PinIt: Influencing OS Scheduling via Compiler-Induced Affinities

Girish Mururu
Georgia Institute of Technology
Atlanta, GA, USA
girishmururu@gatech.edu

Kangqi Ni
Georgia Institute of Technology
Atlanta, GA, USA
vincent.nkq@gatech.edu

Ada Gavrilovska
Georgia Institute of Technology
Atlanta, GA, USA
ada@cc.gatech.edu

Santosh Pande
Georgia Institute of Technology
Atlanta, GA, USA
santosh.pande@cc.gatech.edu

Abstract

In multi-core machines, applications execute in a complex-co-execution environment in which the number of concurrently executing applications typically exceeds the number of available cores. In order to fairly and efficiently utilize cores, modern operating systems (OS) such as Linux migrate threads between cores during execution. Although such thread migrations alleviate the problem of stalling and load balancing yielding better core utilization, they also tend to destroy data locality, resulting in fewer cache hits, TLB hits, and thus performance loss for the group of applications collectively. This problem is especially severe in embedded servers which execute media and vision applications that exhibit high data locality. On one hand, mitigating this problem across a group of applications based on OS only solution is infeasible since OS treats applications as blackboxes and has no knowledge of applications' locality and other behavior. On the other hand, to-date, compiler optimization have focused on analysis, transformations and performance enhancement of applications in isolation ignoring the problem of optimizing performance for applications as a *group*. This is because of the infeasibility of global-compiler analysis *across* applications as well as due to the dynamic nature of inter-application interactions which is statically unknown.

To address this problem, we propose PinIt, a *compiler-directed methodology* that analyzes applications individually yet induces the operating system to mediate actions across applications to minimize harmful migrations and maintain locality. PinIt determines the regions of a program in which the process should be pinned onto a core so that adverse migrations causing excessive cache and TLB misses are avoided. PinIt first calculates memory reuse density, a new measure that quantifies the reuses within code regions which may not be migrated. Pin/unpin calls are then hoisted at the entry and exits of the region which exhibit high values of reuse density. The paper presents new analyses and transformations that optimize the placement of such calls. In an overloaded environment compared to priority-cfs, PinIt speeds up high-priority applications in Mediabench workloads by 1.16x and 2.12x and in vision-based workloads by 1.35x and 1.23x on 8cores and 16cores, respectively, with almost the same or better throughput for low-priority applications.

CCS Concepts: • **Computing methodologies → Concurrent computing methodologies**.

Keywords: dynamic compiler optimization, cache-affinity, server consolidation, process affinity

ACM Reference Format:
Girish Mururu, Kangqi Ni, Ada Gavrilovska, and Santosh Pande. 2023. PinIt: Influencing OS Scheduling via Compiler-Induced Affinities. In *Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '23), June 18, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3589610.3596279

1 Introduction

Modern servers based on multi-core systems execute several applications simultaneously. This is facilitated by the operating system (OS) that efficiently schedules multiple processes and manages shared resources. OS schedulers dynamically determine both the CPU time and/or the core that should be allocated to each workload component through time- and space- multiplexing workloads on available cores. As a result, workload threads are *continuously dispatched* on CPU cores, *preempted* after a period of time, and potentially *migrated* to another core on the same or a different socket on modern multi-core platforms. Such migrations are effected by the OS to achieve load balancing (and thus, better utilization) for

LCTES '23, June 18, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0174-0/23/06.
https://doi.org/10.1145/3589610.3596279

gaining higher performance; However, such actions could lead to the loss of a cached working set of a process resulting in cache misses, TLB misses, memory stalls, and degraded process performance [9, 13, 25]. This effect intensifies when processes are migrated by the OS to a processor on another *socket*, which has its own low-level cache. In such cases, when a migration occurs, multiple levels of the cache have to be warmed up by transferring data from a socket to another over the system interconnect. Such process migrations can be especially problematic for embedded media servers which execute applications involving multi-media and vision that exhibit high data locality and reuse. Therefore, the OS entails a tradeoff between moving a process around for better use of computing resources in terms of their utilization and for the purpose of load balancing vs. improving the cache efficiency of the process by leaving it where it is without migrating it which could leave all its data intact in its cache undisturbed. This tradeoff is a very common scenario, especially when the number of executing applications exceeds the available resources.

To maximize cache usage, applications have traditionally been optimized in isolation by a compiler [1, 4, 5, 15]. However, because of shared resources and the tradeoff mentioned above, modern applications must be optimized as an *ensemble* executing together; unfortunately, performing global compiler analysis across applications is not only infeasible but is unrealistic due to lack of knowledge about the dynamic concurrency of individual applications cache-heavy regions. This work proposes a methodology that limits analysis to individual applications yet induces the operating system to mediate actions across applications that minimize harmful migrations and maintain data locality in the cache. The problem of cache misses is well known to schedulers in OSes such as Linux, which strive to maintain the natural affinity of processes in the execution queue. By scheduling a process on the processor on which it had executed before, the scheduler attempts to maintain cache affinity. When OS must perform migration for the purpose of load-balancing, the process is scheduled onto a nearby free processor [24]. Such a migration to a nearby free processor will not affect the performance if a process is executing in a program region with very few access to memory or one which exhibits very little data locality. However, if that region is memory intensive and the data locality/reuse is significant, the effect can be quite adverse [12]. To avoid the effects of migration, hardware resources such as memory can be partitioned such that certain workload components are always executed on a dedicated CPU by pinning (processor affinity). However, pinning the workload components permanently to a given core takes away the scheduling flexibility to load-balance in an overloaded environment and thus, adversely affects the aggregate performance of the workload. Therefore a solution that pins a process to a core *only* in a region of heavy reuse and also allows free process migrations otherwise, achieves the best of both worlds.

We developed PinIt – a compiler-assisted technique, which (1) automatically identifies critical program regions with high cache requirements and sensitivity, (2) instruments the program around critical regions to generate *pinit* requests to the underlying scheduler, and then (3) uses this information to dynamically inform the underlying OS scheduler of potentially harmful thread migrations which are then prevented. The outcome is a scheduling framework that migrates processes for resource management only when the cache locality is not hurt due to migrations. In developing PinIt we make the following technical contributions:

- A new metric Memory Reuse Density (MRD) that characterizes cache affinity/property of a program in terms of its locality.
- Compiler transformation/optimization to insert/hoist PinIt calls minimizing overhead – number of calls and avoiding pinning of non-critical regions.
- Implement and evaluate PinIt framework and demonstrate practical and effective scheduling of workloads utilizing the above.

