Abstract

We provide a work-stealing scheduling method for nested fork/join parallelism that is mathematically proven to self-adapt multiprogrammed applications resource allocation to the current workloads’ individual needs while it takes available resources into account. The scheduling method both scales up the allocated resources when needed and down, when possible.

The theoretical model has been implemented in the Barrelfish distributed multikernel operating system and demonstrated to function on a simulated x86_64 multicore platform.

The work presented here is the first step towards a complete framework for the system-wide scheduling and load balancing of multiprogrammed many-core systems, assuming a variety of workload types and guaranteeing at least average execution for each running program.

Categories and Subject Descriptors D.4.1 [Operating Systems]: Process Management—Multiprogramming

General Terms multiprogramming, tasks, fork-join, parallelism, workload, mathematical model

Keywords

1. Introduction

We believe that future manycore applications will be multi-programmed and of dynamic nature with varying degree of parallelism during an application’s life time. We also believe that resource management will only become more important as there are more and more resources on-chip to manage.

Nested fork/join, or task-centric, programming models have been shown to behave gracefully in the presence of varying amount of resources. There is an opportunity to be able to scale down or up the amount of resources depending on the current workload requirements and the available power budget at the moment. Current task-centric run-time systems do not utilize this opportunity.

Furthermore, multiprogrammed parallel applications complicates the situation. Current operating systems can not handle this situation well as threads from different applications are not differentiated from each other and the system is at the users’ mercy to not overcommit the resources with more threads than the system can handle well. For these reasons we propose a mathematically proven technique that automatically adapts the amount of resources available to each application both individually and when multiprogrammed.

The algorithm is completely distributed and without a central manager which could become a bottleneck.

In this paper we present a theoretical model of a self-managed distributed, work-stealing scheduler that maintains islands of cores that grow or shrink with application needs. The adaptation of cores (resources) to an island is completely self contained by an application and no central system-wide scheduler needs to be consulted. We implemented this theoretical model in the Barrelfish operating system [3] to demonstrate its effectiveness and performance.

Experiments show that both the mathematical model performs as expected; although tested through an unoptimized implementation, the behavioral performance of several test applications validates the use of non-random work-stealing.

Our main contributions are (i) the ability to theoretically model an application’s processing needs with respect to the number of cores needed and, (ii) the implemented framework that greatly reduces the need for system-wide synchronization and load-balancing.

2. Principles

The main focus of the work reported here it to maximize the overall utilization and throughput of such a multiprogrammed system. Resources are typically wasted in such configurations due to deficiencies in programming models that have been designed for and tested on dedicated hardware.

Our work is based on three major factors:
The presented model is based on three fundamental principles:

- The benefits of space sharing resource management, regarding scalable manycore systems and system-wide load balancing [15].
- The performance prevalence of task-centric work-stealing programming models.
- The Barrelfish distributed operating system, providing scalability and portability [3].

Multiprogrammed manycore systems’ successful load balancing needs two main ingredients; knowledge of available resources and each application’s effective resource usage. Traditionally, gathering this information is based on feedback mechanisms to a centralized system scheduler [1]. We have previously highlighted a crucial question [14]: which part of a system can optimally decide about system-wide allocation of resources? Related projects have delegated the full responsibility of that task to a system-level scheduler that aggregates efficiency measurements on all running applications and periodically partitions the resources.

It is our belief that such schemes are inflexible when scaled, mainly due to their dependence on the periodic gathering of measurements. The variety in target applications makes the determination of such period very subjective. Our model, in contrast, tries to decouple judgment of application efficiency and decisions on resource allocation. We achieve this by allowing the application to determine on its own if it should change its resource allocation. Realization of such a change translates to a request to the system scheduler which in turn can reject it according to current availability. When performing system-wide resource management, it isn’t possible to avoid the need for periodic events; however, delegating decision making to the application, allows their auto-tuning and optimization.

For the purposes of this project we assume a custom fork-join programming model that is quite similar to Cilk [6] and WOOL [10]. We have made two major customizations to the semantics of those models. The use of persistent shared worker threads, created at boot-time as a single worker per physical core. Second is a complex non-random work-stealing protocol between those workers.

Without any control over the worker threads, each application can be considered as a simple task-generator and in our model called a taskset; such a set includes all the tasks to be spawned during the execution of an application. Each taskset is assigned to a subset of the existing workers, forming a work-stealing island. Workers can be shared between islands. All such islands can change their sizes, meaning a change in the number of workers allowed to process each taskset. The mechanism for deciding any such change is self-contained and consists our main contribution.

The presented model is based on three fundamental principles:

1. **Island size containment**: due to significant latencies occurred transferring data between distant core, expanding the island should occur only when the existing parallelism allows it, while it should be reduced when there is not enough work produced.

2. **Space sharing prioritization**: inter-island sharing of workers should be minimized to the ones on the virtual borders of the islands involved.

3. **Malleability cost minimization**: with variable work-loads, malleability of the island size is usually frequent. For that reason adding and removing workers should be fast.

### 3. Architectural assumptions

The proposed model is based on a specific set of assumption for the underlying architectures to be deployed on.

1. **Topology formation**: We have assumed that cores form a mesh grid, where cores are connected vertically but not diagonally. One example of such architecture is the Tilera platform.

2. **Topological dimensions**: The model has been built with any number of dimensions $m$ in mind; however, this paper assumes either one or two dimensions.

3. **Topological border isolation**: It is assumed that there is no direct interconnect between cores placed on the outer parts of the chip. This is a restriction enforced by our current implementation and should not break the model in such cases, i.e. a taurus.

