<http://www.diva-portal.org>

This is the published version of a paper presented at *PMAM'14: The 2014 International Workshop on Programming Models and Applications for Multicores and Manycores*.

Citation for the original published paper:

Varisteas, G., Brorsson, M. (2014)

Palirria: Accurate On-line Parallelism Estimation for Adaptive Work-Stealing.

In: *PMAM'14 Proceedings of Programming Models and Applications on Multicores and Manycore* (pp. 120-130). ACM New York, NY, USA ©2007: ACM Press

<http://dx.doi.org/10.1145/2560683.2560687>

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-128184>

Palirria: Accurate On-line Parallelism Estimation for Adaptive Work-Stealing

Georgios Varisteas
KTH Royal Institute of Technology
yorgos@kth.se

Mats Brorsson
KTH Royal Institute of Technology
matsbror@kth.se

ABSTRACT

We present Palirria, a self-adapting work-stealing scheduling method for nested fork/join parallelism that can be used to estimate the number of utilizable workers and self-adapt accordingly. The estimation mechanism is optimized for accuracy, minimizing the requested resources without degrading performance. We implemented Palirria for both the Linux and Barrelfish operating systems and evaluated it on two platforms: a 48-core NUMA multiprocessor and a simulated 32-core system. Compared to state-of-the-art, we observed higher accuracy in estimating resource requirements. This leads to improved resource utilization and performance on par or better to executing with fixed resource allotments.

Categories and Subject Descriptors

D.4.1 [Operating Systems]: Process Management—*Multiprogramming*

Keywords

multicore, parallel, scheduler, workload, runtime, task, adaptive, resource management, load balancing, work-stealing

1. INTRODUCTION

Many parallel workloads exhibit a varying degree of parallelism during their life time. Examples include web-servers with variable load, the unbalanced working set of reductions in a MapReduce system and the dynamic resource management between multiple VMs over a hypervisor. Task-centric programming models can flexibly adapt to irregular parallelism and fluctuations in resource availability; tasks and associated data structures are logical and not bound to the underlying hardware [26]. However, current task-centric runtime systems do not utilize the opportunity to scale down or up the amount of resources depending on the workload requirements and the available power budget at the moment.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
PMAM '14, February 15–19 2014, Orlando, FL, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM. ACM 978-1-4503-2655-1/14/02 ...\$15.00
<http://dx.doi.org/10.1145/2560683.2560687>.

Currently, the system is at the users' mercy to not over-commit the resources with more threads than the system can handle and threads from different applications are not differentiated from each other in modern operating systems such as Windows and Linux. There is room for improvement in how operating systems manage heavy multiprogrammed parallel workloads. Fluctuating parallelism [23] makes static resource management inefficient on large scale multiprogrammed systems [5]. Dynamically adaptive resource management promises to optimize such patterns [3, 18, 9] by recognizing a workload's requirements and automatically altering the allotted resources to match.

Adaptive resource management requires scheduling abstractions to be split in two cooperating layers. The system layer which is aware of the availability of resources and the application layer which can accurately estimate its workload's true resource requirements. The application's runtime scheduler iteratively estimates the number of workers it can actually utilize, based on the parallelism existent in its current workload. Upon receiving such estimations, the system scheduler can adapt the workload's allotment of workers.

We focus on the automatic estimation of a task-parallel application's resource requirements. Current algorithms are based on measuring the number of cycles spent on useful actions [2, 9]. However, such metrics attest to the utilization of resources up to that time point; the assumption that past behavior will persist can be false, if the parallelism is fluctuating. Moreover, the constant reading of cycle counters is a non-negligible overhead, since work-stealing runtimes have reached the maturity of performing steal and spawn actions in just a few hundred cycles [16].

We present a work-stealing scheduler which can produce accurate estimations with low overhead. At its heart is the Deterministic Victim Selection policy (DVS) as a replacement of traditional random and semi-random techniques. DVS controls the relocation of tasks by restricting the victim selection to predefined candidates. This control makes the concentration of tasks predictable even when the parallelism is fluctuating. With DVS, the size of the task-queue of the workers (the number of stealable tasks, spawned but not processed or stolen) can be used as the metric for resource requirements. Acquiring such value does not involve overhead since its calculation is performed during the spawn and sync operations. Furthermore, the size of the task-queue reflects the amount of the work that has to be performed in the future. Hence increasing an allotment is a response to actual needs.

We evaluate Palirria in comparison to ASTEAL, a similar

self-adapting algorithm [2], using a non-adaptive work stealing scheduler to provide a baseline of expected performance. We use implementations on the Barrelfish [6] and Linux operating systems. On average, Palirria performs similarly to the baseline while utilizing less resources; for irregular workloads it achieved execution time improvement up to 9% and up to 7% reduction of wasteful runtime operations, like trying to steal from victims that have no stealable tasks. Compared to ASTEAL there is an average of 10% (real hardware) and 8% (simulator) better performance coupled with improved resource utilization across all experiments.

