Automatic Generation of Program Affinity Policies Using Machine Learning

Ryan W. Moore and Bruce R. Childers

University of Pittsburgh,
Pittsburgh, USA
{rmoore,childers}@cs.pitt.edu

Abstract. Modern scientific and server programs require multisocket, multicore machines to achieve good performance. Maximizing the performance of these programs requires careful consideration of program behavior and careful management of hardware resources. In particular, a program's affinity can have a critical performance effect. For such machines, there are many possible affinities for a multithreaded program. In this paper, we present AutoFinity, a solution to automatically generate program affinity policies that consider program behavior and the target machine. The policies are constructed with machine learning and used online to select an affinity. We implemented AutoFinity on a 4-processor, 48-core machine and evaluated it on 18 multithreaded programs with varying thread counts. Our results show that in 12 out of 15 cases where affinity impacts runtime, the policy generated by AutoFinity chose affinities that improved performance versus assignments that do not consider program and machine behavior.

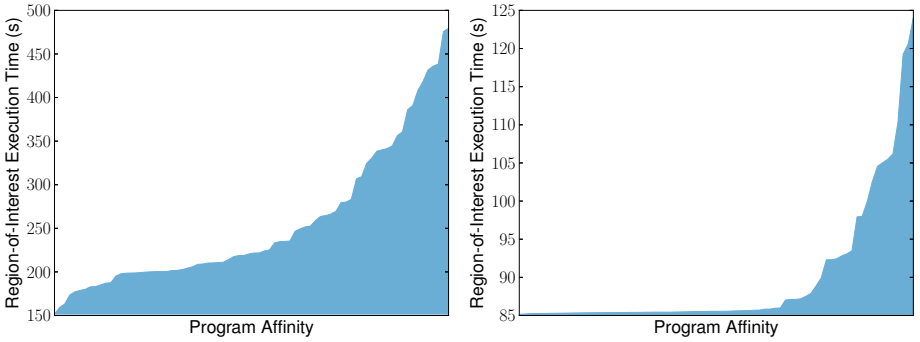
Keywords: policy generation, runtime adaptation, parallel performance.

1 Introduction

Today's computers for scientific and server workloads are large multisocket, multicore machines, capable of large amounts of parallelism. The cores in these machines share multiple resources, such as caches, memory controllers, and interconnects. For a multithreaded program, allocation decisions can be made for threads to share the hardware resources. Alternatively, threads can be given exclusive access to resources (e.g., executing with their own last-level cache). Because resource allocation impacts performance, it must be performed carefully and in response to runtime conditions, such as core availability, thread count, and workload demand. The assignment of program threads to cores, i.e. *program affinity*, is one way resources can be allocated.

It is well known that there is no single "best" program affinity to use across all programs. For example, Fig. 1a shows the program *streamcluster*'s region-of-interest (ROI) execution time when executed with different affinities [1]. The ROI is the program's parallel section where the "work" is done. This figure shows how performance (y-axis) changes with different affinities (x-axis). The program

K. De Bosschere and R. Jhala (Eds.): CC 2013, LNCS 7791, pp. 184–203, 2013.



(a) *Streamcluster* (b) *Swaptions*

Fig. 1. Execution times of programs' region-of-interest across different program affinities on a 48-core machine (each program has 8 threads)

affinities are sorted by ROI execution time. *Streamcluster* uses 8 threads on a 48-core AMD machine in these experiments.

For *streamcluster* (Fig. 1a) the first 5% of affinities achieve an execution time of around 150 seconds. Beyond this point, quality diminishes, resulting in an execution time of about 200 seconds (33% slower). The remaining execution times steeply increase as the affinity quality continues to diminish. The worst affinity has an execution time of over 450 seconds! Because so few affinities are good, it is unlikely that a randomly chosen affinity will result in good performance.

In contrast, *swaptions* (Fig. 1b) is relatively insensitive to affinity [1]. The majority of affinities have nearly identical performance. If an affinity was randomly selected, the resulting execution time would likely be good because *most* affinities performed well. However, some affinities cause *swaptions* to perform poorly. The worst affinity has an execution time of 124 seconds (45% slower than the best one). Even in this insensitive program, it can be beneficial to select its affinity because there are cases that should be avoided.

It is important to understand why *streamcluster* and *swaptions* behave so differently (Fig. 1a and 1b). Certain affinities allow programs to communicate more quickly (e.g., by sharing cache space). However, sharing caches among threads can reduce the cache space, causing the working set to overflow the cache. Other affinities give a thread more cache space, by spreading the threads across sockets and caches, at the cost of communication speed. Cache and communication demands of a program directly influence which affinities are best.

In a study of PARSEC, *streamcluster* made frequent use of barriers [1]. Because affinity affects communication, and thus, the speed at which threads reach and pass through barriers, *streamcluster* is affected dramatically by its affinity. *Swaptions* does not make much use of barriers or locks [1]. Furthermore, its working set size is relatively small: A thread's working set can fit in our experimental machine's private L2 cache. Thus, *swaptions* is more resilient to affinity changes.

As these figures show, there is diversity in program behavior across affinities. The runtime resource manager (e.g., operating system) should support the user

and select a well-behaving affinity for the user's program. Furthermore, selection should support any thread count as well as never-before-executed programs.

In this paper, we present *AutoFinity*, an automated system that generates an *affinity policy* based on machine learning and training data. The generated policy is used at runtime to select a program's affinity. The policy can handle programs that were not part of the training and/or thread counts that have not been considered by training. AutoFinity can update its policy as it discovers and records new information about programs.

A policy chooses a thread-count-independent affinity class, which we call an *affinity hint*. The system resource manager (e.g., the operating system) uses a hint to select an affinity. Affinity hints enable policies and affinity selection across thread counts in the presence of runtime resource constraints.

We discuss the choices and trade-offs behind AutoFinity's design. We evaluate several design decisions on how AutoFinity can improve program performance. The contributions of this paper are:

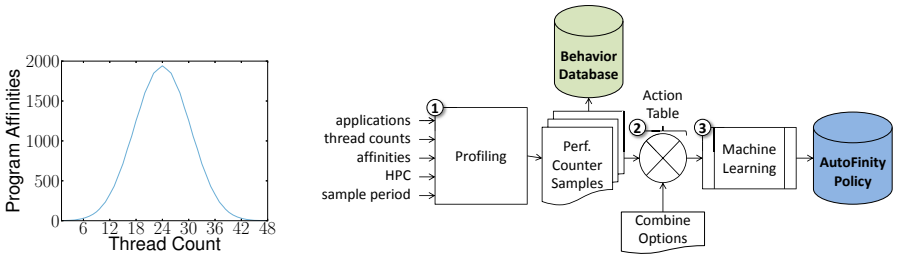
1. We show the importance of choosing program affinities;
2. We present AutoFinity, an automated, thread-independent solution to select program affinity;
3. We provide guidelines to find the proper set of hardware performance counters (HPCs); and,
4. We evaluate AutoFinity and demonstrate its ability to produce policies that guide affinity settings across a range of programs and thread counts.