In other words, while forming islands and distributing workers among them we have assumed either a serial formation of all cores, or a mesh grid, as shown in figure ??.

An immediate corollary of the above assumptions is the number of immediate neighbors $N_i$ for each core $i$.

$$N_i \leq 2m$$

(1)

while it is obvious that for the outer cores the available neighbors are at most half.

### 4. The mathematical model

#### 4.1 Definitions

Here are some definitions that are used in defining the mathematical model.

$\|\cdot\|$: Measure is a function over any set in this model which provides the size of that set as a count over the elements it includes.

$W$: Workers set. There is one worker for every available physical core in the underlying architecture. $^1$

$^1$Available are the cores that are not reserved for the operating system and other functions.
\(w_i \in W\): a **worker** \(i\)

\(s\): **Task-set**, is defined as the virtual set of all the tasks that originate from a single initial task run on a source \(S_s\). Practically it corresponds to a single application executed using this model.

\(S_s\): the **source** worker that initiates a task-set \(s\). This is not selected randomly and it is perceived as the topological center of control for the dissemination of tasks.

\(hc(w_1, w_2)\): **distance** of a worker from another worker as the shortest-path (hop-count) between their respective cores, according to the architectural assumptions. We write \(hc_{i,s}\) (or just \(hc_i\) if talking about a single task-set) and mean \(hc(t_i, S_s)\).

\(d_s\): **Diaspora** of \(s\). Maximum \(hc\) of a worker from \(S_s\) to be allowed to steal from \(s\). Diaspora is used to contain a task-set within a specific island, while it also defines and controls the size of the island. Diaspora is non-periodically malleable according to the requirements of the task-set, in correlation to those of the system.

\(T_s\): **thieves**, defined as the set of workers within a distance \(d_s\) from \(S_s\). These are the only workers allowed to steal a task belonging to a task-set \(s\).

\[T_s = \{w_i \in W : 0 < hc(w_i, S_s) \leq d_s\} \quad (2)\]

\(I_s\): **Island** of task-set \(s\). This is the set of all workers assigned to process a task-set \(s\). It is equal its set of thieves \(T_s\) plus the source \(S_s\). Thus:

\[I_s = T_s \cup \{S_s\} \quad (3)\]

\(t_i, s\): a worker acting as a **thief**, where \(t_i \in T_s\).

\(B_l, s\): **Bag of tasks** of thief \(t_i\) from task-set \(s\). Each bag includes only the stealable tasks that were spawned by worker \(w_i\). Conceptually each worker has a separate bag for each island it belongs to. Tasks that enter a bag are called *(stagnant tasks)*.

\(Z_{i,s}(w_i)\): **Distance Zone** from \(w_i\). This defines the set of workers that have the same distance from a specific worker \(w_i\). Thus:

\[Z_{i,s}(w_i) = \{w_j \in W : hc(w_j, w_i) = c\} \quad (4)\]

We will say \(Z_l\) and mean \(Z_{d_l}(S_s)\) over task-set \(s\), where \(0 \leq d_l \leq d_s\). Also, assuming \(Z_j\) then we call \(Z_{j-1}\) and \(Z_{j+1}\) its inner and outer zones respectively. Finally, \(Z_1\) and \(Z_s\) are the innermost and outermost zones of the island \(I_s\); \(Z_0 = \emptyset\) since no worker can be at 0 distance with any other.

\(X_h\) & \(X_v\): respectively **horizontal** and **vertical Diaspora control axes** or simply **axes**\(^2\). These are defined as the sets of thieves neighboring only one worker of their inner zone, excluding the outermost when \(d_s > 1\).\(^3\)

\[X_s = X_h \cup X_v = \begin{cases} T_s & d_s \leq 1 \\ T_{s'} & d_s > 1 \end{cases} \quad (5)\]

\(F_s\): **Peripheral** thieves are the non-outermost ones that are not in the axes, in a task-set \(s\).

\[F_s = T_s \setminus (X_s \cup Z_s) \quad (6)\]

Figure 1: Each shade of grey is a different distance zone. The boxed workers (excluding \(S_s\)) are part of the \(X\) set. The ones with dashed borders are in \(F\).

After these necessary definitions it should be clarified that writing worker \(w_i\) corresponds to any arbitrary worker in the system \((W)\); a thief \(t_i\) [in a task-set \(s\)] is any worker that is part of a specific island \(I_s\) and can steal tasks from an active task-set \(s\); a bag \(B_l\) belongs to the thief \(t_i\). When dealing with a single task-set \(s\), the indicator \(s\) will not be written but assumed, to avoid heavily dense text.

### 4.2 Topological properties

According to the topological assumptions and the definitions of the previous section, several properties can be calculated.

**Lemma 1 (Distance zone size).** The size of each distance zone can be calculated in relation to its distance from the source \(i\) and the topology dimensions:

\[\|Z_k\| = k^{m-1}2m, \quad 0 \leq k \leq d_s \quad (7)\]

As a direct result there is formula for calculating the complete size of the thieves set.

\[T'_s = \{t_i \in T_s : \exists t_j \in T_s (d_s > hc_i = hc_j + 1 \land hc(t_i, t_j) = 1)\}\]

\(^2\)In the absence of a coordinate system to distinguish between these sets, we mathematically define only their union \(X_s\) as it’s considered trivial for the reader to realize the distinction.

