Model-Based, Memory-Centric Performance and Power Optimization on NUMA Multiprocessors

ChunYi Su[†] Dong Li[‡] Dimitrios S. Nikolopoulos[§]
Kirk W. Cameron[†] Bronis R. de Supinski[♮] Edgar A. León[♮]

Department of Computer Science[†]
Virginia Tech, VA 24060
{sonicat,cameron}@vt.edu

Oak Ridge [‡]
National Laboratory
Oak Ridge, TN 37831
lid1@ornl.gov

Lawrence Livermore[♮]
National Laboratory
Livermore, CA 94550
{bronis,leon}@llnl.gov

Queen's University[§]
of Belfast
Belfast,Northern Ireland, UK
d.nikolopoulos@qub.ac.uk

***Abstract*—Non-Uniform Memory Access (NUMA) architectures are ubiquitous in HPC systems. NUMA along with other factors including socket layout, data placement, and memory contention significantly increase the search space to find an optimal mapping of applications to NUMA systems. This search space may be intractable for online optimization and challenging for efficient offline search. This paper presents DyNUMA, a framework for dynamic optimization of programs on NUMA architectures. DyNUMA uses simple, memory-centric, performance and energy models with non-linear terms to capture the complex and interacting effects of system layout, program concurrency, data placement, and memory controller contention. DyNUMA leverages an artificial neural network (ANN) with input, output, and intermediate layers that emulate program threads, memory controllers, processor cores, and their interactions. Using an ANN in conjunction with critical path analysis, DyNUMA autonomously optimizes programs for performance or energy-efficiency metrics. We used DyNUMA on a variety of benchmarks from the NPB and ASC Sequoia suites on three different architectures (a 16-core AMD Barcelona system, a 32-core AMD Magny-Cours system, and a 64-core Tilera TilePro64 system). Our results show that DyNUMA achieves on average 8.7% improvement in performance (12.9% in the best case), 16% improvement in Energy-Delay (30.6% in the best case) and 9.1% improvement in MFLOPS/Watt (10.7% in the best case) compared to the default Linux scheduling.**

I. Introduction

Non-Uniform Memory Access (NUMA) is now the dominant memory system architecture for multiprocessors. NUMA has been the leading design paradigm in scalable, cache-coherent, multi-processor architectures since the 1990s. On a typical NUMA system, each processor has a local memory node accessible over dedicated links, while remote memory nodes are accessible via an interconnect and through network interfaces. The latency of accessing the local memory node is markedly lower than the latency of accessing a remote memory node. More recently, non-uniform memory access latency is also present between cores in the same socket. The processor uses multiple memory controllers on-chip to serve its cores, with each controller connected to a different memory node. NUMA is therefore becoming pronounced also within the boundaries of a single chip. For example, the Tilera TilePro64 processor has four memory nodes on the same die [1]. It implements a shared physical address space via a mesh interconnect between cores. When a core accesses the closest memory node, it incurs lower access latency than when accessing other memory nodes. Similar asymmetric access latencies also appear in the NVIDIA Fermi architecture [2].

NUMA improves system scalability by avoiding bottlenecks in the memory subsystem and by increasing the memory bandwidth available per core. With an increasing number of cores per processor, NUMA is becoming necessary for systems to scale. According to Top500 statistics, over 90% of Top500 supercomputers are based on NUMA nodes [3].

Optimizing applications for performance and energy efficiency on a NUMA architecture has been and remains challenging. While a significant body of prior work has treated non-uniform memory access as one of data distribution and migration, assuming a stationary mapping of threads to cores [4], [5], [6] [7] [8], we consider the problem from the opposite direction: given a distribution of data among memory nodes, what is the optimal mapping of threads to cores? As remapping of threads to cores is orders of magnitude faster than remapping data to memories, such an approach is worth considering as a dynamic optimization strategy.

Application performance is highly sensitive to thread-to-core mapping. Figure 1 shows an example that quantifies performance variance due to different thread-to-core mappings on a NUMA system. We use SP from the NAS Parallel Benchmarks (class A, OpenMP version), running with 8 threads on a single node with 4 AMD quad-core processors. We enumerate 85 different mappings for 9 parallel regions in the benchmark. We observe a performance difference between the best and the worst mapping up to 45%. Compared to the default system mapping (Linux 2.6.32), the best mapping is 18% faster. Therefore, to optimize the performance and energy efficiency of applications on NUMA systems, we must determine the best mapping. However, the search space to determine the best mapping can be very large.

For the 16-core NUMA architecture shown in Figure 2, a system similar to the smallest system that we use in our experiments, there are over 63 million possible mappings of threads to cores, each with different memory access latency and bandwidth available per core. The above calculation excludes the impact of shared caches and assumes statically placed data. If we consider these implications, the search space is even larger.

In addition to the challenges of making optimal static mapping of threads to cores, previous techniques to optimize power and performance dynamically on Unified Memory Access (UMA) systems does not necessarily extend to NUMA systems. Earlier work [9], [10] shows that dynamic concurrency throttling (DCT) is a viable optimization technique for performance and energy efficiency. DCT amounts to modifying (throttling) the number of threads and the mapping of threads to cores used by parallel code at runtime, to avoid oversubscribing hardware resources, such as shared memory bandwidth. DCT is beneficial also when the degree of available algorithmic parallelism in a code region is less than the maximum number of cores available on the hardware. On a NUMA system, any attempt to throttle con-

978-1-4673-4532-3/12/$31.00 ©2012 IEEE

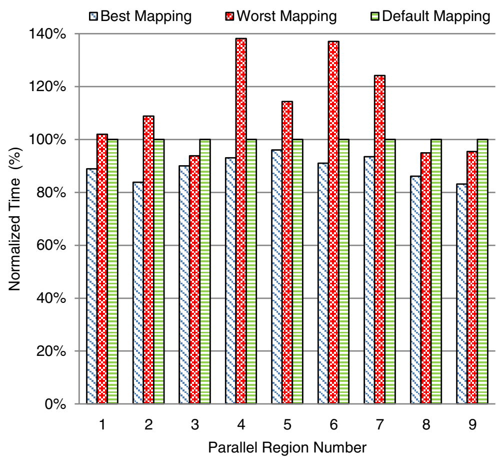

Figure 1: The performance variance of 85 different thread mappings in SP.A benchmark. The performance is normalized to the performance with the Linux default mapping.

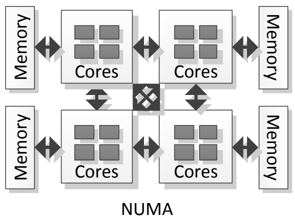

Figure 2: A 16-core NUMA architecture with 4 memory nodes

currency after execution begins will redistribute computation between cores, thereby forcing extraneous cache misses, remote memory accesses, and contention. Prior work on dynamic concurrency throttling overlooks this problem. In fact, any attempt to migrate threads or data in the operating system for the purposes of throughput, power optimization, or reliability, suffers from the same problem.

This paper considers a three-dimensional optimization problem for NUMA systems: (i) finding an optimal degree of concurrency, (ii) mapping threads to cores to reduce remote accesses per core, and (iii) minimizing contention on memory controllers. An optimal degree of concurrency avoids performance loss due to synchronization overhead, contention, or lack of sufficient algorithmic concurrency in the program. Reducing remote memory accesses reduces memory latency but may create contention due to oversubscribing of memory controllers.