This paper is organized in the following way. Section 2 discusses the design and use of AutoFinity. Section 3 performs an evaluation of AutoFinity. Section 4 discusses related work. Section 5 concludes.

2 The Design of an Affinity Policy Generator

This work focuses on choosing program affinity for CPU/memory-bound programs. These programs will most benefit from proper affinity selection. There are several challenges to choosing program affinity. First, the number of affinity settings is large, as shown in Fig. 2a. With 24 threads there are 1941 settings on a 48-core machine. Selecting program affinities across thread counts is even more difficult. Second, the selection process must support unknown programs and yet quickly select a program affinity without evaluating, through trial and error, program affinities. Finally, affinity selection should handle runtime resource constraints (e.g., an unavailable socket). We assume at least one available core per thread and that, if is the system is shared between users, machine resources (e.g., sockets) are statically partitioned to provide strong isolation.

AutoFinity addresses these challenges. It is built on *REEact*, a framework for virtual execution management [13]. AutoFinity automatically builds an affinity policy to maximize a user-defined metric. The policy is consulted at runtime on unseen programs to determine an *affinity hint*, a set of possible affinities that shall work well. A hint is independent from thread count and is used by the resource manager (e.g., operating system) as a guide for satisfying runtime conditions.



(a) Affinity counts (b) Policy generation steps

Fig. 2. Available affinities and building a policy to help choose one

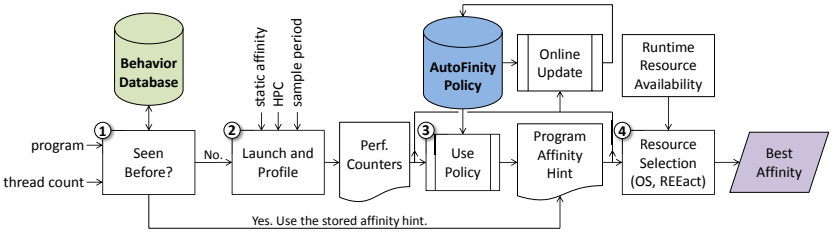


Fig. 3. Policy usage steps

Fig. 2b shows the steps to build a policy. First, AutoFinity profiles training programs for a range of affinities (step 1). This step gathers data that indicates how well particular affinities behave. Program behavior is captured by hardware performance counters (HPC) gathered during the program's ROI. The HPC values are recorded regularly, creating multiple samples. Each sample is timestamped to facilitate the examination of cross-affinity program behavior (e.g., two samples each from the beginning of a program's execution). A sample observes program behavior over a fixed amount of time. As such, multiple consecutive sample periods capture a program's ROI behavior.

The samples are analyzed and transformed into an *action table* (step 2). The action table is built to maximize a user-defined metric (e.g., performance). A row in the action table states what affinity hint (discussed in Sect. 2.1) should be used if a program exhibits a particular behavior at runtime. The samples are also archived in a behavior database to build future policies.

Machine learning is used to compact the rows of the action table thus building the *affinity policy* while also handling contradictory or missing information in the table (step 3). Machine learning resolves the conflicts and "holes" through its statistical analysis.

The generated policy is used at runtime to select an affinity hint. Fig. 3 shows this process. To use the policy, the program's thread count must be known. The thread count can be discovered on the command line, in an environment variable, or by monitoring thread creation syscalls.

With the thread count, the behavior database is consulted. If the program and its thread count have been previously observed then the behavior database specifies the program's affinity hint (step 1). Otherwise, the policy is consulted to obtain the program's affinity hint. To consult the policy, program behavior must be observed. A starting affinity will be used and the program's parallel behavior will be observed. In this paper, we examine the program's behavior once, during a single sample period (step 2). Continuous sampling and adjustment are naturally supported by our approach, but require a migration cost model (future work).

The sampled HPC values are used by the AutoFinity policy to select an affinity hint (step 3). The HPC values and selected hint are saved into the behavior database. The affinity hint can be used to choose a new program affinity. To use an affinity hint, the program's resource controller (e.g., operating system), considers the hint and allocates resources (cores), ultimately choosing a program affinity for the program (step 4).

Because AutoFinity continuously records program behavior, it can generate a new policy as program information is accumulated (e.g., when the behavior database has grown by some percent threshold). If a stored affinity hint corresponds to an old policy, the hint will be removed.

2.1 Affinity Hints

Program affinities explicitly state which cores a program may use (e.g., cores 0–5). For a particular thread count, there may be thousands of possible affinity choices. It is difficult to directly pick an appropriate affinity, and selecting an affinity is made harder by supporting all thread counts. To reduce the number of affinity options that a policy must consider, it is necessary to generalize program affinities into classes (e.g., affinity hints).

An affinity hint suggests a relationship between cores (e.g., cores should be distributed across different cache domains). The generalization of program affinities enables techniques which work *across* thread counts. Using a hint, a program's resource manager will select an affinity based on available resources (e.g., sockets) and number of active threads.

There are two requirements for affinity hints. First, affinity hints should not be overly specific: A hint should be realized at runtime, even under runtime resource constraints (e.g., an unavailable processor socket). Second, affinity hints must capture the effects of core assignments on a thread's: a) cache space, b) communication, and c) access to main memory (DRAM).

Threads may benefit from a large amount of effective cache space (e.g., to hold their private working set). The space available to a thread may be negatively impacted by the presence of one or more threads in the same last-level cache domain (LLC). However, sharing LLCs also allows threads to more quickly communicate. Regardless of whether cores share one or more levels of cache, a thread will occasionally have cache misses. It is important to consider the impact of affinity on NUMA accesses. For example, programs may be able to better exploit memory bandwidth if spread across NUMA domains.