The utility of this work is pronounced for those workloads that exhibit heavy memory reuse and locality plays a dominant role, prominent examples of which are media and vision based applications. Media servers typically house media applications serving thousands of users. Users stream media content online and expect low latency. The stored media must be decoded without causing jitters and delays and hence *decoders* are on a *critical path* and must execute at a *higher priority*. On the other hand, when the media content is either uploaded by users or media enablers such as Netflix, the media can be encoded offline and hence media *encoders* could be executed at a *lower priority*. Similarly, video and images from many sources are typically streamed to a server for analysis through computer vision applications with the expectation of low latency and high throughput from these applications and would be thus executed at higher priority. We evaluated benchmarks from the mediabench and san-diego vision suite (sd-vbs). For effective work consolidation, we classify the jobs into high -and low -priority applications as in [27]. We run these set of high- and low-priority processes such that the total number of processes is more than the number of processors, which is necessary to exercise scheduling scenarios effectively. Compared to low-priority processes, which are limited to a set of encoders, high-priority processes consist of a set of decoders and vision workloads that are in the critical path. In such an overloaded environment, compared to priority-CFS, PinIt speeds up high-priority applications in mediabench workloads by 1.16x and 2.12x and in vision-based workloads by 1.35x and 1.23x on average on 8cores and 16cores, respectively, while completing the low-priority batch 23% faster on 16 cores in mediabench and almost within the same time for other workloads.

2 PinIt Framework

The Pinit framework is best described by the goals that can be summarized as follows:

1. To identify application's regions with large cache reuse where scheduling actions involving process migration can have the most adverse effects.
2. To determine the placement (hoisting) of pin/unpin directives to the underlying scheduler such that intervals within which an application remains pinned to a core is minimised or conversely the scheduler's freedom to migrate applications for load balancing is maximized. The secondary goal is to determine the placement of pin/unpin calls such that their frequency of scheduler interactions is minimized in order to minimize the total overheads of the calls and the frequency of the interruptions they may cause.

In order to meet these goals the framework must strive to achieve the following three objectives.

Maximize reuse density. With respect to the first goal, the sections of code that exhibit large data reuse typically are those loops that can access the same line of memory multiple times within a short span of execution; thus, the migration of a process when executing such loops during such spans is very harmful. Hence, for pinning to be most effective PinIt must find those loops that exhibit high memory reuse. Thus, *one key goal of PinIt is to identify and maximize the density of reuse (or the average reuse exhibited per statement).* PinIt achieves this by finding loops with high reuse per instruction and by splitting the loops if possible by removing statements without memory reuse or dependencies. Section 4.2 gives further details of loop splitting transformation.

Minimize runtime overheads. In order to preclude the scheduler from migrating a process with high reuse density, our framework requests the OS scheduler that this loop region be pinned. Since pin/unpin calls to the OS involve overhead, they must be carefully hoisted. The pinning calls should be ideally moved outside the outermost loop encompassing the pinned regions. In addition, if multiple pinned regions are neighbors, these regions can be merged thus removing unnecessary (intervening) pin/unpin calls.

Maintain scheduler flexibility. In our system, we avoid pinning multiple processes to the same core to avoid cache contention and core sharing which could lead to a slowdown. Restricting only one process to a core also ensures that no other core is starved when processes are waiting to run on a busy core. Based on the above discussion, the overall flow of the above phases is pictured in Figure 1.

3 PinIt Analysis

We define a few terms for developing the analysis necessary for identifying the critical loops that should be pinned.

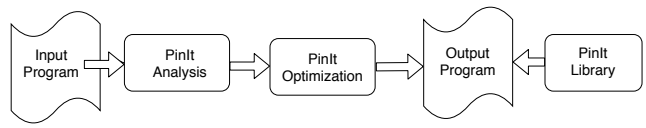


Figure 1. Steps to minimal pinning in PinIt.

3.1 Memory Reuse Density

In order to determine regions with high memory reuse, PinIt analyzes the dependency between the memory accesses of instructions. Consecutive accesses to the memory that was accessed earlier in the loop's iteration space result in reuse. The distance or number of loop iterations after which the memory was reused is the reuse distance which is also equal to the dependence distance. We quantify memory reuse through reuse distance. If the reuse distance between memory accesses is higher, then the penalty resulting from migration in the middle of such accesses is higher since the number of cache misses is equal to the reuse-distance of the accesses. For example, if a process is migrated while executing a statement, $a[i] = a[i-n]$, assuming the size of each element in a is equal to the cache-line size, the subsequent accesses to $a[i-n]$ from iterations i to $i+n$ result in n cache misses assuming the migration takes place during the iteration i. Thus, by considering the reuse distance, memory reuse (MR) is calculated as follows:

$$MR = \Sigma_{i=0}^{n} rd_i, \tag{1}$$

where

$$rd_i = \begin{cases} reuse\_distance, & \text{if } reuse\_distance > 0 \\ 1, & \text{otherwise} \end{cases} \tag{2}$$

where the respective *reuse_distances* are calculated with respect to all other instructions in the loop and Memory Reuse (MR) is the summation of these values over all memory instructions in the loop. If the size of each element in a is not equal to the cache-line size, then MR depends on the number of elements that can be held in a cacheline which is equal to

$$\#Elements_{\$line} = \frac{CachelineSize}{Sizeof(a[i])} \tag{3}$$

and each rd_i is equal to

$$rd_i = \begin{cases} \lceil \frac{reuse\_distance}{\#Elements_{\$line}} \rceil, & \text{if } reuse\_distance > \#Elements_{\$line} \\ 1, & \text{otherwise} \end{cases} \tag{4}$$

For example, if four elements of a can be held within a cacheline, and if the *reuse_distance* is five, the MR for the memory accesses depends on two different cachelines and similarly, if only half an element of a can be held within a cacheline, then for a *reuse_distance* of 1, two cachelines must be accessed as captured in Equation 4. For simplicity, all discussions below assume the size of each element is equal to the size of the

cacheline. Note that the instructions itself when migrated incurs a loss within the i-cache. Every new instruction accessed within the loop incurs one i-cache miss because of migration. If the body of the loop entails a large number of instructions exceeding the i-cache capacity then the previously cached instructions are evicted resulting in more i-cache-misses, however, these are not due to migration and would have occurred, regardless. Modern i-caches have significant capacity and smart prefetching policies which minimize effective i-cache misses. Due to the above reasons, in this analysis we do not include i-cache misses.