Any solution to the optimization problem needs to identify the enumeration and layout of cores with respect to memory controllers and memory nodes (a non-trivial exercise) and also needs to consider phase behavior in programs such as changes in concurrency, memory access patterns or data communication and synchronization patterns [9]. Unfortunately, standard linear regression cannot capture the complexities of such systems. Non-linear regression models (or logistic regression) are often very complicated in formulation and can require substantial computation resources to solve.

To address these challenges, we created DyNUMA, a framework for dynamic optimization of programs on NUMA architectures. DyNUMA is implemented in the runtime system to improve both performance and energy efficiency. The core of DyNUMA is a novel memory-centric performance model. The model captures the non-linear and interacting effects of concurrency, thread mapping, and data placement using an Artificial Neural Network (ANN). ANN's are simpler to implement than logistical regression techniques requiring less formal statistical training. Furthermore, ANN's excel at deriving structure from data samples.

DyNUMA uses an ANN model in conjunction with critical path analysis [11] to predict optimal concurrency and thread mapping, assuming static data placement. Integration of DyNUMA with dynamic data redistribution (migration) is out of the scope of this work.

This paper makes the following contributions:

- A novel memory-centric, non-linear performance model for NUMA architectures which captures the effects of concurrency, data placement, and memory contention on system performance. The model accurately predicts non-linear performance metrics such as the energy-delay product (EDP).
- A flexible and portable framework, DyNUMA, to address the multi-dimensional problem of concurrency control and thread-to-core mapping on NUMA systems. This framework is portable and can be used on a variety of NUMA architectures. It is also flexible allowing the use of different program optimization metrics including energy efficiency.
- A runtime system which implements online and autonomous optimization of NUMA programs using the aforementioned model.

The rest of this paper is organized as follows. In Section II, we review the related work. Section III describes our system design and is followed by our performance evaluation in Section IV. Our conclusions and future work are presented in Section V.

II. Related work

OpenMP performance models: Curtis-Maury et al. [9] and Li et al. [10], [12] map threads to cores using a linear regression model of performance and power. An online predictor enables dynamic optimization based upon measured hardware counter events at runtime. While these techniques are useful, the authors noted limitations in thread placement and model accuracy as the cores and memory scale in number and complexity. Our ANN model is designed to address the accuracy limitations of applying linear models to the non-linear problem of mapping threads to cores to optimize for power and performance as systems and applications scale.

In other work, Singh et al. [13] and Curtis-Maury et al. [14] used ANN's to predict performance and energy efficiency on multicore systems. These models focus on the use of cache miss rates from hardware counters in their predictions. As such, these techniques also suffer inaccuracies on NUMA systems since they ignore significant effects including thread-to-data affinity, non-uniform data access latency, and core bandwidth. In our work, as noted, we include the effects of thread-to-data affinity as well as those of latency and bandwidth. Through collective consideration of these characteristics, we capture thread locality and the full costs of thread affinity in NUMA environments.

Thread-data affinity on NUMA: Terboven et al. [15] propose a NUMA memory placement policy called *next touch*, to migrate pages that are frequently accessed remotely. Ribeiro et al. [16] use data access patterns to guide memory placement on NUMA systems. Nikolopoulos et al. [5], [6] propose a series of user-level dynamic page migration approaches. DyNUMA differs from the above

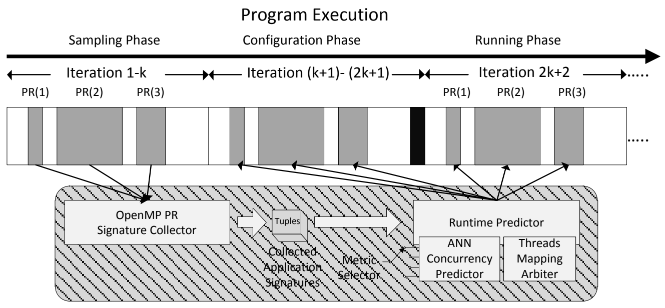

Figure 3: Diagram of the DyNUMA system framework. PR means the parallel region.

approaches, in that it does not perform dynamic data placement or data migration. DyNUMA only migrates threads to improve thread-to-data affinity. Thread migrations are orders of magnitude faster than data migrations that can incur severe performance penalties due to TLB misses and data transfers.

Broquedis et al. [17] introduce a runtime system to optimize thread-to-data affinity by using a *BubbleScheduler* scheme. *BubbleScheduler* remaps threads using a capacity metric to identify memory nodes with the largest concentration of thread data. Threads are then migrated remotely to the identified node to maximize data reuse and minimize data transfer costs. Though this approach considers affinity, the focus is on modeling and optimizing for data movement with threads tightly coupled to data. Despite the focus on minimizing data movement, as threads and cores scale, the need to migrate and the amount of data to migrate increase substantially. DyNUMA addresses this limitation by focusing on optimizing thread affinity without the associated coupling to data movement.

III. System Design

DyNUMA optimizes OpenMP programs where parallelism is expressed with directives that delineate parallel regions. Each parallel region may enclose parallel loops, tasks, or nested regions. The design objective of DyNUMA is to select the best level of concurrency for each OpenMP parallel region and optimize thread placement to cores based on data locality so that the program is optimized for a given performance or energy-efficiency metric. The design of DyNUMA is based on the following characteristics:

- Scalable: system is expected to execute on architectures with massive parallelism.
- Architecture-aware: system should capture key architectural factors that affect performance and power.
- Light-weight: system should incur low overhead to allow for online dynamic optimization.
- Portable: system should be parameterized to allow for ease of porting to different NUMA architectures.

A. Overview

DyNUMA implements a dynamic online predictor for the degree of concurrency and the thread-to-core mapping of each parallel region. The framework is illustrated in Figure 3. The runtime predictor of DyNUMA includes two components. The first component is an architecture-aware, *Artificial Neural Network Predictor* (ANN) which predicts the degree of concurrency. The second component is a *Thread Mapping Arbiter* (TMA) which implements a deterministic algorithm that determines the thread-to-core mapping in linear time. DyNUMA assumes iterative programs where parallel regions are executed a number of time steps. This is common for many HPC applications. In the sampling phase, DyNUMA initially executes a program with maximal concurrency –using as many threads as the number of cores– for first k iterations. The number of k is equal to the number of memory nodes. The ith iteration samples threads' *execution signatures* on the memory node i. The choice of k is determined by a limitation of current hardware counters, that is, hardware counters can only profile one memory node at a time. Overcoming this limitations can significantly reduce k. DyNUMA samples all execution signatures during these k iterations to derive predictions of the best concurrency and thread mapping of each parallel region using ANN. Afterwards, DyNUMA applies TMA to further improve data locality.

