This is the accepted version of a paper presented at MULTIPROG 2014: Programmability Issues for Heterogeneous Multicores.

Citation for the original published paper:

DVS: Deterministic Victim Selection to Improve Performance in Work-Stealing Schedulers.
In:

N.B. When citing this work, cite the original published paper.

Permanent link to this version:
http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-128183
DVS: Deterministic Victim Selection to Improve Performance in Work-Stealing Schedulers

Georgios Varisteas\textsuperscript{1} and Mats Brorsson\textsuperscript{1}

KTH Royal Institute of Technology
\{yorgos,matsbror\}@kth.se

Abstract. Task-centric programming models offer a versatile method for exposing parallelism. Such programs are popularly deployed using work-stealing scheduling runtimes. Work-stealers have traditionally employed randomness dependent techniques, considered optimal for several execution configurations. We have identified certain inefficiencies and leeway for improvement on emerging parallel architectures and workloads of fluctuating parallelism. Our deterministic victim selection (DVS) for work-stealing schedulers was designed to provide controllable and predictable uniform distribution of tasks without degrading performance; stealing is restricted between specific pairs of workers. We experimentally show that DVS offers improved scalability and performance for irregular workloads. We demonstrate DVS on Linux and Barrelfish operating systems, using an 48 core Opteron system and a simulated ideal platform respectively. On real hardware, we observed better scaling and 13\% average performance gains, up to 55\% for specific irregular workloads.

1 INTRODUCTION

Work-stealing is a method for scheduling concurrent computational tasks onto worker threads. In contrast to centralized task-schedulers, each worker has a local task-queue; if a worker runs out of work it selects a victim to steal tasks from. The policy for selecting victims is responsible for distributing tasks among available workers and has the opportunity to enforce specific patterns. In most cases, random victim selection is used because of its stable performance and theoretical properties [1–4]. However, it provides no opportunity to optimize for emerging architectural diversity [5]. When deployed with workloads of fluctuating parallelism, random victim selection can cause sub-optimal scalability [6], leading to resources being wasted [7, 8].

On large scale, randomness does not provide sufficient bounds on the steal attempts required for the discovery of stealable tasks. To address the delays, semi-random victim selection techniques have been introduced, including leapfrogging [9] and sequential selection based on some fixed ordering of the workers. However, work-stealers implementing these techniques still use randomness in the beginning of the execution, and after several failed steal attempts. There is room for improvement.
In this paper we introduce deterministic victim selection (DVS) as an alternative work-stealing scheduling method. In DVS each worker has a predefined partially ordered set of possible victims. DVS controls the relocation of tasks by restricting the victim selection to predefined candidates based on a complex policy that achieves uniform fast distribution of tasks across all workers. Although victim selection is significantly constrained, the uniformity of the distribution increases the probability of stealable tasks on any worker’s task-queue. Failed steal attempts are reduced and performance is maintained or increased, based on the type of parallelism of the workload.

We experimentally show the efficiency and uniformity of DVS-based task distribution. We demonstrate DVS on Barrelfish [10] under a simulated platform and on Linux using real hardware. We comparatively examine the effectiveness and performance of our solution against the original WOOL implementation [11, 12] – to which our DVS implementation is based on. The results are promising with equivalent performance for highly parallel fine grained workloads and consistently improved with more irregular ones. On real hardware we observed an total average of 13% improved performance over various allotment sizes and workload types. Specific highly irregular workloads improved their performance on an average of 40% at large scale.

2 VICTIM SELECTION

Our proposed victim selection policy is deterministic. It uses specific strict rules to redefine the set of possible victims for each worker, thus removing all randomness. Applying DVS in a work-stealing scheduler requires defining a metric topology of the allotted workers. This topology can be virtual and in complete disagreement to the physical interconnect network. To devise it, first each worker thread must be pinned onto a distinct core. Second, a metric must define the communication distance between cores. This distance could -but doesn’t have to- be derived from the architectural interconnect, leading to better data-locality preservation.

