to its distributed nature of placing memory pages. The performance improvement of our *SkelAff* mechanism over the baseline version for *Intruder*, *Kmeans* and *Vacation* were respectively 7%, 14% and 46%.

Although the NUMA32 presents a high NUMA factor between nodes, remote data requests rarely leads to accesses to remote memory banks. It stems from the fact that each node has a large shared L3 cache which is also interconnected to the others through high speed communication channels. Thus, instead of accessing a remote memory bank directly on a data request, a core may find the desired data on remote caches. On the other hand, the shared L3 caches allow the exploitation of helper threads to prefetch data, hopefully increasing the performance gains of individual allocation policies.

Overall, the performance improvements of individual allocation policies (*cyclic* and *bind*) are roughly the same on NUMA32. *Kmeans* (Figure 8d) and *Yada* (Figure 8h) were exceptions and presented better performance gains with *bind*. In some cases, our skeleton-driven memory affinity presented better performance gains than those individual allocation policies on *Intruder* (Figure 8b) and *Vacation* (Figure 8f). In such cases, these applications profited from the use of page allocation policies along with software data prefetching (helper threads). The performance gains for *Intruder*, *Vacation* and *Yada* were respectively 13%, 7% and less than 1%. In the next section we analyze in detail the benefits of using helper threads.

6.2 Analyzing the Benefits of Cache Prefetching

We evaluate the performance of helper threads by using two different metrics: *speedup* and *last level cache miss ratio*. The first one allows us to identify the overall performance gains whereas the second measures the effectiveness of prefetching data on shared caches. Experiments were only conducted on the NUMA32 platform since it has shared L3 caches. We used PAPI [15] interface to access hardware performance counters as well as to compute miss ratios.

Figure 9 compares the benefits of using helper threads with the baseline version. Overall, HTs improved the top performance of the STAMP applications up to 14%, a significant improvement in accordance with the results showed in [24], where the SPEC Benchmark applications achieved up to 22% performance gain by the use of software helper threads. It is important to note that the SPEC Benchmark applications present much more regular behavior, making it easier to predict which data can be prefetched.

As it can be noticed, by applying helper threads we reduced the last level cache miss ratio in all four applications. This confirms the effectiveness of

helper threads: triggering future cache miss events far enough in advance by the main thread reduces the memory miss latency. Although the miss ratio was decreased, such improvement did not significantly reflect well on the overall performance of all applications. In fact, we observe performance improvements compared to the baseline version from 2 to 4 worker threads on most of the applications. However, it did not deliver any performance gains with 16 worker threads (i.e., 16 worker threads + 16 HTs), when most applications stop scaling.

In Figure 9a, *Intruder* showed important performance gains when applying HTs with 2 and 4 worker threads (26% and 14%, respectively). However, even reducing the miss ratio with 8 and 16 threads, HTs did not present the expected performance improvement. The *Intruder* benchmark is characterized by having a high abort ratio which increases proportionally to the number of threads. In STM systems, concurrent transactions are squashed and re-executed on every conflict. Each time a conflict is detected, only one transaction keeps executing while the others involved in the conflict are restarted. When HTs are enabled, they are also restarted to guarantee the correct execution of the next work-unit ahead of time. Thus, this overhead of restarting HTs becomes too high with 8 and 16 worker threads, reducing the benefits of prefetching data and increasing the wasted work.

Kmeans spends most of its execution in non-transactional code as described in [21]. This prevented helper threads from improving the performance of the baseline version of *Kmeans* (Figure 9b). In particular, Kmeans is an iterative application that alternates between a sequential and a parallel phases, the latter implemented as a worklist. At the end of each phase, the worklist is empty and it is repopulated before starting the next phase by a single thread (sequential code). On each iteration, HTs have to be reinitialized in the parallel phase. This causes extra overhead which is significant since *Kmeans* has a short execution time per parallel phase. Thus, this overhead of creating auxiliary threads surpasses the benefits we can obtain from them.