We define an execution signature as a collection of three metrics:

- IPC: Instructions per Cycle
- LMA: Local Memory Accesses per Cycle
- RMA: Remote Memory Accesses per Cycle

The runtime system collects the execution signature of each thread and transforms into a 3-element tuple. Each tuple characterizes a thread with respect to the intensity of computation to memory operations while executing a parallel region. LMA and RMA values are determined by the location of a thread. DyNUMA maintains LMA and RMA per memory node for each thread. DyNUMA uses thread-level tuples coupled with thread mapping information and observed metrics as inputs to the two DyNUMA predictors – ANN and TMA. IPC, LMA and RMA from all threads are used in the ANN to navigate the search space and predict performance on all degrees of concurrency. If an application is processor-bound, IPC should be high while LMA and RMA should be low. In this case, the ANN tends to select higher concurrency. Conversely, a memory-bound application is expected to have low IPC and high LMA and RMA values, in which case the ANN tends to select lower concurrency to avoid oversubscribing the memory system. The optimal degree of concurrency can vary across regions due to variance of execution signature. On the other hand, TMA makes use of LMA, RMA and thread mapping information to redistribute threads in a more balanced way. Following prediction, DyNUMA actuates the selected concurrency and thread mapping for the remaining time of program execution.

B. Metric Selection

DyNUMA predicts performance and energy efficiency using the metrics shown in Table I. MFLOPS/Watt is the metric used to evaluate system energy efficiency in the Green500 list [18], while EDP is a common energy-efficiency metric in HPC environments because it is implementation-neutral. The EDP and MFLOPS/Watt are calculated by Equations 1 and 2 respectively. The system provides the end user with flexibility to define different metrics while using the same unified prediction infrastructure explained in Section III.C.

Table I: Three metrics used for the prediction of performance and energy efficiency.

Wall-clock time	Wall clock time of a parallel region
EDP	Energy-Delay-Product of a parallel region
MFLOPS/Watt	Number of floating point instructions (in millions) per second per Watt of a parallel region

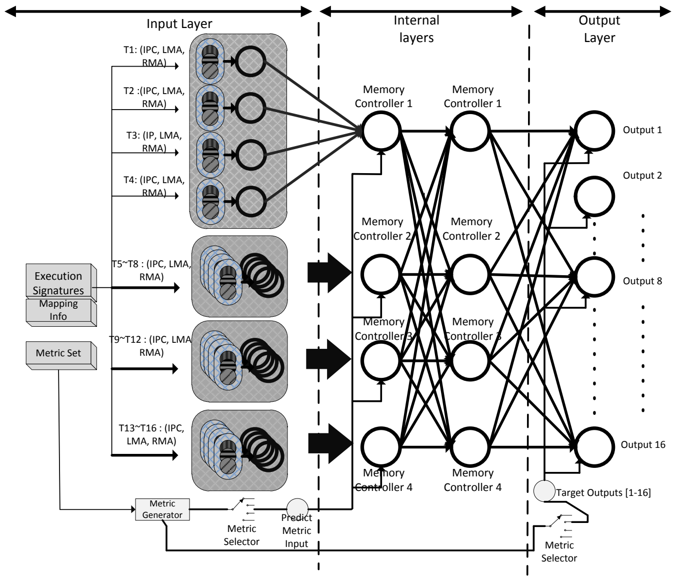

Figure 4: The ANN model for four quad-core processors (16 cores in total) and 4 NUMA memory nodes

$$EDP = Power * (wall\ clock\ time)^2 \tag{1}$$

$$MFLOPS/Watt = \frac{Number\ of\ floating\ point\ instructions}{10^6 * Time * Power} \tag{2}$$

C. Architecture-Aware Artificial Neural Network Predictor

One of DyNUMA's design goals is to be easily portable across platforms with different architectures. This is achieved by using portable metrics in the DyNUMA model of performance, namely IPC, LMA and RMA. The DyNUMA predictor uses a configurable, back-propagation, artificial neural network model [19] which can be ported by changing two parameters: the number of cores and the number of NUMA memory nodes of the target machine.

ANN is an adaptive system that learns its coefficients using training sets fed through the network during a learning phase. Figure 4 shows an example of the configurable ANN model. The topology of the ANN model in this example emulates a node with 4 quad-core processors and 4 NUMA memory nodes. The topology of the ANN model emulates the target architecture. The ANN includes three layers: input, internal and output. The cells in the input layer correspond to cores and receive as input the execution signature of each thread. The cells in two internal layers emulate the controllers of NUMA memory nodes. The links between two internal layers emulate communication among memory nodes. For example, the link between the memory controller 1 and the memory controller 3 emulates data transfers between cores attached to the memory node 1 and cores attached to the memory node 3. The ANN can have multiple outputs. Each output represents the predicted metric at a different degree of concurrency. Output i is the predicted value when running the examined code region with i threads. In the current implementation, the ANN model predicts the three metrics listed in Table I.

The ANN model can be reconfigured for different systems by changing the number of cells in the input and internal layers to correspond to different numbers of cores and memory nodes. The topology of the ANN reflects the system

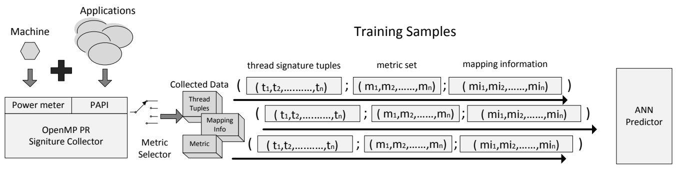

Figure 5: Model training

interconnect topology. It is not fully connected since each core is associated with one NUMA memory node and not all cores directly access all memory nodes. This ANN model can be easily adapted to handle SMT architectures (multiple hardware threads per core) by using the execution signature of each hardware thread as input. There are several advantages of using ANN. First, it can easily capture the hardware architecture by changing its internal layers and topology. Second, it can generate multiple predictions under different levels of concurrency in parallel, contrary to prior linear DCT models that require a different model to predict each level of concurrency [9]. Third, ANN is a non-linear statistical modeling tool that captures complex relationships between inputs (execution signatures) and outputs (performance and power efficiency metrics). Such relationships cannot be easily captured by other models. We demonstrate this advantage by comparing the ANN model to a state-of-the-art linear regression model proposed by Curtis-Maury et al. [14].

1) Data collection: The ANN model in DyNUMA is trained offline. Figure 5 shows the data collection framework for offline training. The *OpenMP PR Signature Collector* uses a set of APIs for application instrumentation. The instrumentation enables the collection of signatures of parallel regions, thread mapping information and metrics targeted for optimization. The signature collector uses PAPI [20], Oprofile [21] and WattsUp [22] power meters. The collected data is transformed into training samples. A training sample consists of: (1) a set of metric values (wall-clock time, EDP or MFLOPS/Watt), (2) a set of thread signature tuples, and (3) thread mapping information. LMA and RMA are hardware events and are collected using architecture-specific counters. The thread mapping data is collected with the portable POSIX *sched_getcpu()* interface.

2) Power Measurement: To compute energy efficiency metrics (EDP and MFLOPS/Watt), DyNUMA collects power consumption of each parallel region. The runtime system uses an API to connect to external WattsUp power meters and record power for each region. The dynamic power of the two components varies as DyNUMA changes the number of active threads, memory access rate, and access pattern per thread. There are other hardware components that might exhibit dynamic power variance under DyNUMA; however, their power variance is expected to be relatively small, comparing to the processors and main memory [23].