The total memory reuse, however, does not capture the total cache penalties due to migration. In other words, loops can have a large number of unique memory accesses that can result in a higher total reuse across them but could still exhibit very little individual reuse for a given access. Such loops are classified as streaming loops. Pinning such streaming loops is not of much value because these hardly require any cache misses upon process migration due to low individual array element reuse. This contrast is clearly explained below in the following Fibonacci and summation examples.

```
for(int i=0; i < N; ++i)
  a[i] = a[i-1] + a[i-2];
```

Listing 1. Fibonacci

```
for(int j = 0; j < M; ++j)
  for(int i=0; i < M; ++i)
    b[j] = b[j] + b[i];
```

Listing 2. Summation

In Fibonacci (Listing 1), $a[i]$, $a[i-1]$, and $a[i-2]$ access an individual array element only three times. If the process is migrated while executing this loop (at iteration i), two array elements must be transferred between the caches for access $a[i-2]$ and one element for access $a[i-1]$ (if the loop was migrated between the execution of a[i-1] and a[i-2])[1], which hardly incurs any cost. However, in Summation ((Listing 2), $b[i]$ leads to M accesses of each array element, and every iteration touches M array elements. If this loop is migrated, then in the worst case, M cache elements must be transferred. To account for the penalty during migration, we define "Unit Memory Reuse (UMR)," which is the memory reuse generated by each unique access. i.e.

$$UMR = \frac{Memory\_Reuse(MR)}{Unique\_accesses}. \quad (5)$$

Unit memory reuse (UMR) is $(4 * N)/N$ or 4 for Fibonacci and $(M * M)/M$ or M for summation. We now co-relate the migration penalty to the Unit Memory Reuse (UMR).

[1]The analysis is worst case analysis and considers migration could happen at any and all points

In the absence of any optimization and for large elements where the register allocation cannot be done across array elements, the code produced for Fibonacci sequence generator in Listing 1 will do the following:

1. Prefetch a in cache
2. Load $a[i-1]$ into a register
3. Load $a[i-2]$ into another register
4. Add the two
5. Store the result into $a[i]$

Consider the above instruction sequence; migration during the execution of a certain instruction could be worse than others. Migration could occur at any point during the above five steps. Worst case occurs if the migration takes place at the beginning of the first step, then all four accesses a, $a[i-1]$, $a[i-2]$, and $a[i]$ requires the memory to be transferred into the new cache. Migration at subsequent steps will require one less access to the cache, i.e., migration at step 2, step 3, step 4, and step 5 will require 3, 2, 1, and 1 new cache access for completing the above steps. Also since this is a streaming loop, as discussed before, the subsequent iterations will require one new access for each of $a[i]$. Assuming that the migration occurs at each of the above steps, a total of 11 misses occur during 4 migrations or on an average a total of 2.5 cache misses could occur during each migration; in other words, pinning this loop will save 2.5 misses on an average. On the other hand, similar analysis for summation code(Listing 2) shows a saving of M cache misses(migration occurring after each reuse of $b[j]$ in inner loop and there are M such reuses). Thus, pinning the summation loop will save M cache misses in the worst case and thus, pinning summation loop will be advantageous over the Fibonacci loop in case one has to make a choice in terms of pinning decision on a given core.

Apart from memory reuse, another factor which plays a critical role in terms of pinning one loop vs. the other relates to the size of the loop bodies. The ratio of Unit Memory Reuse (UMR) to the total number of instructions in the loop body is defined as ***Memory Reuse Density (MRD)*** (Equation 6), and is a measure of the trade-off between reuse and the size of the regions that participates in the reuse. As indicated earlier, for legal execution of loops, not only the statements that directly participate in reuse must be encapsulated in the same loop nest but also all the other statements that exhibit the dependencies with rest to these must also be encapsulated in the same loop nest. Due to this reason, one must account for the extra (dependency carrying) statements that are included in the loop nest which is captured by the MRD measure. Given the same UMD values, compared to regions with larger loop bodies (and correspondingly lower reuse densities), regions that smaller loop bodies (and the ones exhibiting higher reuse densities) are more desirable candidates for pinning since they are likely to place lesser

restrictions on the scheduler flexibility (due to faster execution of smaller loop bodies). This motivates the calculation of MRD as follow:

$$MRD = \frac{UMR}{instructions} \tag{6}$$

The following code excerpt will help illustrate the key idea behind MRD.

```
1 void foo(int M, int N)
2 {
3     ...
4   for(int j = 0; j < M; j++)
5   for(int i = j; i < (j+N); i++)
6   {
7       A[i] = A[i-1] + A[i-2];
8       B[i] = B[i-3] + B[i-2];
9       C[i] = C[i-5] + k;
10      x = i + 2 + x;
11      y = n + i + y;
12      z = z+1;
13      m = m--;
14      k = i + k;
15  }
16  ...
17 }
```

Listing 3. Function foo()

Within the inner loop, the first statement has three memory accesses in $A[i], A[i-1], A[i-2]$. The write to $A[i]$ is read in the next iteration as $A[i-1]$ and the iteration after the next in $A[i-2]$. From Equation 1, the memory reuse is 1 for $A[i-1]$ and 2 for $A[i-2]$ for accessing an element accessed by $A[i]$. The memory reuse of $A[i-1]$ is 1 for accessing an element accessed by $A[i-2]$. Again note that the reuse between $A[i-1]$ and $A[i-2]$ is considered to account for a migration between these accesses. The total MR from this statement within the inner loop is $N + N * 2 + N$, where N is the iterations of the inner loop. Hence, for the loop nest with outer loop iterations M, the MR is $M*[N+2*N+N]$. Similarly, the second statement contributes $M * [3 * N + 2 * N + N]$, and the third statement adds $M * [5 * N]$ to the total memory reuse of the loop nest. In other words, the total memory reuse MR of the loop nest is $15 * M * N$. This reuse is generated by $3N$ unique accesses. Note that, the total accesses are still $3 * N * M$, that is $3N$ array elements accessed M times, but unique accesses are only $3N$. By substituting these accesses in Equation 5, UMR is $15 * M * N/3N = 5 * M * N$. The loop contains ten instructions and by substituting the values in Equation 6, we find memory-reuse density equal to $5 * M/10$. Examining the dependencies, one can notice that statements 10 through 13 can be moved out of the loop, reducing the number of instructions to only 4 and improving MRD to $5 * M/4$, *such a loop with higher MRD will execute faster than the original loop, will be pinned by the scheduler for a smaller amount of time giving it more flexibility to perform better load balancing across cores.* MRD drives the loop optimizations for this reasons as explained in section 4.2

4 PinIt Optimization

PinIt optimization focuses on determining the location where pin/unpin calls should be hoisted to reduce call overheads and on transforming loops to maximize their memory reuse density (MRD).