Helper threads led to interesting performance improvements on *Vacation* (Figure 9c) with 2, 4 and 8 worker threads (25%, 15% and 6%, respectively). It stems from the fact that *Vacation* has short transactions. This allows actual and prefetched data to coexist in the shared cache without causing extra cache misses. Additionally, *Vacation* presents very low abort ratio avoiding the side-effect identified on *Intruder*. With 16 worker threads we observe that the miss ratio is not too much decreased in comparison with the baseline one. As a result, there is no significant difference between the overall performance obtained with HTs and the baseline version.

As opposed to *Vacation*, *Yada* has long transactions. This makes harder to attain fair timing between the worker and helper threads. In fact, it increases the probability that a helper thread takes a wrong path of execution due to premature execution. Even though, HTs reduced the cache miss ratio of *Yada*.

7 RELATED WORK

There are three main software-based approaches that are commonly employed to enhance memory affinity in parallel applications: thread scheduling, memory page allocation policies and cache prefetching. In this section we highlight some related work concerning these three techniques.

Thread scheduling. In [7], the authors proposed a machine learning-based approach to automatically infer a suitable thread mapping strategy for transactional memory applications. Some STM applications are profiled to build a set of input instances. Then, such data feeds a machine learning algorithm, which produces a decision tree able to predict the most suitable thread mapping strategy for new unobserved instances. In [12], two thread mapping algorithms are proposed for applications based on the shared memory programming model. These algorithms rely on memory traces extracted from benchmarks to find data sharing patterns between threads. In [17], the authors proposed a dynamic thread mapping strategy for regular data parallel applications implemented with OpenMP.

Memory page allocation policies. In [23], the authors explored different page allocation policies to improve the memory affinity of two geophysics parallel applications. They showed that these policies can achieve better performance gains than other solutions available on Linux. In [2], the authors designed dynamic mechanisms able to decide the data placement over the physical memory of a NUMA platform. The proposed mechanisms must use hardware counters to compute the most suitable data placement (e.g., queuing delays, on-chip latencies, and row-buffer hit-rates of on-chip memory controllers).

Software cache data prefetching. Helper threads have been used as a means of exploiting idle hardware resources to prefetch data [9]. In [24], the authors proposed a compiler framework to automatically generate software helper threads code for profitable loops in sequential applications. In [22], authors manually coded helper threads within hardware transactional memory barriers to improve the performance of a sequential implementation of Dijkstra's algorithm.

While thread scheduling techniques have been already studied in transactional memory applications, page allocation policies and software cache prefetching are still a daunting task, deserving a more detailed analysis. To the best of our knowledge, this is the first work that explores both techniques (page allocation policies and helper threads) in an automatic fashion for transactional memory applications. This was possible thanks to the OpenSkel framework, which allowed us the implementation of such skeleton-driven mechanism.

8 CONCLUSIONS

In this paper, we presented a new automatic memory affinity mechanism for transactional worklist applications. It employs an algorithmic skeleton approach to improve affinity in the main memory and cache levels. This approach provides compile and runtime information about the application behavior. This enabled *SkelAff* to automatically employ page allocation policies and helper threads to enhance memory affinity.

We performed experiments on two representative NUMA machines using the STAMP benchmark. Our results showed that *SkelAff* yield performance improvements of up to 46%, with an average of 11%, over a baseline version. We can also conclude that tackling affinity in different levels of memory is essential to improve performance of transactional applications on NUMA platforms.

As future work, we intend to implement an autotuning mechanism to dynamically adjust the helper threads parameters such as the lifespan and the buffer size. This can potentially lead to a more profitable data prefetching, avoiding that wrong data is brought into the cache memory. Furthermore, we aim at investigating and including a bigger range of page allocation policies in our mechanism.