We use Equation 3 to compute the power variance of processors and memory:

$$Power = Power_{exec} - Power_{system\_idle} \tag{3}$$

$Power_{exec}$ and $Power_{system\_idle}$ are collected from the WattsUp power meter. Because we are unable to physically access the TilePro64 machine that we use in our experimental analysis, power consumption of the TilePro64 processor is obtained from the TilePro64 technical specification, assuming that processor power scales linearly from idle (17 Watt) to maximum (23 Watt), with the number of cores.

D. Thread Mapping Arbiter

TMA uses an algorithm based on the critical path analysis to identify the optimal thread mapping. In most cases, programmers want to distribute workload (computation) evenly in their parallel execution. However, the execution time from one thread to another may still vary. This is because of different memory access patterns across threads and uneven distribution of data across memory nodes. The thread with the longest execution time in any given parallel region is said to be on the critical path. Note that TMA cannot be combined with ANN and has to be applied after ANN, because the critical path analysis can only be performed after the thread concurrency is determined.

We use Figure 6 to further explain the critical path problem. Figure 6 displays remote and local memory accesses per socket collected from the first OpenMP parallel region in the NAS FT benchmark (class B). The test was deployed on a platform with four quad-core processors (16 cores total), each with one memory node. We used 8 threads to run this parallel region and all threads are evenly distributed to 4 sockets (i.e., 2 threads per socket). We traced LMA and RMA per socket for 40 iterations. From the figure, we observe that each socket has different RMA and LMA. Socket 1 attains the lowest RMA and the highest LMA. We further mapped threads to cores in different ways, but a similar distribution of memory accesses was observed. The difference in the number of memory accesses results in asymmetric execution time between threads and causes the critical path problem.

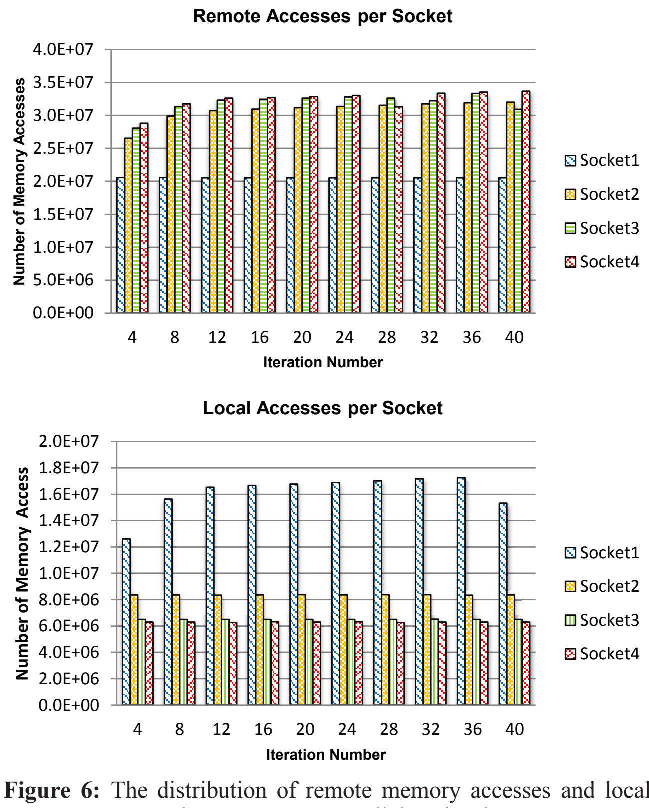

Figure 6: The distribution of remote memory accesses and local memory accesses in an OpenMP parallel region in FT.B

We present an algorithm that attempts to reduce the critical path by modifying thread placement, hence the ratio of local to remote memory accesses from each thread. The algorithm attempts to evenly distribute accesses between memory nodes, reduce remote memory accesses, and avoid contention on any memory node. The pseudo-code is shown in Algorithm 1.

Algorithm 1 Thread Mapping Arbiter Algorithm

```
Input: TNT $t$
Output: Map $mapMinCp$
 1: Map $mapMinCp = \Phi$
 2: Int cpImpact[Nd]= 0;  //The critical path impact on $Nd$
    //memory nodes
 3: ElementList sl= SortElementInTable($t$);
 4: while $sizeof(mapMinCp) \neq \#threads$ do
 5:     Element $e(T_i, D_j)$=GetMinCritcalPathElement(sl);
 6:     $mapMinCp$.Add($e(T_i, D_j)$);
 7: end while
 8: Return $mapMinCp$

 9: GetMinCritcalPathElement($ElementList\ x$)
Input: ElementList $x$  // A list of the candidate elements
Output: Element $e_{decide}$  // An element with the smallest
    //impact to the critical path
10: Element $e_{max}$ = GetFirstMaxElement($x$);
11: ElementList lc=FindAllPossibleCandidateElements($e_{max}$);
12: $e_{decide}$=FindLowestCPElement(cpImpact,lc);
13: RemoveThreadFromList($x$,$e_{decide}.T_i$);
14: AppendCriticalPathImpact($cpImpact$,IF($e_{decide}$));
15: Return $e_{decide}$

16: FindLowestCPElement(UINT64 $cpImpact[], List\ x$)
Input: ElementList $x$  //A sorted element list
Output: Element $e_{decide}$  //An element with the minimal
    //impact to the critical path
17: $minVal$=$UINT64\_MAX$; element $e_{decide}$=$\Phi$;
18: for all Element e in x do
19:     if $(IF(e) + cpImpact[e.D_j]) < minVal$ then
20:         $minVal$=$IF(e) + cpImpact[e.D_j]$);$e_{decide} = e$;
21:     end if
22: end for
23: Return $e_{decide}$
```

The input to the algorithm is a thread to node mapping table (TNT). The output is the predicted best thread mapping ($mapMinCp$). The TNT is a data structure collects the number of memory accesses from each thread to each memory node, derived from the execution signature of the program collected during sample iterations. An example of a TNT is shown in Table II. This TNT records the number of memory accesses from four threads to four memory nodes. Each element ($e(T_i, D_j)$), corresponds to the number of memory accesses to memory node j (i.e., D_j) from thread i (i.e., T_i). The algorithm first sorts all elements in the TNT in descending order of number of memory accesses (line 3 of Algorithm 1). This sorting step facilitates quick thread mapping in later steps of the algorithm. In the implementation, we use parallel radix sort to reduce sorting complexity. The sorting result is saved in a list (sl).

Following the sorting, the algorithm iteratively selects an element from sl and places the selected element, $e(T_i, D_j)$, in $mapMinCp$ (line 6) until all threads are selected. The selected element represents a decision of placing thread T_i on memory node D_j.

The selection criterion is implemented in **GetMinCriticalPathElement** (line 9). Generally speaking, this function chooses an element whose corresponding thread placement introduces the minimum imbalance of memory accesses between memory nodes. The function initially selects the first element from the sorted list (line 10), and then considers elements in other memory nodes (line 11) whose number of memory accesses are close (within 75% in our cases) to that of the first element in the input sorted list. The reason why the algorithm considers multiple candidates instead of choosing the first element is that the first candidate from the list may not necessarily avoid imbalance of memory accesses between memory nodes. In particular, the first candidate may have a significant imbalance between LMA and RMA which creates unbalanced memory accesses across memory nodes.