Affinity Hint Parameter	Parameter Description	Values	Symbol
spreadAcrossSockets	Prefer use of many sockets	{True, False}	S
spreadAcrossLLCs	Prefer to not share LLCs	{True, False}	L
socketOptionGetsPriority	Socket preference is priority	{True, False}	P

Fig. 4. Affinity hint parameters

```
GetAffinity(resources, threadCount, spreadAcrossSockets, spreadAcrossLLCs,
socketOptionGetsPriority)
 1  affs = AllAffinities(resources, threadCount)
 2  if spreadAcrossSockets
 3       socketSortFunc = PrioritizeByLargerUsedSocketCount
 4  else
 5       socketSortFunc = PrioritizeBySmallerUsedSocketCount
 6  if spreadAcrossLLCs
 7       LLCSortFunc = PrioritizeBySmallerLLCOccupancy
 8  else
 9       LLCSortFunc = PrioritizeByHigherLLCOccupancy
10  if socketOptionGetsPriority
11       prioritizedAffs = Sort(affs, socketSortFunc then by LLCSortFunc)
12  else
13       prioritizedAffs = Sort(affs, LLCSortFunc then by socketSortFunc)
14  return HighestPriority(prioritizedAffs)
```

Fig. 5. Algorithm for selecting a program affinity

To meet these requirements, the affinity hints have three boolean parameters, leading to eight possible affinity hints (Fig. 4). The first affinity hint parameter dictates whether threads should be distributed across sockets (*spreadAcrossSockets*, symbol: S). The use of multiple sockets may allow greater memory bandwidth, by simultaneously employing memory banks. On the contrary, assigning threads to the same sockets allows better communication in the same memory domain, potentially avoiding the need to access remote nodes.

The second parameter, *spreadAcrossLLCs* (symbol: L), controls whether threads should be assigned to cores that share a LLC. Threads that share a cache may communicate with each other more quickly or may prefetch data for the sharers. However, sharers may contend for cache space. The third parameter of an affinity hint (*socketOptionGetsPriority*, symbol: P) captures whether the sharing sockets or sharing LLC parameter is more important.

The three parameters define eight hints. We use boolean notation to represent each hint. For example, $\mathrm{S}\overline{\mathrm{LP}}$ specifies *spreadAcrossSockets* = True, *spreadAcrossLLCs* = False, and *socketOptionGetsPriority* = False. This hint dictates that application threads share as few LLCs as possible ($\overline{\mathrm{L}}$)[1]. The LLCs are preferred to be on separate sockets (S). The priority in this example hint is to pack threads onto as few LLCs as possible ($\overline{\mathrm{P}}$).

[1] There is always at most one thread per core.

With an affinity hint, the OS can consider runtime resource availability to choose an affinity. This process is shown in Fig. 5. GetAffinity takes a list of available resources (cores) and the amount needed (*threadCount*). A list of affinities that can be made from the available resources is obtained (line 1). Only affinities that use as many cores as threads are considered. The value of *spreadAcrossSockets* is used to prioritize the potential affinities (lines 2–5). If threads should be spread across sockets then affinities which utilize more sockets are prioritized, otherwise affinities which use fewer sockets will be preferred. In lines 6–9 *spreadAcrossLLCs* is similarly processed. A hint which states that threads should spread across LLCs will prioritize affinities accordingly. Next, the potential affinities are sorted based on *socketOptionGetsPriority*'s value (lines 10–13). Ties are broken by the secondary priority. The highest priority affinity (i.e., the one that best matches the affinity hint) is returned (line 14)[2].

2.2 Action Table Generation

AutoFinity analyzes program behavior to generate a policy that specifies an affinity hint for a program under runtime conditions. This process consists of two steps: 1) building the action table and 2) condensing the table into a policy.

The action table contains observed program behaviors and the affinity hint(s), that will improve the user-defined performance metric in those situations. It is built with initial training data: programs in various affinities will have their behavior (HPC values) recorded. Additional behaviors can be gathered online.

Fig. 6 shows BuildActionTable. This function condenses performance information (*observations*) into an action table. BuildActionTable has two more parameters: *condense* (a function) and *leeway* (an integer). *Condense* resolves different performance values coming from similar HPC samples across multiple programs. This phenomenon is expected to occur occasionally, as programs may have similar behavior (e.g., cache misses) but different performances.

GetAffinity selects one configuration per affinity hint (assuming a fixed thread count). Each of the eight affinity hints will correspond to a best match affinity (eight affinities total). To allow training on affinities which *almost* would be selected by GetAffinity, we introduce *leeway*. *Leeway* is an integer that allows an affinity hint to correspond to additional affinities, beyond its best match. A leeway of 0 means that only the best match affinities are considered.

Leeway is a maximum edit distance between two affinities. A leeway of l allows a program affinity to be considered as belonging to an affinity hint if, by moving up to l threads across LLC domains, the affinity becomes identical to that affinity hint's best match affinity. The movement of threads across LLC domains may also move threads across sockets. Therefore leeway also places a limit on affinity similarity in terms of socket usage.

For example, an affinity that has six threads share a LLC might be selected for the hint $\overline{\text{SLP}}$ as its corresponding best match affinity (for 6 threads). With

[2] Multiple affinities may tie for the same priority. In this case, the affinities will be isomorphic and one will be chosen arbitrarily.

```
BuildActionTable(observations, condense, leeway)
 1  possibilities = ∅, actionTable = ∅
 2  for program ∈ observations
 3      for samplePeriod ∈ program
 4          for aff ∈ samplePeriod
 5              behavior = aff.observedPerformanceCounters
 6              for destAff ∈ samplePeriod
 7                  perf = destAff.perf
 8                  possibilities[behavior][aff][destAff].append(perf)
 9  for behavior ∈ possibilities
10      for aff ∈ behavior
11          for destAff ∈ aff
12              affinityHints = AffinityHintFromConf(destAff, leeway)
13              if affinityHints = NONE
14                  continue // Configuration does not map to an affinity hint.
15              consequence = condense(possibilities[behavior][aff][destAff])
16              // An affinity may map to > 1 affinity hints (e.g., due to leeway).
17              for hint ∈ affinityHints
18                  actionTable[behavior][aff][hint] = consequence
19  for behavior ∈ actionTable
20      for aff ∈ behavior
21          bestAffinityHint = MAX(actionTable[behavior][aff])
22          WriteTrainAction(behavior, bestAffinityHint)
```

Fig. 6. Algorithm for building the action table

a leeway of 1, the affinity which causes 5 threads to share a LLC while putting another thread on a separate socket, would still be considered as corresponding to $\overline{\text{SLP}}$ because only 1 thread was moved as compared to the best match affinity.