4.1 Call Hoisting

The pin (pin/unpin) calls are hoisted at the outermost loops. Hoisting prevents repeated calling of pin/unpin functions when present inside a loop. However, some of the pin/unpin calls could still remain inside some inter-procedural-external loops, that is, the call site is located inside the caller's loop, the pin/unpin calls being hoisted at the outermost loop level of callee. With the use of call graphs, such external loops can be determined, and pin/unpin calls can be moved outside the external loop. Because the external loops can reside in various files such as header files and libraries, hoisting is accomplished after the entire program is linked. After calculating the intra-procedure hoist point, the reuse analysis is extended interprocedurally. The reuse density (MRD) is recalculated at the inter-procedural hoist point to check if the external loop nest has enough memory reuse to negate the loss of scheduler flexibility. We calculate unit memory reuse (UMR) and count the instructions along the path of the external hoist point enclosing several procedural calls. Using these values, we calculate the new MRD at the hoist points. At runtime, the new MRD is checked against RDT (reuse density threshold), a threshold that decides if a loop should be pinned or not. RDT's role is to avoid overpinning of short loops and is explained in details in section 5.

Formally, let f_p be a function originally containing pin calls and I_p the set of instructions within f_p; f_p is called by a set of functions, $F = \{f_1, f_2....f_n\}$. F_L, which calls f_p inside a loop, is a subset of F; that is, $F_L = \{f_i, f_j....f_m\} \subset F$, and I_i is the set of instructions corresponding to the loop in $f_i \in F_L$. If MRD_{f_p} is the reuse density of the pinned loop in f_p, MR_{f_p} is the memory reuse inside f_p, and MR_{L_i} is the reuse in external loop, L_i in $f_i \in F_L$, the pin call must be hoisted outside the loop if

$$\frac{MR_{f_p} + MR_{Li}}{I_p + I_i} > RDT. \tag{7}$$

For functions in F/F_L, in which the calls to the function f_p are not inside a loop, the pin/unpin calls must surround the call to f_p if

$$MRD_{f_p} > RDT \quad \& \quad F_L \neq \emptyset, \tag{8}$$

where RDT is the "Reuse Density Threshold". The steps for minimizing pinning and hoisting pin calls are summarized in Algorithm 1.

Algorithm 1 Pinning Algorithm

procedure PinLoop
for each Outermost loop L ∈ Function F:
  $N_L \leftarrow$ Number of Iterations of L
  Memory_Reuse(MR_L) =
  Σ Memory_Reuse of Subloops(L)+ Σ(Memory_Reuse)
  $MRD_L = \frac{MR_L}{Instructions}$
  Pin Loop L If $MRD_L > RDT$
End for
procedure Hoisting
for each Pinned Loop L ∈ Function F:
  $RemovePin \leftarrow False$
  $U \leftarrow u_1, u2, ..u_n$ s.t each u_i calls F
for each $u_i \in U$:
  if F is within Loop l_i of u_i **then**
    $RemovePin \leftarrow True$
    Recalculate MRD for l_i with F
    Hoist Pin Outside li
    Pin Loop L If $MRD_L > RDT$
End for
  if $RemovePin == True$ **then**
    if $MRDofF > RDT$ **then**
      Hoist Pin outside non-Loop calls of F, $u_j \in U$
    Remove Pin calls from F
End for

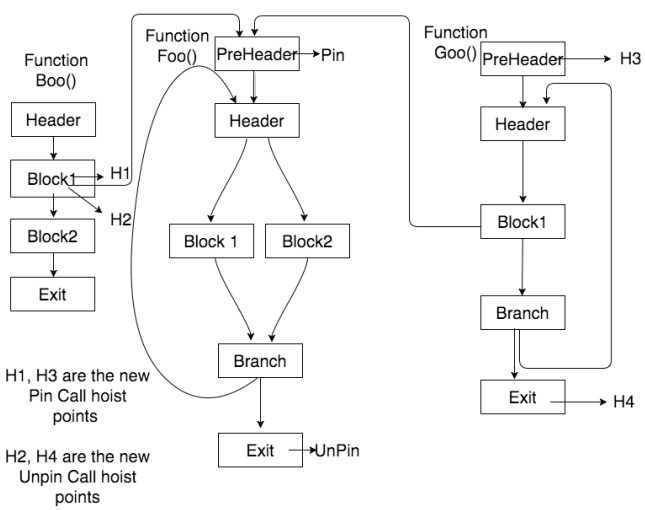


Figure 2. hoisting outside external loops

As an illustration, Figure 2, the loop in function Foo() is initially pinned at the pre-header and unpinned at the exit blocks of the loop based on the intra-procedural analysis. However, inter-procedurally, function Foo() is called in procedure Boo() and within a loop in Goo(). To prevent multiple system call overhead, PinIt by using the information from the call graph hoists the pin/unpin calls at points H3 and H4 in Goo and at points H1 H2 in Boo. At runtime, the loop is pinned through these hoist points only if equations 8 and 7 are satisfied. Note that in function Boo(), calls are hoisted surrounding the call for Foo(), and in Goo(), the calls are hoisted outside the loop.

4.2 Loop Transformation

Pinning a large loop with low memory reuse not only decreases the gain from pinning but also restricts the scheduler from using resources efficiently. To avoid this, we pin the loop only if the reuse density is above a certain threshold. Furthermore, to increase the MRD of a loop, we carry out loop splitting, so that only dependent instructions containing high memory reuse form one loop nest and the rest another.

4.3 Putting It Together

The compiler analysis shown in Figure 1 first calculates UMR followed by MRD for each loop nest. The pin/unpin calls are then hoisted inter-procedurally. After splitting any loop to improve MRD, the framework determines a pin threshold as discussed below. The pin call at runtime checks if the MRD is greater than the threshold before pinning the loop.

5 Pin Threshold

To determine the reuse density threshold value that benefits all the co-executing programs, we chose to cluster the MRD values using kmeans. We first calculate the MRD values for all the loop nests using profile data of the programs to be co-executed. Note that such profiles of applications are commonly used for optimized execution in servers; the profile data useful for MRD calculations involves loop bounds which are functions of problem sizes (image sizes, resolution, etc). We derive k means that lies within the range of MRD value of different loops of a program. We start with $k = 1$ and increment k till we find a mean MRD value that is within the range of MRD values for the application. For example, when $k = 1$, K-means could lead to a first mean such that we might not have any loop in some benchmark with an MRD value higher or equal to this first mean. We increment k to find the second mean, if the mean is still greater than the maximum MRD, we increment k and repeat until we find a mean value that lies within the range of MRDs of the applications. We use this mean value as the **Reuse Density Threshold** for the application. Note that in many hosted servers, especially embedded media servers, the group of programs to be executed on the server is pre-determined and fixed through configuration files. This forms a batch of co-executing applications and is used for finding RDT.