To better illustrate the concepts behind DVS, we model a mesh grid topology. In the model, all nodes are connected horizontally and vertically. We define the communication distance as 1 between adjacent nodes. The connections do not wrap around nodes on the edges. This model can be used for any architecture provides the same number of cores, irrespective of the physical connectivity between them.

The theoretical foundation we developed DVS upon, is based on a generic model where the processor topology can have up to three dimensions. For example cores can be modeled in one dimension, as if placed in a row. Different dimensions produce a different classification although the implications remain the same. For clarity we restricted this paper to the case of two dimensions.

A workload is allotted a set of workers and it’s started on one of them. Throughout this paper this worker will be referred to as the source and symbolized with s. The rest of the workers in the allotment are unambiguously
separated into different classes. The classification is defined by the location of a worker in respect to the source. There are three classes X, Z and F. Class Z includes those workers that are at the same maximum communication distance from the source. Class X includes those workers that span horizontally and vertically from the source, excluding those at maximum distance. Class F includes the remaining workers. Figure 1 uses a mesh grid topology to illustrate these classes, assuming a symmetric allotment of 41 workers.

In summary, stealing is allowed only between close neighbors, at most distance of 2. Workers on the main axes (class X) are responsible for distributing the load away from the source. Members of class F relocate it back inwards. Class Z helps balance the load across all quadrants. Workers in Z will first steal from within their own class (diagonally left and right); only upon failing that, they’ll search for new tasks from the inner parts of the allotment.

If a class’ definition geometrically covers a certain worker which has not been allotted, the class is incomplete. If the class includes all its members it is complete. In a multiprogrammed system resource competition is expected, leading to conserved allotments and incomplete classes (see fig. 2). Hence DVS has been designed to be tolerant of incomplete classes. Victim sets are ordered, starting from those victims that contribute to the main desired flow. However, the sets include other members with lower priority to support towards incomplete classes and fluctuations in the parallelism.

Figure 3. Reimagine the relocation of tasks through stealing, as a flow of the load through the workers. In the figures, each arrow abstractly represents the directed relocation of tasks from the victim to the thief. White for outward flow and gray for inward reflow. Subfigures b-d present one quadrant of a certain allotment size to avoid clutter.
Task parallelism is based on the idea that executing a task will result in spawning more tasks up to a certain recursion depth. Stealing a task relocates it to a new worker. The tasks spawned from that stolen task will be placed in the thief’s task-queue and can be stolen also. This process can be envisioned as a flow of tasks across the workers. A non-random victim selection, like DVS, can control this flow, creating a predictable concentration of the spawned tasks on specific workers.

The DVS ruleset changes according to the maximum distance between the source and the rest of the workers. These are for maximum distance up to 1, equal to 2 and greater than 2. In the first case all workers are members of X; they steal from all workers, mirroring the behavior of random victim selection. The second case introduces class Z to distribute the load across all quadrants. The final case introduces class F as an auxiliary pathway that flows the load back inwards. Figure 3 abstractly represents the flow of tasks generated by applying DVS. The figure does not try to represent the actual rules.

3 ANALYTICAL VIEW

This section consists of a minimal formal presentation of the DVS policy. A complete listing of definitions and proved properties is available at [13].

3.1 Definitions

We refer to the communication distance between any two workers \(w_i\) and \(w_j\) as the hop-count \(hc(w_i, w_j)\); this is the shortest communication path between the physical cores where the two worker threads are located. We define \(I\) as the allotment of workers for a specific workload and \(s\) its source worker. We define as \(d\) the maximum hop-count between the source \(s\) and any other worker in \(I\).

Below is a formal definition of these sets, while fig. 1 presents them graphically.