\(^3\)
Corollary 1 (Thieves set size). Given lemma 1 the total number of thieves in the island \( I_s \) is:
\[
\|T_s\| = \sum_{k=0}^{d_s} \|Z_k\| = \frac{(2d_s + 1)m - 1}{m}
\]

Corollary 2 (Island size). Assuming \( m \) the number dimensions of the chip topology and \( d \) the value of Diaspora for island \( I_s \), then the maximum number of workers consisting the island are:
\[
\|I_s\| = \|T_s \cup \{S_s\}\| = \frac{(2d + 1)m - 1}{m} + 1
\]

Lemma 2 (Zone enlargement). A direct consequence of corollary (1) is that the measure of each distance zone is at least equal or larger to its inner one:
\[
\|Z_i\| < \|Z_j\|, \text{ with } 0 \leq d_i < d_j \leq d_s, \text{ and } m > 1
\]
\[
\|Z_i\| = \|Z_j\|, \text{ with } 0 < d_i < d_j \leq d_s, \text{ and } m = 1
\]

Proof. By using mathematical induction over the value of Diaspora we have that:

- For \( i = 0 \), it holds that
  \[
  \Delta \|Z\|_i = 0, \quad I_s = \{S_s\} \notin T_s
  \]
  \[
  \Delta \|Z\|_i \leq \Delta \|Z\|_{i+1}
  \]
- For \( i = \kappa \), assume that:
  \[
  \Delta \|Z\|_\kappa \leq \Delta \|Z\|_{\kappa+1}
  \]
  \[
  \kappa^{m-1}2m \leq (\kappa + 1)^{m-1}2m
  \]
  \[
  \kappa^{m-1} \leq (\kappa + 1)^{m-1}
  \]
- For \( i = \kappa + 1 \):
  \[
  \Delta \|Z\|_{\kappa+1} \leq \Delta \|Z\|_{\kappa+2}
  \]
  \[
  (\kappa + 1)^{m-1} \leq (\kappa + 2)^{m-1}
  \]
  \[
  \kappa^{m-1} \leq (\kappa + 1)^{m-1}
  \]

\[
\square
\]

4.3 Theft policy

This section defines the policy followed by workers for stealing tasks within an island. Moreover, the basic definition is expanded with a series of proved corollaries and other conclusions. This analysis paves the path to understanding and proving the malleability properties of Diaspora in section 4.4.

4.3.1 Victims set

Definition 1. Thieves in set \( s \) steal from a specific set of workers called the victims set. This set is different for each thief in a task-set \( s \) and its definition is given by the conjunction of two rules. First we have the set of thieves of that are at distance 1 from the thief \( t_i \), called \( V_{C,i} \). Second, if the thief is a member of the control axes \( X_s \) then it has additional victims, defined as all other members of \( X_s \cup F_s \) that are at distance 2; this set is empty for all thieves in \( F_s \); finally thieves in the outermost zone can steal from thieves in the same zone at distance 2. We write \( V_i \) and mean the victims set of thief \( t_i \).

\[
V_{X,i} = \begin{cases} 
\emptyset, & t_i \in F_s \\
\{t_k \in Z_s : hc(t_i, t_k) = 2\}, & t_i \in Z_s \land d_s = 2 \\
\{t_k \in X_h \cup F_s : hc(t_i, t_k) = 2\}, & t_i \in F_s
\end{cases}
\]

\[
V_{C,i} = \{t_j \in T_s : hc(t_i, t_j) = 1\}
\]

\[
V_i = V_{C,i} \cup V_{X,i}
\]

Furthermore, there can be a correlation between the \( V_{C,i} \) and \( V_{X,i} \) and the distance zones of each victim as defined by the following definitions.

Definition 2. A thief’s \( V_{C,i} \) set can be split into tow subsets according to the distance zone of each thief. The first is the Outer victims set \( O_i \), consisting of the victims in the outer zone of the thief. The second set includes all remaining victims and thus is not defined implicitly as it equals the subtraction of \( O_i \) from \( V_{C,i} \). Assuming thief \( t_i \) is in \( Z_j \)

\[
O_i = \{v_k \in V_{C,i} : v_k \in Z_{j+1}\}
\]

The purpose of \( O_i \) becomes apparent when thought as the set of outer to \( t_i \) thieves, that steal from \( t_i \). In other words it’s those workers that may look at \( t_i \) for work.

Corollary 3. Assuming thief \( t_i \) is in \( Z_j \), all members of its \( V_{X,i} \) set are members of the same distance zone.

Proof. Assume \( t_i \in X_h \) and \( v_k \in V_{X,i} \) with \( v_k \notin Z_j \). Then since by definition \( hc(t_i, v_k) = 2 \), \( v_k \in Z_s \) or \( v_k \in X_h \). A similar statement can be reached if \( t_i \in X_v \).

Eventually, all cases are a contradiction to the definition of \( V_{X,i} \). \( \square \)

\( ^4 \) It is obvious that \( V_{C,i} \cap V_{X,i} = \emptyset \) due to lemma 3.

\( ^5 \) this formula is a direct result of the geometrical properties of an island and a proof is considered trivial.
The reason for enforcing this rule is to contain the size of islands to the necessary amount of workers, as required by the overall workload of the task-set; moreover, it provides higher control over the dissemination of tasks and certain guarantees, which are described and explained later on.