To estimate how placing a thread i on memory node j affects the critical path, we define a metric *Impact Factor*, IF, as:

$$IF(T_i,\ D_j) = LMA_{i,j} + \sum_{k=1,k\neq j}^{N} NUMA\_Factor_{i,k} \cdot RMA_{i,k} \quad (4)$$

The equation weighs the number of remote memory accesses by a *NUMA_Factor* because a remote access has longer latency than a local access. The $NUMA\_Factor$ is the ratio of the remote memory access latency to the local memory access latency. The $NUMA\_Factor$ is a variable. Depending on the distance between the core that issues a memory access upon a cache miss and the memory node where the miss is served, the $NUMA\_Factor$ can have different values. The $NUMA\_Factor$ can be calculated by measuring average access time when running a microbenchmark to vary data location between memory nodes. Based on the above equation, an element with a small IF means that this element introduce lowest-unbalanced memory accesses between memory nodes while avoiding remote memory accesses. We also define a counter ($cpImpact$) associated with each memory node that accumulates the IF value for each memory node whenever a thread mapping is determined (line 14). The counter helps us trace the distribution of memory accesses across memory nodes.

FindLowestCPElement (line 12) selects the best candidate. For all candidate elements (line 18), the algorithm first calculates $IF(e) + cpImpact[e.D_j]$, which estimates the impact of the memory accesses of a specific thread to memory node D_j on the critical path. The algorithm selects the element with the minimal value (lines 19 and 20) to minimize memory load imbalance between nodes while avoidng remote memory accesses.

We use an example to further illustrate the algorithm. We assume a system with four threads and four memory nodes, with a TNT as shown in Table II. After applying the algorithm, elements $e(3,2), e(2,3), e(4,1)$, and $e(1,4)$ are considered, which means that threads 3, 2, 4, and 1 are placed on cores close to memory nodes 2, 3, 1, and 4 respectively. We use a specific case to explain the process of choosing the best mapping candidate. In the second iteration of the selection loop (line 4), the algorithm first selects $e(4,1)$ from the sorted list. The algorithm selects this element, because it wants to first handle the element with the highest number of memory accesses. The selection of this element is the key to improve performance and should take the most favorable mapping when possible. However, $e(4,1)$ is not necessarily the best choice because it does not have the lowest IF on the critical path. Hence the algorithm consider other candidates (i.e., $e(4,2)$ and $e(2,3)$). Their number of memory accesses are close to $e(4,1)$. The algorithm then calculates the IF values of the three candidates and checks the $cpImpact[j]$ on each memory node (shown in Table III). The algorithm eventually selects $e(2,3)$ instead of $e(4,1)$ because its $IF + cpImpact[j]$ is the lowest among the three candidates, which intuitively introduces the smallest imbalance between the four memory nodes.

Table II: A TNT for 4 threads whose data is distributed into 4 memory nodes

Thread Id	Mem Node1	Mem Node2	Mem Node3	Mem Node4
1	100	1000	0	2000
2	1300	200	3500	1300
3	220	5000	500	500
4	4500	3800	2000	1000

Table III: An example to show how we choose the best element

element	$IFvalue$	$cpImpact$	$IF + cpImpact$
e(4,1)	IF(e(4,1))=14700	cpImpact[1]=0	14700
e(4,2)	IF(e(4,2))=15050	cpImpact[2]=6830	21880
e(2,3)	IF(e(2,3))=7700	cpImpact[3]=0	7700

E. Overhead and Penalty Control

DyNUMA changes concurrency and thread mapping between parallel code regions. Frequent changes in concurrency may incur performance loss due to cache flushing.

To ameliorate this effect, the runtime system considers remapping threads only for parallel code regions with sequential execution times of 100 milliseconds or higher. In addition to cache flushing, non-optimal concurrency prediction or non-optimal prediction of thread mapping can cause performance loss. DyNUMA uses an additional iteration to measure performance of the selected configuration and compares it with the performance of the system default. If the system default is better, the predicted configuration is discarded, and the system default is taken.

IV. Performance Evaluation

Experimental analysis explores two aspects of DyNUMA: prediction accuracy of the ANN model and effectiveness of model-based optimization.

We use two benchmark suites, the NAS parallel benchmarks (3.1) [24] and the ASCI Sequoia benchmark suite [25]. The benchmarks have 85 OpenMP parallel regions in total. Their workload ranges from compute-intensive to memory-intensive and most benchmarks exhibit phase changes in their memory access patterns. We use the Class D data set for all NAS benchmarks and use two of the Sequoia AMG benchmarks, AMG.Relax and AMG.Matvec.

The number of sample iterations k (see Section III.A) is 4 in our tests. When presenting the results, we use the notation $benchmark\_suite\_name.benchmark\_name.region\_no$ to represent a specific OpenMP parallel region. For example, NPB.FT.1 refers to the first parallel region in the benchmark FT in the NAS benchmark suite.

We present experiments from three platforms listed in Table IV to verify the portability of DyNUMA. We use

Intel's C and Fortran compilers (version 12.0.2) on AMD platforms. On TilePro64, we use the Tilera GCC and Fortran compiler (version 3.0.1) to perform cross compilation on an X86-64 platform.

Table IV: Three test platforms

Processor	#Cores	Speed	Memory Nodes	Memory
Barcelona	16	2.0 GHz	4	64GB
Magny-Cours	32	2.5 GHz	4	128GB
TilePro64	64	866 MHZ	4	64GB

We execute OpenMP benchmarks with static loop scheduling, which is the most appropriate for the selected benchmarks. Nevertheless, DyNUMA is independent of scheduling policy and can be applied as is once an initial distribution of workload between threads is performed by the scheduler. We execute benchmarks using *first-touch* for data placement in memories. First-touch is a page-level placement policy that allocates each page in memory located as close as possible to the processor that first touches the page during program execution. First-touch is an effective common case policy for many operating systems (e.g., Linux and FreeBSD).

A. ANN Model Prediction Accuracy

We evaluate the ANN model prediction accuracy by predicting wall-clock time and EDP. We use a cross validation technique in our experiments. In particular, we use 7 out of the 8 benchmarks for training and the remaining benchmark to verify prediction accuracy. Figure 7 shows the prediction error rate on the three platforms using 1400 samples in total. The error rate for wall-clock time is 2.18% on average and only 7.7% of the samples has an error rate higher than 5%. The prediction error rate for EDP is 3.31% on average and only 13.9% of the samples has an error rate higher than 5%.