2. BACKGROUND

This section presents a summary of previous work on which this paper expands upon.

2.1 TASK SCHEDULING

In this project we focus on task-based parallel applications, thus making the task scheduler a crucial part. The basis is an already established runtime scheduler, *WOOL* by [17, 28] which we augmented and adapted. We implemented and evaluated three versions of our scheduler. One being the original, running with a fixed number of workers. The second was augmented with an existing resource estimation algorithm, without altering the scheduler. The third used our resource estimation algorithm—also replacing the victim selection policy—without altering the main functionality of the scheduler. For all implementations the initialization process was adapted. The structure of the workers, the task-queues, spawn, sync and steal functions were left as is.

WOOL employs the *work-first work stealing* method [8, 4] for distributing tasks among worker threads. Each worker has each own task-queue where spawned tasks are placed. Upon spawning a task, a worker continues processing the current task. Tasks in the queue can be either stolen by other workers or picked up later. When a worker is out of work (its task-queue is empty), it will select a set of victims and try to steal tasks from them. In WOOL, each worker’s task-queue has a predefined number of stealable and non-stealable task slots, with the former being much less but populated first. Stealable tasks are those spawned but not yet processed or stolen. For all implementations we have set the number of stealable tasks to the same constant number that is sufficient for the largest number of workers.

2.2 VICTIM SELECTION

Palirria’s estimation of requirements is possible thanks to its deterministic victim selection policy (**DVS**). DVS replaces the random-based techniques in traditional work-stealing schedulers. The complete DVS method is formally presented at [31]. For completeness we include a summary here.

DVS uses specific strict rules to predefine the set of possible victims for each worker, thus removing all randomness. In order to apply DVS in a work-stealing scheduler, there are two requirements. First that each worker thread is pinned to a different core. Second to define a metric for the communication distance between workers. This metric can be either mapped to the physical interconnect of the architecture, or a virtual graph connecting all cores.

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.

Figure 1: 41 worker allotment, classified as per the DVS rule set, over a virtual mesh grid topology.

Figure 2: Example of real-life deployment of 3 applications. Each allotment consists of incomplete classes.

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.

The theoretical foundation we developed DVS on 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.

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.

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.

Figure 3 abstractly represents the generated flow of tasks, as defined by the DVS ruleset. The figure does not try to represent the actual rules. In summary, stealing is allowed only between close neighbors, (at most distance 2). Workers on the main axes (class *X*) are responsible for disseminating the load away from the source. Members of class *F* relocate it back inwards by stealing primarily from workers in *Z*. 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 but that worker 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 figure 2). Hence DVS has been designed to be tolerant of incomplete classes. Victim sets are ordered, prioritizing those victims that contribute to the primary flow path. However, victim sets include other members with lower priority to overcome the existence of incomplete classes and fluctuations in the parallelism.

3. RESOURCE ESTIMATION

The estimation of requirements could be viewed as a decision on the effectiveness of the current allocation of workers to a specific application. This would result in a request for more workers, or the dismissal of some. The criteria can be either runtime-specific (ASTEAL) or workload specific (Palirria). This section will go through the reasoning, the criteria and the objectives of each of the two methods.

Both algorithms run on a fixed interval, called the **quantum**. This is an important configuration point. A long interval will miss important fluctuations thus negating the effectiveness of the adaptation. A short interval might create unnecessary overhead and impede on the overall performance by confusing short bursts as prolonged behavior, by deciding before the load has had the time to be distributed across the running workers. Although our current implementation uses a small fixed interval, there is logic implemented to identify specific patterns and discard false positives. This mechanism is part of the system level, it is independent of the runtime scheduler in place and thus the same for both ASTEAL and Palirria implementations.

3.1 ASTEAL

This section describes ASTEAL, a mechanism for estimating requirements applicable when using either [semi-]random or deterministic victim selection. ASTEAL was initially presented by K. Agrawal et al. [2]. It is beyond the scope of this paper to analyze this model in length. For an in-depth analysis the reader is referred to [3].

We give a brief summary of the algorithm. Assume an application allotted a set of workers. The runtime measures the cycles spent searching for work (iterating victims, trying to find a stealable task) and those spent conducting a successful steal, excluding processing of the task itself; their sum is the amount of **wasted cycles**. The runtime scheduler calculates the sum of the wasted-cycles for all of its workers.