Also, because of the topological assumptions:

**Lemma 4 (Victims distance).** The distance between of all members \( v_i \in V_{C,i} \) is always 2.

\[
\forall v_i, v_j \in V_{C,i} : v_i \neq v_j \Rightarrow hc(v_i, v_j) = 2
\]

**Proof.**

Let’s assume that the distance is 1:

\[
\forall v_i, v_j \in V_{C,i} : v_i \neq v_j \Rightarrow hc(v_i, v_j) = 1 \Leftrightarrow \\
\begin{align*}
&hc(v_i, v_j) = 1 \\
&hc(v_i, t_k) = 1 \\
&hc(v_j, t_k) = 1
\end{align*}
\]

which is a contradiction.

Now assume that:

\[
\forall v_i, v_j \in V_{C,i} : v_i \neq v_j \Rightarrow hc(v_i, v_j) > 2 \Leftrightarrow \\
\begin{align*}
&hc(v_i, v_j) > 2 \\
&hc(v_i, t_k) = 1 \\
&hc(v_j, t_k) = 1
\end{align*}
\]

\[
\exists v_n \in V_{C,i} : hc(v_n, t_k) = 1 \Rightarrow hc(v_n, v_i) = 1
\]

impossible due to the previous step

\[\square\]

Combining definition 1 and lemma 4 we get the following important conclusion.

**Lemma 5 (Victims set equality).** Each thief \( t \) in any island \( I_s \) has a unique victims set \( V_t \).

\[
V_1 = V_2 \Leftrightarrow t_1 = t_2, \quad t_1, t_2 \in I_s
\]

**Proof.** Let’s assume that

\[
V_1 = V_2 \Leftrightarrow \forall v_1 \in V_1, v_i \in V_2 \Leftrightarrow \\
hc(v_k, t_1) = hc(v_k, t_2) = 1 \Leftrightarrow \\
hc(t_1, t_2) = 1 \Leftrightarrow \\
t_1 \in V_2 \\
t_2 \in V_1
\]

\[\square\]

\[\text{\scriptsize 6 by definition if } hc(v_i, v_j) = 0 \text{ then } v_i = v_j\]

Considering the geometrical properties of an island, it is also possible to deduce some interesting conclusions on the size of the victims set of each thief.

**Corollary 4 (Victims set size).** The size of the victims set for each thief in a task-set \( s \), is given in relation to the number of topological dimensions \( m \) and the thief’s distance from the source \( hc(t_i, S_s) \).

\[
0 < ||V_{C,i}|| \leq \left\{ \begin{array}{ll}
2m & , 0 < hc(t_i, S_s) < d_s \\
m & , hc(t_i, S_s) = d_s
\end{array} \right.
\]

### 4.3.2 Victims prioritization

**Definition 3 (Victims prioritization).** Thieves do not steal from their victims randomly but in search of stealable tasks they will iterate their victims set in a specific order. This order is defined as a partial order over the set of victims of each thief \( P(v_i \in V_{t_i}) \), defined by two criteria establishing the priority of one worker-victim over another.

To sum up:

1. Priority increases proportionally to the distance of the victim’s distance from \( S_s \). This means that the groups of thieves that belong in the same distance zone have the same priority.
2. Within a specific distance zone from the source worker, priority decreases proportionally to the size of their victims set.

\[
P(v_i \in V_{t_i}) > P(v_j \in V_{t_j}) \Leftrightarrow \\
\left\{ \begin{array}{l}
hc(v_i, S_s) \geq hc(v_j, S_s) \\
||V_{t_i}|| < ||V_{t_j}||
\end{array} \right., \text{ with } i \neq j
\]

(9)

and,

\[
P(v_i \in V_{t_i}) = P(v_j \in V_{t_j}) \Leftrightarrow \\
\left\{ \begin{array}{l}
hc(v_i, S_s) = hc(v_j, S_s) \\
||V_{t_i}|| = ||V_{t_j}||
\end{array} \right., \text{ with } i \neq j
\]

(10)

Equation (10) shows that it is possible to have conflicts in \( P(\bullet) \), where multiple victims share the same priority. For this model it has no real value to distinguish between those victims, so it is left up to the implementer to decide on secondary ordering criteria.

In simple words, victims selection prioritizes outer neighbors over inner ones; moreover, among the outer victims the first selected has the least victims in its zone. The purpose for this mechanism is explained in section 4.5.1.

### 4.4 Malleability of Diaspora

As mentioned earlier, this model islands in this model are self managed. This means that the size of the island can be increased or decreased automatically by evaluating specific conditions, called the Diaspora Malleability Conditions or DMC. Increase happens when it is identified that the amount of produced work is enough to utilize more workers,
while the size is decreased when border workers are found underutilized\(^7\).

**Conjecture 1.** The conditions are as follows:

- \(d_s\) is **increased** when the size of the bag of each thief of \(s\) in \(X_s\) increases beyond \(L\).

\[
d_s^+ \iff \|B_s\| > L \geq \|O_i\|, \forall w_i \in X_s \quad (11)
\]

Where threshold \(L\) will be defined later on.

- \(d_s\) is **decreased** when for all thieves of \(s\) that are at distance \(d_s\) from \(S_s\), their bag is empty.

\[
d_s^- \iff \|B_s\| = 0 \quad (12)
\]

### 4.5 Proof of the Diaspora Malleability Conditions

It is important to showcase and later prove how these conditions reflect a change in the workload which can justify adding or removing workers from an island. Before doing so, it will be helpful to interpret section 4.3 from this perspective.

#### 4.5.1 Workload flow

According to the rules of stealing and how the victims set is constructed, from section 4.3, there is a specific flow of workload among the thieves of an island. It is imperative for the validation of this model to identify its laws. This will be done here, first descriptively with a generalized example platform and then formally stated.