Class \(Z\) is defined as the set of workers at distance \(d\) from the source.

\[
Z_n = \{w_j \in I : hc(w_j, s) = n\}
\]

Class \(X\) consists of those workers that are neighboring only one worker at one less hop from the source.

\[
X = \left\{ w_j \in I : \exists! w_i \in I : \begin{array}{l}
hc(w_i, w_j) = 1 \\
hc(w_j, s) = hc(w_i, s) + 1
\end{array} \right\}
\]

Class \(F\) consists of the remaining workers excluding the source \(s\).

\[
F = I \setminus (X \cup Z \cup \{s\})
\]
Victim selection rules This section presents and explains our deterministic victim selection policy. We provide a formal definition of the victim set for each different class.

Definition 1 (Victims set). Each worker \( w_i \in I \) can steal tasks from a subset of workers called the victims set \( V_i \). Members of each class populate this set differently. It is defined as the union of two independently defined sets \( V_a \) and \( V_b \). The former is equal for all workers and includes the immediate neighbors (at distance 1). The rules defining the second set depend on the owner’s class, its distance from the source and the value of \( d \).

\[
V_i = V_a \cup V_b \\
V_a = \{w_j \in I : hc(w_i, w_j) = 1\} \\
V_b = \begin{cases} 
\emptyset, & w_i \in F \\
\emptyset, & w_i \in Z : d = 2 \\
\{w_j \in Z : hc(w_i, w_j) = 2\}, & w_i \in Z : d > 2 \\
\{w_j \in X : hc(w_i, w_j) = 2\}, & w_i \in X : hc(w_i, s) = 1 
\end{cases}
\]

It is important to note that when \( d \) is 1, classes \( X \) and \( Z \) conceptually overlap; thus class \( Z \) is introduced only when \( d \) is 2 or more. Similarly, class \( F \) exists only when \( d \) is 3 or higher. Finally, in the special case of a one-dimensional topology (all cores placed in a row), \( F \) can never be introduced but is also not needed.

Victims prioritization This section presents the order by which a worker selects a victim out of its set.

Definition 2 (Victim prioritization). We define a partial order \( P(v_j \in V_i) \) where \( v_j \in V_i \), where \( V_i \) the victims set of worker \( w_i \) in allotment \( I \) as follows.

- For members of class \( X \)
  1. Priority increases inversely proportional to the victim’s distance from \( s \), thus giving higher priority to victims located closer to \( s \) that \( w_i \)

- For members of classes \( Z \) and \( F \)
  1. Priority increases proportionally to the victim’s distance from \( s \), thus giving higher priority to victims located further away from \( s \) that \( w_i \)

The ordering rules between members of \( X \) and \( Z \cup F \) are opposed. The former prioritize victims closer to the source, while the latter those further away. These rules allow certain ordering conflicts as there can be multiple victims at the same distance from the source. These are intentional as there is no benefit by resolving them a certain way.
Fig. 4. Two example allotments on two different architectures, classified as per the DVS rules. (a) shows a 27 worker allotment on a 8x4 mesh grid. All classes are incomplete. (b) shows a 35 workers on a 8x6 mesh grid, with core 28 being the source. Classes X and Z are incomplete.

4 EXPERIMENTAL SETUP

Our implementation is based on an established work-stealing scheduler, WOOL. We implemented and evaluated two versions of our scheduler. One being the original WOOL implementation [11] (WOOL-LF) that employs semi-random victim selection. For the second version (WOOL-DVS) we replaced only the victim-selection algorithm with DVS; the structure of the workers, the task-queues, spawn, sync and steal operations where left as is.

The workloads we used for evaluation are the irregularly parallel FFT, nQueens, Sort and Strassen from the BOTS benchmark suite [14] along with the regular and highly parallel recursive Fibonacci (Fib), Skew and Stress micro-benchmarks. We use the latter group to establish that DVS does not affect performance. The irregular workloads stress the runtime and evaluate the balanced distribution properties of DVS.

We run our experiments on two different platforms, one simulated and on real hardware. In the Simics v4.6 full system simulator [15] we modeled an ideal parallel platform without memory hierarchy, running the Barrelfish OS [16,17]. We have modeled a 32 core, 8x4 mesh topology where each instruction takes one cycle, including memory operations. The exclusion of a memory hierarchy and the ability to dedicate cores to threads in Barrelfish, makes the scheduling algorithm the only factor for different results. Thus it serves to evaluate the algorithm in itself.