At the end of each **quantum**, this metric is compared against the normalized length of the quantum in cycles and then used to reason on the efficiency of the whole allotment. A secondary classification measures the degree of congestion in the system (as not having enough resources) and is independent to the performance of the allotment; if the current allotment is smaller than desired (i.e. a previous request was rejected) the workload is labeled as **deprived**, otherwise **satisfied**. The reasoning is that if an efficient workload did not get its desired resources, the system cannot offer them.

The actual decision policy is as follows:

- **Inefficient:** decrease allotment. The workload is unable to fully utilize its allotted workers.
- **Efficient and Satisfied:** increase allotment. The workload was allotted the desired resources and successfully utilized them.
- **Efficient and Deprived:** unchanged allotment. The workload was allotted less than the desired resources and successfully utilized them. The system is probably congested.

When a workload is inefficient, the secondary classification is irrelevant. A classification of *efficient and deprived* means that although efficient a taskset has been denied the desired amount of workers; that is due to external factors like contention. Assuming no change in the workload, during the next quanta the secondary classification will be satisfied, thus re-testing the ability of the system to increase the allotment.

3.2 Palirria

This section describes our proposed mechanism **Palirria** (Greek *παλίρροια*, meaning *tide*) for estimating requirements and can be used only when combined with deterministic victim selection. First we give an overview of the technique and then continue to present a more analytical view (section 4) that supports our claims.

Palirria requires deterministic victim selection but it can be applied on any work-stealing scheduler employs DVS. This is not a taxing requirement, since victim selection is usually an independent component. Palirria makes use of the controlled distribution and concentration of tasks DVS creates, to infer the future utilizability of the allotted workers. Based on a predefined threshold and the classification of workers (see figure 1), the decision policy is as follows:

- **Under-utilized:** decrease allotment. If the size of the task queue of each worker belonging to class Z is 0.
- **Over-utilized:** increase allotment. If the size of the task queue of each worker belonging to class X is above a threshold L .
- **Balanced:** unchanged allotment. If the size of the task queue of any worker belonging to class X is less than L or of any worker belonging to class Z is above 0.

The benefit of restricting the victim selection is deterministic flow and distribution of the workload among all workers. Simple conditions can be used to classify the utilization state of the workload as a whole. Moreover, these conditions need be evaluated for only a small but specific subset of the workers; also the required data (the size of the task-queue) are already calculated by the scheduler. this method produces lower overall overhead than constantly measuring spent cycles.

The time spent by the workers performing certain actions characterizes the workload up to that point. The accuracy of a decision for the future—based on this past behavior—is heavily dependent on the behavioral persistence into the future. However, most real life applications exhibit fluctuating parallelism during their execution. So, the persistence assumption is highly possible to become false. Contrary, the size of the task-queue of a worker attests to work that needs to be performed in the future. Thus the decision is based on criteria that characterize future behavior, enabling the change in the allotment to be in place on time to handle that work; may that be either avoiding wasting resources during a lack of parallelism or increasing the resources to efficiently handle a sudden burst of parallelism.

4. ANALYTICAL VIEW

This section consists of a limited formal presentation of the theoretical foundation behind Palirria, supporting the validity of the decision conditions. Formal proofs of the conditions and necessary properties are published in a technical report [31].

4.1 Definitions

We define the communication distance between 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 as **diaspora** (**d**) the maximum distance between the source worker s and any other worker in its allotment **I**. Finally, the estimation conditions will be referred to as the *Diaspora Malleability Conditions*, or **DMC** for short.

A **zone** is a subset of workers located at the same distance (hc) from the source. A zone's members can be of different classes. Each allotment consists of multiple zones. For any worker w , its inner zone is at 1 hop less, while its outer is at 1 hop more. The outermost zone of an allotment includes all workers at distance d from the source. A zone is important because it is the unit at which the size of an allotment changes. Upon reducing the size of an allotment, members of the outermost zone are removed; when increasing the size, workers at distance $d + 1$ are added.

All workers in an allotment are divided into **classes** according to their geometrical position in the allotment and in relation to the source. Below is a formal definition of these sets.

Class Z is defined as the set of workers that make up zone Z_d (at the maximum distance from the source within the allotment).

$$Z = \{w_j \in I : hc(w_j, s) = d\}$$

Class X consists of those workers that are neighboring (hc of 1) a single worker belonging to their inner zone.

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

Class F consists of the remaining workers excluding the source s .

$$F = I \setminus (X \cup Z \cup \{s\})$$

Furthermore, we make a distinction between certain members of a worker's w_i victim set V_i , as $O_i \subset V_{i_1} \subset V_i$ ¹.

Definition 1. We define the **Outer Victims set O_i** as the set of the victims of worker w_i at distance 1 located in its outer zone.

$$O_i = \{w_j \in V_{i_a} : hc(w_j, s) = hc(w_i, s) + 1\}$$

Since members of O_i are at distance 1 from w_i , w_i is their victim apart from them being its victims. Thus O_i includes those workers whom w_i is stealing from but also providing work to.