In Fig. 6 the first set of loops (lines 2–8), build up a table, *possibilities*, which stores the affinity (*aff*) and its associated *behavior* (HPC values). For each affinity with recorded behavior, alternate affinities (with the same thread count) are considered. These are considered to be destination affinities (*destAff*) that the program *could have* been in. For each alternative affinity, the performance metric of the program in that sample period affinity is recorded (*perf*).

Programs may exhibit similar behavior, and therefore, the first set of loops accumulate a list of multiple performance metrics (line 8). List items are reduced into one entry using the *condense* function (lines 9–18). BuildActionTable also converts destination affinities to their corresponding affinity hint(s), with AffinityHintfromConf (the inverse of GetAffinity). After this step, the action table contains a list of affinities and associated behaviors, affinity hints that could have applied, and the consequent performance for the affinity hints.

Finally, the action table is ready to be written. In lines 19–22 the action table is iterated over. For each type of behavior seen under each experienced affinity, BuildActionTable finds the best affinity hint to be in. These results are written to a file as a series of rules (WriteTrainAction). Each rule states an affinity that should be used if a particular behavior is observed. Contradictions

```
AffinityHintPolicySW(dCacheAccesses, invalidDCacheLinesEvicted,
                     exclusiveReadRequestsToL3CacheFromAnyCore, retiredUops)
1  if retiredUops ∈ (BoundLower0, BoundUpper0]
2      if dCacheAccesses < Bound1 and  invalidDCacheLinesEvicted < Bound2
3          return S̄LP
4      if invalidDCacheLinesEvicted < Bound3 and
                 dCacheAccesses ∈ (BoundLower4, BoundUpper4] and
                 exclusiveReadRequestsToL3CacheFromAnyCore < Bound5
5          return S̄LP̄
6  return S̄LP
```

Fig. 7. Example affinity hint policy

between rules are expected, and may be due to noise and/or different programs exhibiting similar behavior, yet operating best in different affinity hints.

2.3 Building the Affinity Hint Policy

To convert the action table to a policy that can be consulted at runtime, we use JRIP from WEKA [4]. JRIP is an implementation of Repeated Incremental Pruning to Produce Error Reduction (RIPPER), a propositional rule learner that produces a series of rules for classification. Each rule is a set of conditions joined by `and` operators. RIPPER fills in "holes" due to incomplete data (i.e., combinations of HPC values that were not observed). It also prunes rules that are statistically insignificant. We use RIPPER because it creates rules for missing data, produces simple, human-readable policies, and its rules are easily converted into a code implementation to use at runtime. We use WEKA's default settings.

An example policy produced by JRIP is shown in Fig. 7. This policy is used by *swaptions* in Sect. 3. For readability, numeric values have been replaced with constants. The policy's parameters are the values of four HPCs. The selection of these counters is discussed later (Sect. 3.1). Depending on the values of these counters, one of three affinity hints will be selected ($\overline{\text{S}}$LP, $\overline{\text{S}}\text{L}\overline{\text{P}}$, or $\overline{\text{SL}}$P).

These affinity hints have been selected by the RIPPER algorithm as the only necessary affinity hints (i.e., there is no fourth affinity hint that the policy will select). As more program performance data is obtained and the policy is rebuilt, additional affinity hints may become available for selection.

3 Evaluation

The previous section described how AutoFinity generates affinity hints, builds the action table and its policy, and uses its policy. This section presents experimental results for AutoFinity on a diverse set of programs and thread counts.

All evaluations were performed on the machine described in Fig. 8. The machine is a 48-core AMD Opteron 6164 NUMA system. Each of the four sockets has 2 NUMA domains (8 NUMA domains total). Each NUMA domain has 6

Machine Component	Component Details
Processors	4 AMD Opteron 6164 HE sockets (48 cores total)
Socket	2 NUMA nodes
NUMA Node	1 L3 cache (LLC)
L3 Cache	5 MiB cache, 6x cores
Core	1.7 GHz, private L1 $ (64 KiB), private L2 $ (512 KiB)
Operating System	Linux 2.6.39.1

Fig. 8. Experimental Machine

cores that share an L3. The cores have private L1 and L2 caches. Programs are executed on an otherwise idle system.

The 18 evaluated benchmarks are culled from PARSEC 2.1 [1], OmpSCR 2.0 [3], and the NAS Parallel Benchmark Suite (NPB) 3.3.1 [7]. The PARSEC benchmarks use the largest input size ("native") and the NPB benchmark uses the "A" input size. Because OmpSCR does not provide official inputs, we configured its programs to execute in a similar amount of time as the PARSEC benchmarks (about one to two minutes with eight threads).

We use the following PARSEC programs: *blackscholes* (BS), *bodytrack* (BT), *canneal* (CN), *dedup* (DD), *facesim* (FS), *fluidanimate* (FA), *freqmine* (FM), *raytrace* (RA), *streamcluster* (SC), *swaptions* (SW), *vips* (VP), and *x264* (X). From OMPSCR we use *c_fft* (FT), *c_fft*6 (FT6), *c_lu* (LU), *c_mandel* (MN), and *c_md* (MD). Lastly, we use *dc* (DC) from NAS. Some programs (e.g., PARSEC's *ferret*) were not used due to compilation errors, framework incompatibilities, or short execution times (i.e., a few seconds).

We use AutoFinity to maximize instructions per second (IPC), consequently minimizing program execution time. We chose this metric because it captures the rate of application progress and is easy to obtain via hardware performance monitoring. As we show later, choosing IPC works well to reduce execution time.

Before using AutoFinity to select a program's affinity hint and program affinity, we remove that program's data from the training data. This causes the program to be treated as a never-before-executed program. The AutoFinity policy is built on the remaining programs' data (leave-one-out cross-validation).

To evaluate the quality of the generated policy, we launch the program in a fixed affinity and observe one sample's HPC values (as shown in Fig. 3). The AutoFinity policy uses the HPC values to select an affinity hint. An affinity is selected from the hint. We then execute the program with that affinity and observe its ROI runtime. To find average performance, we take the geometric mean of each program executed under their selected affinity.

By default, previously unexecuted programs and thread counts are executed under a *distributed* affinity. A distributed affinity allows the AutoFinity policy to make better affinity hint suggestions. A distributed affinity spreads a program's threads across sockets and LLCs.