6 Pinit Runtime

In order that too many processes do not get pinned to the same core and slow each other down (due to time multiplexing), we perform a runtime optimization. Pin calls are

implemented using the cpu_schedsetaffinty system call in a shared library that maintains a shared bit-mask, a mask of reserved processors in the system. Shared bit-mask tracks the CPU cores to which processes are currently pinned; the bit mask is set when a process is pinned to a respective CPU core and is unset when no process is pinned to that core. The library first invokes *get_cpu* system call which returns the CPU mask on which the process is currently executing. PinIt always attempts to pin a process where it is currently executing to preserve its data locality and affinity. The library then checks if the corresponding CPU core in the shared bit-mask is free for pinning. If the process can be pinned, the mask bit is atomically set in the shared bit-mask and any subsequent requests to the same CPU are denied. In case the core is unavailable for pinning the process, the pin call attempts to find a free core on the same socket following the above process in order to preserve last level cached data of the process. Unpin call works by resetting the CPU core mask bit in shared bit-mask to allow other processes to claim the core for pinning.

The shared library in the pinit framework consists of four exposed functions–$pininit, pin, unpin, andpinfree.$ $pininit$ is inserted at the entry block of the main function and $pinfree$ is instrumented at all the exit blocks of the program. We track the shared processors with the help of *shm* semantics provided by Linux and use the $cpu\_sched\_setaffinity$ system call to pin a process inside the pin function. The average overhead of each call is in the order of few microseconds. The applications were instrumented with passes written in LLVM 3.8 and compiled with O3 optimization.

7 Experiments

7.1 Experimental Setup

Testbed: We used Linux OS running on two different Intel Xeon Processor machines, one with 8 cores and the other with 16 cores with configurations as mentioned in Table 1 in a carefully controlled environment. Specifically, to prevent the effects from warmup, we cleared shared resources such as the cache before every run and disabled hyper-threading. In such an environment, the machine was overloaded by 50%, that is, the number of processes scheduled was 1.5 times the number of processors. Consequently, in 8 core machine we ran 12 processes, and in 16 core machine we scheduled 24 processes for execution. Among the processes in the load, 100% processes was of high-priority and the rest overloaded 50% was of low-priority. In other words, 8 high-priority applications and 4 low-priority applications were scheduled on 8 core machine, and 16 high-priority and 8 low-priority applications were executed on 16 core machine.

Benchmarks: To demonstrate the usage of Pinit, we used Mediabench II benchmark suite [19] and san-diego vision based benchmark suite (sd-vbs) [26]. MediaBench and sd-vbs represent real-world media and computer vision workloads very typical of embedded software. Such workloads are typically served near the edge via the fog compute servers that play a critical part of IoT and Edge based infrastructure. Mediabench are large benchmarks with between 7602 to 35162 LOC (Lines of Code) and execution time ranging from 2 to 12 seconds with complex and multiple loops in each benchmark. The seven sd-vbs benchmarks encompasses around 20 different kernels that form the core of vision processing including some prominent full applications such as sift and stitch [23]. All the benchmarks are written in C/C++. The applications were classified into higher- and lower-priority applications. As mentioned previously, media decoders are on the critical path of Media Servers and demand higher-priority compared to the media encoders. Hence for Mediabench, we use 12 different combinations of benchmarks from the set of decoders– djpeg, h263dec, h264dec, and mpeg2decode– as high-priority mixes along with all the encoders as low-priority set. With sd-vbs as high-priority set, we used the same set of low-priority applications consisting of encoders, that is, all the experiments entailed the encoders in mediabench executing simultaneously as low-priority set coupled with each Mix shown in Table 2. Different applications used have different characteristics in terms of the loops that are pinned and their execution times. While we demonstrate the effectiveness of PinIt on different hetero-and-homogeneous mixes of Mediabench, we only consider homogeneous mixes of sd-vbs. Different mixes of the above applications allow us to stress test the scheduling behavior of PinIt with different overlaps of pinned and non-pinned regions. In the homogeneous mix in high-priority set, the pin phases of all the tasks overlap thus pressurizing the demand of resources, while the heterogeneous mixes provide options for the scheduler to flexibly migrate processes as necessary. Altogether these mixes of such diverse applications push the scheduler to capture various scenarios thus testing the robustness of PinIt.

Table 1. Configuration of experiment machines

Features	8 core	16 core
Core count	8	16
CPU Base frequency	2.40 GHz	2.2 GHz
CPU architecutre	Nehalem	SandyBridge
Sockets	2	2
Core/socket	4	8
L1 Cache	32 KB	32 KB
L2 Cache	256 KB	256 KB
LLC Cache	8 MB	20 MB
Linux kernel	4.5.0	4.5.0

7.2 Scheduler Setting and Experimental Goals

The process when pinned must be non-preemptive, otherwise the cache of the pinned process will be polluted by

Table 2. Mixes of high-priority applications in MediaBench

MixBench	High Priority Set
Mix1	djpeg h263dec h264dec mpeg2decode
Mix2	h263dec h264dec mpeg2decode
Mix3	h264dec mpeg2decode
Mix4	mpeg2decode
Mix5	djpeg
Mix6	h263dec
Mix7	h264dec
Mix8	djpeg h263dec h264dec
Mix9	djpeg h263dec
Mix10	djpeg h264dec
Mix11	djpeg mpeg2decode
Mix12	h263dec mpeg2decode

the process that pre-empted the pinned process. The default CFS scheduler pre-empts pinned processes and is not suited for PinIt. In the function call PinInit, the scheduling for the high-priority applications is changed from CFS to FIFO, which prohibits pre-emption of pinned process except only when an IO operation is requested by the process. The IO request can be very time consuming to skip the precious cpu cycles. Only the high-priority applications are scheduled non-preemptively. The low-priority applications are scheduled using the default CFS in PinIt. We compare PinIt against priority CFS that differentiates the high- and low-priority applications through nice values. The nice value was set to -20 (highest) for high-priority applications and default 0 for low-priority applications. In the red-black tree based run-queue of CFS, the nice value plays a part in selecting the next application for execution. Lower the nice value, higher the priority for an application to be picked up for execution.

7.3 Experimental Results

We demonstrate the speedups gained by PinIt over priority CFS scheduler. Each overloaded batch of mix was run for 12 iterations with each framework – PinIt and priority CFS, to collect various results of the applications in the batch. The speedup reported in the figures is normalized to that of the priority CFS scheduler.