The second platform is based on Linux (v2.6.32) running on real hardware. The architecture is Opteron 6172 (AMD Magny-Cours) ccNUMA system with a total of 48 cores. There are 4 sockets, holding 2 NUMA nodes each. A node has 6 processing cores and 8GB of locally controlled RAM. Threads were pinned using pthread affinity while the system was running a minimum of necessary services.

On both platforms we tested allotments of varying sizes; namely 5, 12, 20 or 27 cores for Barrelfish and 5, 13, 24, 35 cores for Linux. It’s important to note that due to the topology’s geometry, these allotments are not complete in respect to the classes as defined by DVS. Figure 4 visually presents the classification for the largest allotment on each platform.

The input dataset used for each program can be viewed in fig. 5. The input field corresponds to basic parameters, while the cut-off controls the maximum
recursion depth and has a significant impact on the produced parallelism. We have selected small cut-off values to expose high irregularity in the parallelism profile of some workloads. On the Linux platform we used larger inputs to minimize the effect of interference from the operating system and other services, without altering the parallelism profile.

**WORKLOAD INPUT DATA SETS**

<table>
<thead>
<tr>
<th></th>
<th>FFT</th>
<th>FIB</th>
<th>nQueens</th>
<th>Skew</th>
<th>Sort</th>
<th>Strassen</th>
<th>Stress</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>input</strong></td>
<td>32768*512</td>
<td>40</td>
<td>13</td>
<td>10000,20,1,1</td>
<td>32768*1024</td>
<td>1024,32</td>
<td>10000,20,1</td>
</tr>
<tr>
<td><strong>cut-off</strong></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>2048,20</td>
<td>64,3</td>
<td></td>
</tr>
<tr>
<td><strong>input</strong></td>
<td>32768*1024</td>
<td>42</td>
<td>14</td>
<td>10000,44,3,1</td>
<td>32768*1024</td>
<td>1024,32</td>
<td>10000,44,3</td>
</tr>
<tr>
<td><strong>cut-off</strong></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>2048,20</td>
<td>64,3</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 5. Arguments used for each test program. Top for Barrelfish, below for Linux. The *input* field corresponds to basic parameters, while the *cut-off* controls the maximum recursion depth.

5 EVALUATION

Figure 6 presents performance results. At the top two rows with the highly parallel regular workloads, the two runtimes performed and scaled equally. However, with irregular workloads we observed better scaling with WOOL-DVS.

FFT had a 52% improved performance on 24 cores and 32% on 35 cores. nQueens gained 51% on Barrelfish and 55% on Linux. Strassen also exhibited improved scaling with 7%, 27%, 30% and 10% improvement between allotment sizes on Barrelfish. On Linux the improvements where 14%, 17%, 23% and 43% in comparison to WOOL-LF.

Strassen is a peculiar parallel application, that is known get a slow down with many workers. The task tree is unbalanced and the amount of parallelism just enough to utilize the workers. With random victim selection, workers fail to find tasks quickly; tasks are stolen by the same workers that spawned them, which results into sequential execution. WOOL-DVS manages to fix this behavior achieving a better distribution to all workers.

Finally with Sort the results are different. This irregular workload is the only that performed 9% worse with WOOL-DVS and only with 20 workers on Barrelfish. Sort produces a sequence of parallel sections of variable size. These are all started on the source worker; thus after every section, parallelism syncs back to the source, and the new parallel load must be a redistributed. DVS is algorithmically slower in managing such pattern. However, the results on real hardware showed an small increase in performance with greater number of workers. The reason is a reduction of cache misses irrespective of the physical memory layout, because workers always steal from the same victims.
The per worker utilization is uniform in most of our runs. We included results only for Sort (figure 9) due to space limitations, although similar charts for other runs can be produced on demand.