3.1 Choosing and Gathering Hardware Performance Counters

Before an AutoFinity policy can be used, the policy must be built. This requires profiling programs using HPCs. We use feature selection to choose appropriate counters automatically. The chosen counters will be measured during the initial training period as well as online (e.g., to consult the policy). To determine which counters were the best to help AutoFinity maximize IPC, we profiled several programs multiple times, collecting HPC values periodically with 60 manually selected counters.

Our experimental machine supports recording up to four counters at a time. Additional counters bring dwindling returns in terms of utility. The machine learning algorithm (i.e., RIPPER) will ignore counters and/or counter values that are statistically insignificant.

We used WEKA's `CfsSubsetEval` feature selection algorithm to choose the four performance counters that best correlate with our objective metric, IPC. This method evaluates features by considering the individual predictive ability of each feature along with the degree of redundancy between them. Features were selected and "grown" with a greedy search method.

Out of the 60 counters considered, the feature selection process selected the following features: 1) data cache accesses, 2) invalid data cache lines evicted, 3) exclusive read requests to the LLC from any core, and 4) retired micro-ops. These HPCs capture well how affinities affect IPC due to:

1. Data cache accesses indicate the memory demands of a program.
2. Cache lines become invalid if a shared line becomes modified by another core. Therefore, this counter indicates thread communication.
3. Exclusive read requests to the LLC cache also indicates communication and memory demand. A cache line in the exclusive state may soon become shared if another thread accesses that line. Exclusive LLC read requests may also allow one thread to prefetch another's data.
4. Retired micro-ops counts the number of operations completed by the processor. Our experiment machine, an x86-64 processor, executes instructions that decode into one or more RISC-like operations. Therefore, this metric captures application performance (IPC).

Lastly, for our techniques to work across a range of thread counts and sampling period lengths, we normalize each performance counter value to the number of threads and the number of seconds over which the counter was gathered.

Knowing which four HPCs were necessary to observe and maximize IPC, we gathered the selected counters and IPCs for each program. The programs are executed with 8 threads and counters are recorded for each of the 78 program affinities (8 threads). This step gathers the initial training data.

3.2 Selecting a Discretization Method

To build the action table, the training data must be put into discrete bins. After discretization, the data is analyzed and the action table is constructed (as discussed in Sect. 2.2). We consider two methods to discretize the data:

Method	Geo. Mean
equal-width	79.4
equal-frequency	83.1

(a) Impact of the discretization method

Function	Geo. Mean
average	80.6
gmean	80.6
hmean	80.6
max	81.5
median	79.4
min	83.1
sum	80.6

(b) Impact of the condense function

Fig. 9. Effects on average program execution time

1. Equal-width binning: Each bin corresponds to a constant range over the possible values of a HPC. Equal-width binning uniformly covers the range of observed performance counter values.
2. Equal-frequency binning: Bin widths are uneven to allow higher resolution where HPC values are most concentrated. Bins have the same item counts.

While we tried a range of numbers, we found that five bins for each method worked well, without making individual bins too wide or small. WEKA's equal-width discretization supports the automatic adjustment of the number of bins to be data-appropriate. We enabled this feature. Equal-frequency binning does not support such an option. In our experience, the adjustment is influenced by the number of requested bins.

To evaluate the discretization method, we perform cross-validation. The geometric mean of the program execution times is computed. Because the method of discretization is not the only adjustable parameter, we fix the other parameters (e.g., the condense function) to their best values (described later).

Fig. 9a shows the results of varying the discretization method. The results show that equal-width binning improves the geometric mean by approximately 5% (a mean of 79.4s versus 83.1s). Equal-width binning does better because equal-frequency binning creates very wide bins to allow for occasional narrow bins. However, a policy built with equal-frequency binning may be unable to later differentiate between HPC values which, though the difference between them is large, are grouped into the same wide bins. Thus, we conclude that equal-width binning is the best discretization method.

3.3 Selecting a Condense Function

We considered multiple condense functions to determine which method works best. The condense function combines performance measures from similar behaving samples gathered from the same affinity. Similar to Sect. 3.2, we set the other parameters to their best values. The condense function can be: a) arithmetic mean (AVERAGE), b) geometric mean (GMEAN), c) harmonic mean (HMEAN), d) MAX, e) MEDIAN, f) MIN, and g) SUM.

For example, suppose that program a is similar to b in that both have high data cache miss rates among other counter values. After analyzing their performance in different affinities, it is determined that both programs would execute fastest in affinity hint h. However, due to differences in program behavior, a will obtain an IPC in h of 2, whereas b will obtain an IPC of 1.5.

During action table and policy construction, AutoFinity must consider the expected IPC of a program, c, running under hint h, that is similar to a and b (e.g., c is a not-yet-encountered program). The action table could consider c to have a runtime of 2 (MAX), 1.5 (MIN), or 1.75 (AVERAGE). The condense function determines the expected value.

Fig. 9b shows the results of varying the condense function. The best function, MEDIAN, has a geometric mean of 79.4s. The worst condense function is MIN, with a geometric mean of 83.1s, about 5% worse than MEDIAN's result. MIN results in the worst performance because it creates a pessimistic policy. It penalizes types of behavior which are exhibited by multiple programs (e.g., if three programs exhibit the same behavior, the worst performing one is the only one whose performance will be considered).

Although MIN makes a pessimistic policy, and MAX makes an overly optimistic one, MEDIAN function is balanced and works well. Similarly, each of the means had good behavior.

3.4 Other Parameters

There are three other parameters for AutoFinity: 1) whether to normalize IPCs, 2) whether to include affinity-insensitive programs in the training data, and 3) leeway.

A program's IPC can be normalized to the smallest IPC recorded for that program. This allows for easier comparison of IPCs across programs. For example, if using a particular affinity hint can boost a program's IPC by 0.1, then this may be a relatively minor performance increase (e.g., 3% if the original IPC is 3.0) or a relatively larger increase (e.g., 10% if the original IPC is 1.0). Normalization allows these trends to be captured.

However, using the best parameter settings for the other policy parameters (e.g., the best condense function, the best discretization method) causes normalization to have no significant benefit. Nevertheless, we normalize IPCs because it did, for suboptimal parameter settings, result in a better policy.

The second parameter is whether affinity-insensitive programs are included in the training data. We define program insensitive programs as programs whose ROI execution time standard deviation is less than 1% of the program's average execution time across all affinities. Intuitively, programs whose behavior is insensitive to affinity contain no useful information to guide affinity selection. At worst, the data from insensitive programs may dilute important information.