4.1.1 Malleability of Diaspora

This section presents and explains the conditions which can be used for estimating the utilization level of a workload. An increase of workers should be performed when the amount of already produced work is enough to utilize the added resources; when the outermost workers (Z) are found underutilized they can be removed without risking a decrease in performance.

For the following formulas, Q_i is the set of tasks placed in a worker's queue. These are tasks spawned but not yet stolen or processed. Also $\mu(\bullet)$ is defined as the measure of a set, evaluating to the set's amount of members, with $\mu(\emptyset) = 0$.

¹For a formal definition of V_i and V_{i_a} please see [31]

Claim 1 (DMC). *Diaspora Malleability Conditions:*

- **Increase** d when the size of the task queue Q_i of each worker w_i in X increases beyond L .

$$d+ \Leftrightarrow \mu(Q_i) > L > \mu(O_i), \forall w_i \in X \quad (1)$$

- **Decrease** d when the bag of each worker in Z is empty.

$$d- \Leftrightarrow \mu(Q_i) = 0, \forall w_i \in Z \quad (2)$$

- **Balanced** otherwise.

$\mu(O_i)$ is the number of workers in the outer zone that can steal tasks from w_i and is different for every worker. L is theoretically bound at $\mu(O_i)$; using different values for L (like $\mu(O_i) + 1$, but not constant) can tune the tolerance of the model. Conceptually L guarantees that momentarily there is immediately enough work for the new workers to steal. If these tasks are leaves, the allotment will most probably shrink in the next quantum. If not, the load will be distributed very fast outwards, generating new stealable tasks further from the source.

For example a case of an allotment of 5 workers (1 zone plus the source). All workers are part of X and their respective value of L is zero. That means that unless all their task-queues are empty, the allotment will always increase. Such behavior counters cases like the LOOPY program from [30] which is synthesized to be highly parallel, although the workers will never have more than one task in their queues. DVS scheduling complements this design by guaranteeing task discovery by all workers.

Increasing the value of d , increases the maximum distance from the source that an allotted worker can be at. Effectively this change allows all workers located at distance d to be added; these workers make up zone Z_d . It is up to the OS to check the availability of those workers and apply the change. We assume that the complete zone will be added, although it is not necessary for the model to function. Actually, it is presumed that a complete zone will rarely be available, unless the system is not multiprogrammed.

Contrary to that, decreasing diaspora from will effectively remove all workers in zone Z_d . Intuitively this can be tricky since it's possible that their task-queue is not empty. This is fairly common when random based victim selection is employed, as with ASTEAL, since there is no knowledge of a certain subset of workers that is underutilized. Our condition for decreasing ensures that at that time-point the queue of each worker in Z_d is empty. However, these workers will exit only after processing their current task, thus it's possible for new tasks to be spawned after requesting removal. In our implementation, a removed worker is forbidden to steal but can continue processing its own queue; it also continues to be selected as a victim. Each worker exits automatically when its task-queue is empty and no task is being processed; the removal can also be revoked at a later quantum, if the status changes according to DMC.

5. EXPERIMENTAL SETUP

For evaluating our work we have used several applications from various sources, faithfully ported to our runtime's programming model, WOOL. Among them are FFT, nQueens, Sort and Strassen from the BOTS benchmark suite [14], selected as distinctive and popular examples of specific workloads. Their parallelism profiles range from the fine grained

with a wide and balanced tree nQueens, to the quite irregular and coarser grained Strassen. FFT and Sort are thrashing the caches with the latter also being irregular. We have also included some micro-benchmarks; recursive Fibonacci (Fib), is embarrassingly parallel and rather finely grained which makes it scale linearly; Stress strains the runtime by varying the grain size; Skew is an adaptation of Stress producing a unbalanced task tree. nQueens, although highly parallel comprises multiple tasks of varying granularity, scaling sub-linearly with a small cut-off.

We run our experiments on two different platforms. One simulated and on real hardware. In the Simics v4.6 simulator we modeled an ideal parallel platform, running the Barrelfish OS. Simics is a full system simulator [25]. We have modeled a 32 core, 8x4 mesh topology where each instruction takes one cycle, including memory operations. The simulated model purposefully does not include a memory-hierarchy to isolate the behavior of the estimation algorithms.

Barrelfish is based on an interesting design which might be well suited for manycore architectures; it provides scalability and portability [29], major benefits when combined with widely distributed architectures [7], like most tiled manycore prototypes [24, 1, 19]. Also, it has no migration of execution between cores making worker threads pinned by default, while also allowing to execute programs in true isolation of other processes even the OS. All system services and auxiliary functions were executed on cores 0 and 1 which were never allotted to our test programs. The test environment of the two schedulers is controlled and the victim selection algorithm is the only difference in implementation; thus it can be argued that the adaptive scheduling is the only factor responsible for changes in the workload's behavior.