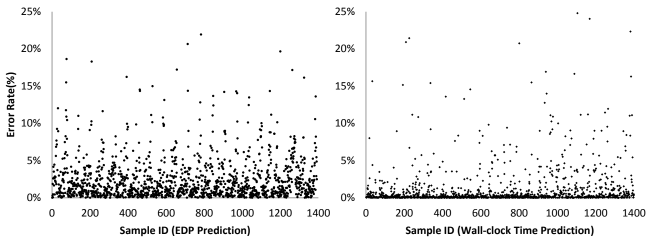

Figure 7: The distribution of ANN prediction error rate for EDP and wall-clock time

To investigate the variance of prediction accuracy across benchmarks, we look into the prediction results for each benchmark. Figure 8 displays the EDP prediction results for one OpenMP region of each benchmark. Similar variance of prediction accuracy is observed in other OpenMP regions. We notice that the predictor achieves high accuracy no matter how many threads are chosen to run a parallel region. We also notice that the prediction error rate for NAS SP is relatively high. We suspect this is due to a shift in the memory access pattern within the benchmark region studied. Our model cannot capture well oscillating memory access patterns within the same OpenMP region. Prediction accuracy can be improved, if the model is applied at a granularity finer than that of an OpenMP parallel region.

B. Comparison between ANN Model and Linear Regression Model

Linear regression models have been used for performance prediction in earlier work [9], [10], [12]. They are a realistic baseline to compare against the ANN model. We compare the prediction accuracy of the ANN model with that of a linear regression-based model proposed by Curtis-Maury et al. [26]. This linear regression model is briefly explained in Equation 5.

$$p_i = P_{max} * \ H_i(m_1, m_2, m_3, m_4) + e_i \tag{5}$$

where p_i is the prediction target (e.g., wall-clock time, EDP or MFLOPS/Watt) for the case of using i threads. P_{max} is the measured value using maximal number of threads and $H_i()$ is a transfer function to scale the observed P_{max}. The transfer function is a linear combination of four hardware event rates, m_1, m_2, m_3, and m_4, with significant contribution to the observed metric, in a statistical sense. For the 16-core Barcelona system, these rates are IPC, LMA, RMA and branch misses per cycle. e_i is a constant residual.

Figure 9 shows the prediction results from 21 parallel regions of NPB FT, CG, SP and MG benchmarks using the linear regression model. The benchmarks run with 8 threads on the 16-core Barcelona platform. The curves within the figure represent prediction values normalized to the measured values. We find that linear regression predicts EDP poorly. The prediction error is up to 60%. We further compare the linear regression model and ANN models in Table V, which summarizes the prediction error rates for wall-clock time and EDP, collected from the 16-core Barcelona platform. The results are averages of 21 parallel regions. In terms of wall-clock time prediction, the ANN model is about 7% better than the linear model, with the standard deviation being 10 times less. In terms of EDP prediction, the ANN model is much better (18%) than the linear model, with the standard deviation being 25 times less. The ANN model achieves better prediction accuracy than the linear model. This is because there is inherent non-linear relationship between hardware counter event rates and the prediction target, due to the implications of data locality and contention. The linear model lacks the ability to emulate the NUMA architecture, as all remote memory accesses are treated equally and summarized as a single term with only one coefficient within the model, despite varying latency due to the interconnect topology and contention. In contrast, the ANN model can map data locality and architecture details into the model illustrated in Figure 4, hence is able to make prediction with higher accuracy.

Table V: Comparison of the linear regression (LR) and ANN models for time and EDP predictions

Model	LR	ANN
The averaged error rate for time prediction	9.90%	2.18%
The standard deviation for time prediction	1.591	0.156
The averaged error rate for EDP prediction	22.61%	3.31%
The standard deviation for EDP prediction	2741.3	106.78

C. Thread Mapping

We compare thread mapping in DyNUMA to the default thread mapping scheme used in Linux. Table VI displays selected results. For each benchmark, we choose a specific number of threads and then execute it with the two methods to decide the thread mapping. We run each test 100 times on the 16-core Barcelona machine. Table VI reports the best performance improvement with DyNUMA for each test

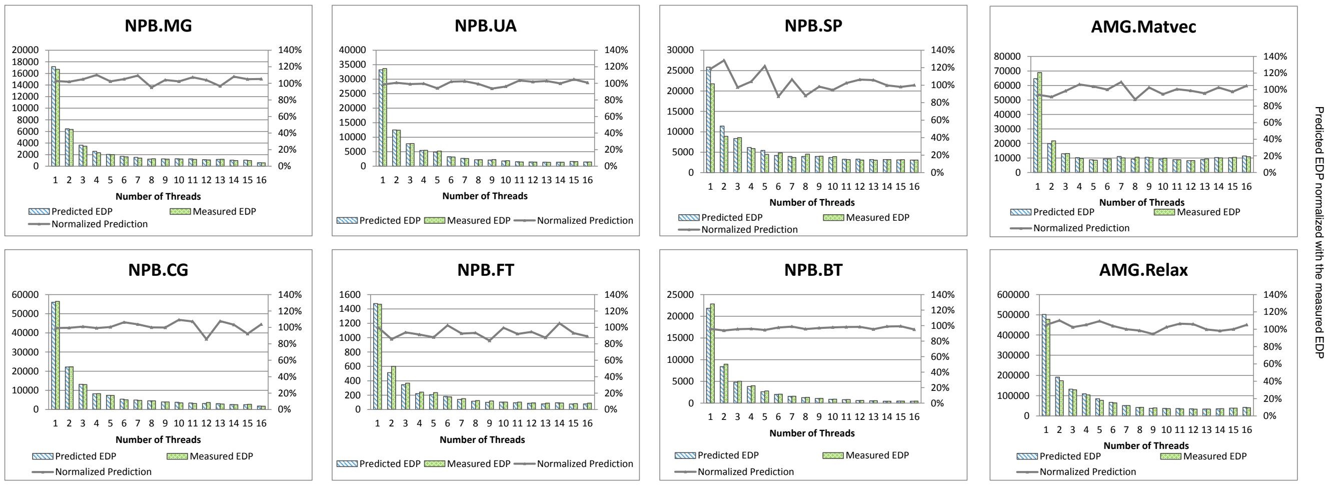
Figure 8: The EDP prediction results for the 16-cores system with the ANN model. The *Normalized Prediction* refers to the predicted value normalized by the measured one.

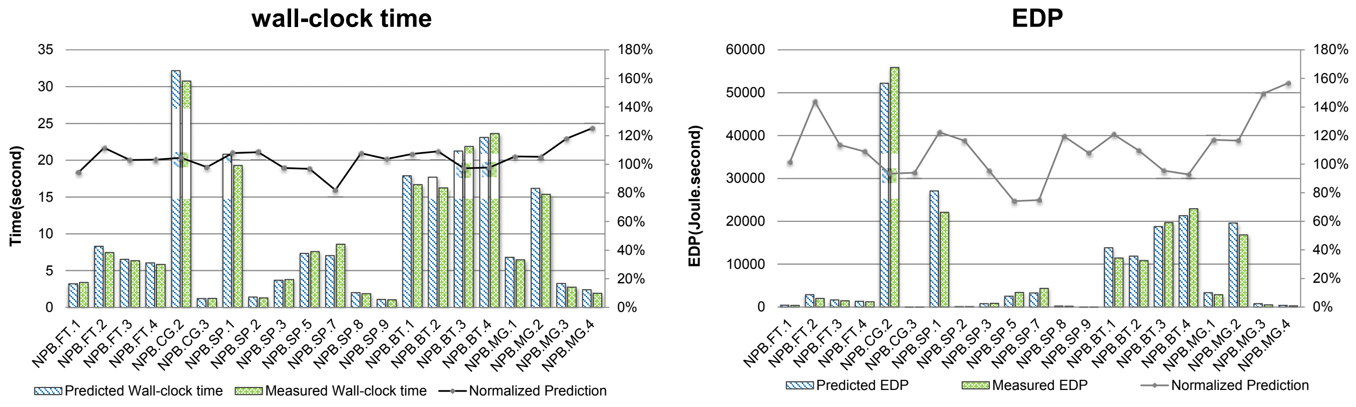
Figure 9: Prediction accuracy of the linear regression model

case. The results indicate that optimized thread mapping can significantly improve performance.