Assuming a 2-dimensional topology, with a single running task-set \(s\), a source \(S_s\) and diaspora \(d_s = 0\). This is the case for single-core execution in this model.

By increasing diaspora to 1, at maximum 4 more thieves are added into the island (1). These are all at distance 1 from the source and at distance 2 from each other. Since all are in \(X_s\), they steal tasks from the source and each other; the source worker can also steal from any of them. Hence, the workload is quickly disseminated and shared among all workers. Although these workers maintain the same behavior with all higher values of \(d_s\), some new thieves behave much differently.

Increasing diaspora to 2, will increase the island by 8 more thieves (1), totaling 12 (1) plus the source. An example is given in figure 1a, going up to \(d_s = 3\). The old thieves of \(Z_1\) maintain the aforementioned behavior. However, all the new thieves are in \(Z_s\) and since \(d_s < 3\), their \(V_{X,i}\) is empty; thus they can steal work from inner thieves according to figures 2a and 2b but not each other. So, one can say that their purpose is to just lighten the load in \(Z_1\).

Assuming that the load increases and condition (11) is met, diaspora is increased to the value of 3. At that point \(X_s\) is extended to include a subset of \(Z_2\), as shown in figure 1, while the rest of \(Z_2\) populates set \(F_s\); all new thieves are in \(Z_s\) but now their \(V_{X,i}\) set is not empty. The flow of workload in this scheme is very different. The new members of \(X_s\) at \(Z_2\) can steal tasks from \(F_s\) while the latter workers steal from \(Z_1\). Thieves from \(Z_s\) will create a new ring distributing the workload, while the main purpose of the \(F_s\) thieves is to reflow it inwards (due to prioritization).

If the existing workload justified the increase of \(d\), then new tasks will be stolen outwards and the workload will eventually be taken inwards again through \(F_s\). Thus the dissemination of work is balanced. If the load did not justify the increase, if there are multiple task-sets bordering or even overlapping one another, the thieves of \(Z_s\) will be able to steal work from them thus negating the bad estimation. Eventually, as well as in the case of a single task-set, diaspora will quickly decrease again.

To prove that the DMC conditions are correct it is required to prove their necessity and sufficiency. The first part is straightforward, just show that when each condition is true there is indeed an increase or decrease in workload enough to utilize one more or less zone of workers. For that it is necessary to add a few new definitions:

**Definition 4.** Task generation rate. \(\frac{\Delta G_i}{dt}\) is the rate at which a worker spawns new tasks in an island \(I_s\). We write \(\frac{\Delta G_X}{dt}\) for the overall rate in the axes set \(X_s\) or \(\frac{\Delta G_s}{dt}\) for the whole island. In case of referring to a different set the notation will be \(\frac{\Delta G(A)}{dt}\).
Definition 5. **Workload.** \( \text{Load}_i \) is the minimum required amount of stagnant tasks of a thief \( t_i \) in an island \( I_s \) and is defined as its task generation rate minus the number of its outer victims. Thus:

\[
\text{Load}_i = \frac{\Delta G_i}{dt} - \|O_i\| \Rightarrow \|B_i\| \leq \frac{\Delta G_i}{dt} - \sum_{j \in O_i} \left( \frac{\Delta G_j}{dt} \right) > 0?0 : 1
\]

We write \( \text{Load}_i \), \( A \) being a set, and refer to all members of that set individually.

Definition 6. **Thief’s work potential.** \( \frac{\Delta N_i}{dt} \): This corresponds to the potential of a thief to have work and is defined as its own task generation rate plus the load of its victims set.

\[
\frac{\Delta N_i}{dt} = \frac{\Delta G(t_i)}{dt} + \text{Load}_i
\]

At this point it becomes obvious that a positive value for \( \text{Load}_i \) denotes an excess in workload, while a negative lack of it. Furthermore, a positive work potential for the outermost thieves speaks towards the workload stability of the island. This conjecture stems from the stealing scheme that creates a dependency for these thieves to their inner zone. To elaborate, a high such value means that either they can produce enough work themselves or do their inner victims.

4.5.2 **Workload increase**

To prove the validity of condition (11), it is required to prove its necessity and sufficiency. The first part is straightforward; just show that when the condition is true, there is indeed an increase in workload enough to utilize one more zone of workers.

It is obvious that the given condition should translate in an increase in workload which can be expressed as follows:

\[
\|B_i\| > L \geq \|O_i\| \Rightarrow \text{Load}_i > 0 \forall w_i \in X_s \Rightarrow \text{Load}_j > 0
\]

**Proof (part 1: sufficiency).**

Assume \( \|B_i\| \geq \|O_i\| \Rightarrow \text{Load}_i > 0 \forall w_i \in X_s \Rightarrow \|B_j\| > \|O_j\| \forall w_j \in I_s \setminus X_s \Rightarrow \text{Load}_j \leq 0 \forall w_j \in I_s \setminus X_s \Rightarrow \|B_j\| \leq \|O_j\| \forall w_j \in I_s \setminus X_s \Rightarrow \text{Load}_j = 0 \Rightarrow \|B_j\| = \|O_j\| \}

However, due to how an island is constructed it holds that:

\[
\forall w_j \in I_s \setminus X_s \exists w_i \in X_s : w_j \in O_i
\]

This last part contradicts the initial hypothesis which guarantees the existence of stealable work for all of them. Furthermore, \( \forall w_j \in (I_s \setminus X_s) \setminus Z_s \) there exist at least \( m \) (topology dimensions) \( w_i \in X_s \) such that \( w_j \in O_i \), meaning the availability of excess work, comparing to the number of workers in the island. \( \square \)

However, the actual condition speaks of \( \|B_i\| > \|O_i\| \) so it requires at least one more stagnant task to justify a Diaspora increase. Practically it is left up to the implementer of this model to decide on a value of the \( L \) threshold, with a minimum of \( \|O_i\| + 1 \).