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.

The input dataset used for each program can be viewed in table 4. 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 significant 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.

Our evaluation strategy is based on a comparison between three implementations. First is the original non-adaptive version of the runtime scheduler (referred to as **WOOL**), using different fixed sets of workers. The same scheduler is augmented with each of the resource estimation algorithms, **ASTEAL** and **Palirria** as described in their respective sections. WOOL is used as a baseline for all comparisons, revealing the unavoidable overhead introduced by any resource estimation algorithm and the adaptation mechanism, and to provide a baseline for the expected performance.

The operating system's scheduler removes and adds workers in sets. These sets are the zones as defined in section 4.1; they are of fixed size so the total number of workers sequentially scales between them. With both ASTEAL and Palirria each application starts with the minimum set of 5 workers

		FFT	FIB	nQueens	Skew	Sort	Strassen	Stress
Barrelfish	input	32*1024*512	40	13	10000,20,1,1	32*1024*1024	1024,32	10000,20,1
	cut-off	N.A.	N.A.	3	N.A.	(2*1024),20	64,3	N.A.
Linux	input	32*1024*1024	42	14	10000,44,3,1	32*1024*1024	1024,32	10000,44,3
	cut-off	N.A.	N.A.	3	N.A.	(2*1024),20	64,3	N.A.

Figure 4: Workload input data sets. The *input* field corresponds to basic parameters, while the *cut-off* controls the maximum recursion depth. When multiple values are required, they are given separated with commas.

(zone 1 plus the source) and is left to adapt automatically. Using WOOL, we executed each application with multiple fixed allotment sizes. Namely, 5, 12, 20 and 27 workers for the 32-core simulated platform and 5, 13, 24, 35, 42 and 45 workers on the 48 core real machine.

The OS scheduler always satisfies allotment increment requests, up to the total number of cores available. The absence of external constraints in the expansion of the allotment evaluates the ability of the adaptive algorithm to recognize the existing parallelism while also conserving resources. We perceive these properties as fundamental and necessarily evaluated before investigating true multiprogramming scenarios.

The adaptive implementations (ASTEAL, Palirria) were started with 5 workers and let to expand automatically. Highly parallel workloads like FIB, loose performance this way. However, this strategy can be beneficial if resource conservation is the primary goal. To maximize performance such workloads can start with a big allotment, and shrink automatically.

It’s important to note that due to the topology’s geometry, most of the larger allotments are not *complete* in respect to the zones and classes as defined by DVS. Figure 9 visually presents the classification of a 4-zone allotment on each platform.

Cores 0 and 1 are reserved for the system scheduler and the helper threads respectively, hence all workers have uninterrupted, exclusive access to their cores. The helper threads are responsible for running the estimation and decision algorithms, bootstrapping and removing workers. With this design, changes in the allotment happen in the background with minimal interruption of the actual execution. Note that the data used by these algorithms are generated by the workers themselves, constituting overhead to the execution. In all time measurements the initialization time has been excluded, mainly to avoid unfairness due to the fact that fixed-allotment runs have to sequentially allocate a different number of data structures, based on the amount of workers.

Finally, the results reported were of the second best run among 10. Time measurements are in cycles, acquired by reading the *rdtsc* register of the underlying core. Thread creation and other runtime initialization is excluded. Total execution time is measured by the source worker, placed on the same core for all experiments.

6. EVALUATION

Our evaluation focuses exclusively on ASTEAL and Palirria. The criteria are performance, wastefulness and accuracy. Regarding performance we judge based on the ability

Figure 9: Two different allotments on two architectures, classified as per the DVS rules. Although each is modeled as a tiled grid, the actual topology is not. (a) shows an allotment of 27 workers on a 8x4 mesh grid, with core 20 being the source; all classes are incomplete. (b) shows an allotment of 35 workers on a 8x6 mesh grid, with core 28 being the source; classes X and Z are incomplete.

of each implementation to reach or surpass the performance obtain with fixed resources. For wastefulness, we borrow the definition of wasted cycles from ASTEAL (section 3.1), as the cycles spent on non-productive functions, namely failed steal attempts. Accuracy is the analogy between performance and resource utilization; an estimation is accurate when it reduces the resources utilized without degrading performance.