Table VI: Performance improvement with our thread mapping algorithm

Benchmark	# Threads	Performance Improvement
SP.C	4	20%
FT.B	8	28%
MG.B	12	6%
MG.B	16	14%

D. ANN versus TMA

We use two benchmarks, AMG.Relax and AMG.Matvec to show if the ANN predictor provides performance improvement over a system that uses only TMA as an optimizer before showing the performance of the two optimizers combined in next subsection. Figure 10 shows that concurrency control with the ANN provides significant additional improvement in performance and energy-efficiency compared to mere thread mapping optimization. This behavior is more pronounced in memory-bound code regions.

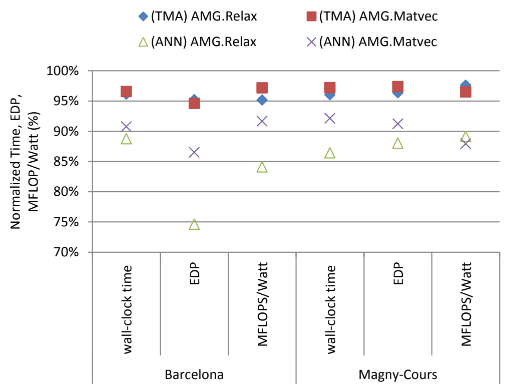
Figure 10: Performance comparison of ANN over TMA

E. DyNUMA Results

We report results in Figures 11-14 and Table VII. These results are normalized to the respective metrics with maximum concurrency and the default Linux thread mapping. On TilePro64, we test DyNUMA with a limited subset of the benchmarks due to hardware instability. The TilePro64 provides a platform-specific Oprofile tool for collecting hardware event rates. Oprofile, unlike PAPI, does not have the ability to collect data at runtime. Therefore, the TMA algorithm cannot collect application signatures on TilePro64. Hence, we only use the ANN model to predict thread concurrency without applying TMA on the Tilera platform.

Figure 11 summarizes the performance of DyNUMA and Table VII presents averages. We notice significant improvement in EDP and noticeable improvement in wall-clock time on the TilePro64. The improvement stem exclusively from concurrency throttling, as applications do not scale perfectly on the TilePro64. By choosing appropriate thread-level concurrency, DyNUMA improves EDP by 30%. Improvements in performance and energy-efficiency on other platforms are more modest but still measurable and consistent.

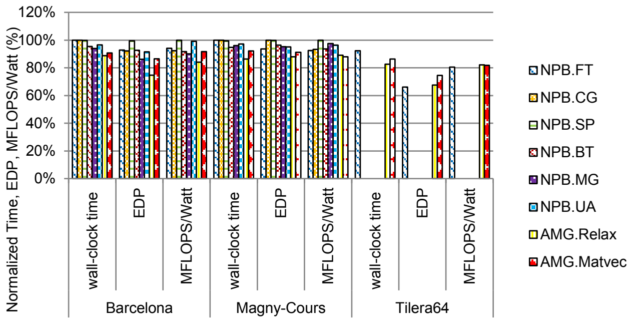

Figure 11: Performance improvement with DyNUMA on the three platforms

Table VII: Performance improvement with DyNUMA on the three platforms

Metrics	Barcelona	Magny-Cours	TilePro64
wall-clock time	6.74%	6.58%	12.88%
EDP	10.45%	6.90%	30.58%
MFLOS/Watt	10.66%	7.60%	18.49%

To further explore DyNUMA results, Figures 12-14 break down the metrics presented in Figure 11 between OpenMP parallel regions longer than 100 milliseconds. On the 16-core Barcelona system, DyNUMA achieves improvement in performance in 45% of the OpenMP parallel regions and energy efficiency in 72% of the OpenMP parallel regions; on the 32-core Magny-Cours machine, DyNUMA achieves improvement in performance in 59% and energy efficiency in 56% of OpenMP parallel regions; on the Tilera platform, all parallel regions benefit from DyNUMA in both performance and energy efficiency. However, not all parallel regions present opportunities for optimization. Compute-intensive regions tend to be more scalable and less sensitive to thread mappings than memory-bound regions. This is the case, for example, in NPB.FT.4, NPB.BT.1, NPB.BT.4 and NPB.UA.18. In these parallel regions, DyNUMA leads to negligible performance loss.

V. Conclusions and Future Work

Performance and energy efficiency optimization depends on effective control and mapping of parallelism to the system architecture. NUMA architectures expand significantly the search space of optimality. Programmers are often unaware of or unwilling to navigate this space via experimentation. Effective automatic control of concurrency and mapping needs to consider not only workload characteristics but also specifics of the underlying NUMA architecture.

This paper presents a framework combining a memory-centric, architecture-aware ANN model and a thread mapping arbiter to help parallel programs to autonomously optimize their concurrency and thread mapping at runtime.

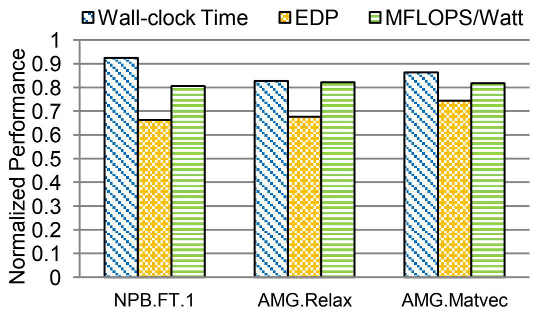

Figure 12: Performance with DyNUMA on the 64-cores Tilera platform

We evaluate the framework using the NAS and Sequoia Benchmarks on three different NUMA platforms. DyNUMA achieves on average 8.7% improvement in wall-clock time, 16% improvement in EDP and 12.3% improvement in MFLOPS/Watt.

For future work, we will incorporate DyNUMA with dynamic data migration to achieve better thread-data affinity. We will also develop a strategy to combine small parallel regions into bigger ones to explore new opportunities for performance improvement.

Acknowledgement

The paper has been authored, in part, by Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract #DE-AC05-00OR22725 to the U.S. Government.Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes. This research is sponsored by the Office of Advanced Scientific Computing Research in the U.S. Department of Energy. The research leading to these results has received funding from the European Union 7th Framework Programme [FP7/2007-2013], under the ENCORE (grant agreement no 248647), TEXT (no 261580) and I-CORES (no 224759) Projects. This material is based upon work supported by the National Science Foundation under Grant No. 0910784 and 0905187.

References

[1] Tilera, "TILEPro64 Processor-Product Brief," Tilera, Tech. Rep., 2012.