The second and last part of the proof deals with the necessity of the condition. In other words:

\[
\text{Load}_j > 0 \forall w_j \in I_s \Rightarrow \forall w_i \in X_s : \|B_i\| > L \geq \|O_i\|
\]

**Proof (part 2: necessity).**

\[
\forall w_j \in I_s : \text{Load}_j > 0 \Rightarrow \forall w_j \in I_s : \frac{\Delta G_j}{dt} > \|O_j\|
\]

\[
\Rightarrow \forall w_j \in I_s : \|B_j\| = \frac{\Delta G_j}{dt} > \|O_j\|
\]

\[
\Rightarrow \forall w_i \in X_s : \|B_i\| > \|O_i\|
\]

\( \square \)

4.5.3 **Workload decrease**

Proving the condition for reducing the value of diaspora is simpler. First it has to be shown that in the absence of excess workload the bags of the \( X_s \) set’s workers are empty.

**Proof (part 1: sufficiency).**

Assume \( \forall w_i \in X_s : \|B_i\| = 0 \Rightarrow \frac{\Delta G_i}{dt} \leq \|O_i\| \Rightarrow \forall w_i \in X_s : \|B_i\| = \frac{\Delta G_i}{dt} = 0 \Rightarrow \forall w_i \in I_s : \text{Load}_i < 0\)

\( \square \)

**Proof (part 2: necessity).** Due to the rules of stealing, excess work travels outwards while lack of it will concentrate the work inwards (lack of victims for the outermost workers). Hence:

\[
\forall w_i \in X_s \Rightarrow \forall w_j \in O_i : \frac{\Delta G_j}{dt} = 0 \Rightarrow \forall w_i \in X_s : \|B_i\| = \frac{\Delta G_i}{dt} - \|O_i\|
\]

However, lack of work means that \( \text{Load}_i < 0 \) so:

\[
\forall w_i \in X_s : \text{Load}_i < 0 \Rightarrow \forall w_i \in X_s : \|B_i\| = \frac{\Delta G_i}{dt} = 0 \Rightarrow \forall w_i \in X_s : \|B_i\| = \|O_i\|
\]

\( \square \)
5. Implementation

In order for the evaluation number to make complete sense it is important to spend some space describing the our implementation of the model. We have opted for a convenient yet flexible design. At the point of this writing it is not yet clear if our design decisions have been the best possible.

5.1 Barrelfish

The work presented in this paper has been part of the overall Barrelfish project[3]. Barrelfish is a novel approach to a distributed OS; it is based solely on message passing for inter-core communication hence uses no shared-memory, although it won't disallow user-space applications to use it. Of course in Barrelfish, almost everything is run in userspace, as the microkernel is nothing but a driver to the underlying core architecture. Thus the purpose is to investigate the benefits of a system that tries to get the best out of both paradigms.

All implementation and experimentation for this project has been performed on top of Barrelfish and currently only that. This OS comes with specific idiosyncrasies that are uncommon to other operating systems. There is no inter-core scheduler, or in other words any intelligent mechanism for distributing and migrating work across multiple cores.

Spawned threads are statically positioned to their initial hosting core unless explicitly migrated by user-space code. That’s were project lies as a unique intermediate layer that tries to do exactly that. Although Barrelfish does not use shared-memory for all system cross-core functionality, a userspace application is able to spread across multiple cores with all of its threads sharing the same address space. This model fits perfectly with our design for static, persistent, shared-by-all worker threads.

Concluding, our design and implementation comes to complement the message-passing platform of the Barrelfish operating system; it takes advantage of its benefits [4] but also had to overcome most of its peculiarities through the absence of any other scheduler.

5.2 The programming model

As stated before, we have developed a custom fork-join programming model that employs the work-stealing paradigm. It is a combination of lessons learned from various other models including Cilk++, WOOL but also OpenMP. It includes the following usual constructs:

- task = SPAWN(function, arguments): Spawns function with specific arguments. It returns a pointer to the task descriptor allowing, out-of-order sync.
- CALL(function, arguments): Locally calls function with specific arguments.
- SYNC(task): syncs a specific task, out of order.
- FOR(function, chunk, from, to, step, nowait, arguments): Spawns a function to be called a number of times, according to parameters. the caller will wait for the loop to sync unless the nowait flag is passed true. Chunk denotes how many iterations to be packed together in each task. A value of 0 will activate a mechanism similar to OpenMP dynamic strategy[8].

5.3 Workers

Each worker has one separate task queue per island it belongs to. These are implemented as array-based circular buffers, allowing for non breaking spawning of tasks. Thieves acquire a lock on their victims before proceeding with a steal.

To steal a worker will iterate his victim list according to protocol (section 4.3.2). If a victim is busy, the thief will continue to the next instead of waiting. Syncing out-of-order allows the possibility of bogus tasks in the queue. A thief will iterate a victim’s queue until it finds a task or the queue is exhausted (empty). An outcome of a DMC hit is that the victims list of each worker is dynamically malleable. Thus there is version counter attached to it. If a worker realizes a change in version will abort the current steal attempt. If trying to sync a not yet stolen task, it will be stolen and executed at that time. If the task is stolen but not done, the thief will proceed to steal.