As with IPC normalization, we found that using the best possible parameter settings causes the removal or inclusion of affinity-insensitive programs to make little difference (less than 1%). We did find it was occasionally beneficial for less

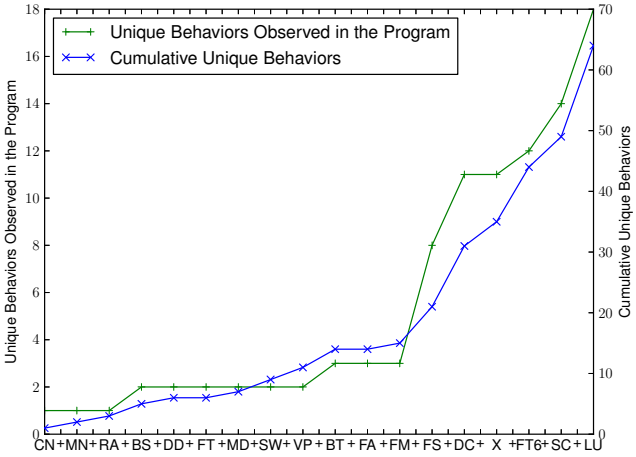


Fig. 10. Cumulative and per-program discretized behavior coverage

optimal settings. Therefore in our later evaluation, programs that are affinity insensitive are removed from the training data.

Lastly, we set leeway to 2. This leeway setting was determined experimentally (not presented due to space constraints). For 8 threads, this value resulted in 31 possible destination affinities while building the initial AutoFinity policy.

3.5 Program Behavior Coverage

Exposure to diverse program behavior allows the action table and policy to appropriately respond to unseen programs. To gain insight into this diversity, we analyzed program behavior across all possible affinities for 8 threads.

We define a program as exhibiting a behavior, b, if during the training process it produces a discretized HPC sample that is equal to b. Each program, p, produces a set of unique behaviors, B_p, where $B_p = \{b_0, b_1, ...\}$. Examining the discretized behaviors, we can determine the diversity of a program ($|B_p|$) and whether programs exhibit the same number of behaviors.

Other questions that we sought to address include whether some programs have more diverse behavior than others (i.e., whether $|B_{p_i}| < |B_{p_j}|$), and what is the effect of a program's behavior on the cumulative training information.

Figure 10 shows how coverage is influenced by the programs. The figure's x-axis is sorted by the number of unique behaviors in each program. It has two plots. The "+" line shows the number of unique behaviors in each program, while the "×" line shows the number of cumulative behaviors as the programs are added (left to right).

From the figure, we broadly place each program into one of two categories: 1) programs that have few unique behaviors and 2) programs that exhibit a diverse range of behavior. Approximately half of the programs, those before and including FM (*freqmine*), have few behaviors, and therefore, exhibit consistent

behavior across their lifetime and affinities. This is shown by the low height of the "unique behaviors" line. Even though 12 programs fall into this category, each with an average of 2 unique behaviors per program (24 total), there are only 15 unique behaviors between them (i.e., at the point on the x-axis for FM, the cumulative unique behaviors line has a y-value of 15). Each program, therefore, contributes on average only 1.25 unique behaviors to the cumulative training data (15 behaviors over 12 programs = 1.25 behaviors per program).

The other programs (after FM) have great diversity. This diversity may be due to phases and/or sensitivity to affinity. These 6 programs contribute 49 unique behaviors to the training data, for an average of more than 8 unique behaviors per program. *C_lu* has the most unique behaviors. It exhibits 18 unique behaviors and contributes 15 to the cumulative list. *Streamcluster* has the second highest number of unique behaviors (14) and contributes 5 unique behaviors.

From this data, we believe that only a few training programs (i.e., those that exhibit a range of behavior) may be necessary to produce a good policy.

3.6 Performance Evaluation

Next, we evaluate how the policies from AutoFinity choose affinity hints for unknown programs. We compare the execution times of the programs using the AutoFinity-selected affinities against two static policies:

1. *Packed*: This policy places threads together, causing them to share LLCs whenever possible, and preferring to use as few sockets as possible ($\overline{\text{SLP}}$).
2. *Distributed*: This policy distributes threads uniformly across sockets and LLCs (SLP).

AutoFinity uses a leeway of 2 and affinity insensitive programs are removed from the training data. IPCs from a program are normalized to the slowest IPC from that program. HPC monitoring uses a sample size of 10 seconds for the four counters described in Sect. 3.1. Lastly, the condense function is MEDIAN.

Fixed Thread Count Evaluation. First, we compare the runtime of each program with 8 threads under the affinities chosen by AutoFinity. Training data also consisted of programs with 8 threads. Figure 11 shows these results.

The y-axis is ROI execution time (lower is better). The x-axis shows each program, with the geometric mean at the far right. Subscripts indicate thread count. For each program, three bars are shown. The first bar is the runtime under a packed affinity. The second bar is runtime under a distributed affinity. Finally, the third bar is runtime for the affinity chosen by the AutoFinity policy.

Some programs are insensitive to the policy. *Blackscholes*, *freqmine*, *c_md*, *c_mandel*, *raytrace*, *swaptions*, *vips*, and *x264* are not significantly affected by affinity. To a lesser extent, *fluidanimate* is insensitive too. However for most programs, the affinity is important. Five of the programs prefer a packed affinity (*bodytrack*, *canneal*, *dedup*, *c_fft*, *streamcluster*). These programs' threads communicate and/or share data. For *bodytrack*, *dedup*, and *c_fft*, AutoFinity selects an affinity that performs nearly as good or just as good as the best static policy.