[2] NVIDIA, "NVIDIA Next Generation CUDA Compute Architecture:Fermi," 2012. [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[3] TOP500, "TOP500 Supercomputer Site." [Online]. Available: http://www.top500.org

[4] J. Marathe, V. Thakkar, and F. Mueller, "Feedback-Directed Page Placement for ccNUMA via Hardware-Generated Memory Traces," *Journal of Parallel and Distributed Computing*, vol. 70, no. 12, pp. 1204–1219, 2010.

[5] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguadé, "UPMLIB: A Runtime System for Tuning the Memory Performance of OpenMP Programs on Scalable Shared-Memory Multiprocessors," in *The 5th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, 2000, pp. 85–99.

[6] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguade, "User-level dynamic page migration for multiprogrammed shared-memory multiprocessors," in *Proc. of the 2000 International Conference on Parallel Processing*, 2000.

[7] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors," in *Proc. of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07, 2007.

[8] R. Azimi, D. K. Tam, L. Soares, and M. Stumm, "Enhancing operating system support for multicore processors by using hardware performance monitoring," *SIGOPS Oper. Syst. Rev.*, 2009.

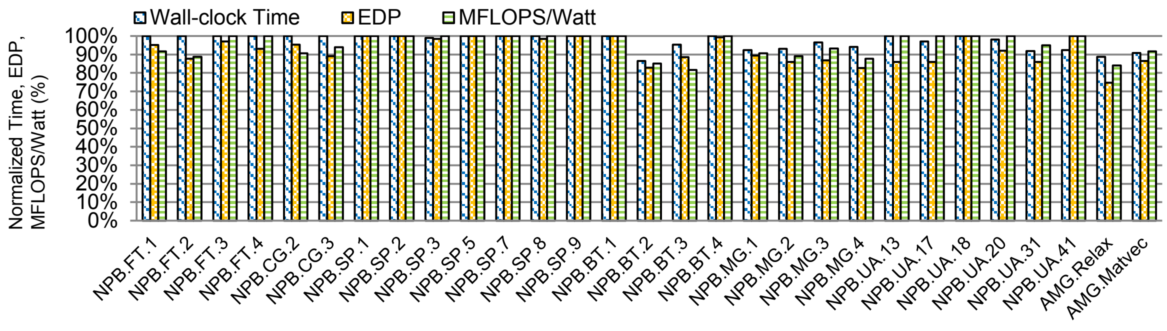

Figure 13: Performance with DyNUMA on the 16-cores Barcelona platform

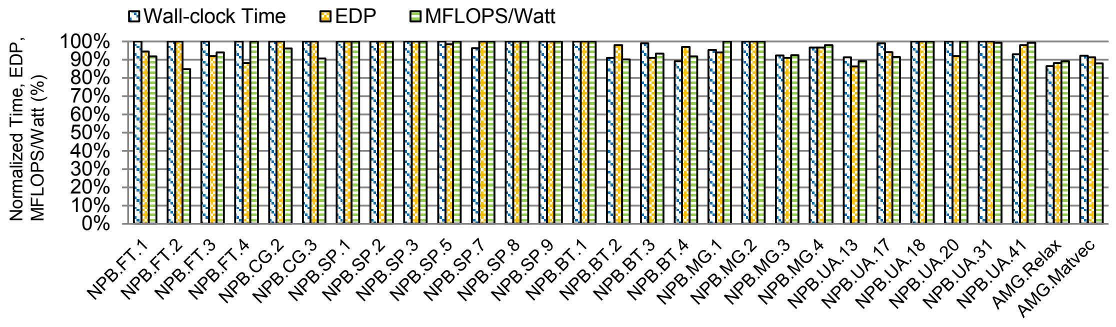

Figure 14: Performance with DyNUMA on the 32-cores Magny-Cours platform

[9] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz, "Prediction Models for Multi-Dimensional Power-Performance Optimization on Many Cores," in *Proc. of the 17th international conference on Parallel architectures and compilation techniques*, 2008.

[10] D. Li, B. de Supinski, M. Schulz, K. Cameron, and D. Nikolopoulos, "Hybrid MPI/OpenMP Power-Aware Computing," in *IEEE International Symposium on Parallel Distributed Processing*, 2010.

[11] C. Su, D. Li, D. Nikolopoulos, M. Grove, K. W. Cameron, and B. R. de Supinski, "Critical path-based thread placement for numa systems," in *The second international workshop on Performance modeling, benchmarking and simulation of high performance computing systems*, 2011.

[12] D. Li, D. Nikolopoulos, K. Cameron, B. de Supinski, and M. Schulz, "Power-Aware MPI Task Aggregation Prediction for High-End Computing Systems," in *IEEE International Symposium on Parallel Distributed Processing*, 2010.

[13] K. Singh, M. Curtis-Maury, S. A. McKee, F. Blagojević, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz, "Comparing Scalability Prediction Strategies on an SMP of CMPs," in *Proc. of the 16th international Euro-Par conference on Parallel processing*, 2010.

[14] M. Curtis-Maury, K. Singh, S. A. McKee, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz, "Identifying Energy-Efficient Concurrency Levels Using Machine Learning," in *Proc. of the 2007 IEEE International Conference on Cluster Computing*, 2007.

[15] C. Terboven, D. an Mey, D. Schmidl, H. Jin, and T. Reichstein, "Data and Thread Affinity in OpenMP Programs," in *Proc. of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem?*, 2008.

[16] C. Ribeiro, J.-F. Mehaut, A. Carissimi, M. Castro, and L. Fernandes, "Memory Affinity for Hierarchical Shared Memory Multiprocessors," in *Computer Architecture and High Performance Computing, 2009. SBAC-PAD '09. 21st International Symposium on*, 2009.

[17] F. Broquedis, N. Furmento, B. Goglin, R. Namyst, and P.-A. Wacrenier, "Dynamic Task and Data Placement over NUMA Architectures: An OpenMP Runtime Perspective," in *Proc. of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, 2009.

[18] W. Feng and T. Scogland, "The green500 list: Year one," in *Proc. of IEEE International Symposium on Parallel Distributed Processing*, 2009.

[19] S. Haykin, *Neural Networks – a Comprehensive Foundation, Prentice Hall*, 2nd ed. Prentice Hall, 1999.

[20] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A Portable Interface to Hardware Performance Counters," in *In Proceedings of the Department of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.

[21] "Oprofile Performance Monitoring Tool." [Online]. Available: http://oprofile.sourceforge.net/news/

[22] "WattsUp Meter Tool." [Online]. Available: https://www.wattsupmeters.com

[23] R. Ge, X. Feng, and K. Cameron, "Modeling and Evaluating Energy-Performance Efficiency of Parallel Processing on Multicore Based Power Aware Systems," in *IEEE International Symposium on Parallel Distributed Processing*, 2009.

[24] D. H. Bailey, "Performance and the NAS Parallel Benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, 1994.

[25] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith, "The Sequoia 2000 Benchmark," in *Proc. of the 1993 ACM SIGMOD International Conference on Management of Data*, 1993.

[26] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos, "Online Power-Performance Adaptation of Multithreaded Programs Using Hardware Event-Based Prediction," in *Proc. of the 20th annual international conference on Supercomputing*, 2006.