Figures 5 and 7 present the required data to decide on those criteria. Column (a) shows the execution time between all implementations, shorter bar is better. The execution with fixed-5 workers is consistently slower, thus taken as 100% with all other results normalized against it. Column (b) measures the wastefulness as the percentage of the average wasted cycles proportionally to the individual execution time. Column (c) shows the automatic adaptation of the worker set size over time and applies only to ASTEAL and Palirria. The amount of workers is on the Y axis and the time is on X; both plots are synchronized. In effect the implementation whose plot ends closer to point (0,0), denotes better accuracy as it executes faster and with less resources.

Figures 6 and 8 show the useful and wasted cycles spent by each worker on the two platforms. W4 and W5 are selected as baseline, since they exhibit overall best performance and less wastefulness on the simulator and real hardware respectively. These figures are meant to shed more light into were are the wasted cycles spent, but also reveal the uniformity in work distribution by the two adaptive schedulers.

For the rest of the analysis, all metrics are given relative to the best value achieved by a fixed allocation configuration.

We will start our analysis with the results on the simulated platform. Note that the absence of a memory hierarchy can

Performance measurements, on Barrelfish (simulator)

Figure 5: One workload per row. Results for fixed 5, 12, 20, 27 workers, ASTEAL (AS) and Palirria (PA) self-adapting implementations. (a) presents the total execution time, normalized to 5 workers as 100%. (b) presents the average percentage of wasted cycles of all workers compared to the individual total execution time. (c) shows the worker utilization over time for ASTEAL and Palirria; shorter execution time (x-axis) means better resource utilization thus higher accuracy of estimation.

lead to different results for specific workloads; there is no penalty for sharing data across different memory banks and load time is always 1 cycle. Regarding performance and column (a) of figure 5, both adaptive algorithms are not able to reach the performance achieved with the maximum fixed allotment size. Slowdown is on average 19% for ASTEAL and 14% for Palirria. Nevertheless, Palirria is faster than ASTEAL for all but one workload, Sort. We will discuss

Sort in detail separately.

Moving on to column (b), both algorithms reduce wastefulness in respect to non-adaptive WOOL. The reduction is on average 5% for ASTEAL and 6% for Palirria between all applications. ASTEAL is just 1% more effective for finely grained workloads. The timeline plots in column (c) attest to the overall better accuracy of Palirria against ASTEAL. With the exception of Sort, all workloads achieve better per-

Per worker useful time (normalized to 27 workers), on Barrelfish (simulator)

Figure 6: Useful (dark) and non-useful (light) normalized time per worker ordered by *zone*. Useful are the cycles spent successfully stealing and processing tasks. The variance between groups of workers, denotes different zones that are dynamically added. Plots per row are normalized to the first bar (source worker) of the first column.

formance. This is significant with FFT, Strassen and Stress as less resources are utilized. With FIB, nQueens and Skew Palirria recognized the existence of parallelism faster utilizing more workers and improving performance; given the availability of the extra resources, this behavior attests to the ability of Palirria to maximize performance. However, Palirria fails to recognize the existence of parallelism with Sort.

In detail *Sort* consists of a sequence of sections with a varying degree of parallelism. Parallelism is spawned quickly and then syncs back again. Each such section is started at the source worker, which means that tasks have to be redistributed each time. If the estimation interval’s length is such that the increase in parallelism is missed, the algorithm might falsely perceive a constant lack of parallelism thereof.

Moving on to the results on real hardware, behavioral patterns are different mainly due to caches. Regarding performance we see both adaptive schedulers, able to match the baseline execution time and in some cases surpass it. For ASTEAL the average slowdown is 5% across all runs, while it reaches 19% on Strassen. On the other hand, Palirria achieves an average 1% increase in performance and up to

8% for Strassen; worst slowdown is just 3% on FIB. FIB, Skew and Stress produce low wastefulness; they are fine grained, spawning enough tasks quite early. Regarding the other workloads ASTEAL manages a 3% average reduction of wastefulness and a maximum of 5% with nQueens; the average reduction for Palirria is 4% and a maximum of 8% with nQueens.

Strassen has been configured to produce just enough tasks to utilize a small number of workers. These tasks are also spawned gradually. In effect Palirria recognizes the lack of parallelism and distributes the available tasks more uniformly to all workers; this allows for the significantly better performance compared to both ASTEAL and the non-adaptive runtime.

The timeline plots (column c), show a consistently higher accuracy on behalf of Palirria as it manages better performance and resource utilization with FFT, nQueens, Sort, Strassen and Stress. For FIB and Skew the two schedulers performed almost identically.

Looking at the useful cycles charts (figures 6, 8), both ASTEAL and Palirria manage to uniformly reduce wastefulness and distribute tasks among all workers. In some

Performance measurements, on Linux (real hardware)

Figure 7: One workload per row. Results for fixed 5, 13, 24, 35, 42, 45 workers, ASTEAL (AS) and Palirria (PA) self-adapting implementations. (a) presents the total execution time, normalized to 5 workers as 100%. (b) presents the average percentage of wasted cycles of all workers compared to the individual total execution time. (c) shows the worker utilization over time for ASTEAL and Palirria; shorter execution time (x-axis) means better resource utilization thus higher accuracy of estimation.

cases this pattern is a significant reason for the performance gains. However, Palirria utilizes resources better; even at a small percentile difference it results into improved overall execution time.