References

1. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: A View of the Parallel Computing Landscape. Communications of the ACM **52**(10), 56–67 (2009)
2. Awasthi, M., Nellans, D.W., Sudan, K., Balasubramonian, R., Davis, A.: Handling the Problems and Opportunities Posed by Multiple on-Chip Memory Controllers. In: PACT, pp. 319–330. ACM (2010). DOI http://doi.acm.org/10.1145/1854273.1854314
3. Baek, W., Minh, C.C., Trautmann, M., Kozyrakis, C., Olukotun, K.: The OpenTM Transactional Application Programming Interface. In: PACT 2007, pp. 376 –387. IEEE Computer Society (2007)
4. Broquedis, F., Aumage, O., Goglin, B., Thibault, S., Wacrenier, P.A., Namyst, R.: Structuring the Execution of OpenMP Applications for Multicore Architectures. In: IPDPS, pp. 1–10. IEEE Computer Society (2010)
5. Broquedis, F., Clet Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In: PDP, pp. 180–186. IEEE Computer Society (2010)
6. Castro, M.B., Góes, L.F.W., Fernandes, L.G., Méhaut, J.F.: Dynamic thread mapping based on machine learning for transactional memory applications. In: Euro-Par, pp. 465–476 (2012)
7. Castro, M.B., Góes, L.F.W., Ribeiro, C.P., Cole, M., Cintra, M., Méhaut, J.F.: A machine learning-based approach for thread mapping on transactional memory applications. In: HiPC, pp. 1–10 (2011)
8. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press & Pitman (1989)
9. Collins, J.D., Wang, H., Tullsen, D.M., Hughes, C., Lee, Y.F., Lavery, D., Shen, J.P.: Speculative Precomputation: Long-Range Prefetching of Delinquent Loads. In: ISCA, pp. 14–25. ACM (2001)
10. Dalessandro, L., Dice, D., Scott, M., Shavit, N., Spear, M.: Transactional Mutex Locks. In: Euro-Par, pp. 2–13. Springer-Verlag (2010)
11. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: OSDI, pp. 137–150. USENIX Association (2004)

12. Diener, M., Madruga, F., Rodrigues, E., Alves, M., Schneider, J., Navaux, P., Heiss, H.U.: Evaluating Thread Placement Based on Memory Access Patterns for Multi-core Processors. In: HPCC, pp. 491–496. IEEE Computer Society (2010)
13. Felber, P., Fetzer, C., Riegel, T.: Dynamic Performance Tuning of Word-Based Software Transactional Memory. In: PPoPP, pp. 237–246. ACM (2008). DOI 10.1145/1345206.1345241
14. Felber, P., Fetzer, C., Riegel, T., Sturzrehm, H.: Transactifying Applications Using an Open Compiler Framework. In: TRANSACT. ACM (2007)
15. Garner, B.D., Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A Portable Programming Interface for Performance Evaluation on Modern Processors. International Journal of High Performance Computing Applications **14**, 189–204 (2000)
16. Goes, L.F.W., Ioannou, N., Xekalakis, P., Cole, M., Cintra, M.: Autotuning skeleton-driven optimizations for transactional worklist applications. IEEE Transactions on Parallel and Distributed Systems **23**(12), 2205–2218 (2012)
17. Hong, S., Narayanan, S.H.K., Kandemir, M., Özturk, O.: Process Variation Aware Thread Mapping for Chip Multiprocessors. In: DATE, pp. 821–826. European Design and Automation Association (2009)
18. Kleen, A.: A NUMA API for Linux. Tech. Rep. Novell-4621437 (2005)
19. Larus, J., Rajwar, R.: Transactional Memory. Morgan & Claypool Publishers (2006)
20. McCool, M.: Structured Parallel Programming with Deterministic Patterns. In: HotPar, pp. 25–30. USENIX Association (2010)
21. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford Transactional Applications for Multi-Processing. In: IISWC, pp. 35–46. IEEE Computer Society
22. Nikas, K., Anastopoulos, N., Goumas, G., Koziris, N.: Employing Transactional Memory and Helper Threads to Speedup Dijkstra's Algorithm. In: ICPP, pp. 388–395. IEEE Computer Society (2009)
23. Pousa Ribeiro, C., Castro, M., Carissimi, A., Méhaut, J.F.: Improving Memory Affinity of Geophysics Applications on NUMA platforms Using Minas. In: VECPAR. Springer-Verlag (2010)
24. Song, Y., Kalogeropulos, S., Tirumalai, P.: Design and Implementation of a Compiler Framework for Helper Threading on Multicore Processors. In: PACT, pp. 99–109. IEEE Computer Society (2005)
25. Wanderley Góes, L.F.: Automatic skeleton-driven performance optimizations for transactional memory. Ph.D. thesis, School of Informatics, University of Edinburgh, UK (2012)