6 RELATED WORK

Task-centric models and work-stealing schedulers have been the focus of much research the past few years. Tasks are logical entities, not bound to hardware, which provides a large level of flexibility in expressing parallelism. Also, the distributed nature of work-stealing and its documented high performance have led to apply it in a series of diverse environments [6,18,19]. Many have successfully experimented with alternative methods for more intelligently controlled selection of victims.

SLAW (Scalable Locality-Aware Work-stealing) [20] provides and uses locality awareness in a scheduler that can choose to follow either work-first or
Fig. 7. Delay before first successful steal normalized to WOOL-LF (as 100%). FIB, Skew and Stress provide high parallelism very early, benefiting random victim selection. DVS reduces the worst delay, while for irregular workloads the average is the same.

Fig. 8. Successful steals, normalized to WOOL-LF (as 100%) per allotment size. DVS results in less steals with fewer workers; thus workers are able to steal tasks higher on the task tree, running sequential thereafter (stealing from themselves).

Fig. 9. Per worker useful execution time for Sort on real hardware. Useful is the aggregated time spent processing and successfully stealing tasks. The first worker is the source which includes some initialization.

help-first on a per-task basis. Semi-random victim selection is being used, with the search space being restricted to a specific subset for locality preservation.

HotSLAW [21] expands on SLAW, and introduces HVS (Hierarchical Victim Selection) increasing the locality benefits. Different components create a hierarchical ladder. Workers arbitrarily pick victims on the bottom of the ladder (highest locality) and move upwards until a steal is successful.

Quintin et. al. [22] also embraces the idea of hierarchy, presenting two variants of a work-stealing scheduler. PWS restricts the victim search space to immediate
neighbors, based on the hardware interconnect network. HWS applies a more complex notion of hierarchy in clusters where its is possible to transfer chunks of tasks between clusters while taking advantage of network properties to maximize performance.

Finally, Saraswat et al. [23] has also tackled the problem of deploying a work-stealer on distributed memory and at great scale of cores, through the introduction of lifeline load balancing. Lifeline graphs denote a connectivity network between workers as nodes. After a certain threshold of failed steals a node becomes inactive while informing its lifeline. Inactive nodes are not picked as victims until their lifeline provides work and reactivate them.

7 CONCLUSIONS

While random-based victim selection has been shown to be optimal for a specific model of parallel computation, recent architecture designs and other parallel configurations exhibit scale and resource diversity not captured by it. Hence, there is leeway for improving scalability while allowing better control on the placement of parallel tasks through deterministic distribution.

We presented a deterministic victim selection policy, called DVS. Assuming one per core pinned workers, DVS makes use of a virtual geometrical topology to restrict steals between neighboring workers. Through a predefined ordered victim set, stealing overhead is decreased while tasks are relocated faster and uniformly. These benefits result in a significant reduction of failed steal attempts.

Finally, DVS provides the guaranties required for better resource requirements estimation. By controlling the concentration of tasks, it allows the use of task-queue size as the requirements estimation metric. Such metric quantifies future processing and is independent of past behavior. Furthermore, knowing the availability of tasks does not require knowledge of their granularity as long as the interval at which the status is checked is comparable to their execution time. Our future work is focused on adaptive resource management by exploiting the properties of DVS to accurately estimate resource requirements online.

References

   In: Job Scheduling Strategies for Parallel Processing. Springer Berlin Heidelberg (2009) 280–299
   In: Job Scheduling Strategies for Parallel Processing, Springer (2007) 1–32
    In: 2010 39th International Conference on Parallel Processing, Ieee (September 2010) 313–322
16. Baumann, A., Peter, S., Schupbach, A., Singhania, A., Roscoe, T., Barham, P., Isaacs, R.: Your computer is already a distributed system. why isn’t your OS?
    In: 2008 37th International Conference on Parallel Processing, Ieee (September 2008) 586–593
    In: Proceedings of the 15th ACM