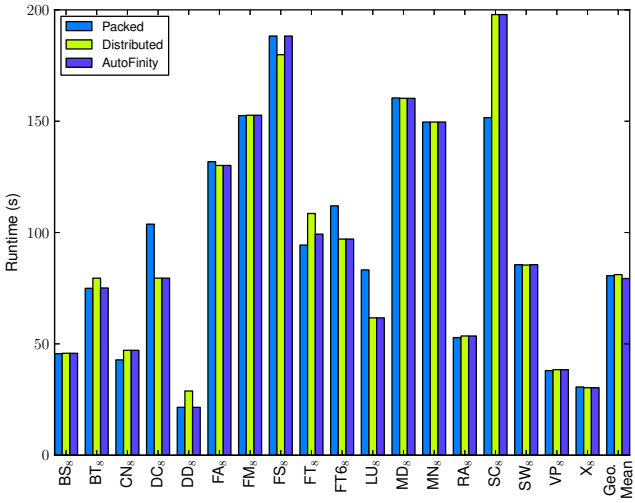


Fig. 11. AutoFinity, packed, and distributed ROI execution times for 8 threads

The AutoFinity policy selects a poor affinity hint for *streamcluster* ($\overline{\text{SLP}}$). The result is due to *streamcluster*'s unique behavior. As shown in Fig. 10 (discussed in Sect. 3.5), *streamcluster* has the second highest number of unique behaviors. Because *streamcluster* exhibits such different behavior (as compared to the other programs), AutoFinity did not build an appropriate rule to handle programs like it. Additional training data (i.e., from *streamcluster* or programs which exhibit behavior like it) would, we believe, allow the AutoFinity policy to make better affinity hint selections.

Dc, *facesim*, *c_fft*6, and *c_lu* prefer a distributed affinity. These programs have large memory footprints, as evidenced by the distributed policy preference. For *dc*, *c_fft*6, and *c_lu*, AutoFinity chooses an affinity hint and program affinity which performs just as well as the distributed static policy.

In 7 out of 9 cases involving affinity sensitive programs, AutoFinity chose a program affinity which performed nearly as well or just as well as their preferred static policy. The geometric mean (last set of bars in Fig. 11) shows that AutoFinity has a slight runtime advantage over either static policy. The advantage to AutoFinity is the user does not have to manually select the proper affinity.

We also compared AutoFinity's performance against an oracle policy, built through exhaustive experimentation. On average, AutoFinity achieved 96% of the performance (ratio of the oracle and AutoFinity's geometric mean). For three programs, it tied with the oracle policy (*dc*, *c_fft*6, *c_lu*).

Varying Thread Count. Next, we evaluate AutoFinity on each program with a range of thread counts (4, 8, 16, and 24 threads)[3]. Training data was obtained

[3] *Facesim* and *fluidanimate* do not support executing with 24 threads.

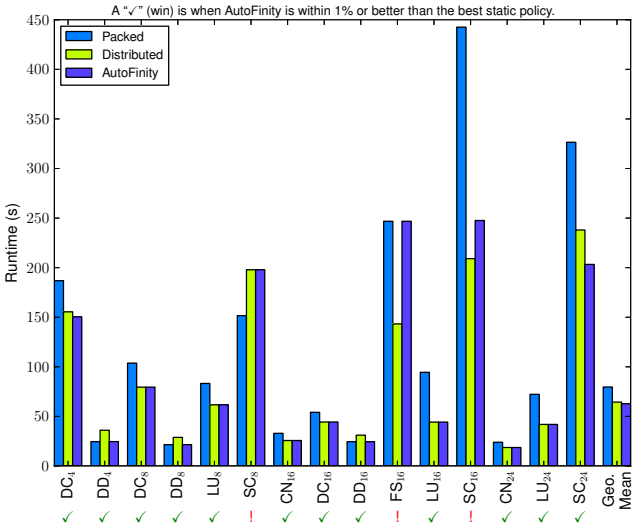


Fig. 12. AutoFinity and static policy ROI execution time comparisons across programs and thread counts for interesting cases

only for a thread count of 8. To test AutoFinity on a program, we remove that program from the training data before building the AutoFinity policy.

As previously shown, some programs have behavior that is not affected by affinity. Many programs, however, do show widely variable behavior. Affinity insensitivity is even sometimes thread count dependent. We deem the cases that have at least a 20% ROI runtime difference between their packed and distributed affinities, as "interesting cases." Interesting cases have a potential for bad or good affinity choices. Due to space constraints, we show and discuss only the interesting cases across the thread counts. There are 15 interesting cases.

Figure 12 shows the interesting cases. The x-axis shows the program's name with subscripts indicating thread count. The y-axis is the runtime of the program's ROI (lower is better). The first bar is the runtime for the packed affinity, the second bar is the runtime for the distributed affinity, and the third bar is the runtime of the program under the affinity chosen by AutoFinity.

The figure is annotated to indicate whether AutoFinity chooses an affinity that is within 1% of the best static policy. Wins (i.e., when AutoFinity is within 1%) are marked by "✓." If AutoFinity does not have a performance within 1% of the best static policy, we mark the graph with "**!**."

Dedup (DD) is an interesting program, appearing 3 times (4, 8, and 16 threads). Each time, *dedup* does best under a packed affinity. Because it is a pipeline parallel program, there is communication between threads as data is passed from stage to stage [1]. Therefore, it is natural that *dedup* would benefit from packed affinities, which cause threads to share cache space. In each case, the AutoFinity policy chooses an appropriate program affinity, and thus allowing

dedup to execute quickly even though the AutoFinity policy was not built on data obtained from observing *dedup*.

Dc also appears 3 times in Fig. 12 (using 4, 8, 16 threads). Each time, it prefers a packed affinity. If *dc* is executed with 4 threads, AutoFinity chooses an affinity hint that performs better than the best static affinity ($\overline{\text{SLP}}$).

Streamcluster (SC) is another interesting case. If run with 8 threads (SC_8), it benefits most from a packed affinity. Unfortunately, AutoFinity chooses an affinity whose runtime is similar to the distributed affinity's runtime. If using 16 threads (SC_{16}), *streamcluster* prefers a distributed affinity. AutoFinity chooses an affinity which performs almost like (in terms of execution time) the preferred distributed affinity, but is not within 1%, and therefore, it is not classified as a win. If 24 threads are used (SC_{24}), *streamcluster* still prefers a distributed affinity. AutoFinity actually beats both the packed and distributed policies and chooses an affinity hint that performs better than either ($\overline{\text{S}}\text{LP}$).

Finally, in *facesim* (FS_{16}) the packed affinity is almost 75% slower than the distributed affinity. Unfortunately, in this case AutoFinity chooses an affinity hint that behaves like the packed affinity. Additional behavior data would improve AutoFinity's hint selection.

Overall, AutoFinity chooses a good affinity in most cases (12 out of 15 cases). It even beats the distributed and packed static policies in two cases. These results across thread counts show that AutoFinity policies and affinity hints work well.