7. RELATED WORK

Multiple past projects have targeted the topic of adaptive resource management for multiprogrammed systems. Adaptive Thread Management [21] optimizes the number of resources to data-parallel loops in automatically parallelized

Per worker useful time (normalized to 42 workers), on Linux (real hardware)

Figure 8: Useful (dark) and non-useful (light) normalized time per worker ordered by *zone*. Useful are the cycles spent successfully stealing and processing tasks. The variance between groups of workers, denotes different zones that are dynamically added. Plots per row are normalized to the first bar (source worker) of the first column.

programs [20] during execution, according to the online comparison of current speedup against expect values. Adaptive Multiblock PARTI runtime library [15] follows a similar approach, providing great portability. Dynamic feedback [13] provides a compiler assisted optimization scheme, adapting to different compiler generated synchronization policy on alternating phases. ADAPT (Automated De-coupled Adaptive Program Transformation) [32] applies loop optimizations via dynamic recompilation, it is based on the Polaris parallelizing compiler [27].

CAB (Cache Aware Bi-tier Task-Stealing) [11] focuses on avoiding cache pollution resulting from the random victim selection. Through the execution directed acyclic graph (DAG) it calculates data dependencies between tasks and schedules them so that they share the same caches. CATS [10], a follow up to CAB, provides the same bounds but great performance benefits and can handle more complicated DAGs. Cao et al. [9] work towards the same goal of adapting resources to workload requirements. They employ a mechanism evolved from ASTEAL that uses the size of the task-queue as metric for requirements estimation. Their method approximates the values using statistical sampling.

There have been several other projects, addressing the same issues on the OS level. Some approaches are similar, like the Factored OS [33], Tessellation OS [12] or ROS [22]. All argue in favor of resource space-sharing scheduling strategies, therefor redefine the process model to be more lightweight and add distributed resource management mechanisms in

the OS. They share our design for a two level cooperative scheduling scheme.

8. CONCLUSIONS

We contribute Palirria, a work-stealing method for the dynamic resource allocation for task-centric parallel applications; dynamically adapting the resources is a necessary component for efficient load-balancing on emerging many-core multiprogrammed systems. We evaluate Palirria using non-multiprogrammed scenarios, in order to establish that the proposed mechanisms are able to match the expected performance while improving resource utilization. High-load multiprogrammed configurations that lead to individual performance degradation is the next step for our project.

Palirria exhibited good estimation accuracy, low overhead and a reduction in wasteful scheduling operations on simulated and real-hardware platforms; performing good on all these metrics is crucial for not degrading performance. Palirria on average performed better on all three metrics, in comparison to the ASTEAL self-adapting algorithm. Utilization of workers is more uniform, leading to improved reduction of wasteful operations.

We show that the effectiveness of the self-adapting algorithms is dependent on the parallelism profile of the workload. Compared to a non-adaptive runtime, performance benefits were observed with irregular and coarser grained applications; performance was similar for finer-grained highly parallel ones.