When a worker belongs to no island it pauses execution using conditional locks. Multiplexing multiple islands happens by prioritizing islands on a "first come, first served" basis. Thus if an island produces a very high workload it is possible for a worker to not timeshare for other islands. In this design, the overlapping that the master scheduler allows plays a very important role, depending on the deployed workloads. Although multiprogramming is the goal of this project, we haven’t yet reached the point of fully investigating all such scenarios.

5.4 Master scheduler

The master scheduler is a single thread running on a reserved core. Its role is to enforce island changes and terminating islands. The master scheduler keeps track of the overall distribution of resources but takes no part in deciding the requirements of each task-set. It can be thought of as a single point of control for the partitioning of workers.

When a task-set realizes a workload increase it will ask the master scheduler for the extra resources. The latter has full control on how to answer that request. In our current design, we allow overlapping only on the outermost zone. Thus the master scheduler will comply with all increase requests, refusing to assign workers that are not idle but belong to a non-outermost zone of another island.

Workload decrease requests are always accepted. However, at this point there is no mechanism to directly reassign newly available resources. Hence it’s up to the other task-sets to request those resources again.

It is important to note that even if a request is not met in full, the model is not affected. Since an island increase happens at one zone at a time, the axes do not include any
missing workers, unless they had been missing as outermost workers earlier. Such asymmetry is provably not affecting the model in a negative way, within certain bounds; such proof has been omitted due to space limitation regarding this paper.

Evaluation of the state of the workers in the \( X \) set, is done through helper threads; one per task-set. Where that thread lies is up to the implementer to decide. In our design we have elected to place them at the same reserved core that hosts the master scheduler. The helper thread will submit a DMC request to the master scheduler. That request will wake the scheduler to act upon it. Thus is a single application scenario there is very infrequent time sharing of the core between these threads. Also we have not cared about the generated on-chip traffic through this design.

A very important configuration point for the performance of the whole system is the period of the helper thread. A very short period can lead to excess overhead, especially for highly unstable workloads, resulting in excessively frequent island changes. Similarly a very long period will defeat the purpose of having this scheduler as it could miss acting upon severe workload changes. For this paper it is a constant value. Later on we provide results showing how this value can affect different kind of workloads and also specify a refined value per application used.

### 6. Evaluation

Evaluation has been performed using the Simics v4.6, a full system cycle-accurate simulator [13]. The simulated target machine consists of 8 Intel Pentium 4e processors, each with 4 hardware threads, resulting in 32 cores total. They are clocked at 2000MHz with CPI of 1. For the purposes of the model the processor topology has been defined as an 8x4 mesh grid. At the time of this writing, development hasn’t reached the maturity to allow performance to be a priority issue. Thus at this stage we have not yet used a target that includes a specific cache hierarchy and on-chip traffic latencies. However, such experimental results will most probably be available very soon.

Figure 3: A two dimensional 8x4 mesh grid.

For our testing purposes we have ported several applications from the BOTS benchmarks [9], selected as distinctive and popular examples of specific workloads. The first group includes algorithms that produce variable and irregular workloads. These programs are Sort, FFT and Strassen as examples of regular and little irregular workloads, where the task graph is either of constant or increasing width. One interesting case is the \textit{nQueens} program, also used, which although highly parallel, exhibited rapid fluctuations. We could use that for our experiments setting the \textit{period} at a low value.

Moreover, we have also used some simple synthetic benchmarks, which demonstrate specific types of workloads. They will make it easier to show-case the functionality of our model, while also acting as a best case comparison point. The following applications were mostly borrowed from the \textit{WOOL} package:

- **Fib**: the perfect example of an embarrassingly parallel application that can and should consume all resources.
- **Stress**: stress the scheduler by varying the grain size.
- **VarLoop**: produces a variable and irregular workload, meant to identify the significance of the helper thread period.

The application we selected are not all very parallel. However they do provide a notion of recursion, in the sense that a tasks can and will spawn other tasks. In programming paradigms where parallelism is expressed through loops that spawn sequential tasks, our model would not perform best as work would not be distributed across multiple zones, thus constraining the maximum size of any island. On way that we attack this problem is with an incremental parallelization of loops, where big chunks of iterations are spread outwards their source, while every recipient (thief) will keep and execute the desired chunk and spawn for stealing all that remain.

#### 6.1 A case for non-random work-stealing

In this section we provide some experimental result to support our choice for \textit{non-random work stealing}. Through the results below we can show that given the complicated policy for victim selection an island behaves quite harmonically.

Figures 4 and 5 shows the behavior of 4 different programs regarding the aggregate amount of local and stolen tasks the executed, across different zones and thief classes. It’s important to remember that each zone comprises of workers from multiple classes (sets \( X, F, Z \)), each of which follows different work-stealing rules. For each worker but the Source, the preferred victim is self. Thus a high value would seem logical yo mean either a high production of work and a luck of thieves to steal tasks. This assumption is supported by looking at plot 5, where most programs exhibit a stable performance across most thieves.

Actually, it’s not surprising that almost all programs show an increase of local tasks in zone 2. For one it’s because of our topology as shown in figure 3. This is the biggest zone possible, while it is positioned close enough to the Source for quick access to stealable work. However, noticing the graph below, the distribution is proportional also per each worker.

Figure 5 shows a different picture for stolen tasks, as workers of all zones exhibit the same results. It’s important...
Figure 4: Aggregate values of locally created and executed tasks. On the right they are average per worker values.