4 Related Work

Wang et al., like our work, use machine learning to choose affinities [14]. Their approach requires a special compiler or dynamic binary instrumentation to extract features. Their offline approach requires a single-threaded execution of a previously-unseen program. Fengguang et al. also use dynamic binary instrumentation to choose affinities. They use an analytical model and gathered information to predict program performance in various thread settings [9]. We do not require dynamic binary instrumentation or special training runs before an affinity can be chosen.

Tam et al. [10] propose a system that uses hardware performance counters to decide thread affinity settings. Their work requires hardware support to track data sharing on a cache line basis. Our system supports responding to runtime resource availability by providing thread-count-independent affinity hints. We present guidelines for using our techniques on new systems, by discussing feature selection, sensitivity studies, and making use of more generally available performance counters. Our techniques adapt online and are more generic, supporting multiple performance metrics (not just IPC). Evaluating AutoFinity on other performance metrics is future work.

Klug et al. [5] propose *autopin*, a tool which uses hardware performance counters to choose thread affinities. Autopin uses a set, evaluate, iterate strategy: threads are given an affinity and the program behavior is measured. Another affinity is then evaluated and eventually the tool chooses an affinity. They ultimately reach well-performing affinity settings. Radojković et al. have a similar

approach (e.g., resource monitoring, trial and error) but focus on network applications [8]. Our approaches decide on an affinity after a single sample.

AbouGhazaleh et. al use machine learning to build a runtime policy that examines hardware performance counters. However, their work is applied to embedded single-core systems, and focuses on optimizing energy-delay product [8].

Our techniques currently assume program behavior with a particular thread count does not change across invocations. Some programs have changing behavior across inputs. Techniques exist to model and predict how an input will affect a program's behavior [12]. Such techniques could be integrated into the AutoFinity flow, and enable the reuse of learned program behavior.

Under AutoFinity, programs whose behavior is not stored in the behavior database may dynamically change their program affinities (i.e., after sampling an affinity hint is chosen). For some programs, dynamic affinities can cause poor NUMA behavior. Blagodurov et al., propose dynamic techniques to improve NUMA page placement, thus solving this potential issue [2]. Terboven et al. also examined data placement and proposed OpenMP-based solutions [11].

Lee et al. discuss why a program might use different thread counts across invocations [6]. They provide an automatic system to change program thread count. This motivates the need for our work.

Although some programs' were insensitive to the affinity setting, Zhang et al. show how to modify programs to better share caches [15]. With their techniques, programs can have improved performance. The modified programs will then, we expect, require good affinity selection like that provided by AutoFinity.

5 Conclusion

Program affinity can have a large impact on performance; it is important to select the "right" affinity to maximize performance. In this paper, we present AutoFinity, a method to automatically generate a policy that can select at runtime the affinity of a previously unknown program. AutoFinity uses machine learning to derive thread-count-independent policy from training data. We evaluate AutoFinity on previously unknown programs, across a range of thread counts on which AutoFinity may not have been trained. In 12 of 15 cases where affinity has a significant impact on performance, we show that AutoFinity's selection performs within 1% of, or better than, fixed assignments that do not consider program behavior. In two of these cases, AutoFinity outperforms the fixed assignments. AutoFinity is a practical and successful solution to the problem of choosing program affinity.

References

[1] Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC benchmark suite: Characterization and architectural implications. In: Proc. of the 17th Int'l Conf. on Parallel Architectures and Compilation Techniques (October)

[2] Blagodurov, S., Zhuravlev, S., Dashti, M., Fedorova, A.: A case for NUMA-aware contention management on multicore systems. In: Proc. of the 2011 USENIX Conf. on USENIX Annual Tech. Conf., USENIXATC 2011. USENIX Assoc., Berkeley (2011)
[3] Dorta, A., Rodriguez, C., de Sande, F.: The OpenMP source code repository. In: 13th Euromicro Conf. on Parallel, Distributed and Network-Based Processing, PDP 2005 (February 2005)
[4] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The weka data mining software: an update. SIGKDD Explor. Newsl. (November)
[5] Klug, T., Ott, M., Weidendorfer, J., Trinitis, C.: `autopin` – Automated Optimization of Thread-to-Core Pinning on Multicore Systems. In: Stenström, P. (ed.) Transactions on HiPEAC III. LNCS, vol. 6590, pp. 219–235. Springer, Heidelberg (2011)
[6] Lee, J., Wu, H., Ravichandran, M., Clark, N.: Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. In: Proc. of the 37th Annual Int'l Symp. on Computer Architecture, ISCA 2010. ACM (2010)
[7] NAS Parallel Benchmarks Team: NAS parallel benchmarks 3.3.1 (2009)
[8] Radojković, P., Čakarević, V., Verdú, J., Pajuelo, A., Cazorla, F.J., Nemirovsky, M., Valero, M.: Thread to strand binding of parallel network applications in massive multi-threaded systems. In: Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, PPoPP 2010. ACM (2010)
[9] Song, F., Moore, S., Dongarra, J.: Analytical modeling and optimization for affinity based thread scheduling on multicore systems. In: IEEE Int'l Conference on Cluster Computing and Workshops, CLUSTER 2009 (2009)
[10] Tam, D., Azimi, R., Stumm, M.: Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In: Proc. of the 2nd ACM SIGOPS/EuroSys European Conf. on Comp. Systems, EuroSys 2007 (2007)
[11] Terboven, C., an Mey, D., Schmidl, D., Jin, H., Reichstein, T.: Data and thread affinity in openmp programs. In: Proc. of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem?, MAW 2008. ACM (2008)
[12] Tian, K., Jiang, Y., Zhang, E.Z., Shen, X.: An input-centric paradigm for program dynamic optimizations. In: Proc. of the ACM Int'l Conf. on Object Oriented Programming Systems Languages and Applications, OOPSLA 2010. ACM (2010)
[13] Wang, W., Dey, T., Moore, R.W., Aktasoglu, M., Childers, B.R., Davidson, J.W., Irwin, M.J., Kandemir, M., Soffa, M.L.: REEact: a customizable virtual execution manager for multicore platforms. In: Proc. of the 8th ACM SIGPLAN/SIGOPS Conf. on Virtual Execution Environments, VEE 2012. ACM (2012)
[14] Wang, Z., O'Boyle, M.F.: Mapping parallelism to multi-cores: a machine learning based approach. In: Proc. of the 14th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, PPoPP 2009. ACM (2009)
[15] Zhang, E.Z., Jiang, Y., Shen, X.: Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? In: Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, PPoPP 2010. ACM (2010)