References

- [1] Adapteva Inc. Epiphany Architecture Reference (G3), 2011.
- [2] K. Agrawal, Y. He, W. J. Hsu, and C. Leiserson. Adaptive scheduling with parallelism feedback. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming - PPOPP '06*, pages 100–109, New York, New York, USA, 2006. ACM Press.
- [3] K. Agrawal, C. Leiserson, Y. He, and W. J. Hsu. Adaptive work-stealing with parallelism feedback. *ACM Transactions on Computer Systems*, 26(3):1–32, Sept. 2008.
- [4] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems*, 34(2):115–144, Jan. 2001.
- [5] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI '99 Proceedings of the third symposium on Operating systems design and implementation*, pages 45–58. USENIX Association, 1998.
- [6] A. Baumann, P. Barham, P. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schupbach, and A. Singhanian. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating Systems Principles*, pages 29–44. ACM, 2009.
- [7] A. Baumann, S. Peter, A. Schupbach, A. Singhanian, T. Roscoe, P. Barham, and R. Isaacs. Your computer is already a distributed system. why isn't your OS? In *Proceedings of the 12th conference on Hot topics in operating systems*, page 12. USENIX Association, 2009.
- [8] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *35th Annual Symposium on Foundations of Computer Science, 1994. Proceedings.*, pages 356–368. IEEE Comput. Soc. Press, 1994.
- [9] Y. Cao, H. Sun, D. Qian, and W. Wu. Stable Adaptive Work-Stealing for Concurrent Multi-core Runtime Systems. In *2011 IEEE International Conference on High Performance Computing and Communications*, pages 108–115. Ieee, Sept. 2011.
- [10] Q. Chen, M. Guo, and Z. Huang. CATS: cache aware task-stealing based on online profiling in multi-socket multi-core architectures. In *ICS '12 Proceedings of the 26th ACM international conference on Supercomputing*, pages 163–172, 2012.
- [11] Q. Chen, Z. Huang, M. Guo, and J. Zhou. CAB: Cache Aware Bi-tier Task-Stealing in Multi-socket Multi-core Architecture. In *2011 International Conference on Parallel Processing*, pages 722–732. Ieee, Sept. 2011.
- [12] J. Colmenares, S. Bird, and H. Cook. Resource management in the tessellation manycore os. In *HotPar10*. USENIX Association, 2010.
- [13] P. C. Diniz and M. C. Rinard. Dynamic feedback: An effective technique for adaptive computing. *ACM SIGPLAN Notices*, 32(5):71–84, 1997.
- [14] A. Duran, X. Teruel, and R. Ferrer. Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *ICPP '09 Proceedings of the 2009 International Conference on Parallel Processing*, pages 124–131. IEEE Computer Society Press, 2009.
- [15] G. Edjlali, G. Agrawal, A. Sussman, J. Humphries, and J. Saltz. Compiler and Runtime Support for Programming in Adaptive parallel environments. *Scientific Programming*, 6(2):215–227, 1997.
- [16] K. Faxen and J. Ardelius. Manycore work stealing. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, page 1, New York, New York, USA, 2011. ACM Press.
- [17] K.-F. Faxen. Efficient Work Stealing for Fine Grained Parallelism. In *2010 39th International Conference on Parallel Processing*, pages 313–322. Ieee, Sept. 2010.
- [18] D. Feitelson. Job scheduling in multiprogrammed parallel systems (extended version). *IBM Research Report RC19790 (87657) 2nd Revision*, 16(1):104–113, May 1997.
- [19] P. Gschwandtner. Performance analysis and benchmarking of the intel scc. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing (Cluster 2011)*, pages 139–149. Ieee, Sept. 2011.
- [20] M. Hall and J. Anderson. Maximizing multiprocessor performance with the SUIF compiler. *Computer*, 29(12):84–89, 1996.
- [21] M. W. Hall and M. Martonosi. Adaptive parallelism in compiler-parallelized code. *Concurrency: Practice and Experience*, 10(14):1235–1250, 1998.
- [22] K. Klues, B. Rhoden, and Y. Zhu. Processes and resource management in a scalable many-core OS. In *HotPar'10: Proc. 2nd Workshop on Hot Topics in Parallelism*. USENIX Association, 2010.
- [23] M. Kulkarni, M. Burtcher, R. Inkulu, K. Pingali, and C. Casçaval. How much parallelism is there in irregular applications? *ACM SIGPLAN Notices*, 44(4):3, Feb. 2009.
- [24] Tiler LTD. Tile processor architecture technology brief [OL]. <http://www.tiler.com>, 2007.
- [25] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Höglberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *Computer*, 35(2):50–58, Feb. 2002.
- [26] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc, 1 edition, 2012.
- [27] D. A. Padua, R. Eigenmann, J. Hoeflinger, P. Petersen, P. Tu, S. Weatherford, and K. Faigin. Polaris: A new-generation parallelizing compiler for MPPs. In *CSR D Rept. No. 1306. Univ. of Illinois at Urbana-Champaign*, pages 1–77, 1993.
- [28] A. Podobas and M. Brorsson. A Comparison of some recent Task-based Parallel Programming Models. In *Proceedings of the 3rd Workshop on Programmability Issues for Multi-Core Computers, (MULTIPROG'2010), Jan 2010, Pisa*, pages 1–14, Pisa, 2010.
- [29] A. Schupbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*. Citeseer, 2008.
- [30] S. Sen. *Dynamic processor allocation for adaptively parallel work-stealing jobs*. PhD thesis, Massachusetts Institute of technology, 2004.
- [31] G. Varisteas and M. Brorsson. Automatic Adaptation of Resources to Workload Requirements in Nested Fork-join Programming Models. Technical report, KTH Royal Institute of Technology, 2013.
- [32] M. Voss and R. Eigenmann. ADAPT: Automated de-coupled adaptive program transformation. In *International Conference on Parallel Processing, 2000.*, pages 163 – 170. IEEE, 2000.

- [33] D. Wentzlaff and A. Agarwal. Factored operating systems (fos). *ACM SIGOPS Operating Systems Review*, 43(2):76, Apr. 2009.