Figure 5: Aggregate number of Stolen tasks per zone. On the right they are average per worker values.

to note that on a 8x4 grid the fourth zone is significantly reduced in size. As expected closer to the outskirts of an island the number of steals is low as the workload is kept inwards. This property provides several benefits when multiprogramming and allowing workers to participate in multiple islands; the outermost workers will be able to compensate for the lack of work.

An important aspect of a work-stealing algorithm is its success rate; especially when its major goal is to reduce wasted resources. According to the data plotted below (fig. 6) our model performed very well, with a success average well above failure. The reason behind this is simply and stems from the initial definitions of the model; a worker is added into an island if and only if there is enough work to utilize him.

Indeed further measurements that do not fit the limited size of this paper, showed that many of the misses were due to the victim being busy, or tasks synced out of order which leave a NULL placeholder on the serially popped task queue. Moreover, we have designed the stealing mechanism so that if a victim is locked, continue to the next. For our specific model this option gave a significant speedup.

6.2 Showcasing malleability

In this section we focus mostly on two specific test applications, namely nQueens and Varloop. Both under certain conditions can provide an acceptable level of workload variability, for testing our proposed model. The necessary configuration is a rather low period, although not too low. We specifically used 100ms, while total execution time for both was above 30sec. Experimental results for each are shown in figures 7 and 8 respectively. On the top of each graph, one sees the change of the diaspora value over time, thus the change in the size of the island. Below and using the same time values on the X axis, there are captured measurements of the average size of the bag of tasks of workers in the island.

Varloop was especially designed with unexpected workloads in mind. It performs an incremental parallel loop with a step of 3. The loop body task with recursively call either sequentially or in parallel, a version of the Fibonacci algorithm. The decision is based upon the loop counter being odd or even. Given a step of three and the variable chunk size of the loop function, the sequence is not regular, but also there is a variance between the execution time of Fibonacci.

On the other hand nQueens was selected for somewhat opposite reasons; usually it’s considered as a very parallel algorithm that would be able to utilize a high number of workers. On the official website of the BOTS benchmarks it is shown to have almost linear speedup. Of course it’s important to remember that what shown in the graph is the amount of stealable tasks in the bags of workers and it does not speak for the steal or spawn rate.
7. Related work

As this work comes to enhance Barrelfish OS with efficient resource management mechanisms for multicore and many-core architecture, there have been several other projects with the same goals. Some approaches are similar, like the Factored OS [15], Tessellation OS [7] or ROS [11], opting for partitioning of resources and a redefinition of the process model for the new requirements and complexity introduced by emerging architectures. A common point between all these attempts and our proposal over Barrelfish, is the two level split of the scheduler between the application runtime and the system level. The first understands the requirements while the second knows about the availability.

A point of divergence of our work is the focus of responsibilities between those two levels. Although a strict taxonomy and classification of real life mixed workloads is still not available, it is commonly accepted that rapid fluctuation are normal if not expected. For that reason, a centralized decider that tries to fit diverse workloads in the same set of rules is inefficient when large scale systems are considered [2]. It is our understanding that for maximizing the utilization of such systems it is necessary to change the process model. In a task-centric programming model over a distributed OS, the only currencies are the number of tasks and the number of workers processing them. The first can be modeled as the resource requirement while the latter is the actual resource to be brokered between applications. This way it becomes clear that dividing the scheduler should be combined with migrating control of the respective roles too; namely moving control of all threading to the system scheduler.

Moving on, the seminal paper on work stealing by R. Blumofe et.al. [5] provides a insurmountable foundation on the performance benefits of random victim selection. However, on large scale manycore systems and in a multiprogrammed context, random work stealing can be the reason for lack of control and flexibility. Through randomness comes unpredictability and in our case the inability to control the flow of the workload. In order to accomplish true resource partitioning and also avoid contention even within a single island, it is very important to predictably control the flow of the workload.

8. Future work

The model presented in this paper, investigates the properties of a configuration of worker threads to adapt to the existing workload produced by a single fork-join program. The next step is to extend the model, both in theory and through implementation, for multiple islands running simultaneously. Several aspects of the model are left to the implementer and one of our points of research would be to investigate different strategies. To name a few, island overlapping against forced release of resources, allocation of secondary resources like the off chip memory bandwidth and a mecha-
nism to keep track of rejected increase requests for immediate reassignment of released resources.

Our ultimate goal is a model that can guarantee average execution for multiple running applications while allowing leeway for per-application performance optimizations.

Finally, as identified recently[12] there is a need for benchmarks that produce mixed, variable and unexpected workloads. The ones used in this paper accommodate the minimum requirements for testing the functionality of the model. To that end we are striving to create a new suite that is more appropriate for testing multiprogrammed distributed systems, representing more closely real-life scenarios.

9. Conclusions & contributions

This project’s contributions are twofold. One hand there is a mathematical per application scheduling model that allows a lightweight method for the application to know its requirement of processing resources. The second is a novel framework (shared worker threads and taskset islands) that reduces the overhead of system-wide load balancing, by delegating the per-application scheduling to the application itself; this delegation is performed in a suggestive way from the application to the system, allowing the latter to reject requests for resources.

Our preliminary implementation has reached the point of adaptable single islands with performance that is worst although comparable to the standards set by other similar programming models. We are currently testing thoroughly synthetic benchmarks with fluctuating workloads to test the work dissemination capabilities of the model. Unfortunately such measurements where not mature enough for inclusion to this paper but are going to be available in the near future.

Acknowledgments

The research described in this paper has been supported by Microsoft Research, Cambridge, under MSR contract number 2011-006.

References