Table 3. Average Latency and batch throughput for 16threads(8threads) of PinIt normalized to priority CFS

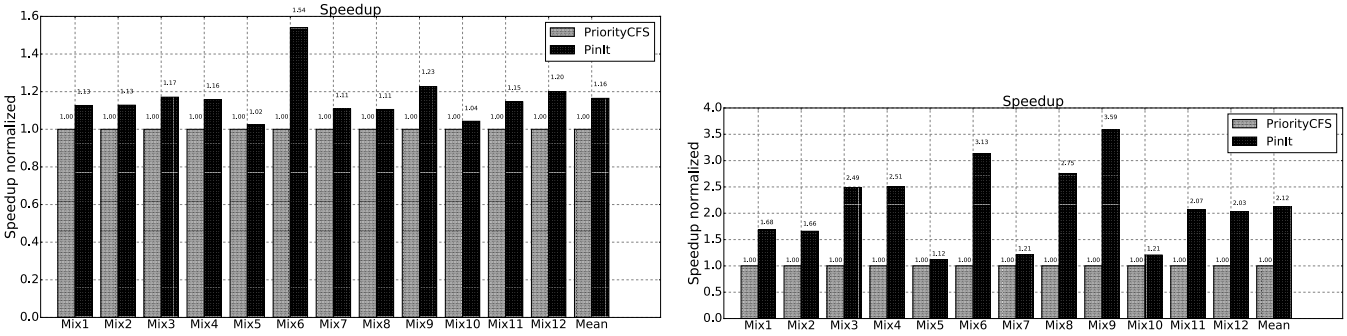
Benchmark Suite	Latency	Throughput
Mediabench	45% (13%)	1.23x (1.01x)
sd-vbs	8% (19%)	0.98x(1x)

7.3.1 Mediabench. The average speedup of high-priority applications in PinIt compared to CFS priority scheduling is 16% in 8 cores as shown in Figure 3a and 2.12x in 16 cores as shown in Figure 3b. Although, the low-priority applications in each framework were scheduled with default CFS scheduling mechanism, scheduling decisions for high-priority applications directly effect the completion time of low-priority applications. For each framework, we also show the completion time of the batch normalized to that of the priority CFS scheduling. In 8 cores, PinIt managed to complete the batch as fast as priority CFS scheduling, and in 16 cores, the ample time saved in completing the high-priority job was used in completing the low-priority jobs improving overall throughput by 23% as shown in Table 3. To demonstrate that the increase in the overall throughput of the high-priority applications is not at the expense of the latency of each application, we calculate the average latency of applications in terms of execution time for the high-priority mixes for each framework. PinIt compared to priority-CFS was faster on average by 13% on 8 cores and by 45% on 16 cores as shown in Table 3.

We compare the system behavior when using PinIt versus the priority CFS by reporting cache misses, page faults and cpu migrations collected using Perf tool. PinIt relies on avoiding dubious migrations than can lead to superfluous cache misses as well as page faults. These three factors together contribute to the speedup of the application. Migrations can not only increase data misses but also can result in inefficient use of cpu cycles. Cache misses and page faults directly affect the execution of an application. In NUMA machines, such as the ones used in our experiments, the cache-misses because of migration from one NUMA node to another can be more drastic than intra-NUMA node migration. In 8 cores, for entire batch, on average , cache misses are reduced by 6% (Figure 4a) , page faults (minor and major together) are decreased by 17.4%, and migrations are reduced by 55.6% in PinIt. Also in 16 cores, for whole batch, on average, cache misses are reduced by 25% (Figure 4b), page faults are reduced by 15%, and migrations are cut down by 63%.The stalls in PinIt are almost the same as priority CFS.

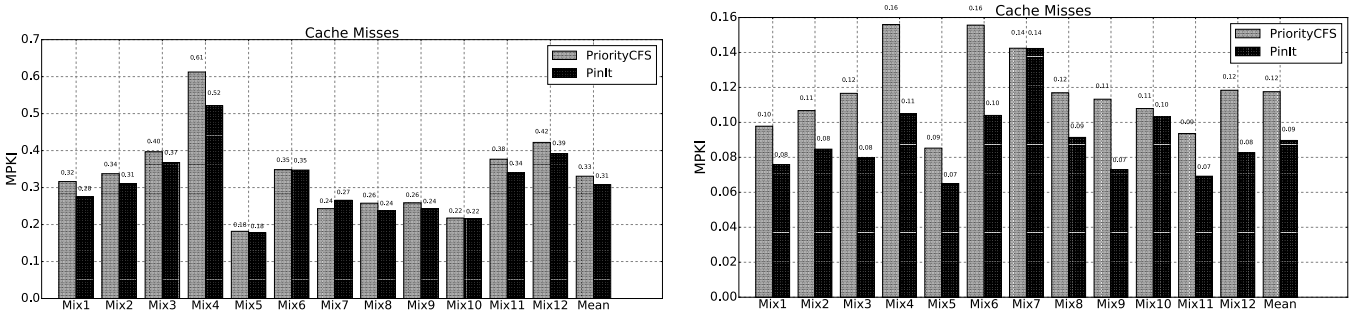
7.3.2 sd-vbs. The average speedup of high-priority ones in PinIt was 1.35 times that in priority CFS on 8 cores as shown in Figure 5a, and 1.23 times on 16 cores as shown in Figure 5b. On 8 and 16 cores, PinIt completed the batch as fast as priority CFS as shown in Table 3. Latency wise, PinIt compared to priority-CFS was faster on average by 19% on 8 cores and by 8% on 16 cores as shown in Table 3.

On 8 core, for the entire batch, on average , MPKI is not reduced, however, PinIt memory reuse benefits manifests as TLB hits which is reduced page faults (minor and major included). Note that minor page faults indicate TLB misses. On 8 cores page faults are decreased by 16.4% and migrations are reduced by 23.3%. On 16 cores, page faults are reduced by 11% and migrations by 25%. Although in PinIt cache misses do not show a considerable difference with priority CFS, fewer migrations in PinIt compared to priority CFS avoid



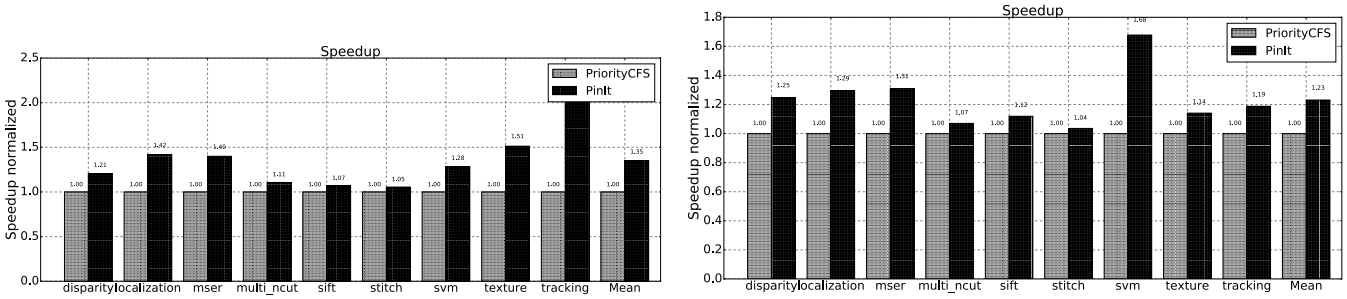
(a) 8 core **(b)** 16 core

Figure 3. Mediabench: speedup of high-priority applications normalized to priority CFS vs PinIt



(a) 8 core **(b)** 16 core

Figure 4. Mediabench: cache Misses (MPKI) in priority CFS vs PinIt



(a) 8 core **(b)** 16 core

Figure 5. sd-vbs: speedup of high-priority applications normalized to priority CFS vs PinIt

the unnecessary halting of a process, migrating to a new processor, and resuming the process overheads thus achieving speedup. The speedup achieved in sd-vbs can be attributed to the reduction in page faults (TLB misses) and migrations.

7.4 Experimental Analysis

PinIt performance. PinIt improves the performance of the applications by reducing all or some of the factors among cache misses, page faults (TLB misses), and migrations. In Mediabench, all three factors reduced thus improving performance phenomenally, and in sd-vbs page-faults (TLB misses) and migrations reduced.

Non-preemptive scheduling: Although PinIt benefits from non-preemption, it outperforms non-preemptive scheduling by as much as 28% in Mediabench and 26% in sd-vbs.

Non-preemptive affinity: We observed that PinIt when compared to complete non-preemptive affinity fairs up to

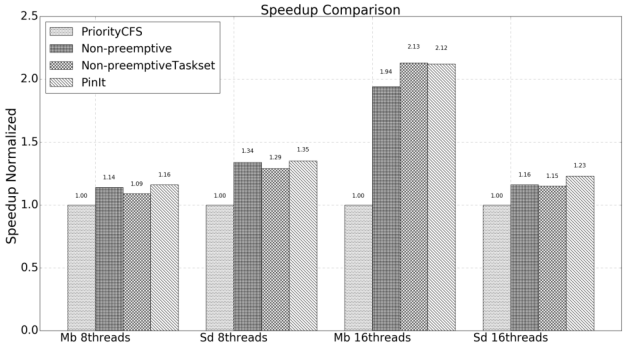


Figure 6. Average normalized speedup: Pinit extracts the best of scheduler flexibility and processor affinity

16% faster in Mediabench and 51% faster in sd-vbs. Few applications in Mediabench, however, gained higher benefits with complete non-preemptive affinity than pinning fewer areas denoted by PinIt. We observed that PinIt missed gains mainly from file read operations that were pinned in complete non-preemptive affinity. Figure 6 compares PinIt with purely non-preempitve scheduling and complete non-preemptive affinity. PinIt*extracts the benefits of scheduler flexibility and processor affinity as and when necessary, thus mostly performing better than both just non-preemptive scheduling and non-preemptive whole process pinning.*

Fairness. The environment in which PinIt is used entails a set of high-priority applications equal to the number of cores and a low-priority set simultaneously executed as a batch. Since each high-priority application is executed by a core in a non-preemptive manner, no high-priority application starves for processing time, and from the above results, we can safely conjure that the low-priority job takes almost same time or less to complete in PinIt as in priority CFS. The standard deviation in the execution time is reduced in PinIt because of reduced migrations due to pinning. That is, the repeatability of the high priority applications is increased because random behavior caused by non-deterministic migrations is reduced.

Loop Splitting: The hoisting optimization hoisted the pin calls to inter-procedural outermost loops, thereafter Loop Splitting pass did not find loops to split for all benchmarks except djpeg. The differences in djpeg before and after loop splitting is shown in Table 4. Note that djpeg has two pinned loops and the MRD of one loop was increased with unchanged number of pin calls.

Table 4. Djpeg before and after loop splitting optimization

Metric	Before	After
MRD	13.97	17.55
Speedup	17.9%	20.9%

8 Related Work

Bubble-up [18] generates a QOS versus memory pressure sensitivity curve to co-locate two programs together. [17] attempts to assign VCPU for entire time to a CPU based on the history of execution. However, it does not dynamically pin/unpin sections or regions of code. Autopin [14], an offline tool similar to valgrind, finds the best thread-to-core mapping through an offline iterative process. To overcome disadvantages of offline profiling, authors of Bubble-up developed Bubble-flux [27], in which low-priority processes are simultaneously executed along with high-priority processes only when QOS of high-priority processes do not drop. We use a similar environment of a set of high- and low-priority processes in our setup.

Thread tranquillizer [22] reduces the side-effects of process migration via scheduling and memory allocation techniques. [8] uses cache space isolation technique and [7] proposes scheduling policies to reduce cache contention for time constrained tasks executing on shared multi-core systems. [10] context-switches only when executing a region that has minimal amount of state to be saved found through live register analysis. [20] proposes compiler-assisted application specific demand-paging for embedded systems with flash memory. [3] uses reuse distance analysis to inform the processor about the cache in which the memory is likely to be found. [16] schedules the process to the appropriate processor, whose cache has the memory region by building a cache map. Works in [6], [2], [21], [11] present loop optimization techniques for improving data reuse by the loop and are orthogonal to our work. We have demonstrated the merits of our work in an overloaded environment, a set of high- and low-priority processes executing together as in Bubble-flux [27]. None of the works to the best of our understanding *influence* the scheduling decisions of the operating system scheduler non-intrusively towards processor affinity to minimize the harmful effects of process migration on latency, which is critical for media applications executing on embedded servers.

9 Conclusion

To achieve optimal pinning, PinIt offers a solution through compiler analysis of memory reuse and carefully balancing the sizes of the regions pinned by relying on a new metric called Memory Reuse Density (MRD) for deciding if the loops must be pinned. In an overloaded environment, PinIt speeds up high-priority applications in Mediabench workloads by 1.16x and 2.12x and in vision-based workloads by 1.35x and 1.23x on 8cores and 16 cores, respectively, on average while completing the low-priority jobs in almost the same time as priority-CFS thus demonstrating the optimization of multiple programs as an ensemble as against in isolation without undertaking infeasible inter-application analysis and without modifying the OS.

References

[1] Jennifer M. Anderson and Monica S. Lam. 1993. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, USA) *(PLDI '93)*. ACM, New York, NY, USA, 112–125. https://doi.org/10.1145/155090.155101

[2] Bin Bao and Chen Ding. 2013. Defensive Loop Tiling for Shared Cache. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO '13)*. IEEE Computer Society, Washington, DC, USA, 1–11. https://doi.org/10.1109/CGO.2013.6495008

[3] Kristof Beyls and Erik H. D'Hollander. 2002. Reuse Distance-Based Cache Hint Selection. In *IN PROCEEDINGS OF THE 8TH INTERNATIONAL EURO-PAR CONFERENCE*. 265–274.

[4] Edouard Bugnion, Jennifer M. Anderson, Todd C. Mowry, Mendel Rosenblum, and Monica S. Lam. 1996. Compiler-directed Page Coloring for Multiprocessors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, Massachusetts, USA) *(ASPLOS VII)*. ACM, New York, NY, USA, 244–255. https://doi.org/10.1145/237090.237195

[5] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. 1994. Compiler Optimizations for Improving Data Locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS VI)*. ACM, New York, NY, USA, 252–262. https://doi.org/10.1145/195473.195557

[6] Jason Cong, Peng Zhang, and Yi Zou. 2011. Combined Loop Transformation and Hierarchy Allocation for Data Reuse Optimization. In *Proceedings of the International Conference on Computer-Aided Design* (San Jose, California) *(ICCAD '11)*. IEEE Press, Piscataway, NJ, USA, 185–192. http://dl.acm.org/citation.cfm?id=2132325.2132368

[7] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. 2013. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*. 1–15. https://doi.org/10.1109/EMSOFT.2013.6658595

[8] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. 2009. Cache-Aware Scheduling and Analysis for Multicores. In *Proceedings of the Seventh ACM International Conference on Embedded Software* (Grenoble, France) *(EMSOFT '09)*. Association for Computing Machinery, New York, NY, USA, 245–254. https://doi.org/10.1145/1629335.1629369

[9] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. 1991. The Impact of Operating System Scheduling Policies and Synchronization Methods of Performance of Parallel Applications. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (San Diego, California, USA) *(SIGMETRICS '91)*. Association for Computing Machinery, New York, NY, USA, 120–132. https://doi.org/10.1145/107971.107985

[10] Pekka Jääskeläinen, Pertti Kellomäki, Jarmo Takala, Heikki Kultala, and Mikael Lepistö. 2008. Reducing Context Switch Overhead with Compiler-Assisted Threading. In *Proceedings of the 2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing - Volume 02 (EUC '08)*. IEEE Computer Society, Washington, DC, USA, 461–466. https://doi.org/10.1109/EUC.2008.181

[11] Aamer Jaleel, Hashem H. Najaf-abadi, Samantika Subramaniam, Simon C. Steely, and Joel Emer. 2012. CRUISE: Cache Replacement and Utility-aware Scheduling. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) *(ASPLOS XVII)*. ACM, New York, NY, USA, 249–260. https://doi.org/10.1145/2150976.2151003

[12] Nagakishore Jammula, Moinuddin Qureshi, Ada Gavrilovska, and Jongman Kim. 2014. Balancing Context Switch Penalty and Response Time with Elastic Time Slicing. In *21st International Conference on High Performance Computing (HiPC)*. Goa, India.

[13] Vahid Kazempour, Alexandra Fedorova, and Pouya Alagheband. 2008. Performance Implications of Cache Affinity on Multicore Processors. In *Euro-Par*.

[14] Tobias Klug, Michael Ott, Josef Weidendorfer, and Carsten Trinitis. 2011. Transactions on High-performance Embedded Architectures and Compilers III. Springer-Verlag, Berlin, Heidelberg, Chapter Autopin: Automated Optimization of Thread-to-core Pinning on Multicore Systems, 219–235. http://dl.acm.org/citation.cfm?id=1980776.1980792

[15] Rakesh Kumar and Dean M. Tullsen. 2002. Compiling for Instruction Cache Performance on a Multithreaded Architecture. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture* (Istanbul, Turkey) *(MICRO 35)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 419–429. http://dl.acm.org/citation.cfm?id=774861.774906

[16] Min Lee and Karsten Schwan. 2012. Region Scheduling: Efficiently Using the Cache Architectures via Page-level Affinity. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) *(ASPLOS XVII)*. ACM, New York, NY, USA, 451–462. https://doi.org/10.1145/2150976.2151023

[17] Zhi Li, Yuebin Bai, Huiyong Zhang, and Yao Ma. 2010. Affinity-Aware Dynamic Pinning Scheduling for Virtual Machines. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. 242–249. https://doi.org/10.1109/CloudCom.2010.51

[18] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (Porto Alegre, Brazil) *(MICRO-44)*. ACM, New York, NY, USA, 248–259. https://doi.org/10.1145/2155620.2155650

[19] MediaBenchII 2006. MediaBench II Benchmark. https://cs.slu.edu/~fritts/mediabench/. Accessed: 2021 Oct 17.

[20] Chanik Park, Junghee Lim, Kiwon Kwon, Jaejin Lee, and Sang Lyul Min. 2004. Compiler-Assisted Demand Paging for Embedded Systems with Flash Memory. In *Proceedings of the 4th ACM International Conference on Embedded Software* (Pisa, Italy) *(EMSOFT '04)*. Association for Computing Machinery, New York, NY, USA, 114–124. https://doi.org/10.1145/1017753.1017775

[21] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. 2013. Polyhedral-based Data Reuse Optimization for Configurable Computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '13)*. ACM, New York, NY, USA, 29–38. https://doi.org/10.1145/2435264.2435273

[22] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. 2012. Thread Tranquilizer: Dynamically Reducing Performance Variation. *ACM Trans. Archit. Code Optim.* 8, 4, Article 46 (Jan. 2012), 21 pages. https://doi.org/10.1145/2086696.2086725

[23] Sift and Stitch 2009. Sift and Stitch Applications in SD-VBS Benchmarking Suite. http://parallel.ucsd.edu/vision/SD-VBS.pdf. Accessed: 2021 Oct 17.

[24] M. S. Squillante and E. D. Lazowska. 1993. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. *IEEE Trans. Parallel Distrib. Syst.* 4, 2 (Feb. 1993), 131–143. https://doi.org/10.1109/71.207589

[25] Josep Torrellas, Andrew Tucker, and Anoop Gupta. 1993. Benefits of Cache-Affinity Scheduling in Shared-Memory Multiprocessors: A Summary. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Santa Clara, California, USA) *(SIGMETRICS '93)*. Association for Computing Machinery, New York, NY, USA, 272–274. https://doi.org/10.1145/166955.167038

[26] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and

Michael Bedford Taylor. 2009. SD-VBS: The San Diego vision benchmark suite. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 55–64.

[27] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. ACM, New York, NY, USA, 607–618. https://doi.org/10.1145/2485922.2485974

Received 2023-03-16; accepted 2023-04-21