Semantical Cognitive Scheduling

By

Shlomi Dolev , Avi Mendelson and Igal Shilman

Technical Report # 13-02
November 2012

Semantical Cognitive Scheduling*

(Extended Abstract)

Shlomi Dolev
Computer Science Department
Ben-Gurion University
Beer-Sheva, 84105, Israel
dolev@cs.bgu.ac.il

Avi Mendelson
Microsoft Research
Herzliya, Israel
avim@microsoft.com

Igal Shilman
Computer Science Department
Ben-Gurion University
Beer-Sheva, 84105, Israel
shilman@cs.bgu.ac.il

***Abstract*—A new scheduling policy called semantical cognitive scheduling is presented. A scheduler that obeys the semantics cognitive scheduling paradigm takes into account the semantical progress of the processes in order to produce a schedule which will imply the maximal semantical progress in executing the tasks of the system. We introduce a framework, define the general problem, provide a lower bound for the optimal algorithm in terms of time complexity and present dynamic and greedy competitive algorithms for the case of bounded (relevant portion) state. We then investigate ways to use additional process dependency information in the form of process dependency graph. The process dependency graph is used to achieve effective scheduling.**

I. Introduction

The process scheduler is a major component of operating systems. The scheduler is responsible for the assignment of CPU time to processes, while maintaining scheduling policy to obtain user interactivity, throughput, real time responsiveness, and more. User interactivity is a major goal in major operating systems such as Microsoft Windows, OS X, and Linux. The most popular scheduling algorithms that adhere to user interactivity are priority based [3] and guaranteed based scheduling [4].

Priority based scheduling is used in Linux Kernel 2.5 [8], Windows NT-based/XP, Solaris, NetBSD and more. Priority based scheduling starts with a predefined priority value for each type of process and adapts the processes priorities by monitoring the IO and CPU usage of the processes. Furthermore the algorithm penalizes CPU-intensive processes and rewards quick, interactive and starved processes (to keep the fairness property) by dynamically adjusting the priority value. Priority scheduling is commonly obtained by the use of multilevel feedback queue adaptive data structure [5].

Newer versions of Linux Kernel use the Completely Fair Scheduler (CFS) [6]. The goal of CFS is to improve user interactivity and CPU usage. CFS is a scheduling algorithm in the class of guaranteed scheduling algorithms, which tries to guarantee a fair share of the CPU to each of the processes in the system by favoring the processes that have the smallest CPU share.

* Research supported in part by Microsoft and also by the ICT Programme of the European Union under contract number FP7-215270 (FRONTS), Deutsche Telekom, Lynne and William Frankel Center for Computer Science, US Air-Force and Rita Altura Trust Chair in Computer Sciences.

The need for new semantic based cognitive scheduling. Classical scheduling techniques, in interactive systems, take into account a static process priority (niceness) and update this value by favoring processes that perform many IO operations (IO bound), while trying to remain fair towards processes that perform mostly computation (CPU bound). In reality process behavior depends greatly on the state of the Operating System. For instance, there is no need to (prioritize and) schedule the *Windows Live Messenger*, (Microsoft's instant messaging application) when there is no Internet connection. Such scheduling will result in repeatedly unanswered requests for a connection to the server. These requests would wrongly imply that the application is IO bound, and would be (wrongly) chose the application as a favorite application by (the classic) scheduler. In real life the manager usually learns the characteristics of hers/his tasks and their needs. The obtained experience is used in distributing resources to gain the maximal total performance. We suggest mechanism to mimic such a cognitive approach for the scheduler.

A *state* of the (operating) system is an abstract description of a snapshot of the memory of the operating system. For example, a *state* can be defined by the collection of the system resources available.

The computation (semantic) value when a process is scheduled in a certain state of the system is termed the *utility value* of the process in the state. We assume the existence of a *utility function* that computes the utility value.

Obtaining the utility value. Computing the utility value for an arbitrary process is not an easy task since the OS cannot possibly know each application and each version that it may execute. Due to this reason, classical scheduling techniques assume no (*a priori*) knowledge of the processes that they are about to schedule. However, we believe that today this assumption might be replaced by a different assumption: (*the diversity principle*) the amount of unique class of applications that the scheduler meets (in a span of the entire user base) is relatively small. We propose approaches to estimate the utility values:

• *White box approach*. The software vendors will provide a concrete implementation of the utility function as an integral part of the application distribution. This approach is most straight forward. An agent that lives inside the process and communicates with the OS-defined API, (similar to [2]) allows the OS to make accurate observation about the utility level of that application. On the down side it is highly unlikely that the software vendors will provide such an accurate information since low utility value will probably result in a degradation of system resources available for that application.
• *Black box approach*. This approach utilizes AI based methods, such as online-learning, in order to classify different applications into different utility categories in the absence of given explicit utility function.
• *Offline learning*. It is possible to classify applications offline and use the classification later during the execution.
• *Online learning*. By learning users feedback, we can assign a state with feedback per application, thus achieving statistical information regarding the compatibility of an application to a state. The users feedback could be obtained implicitly by interpreting the user action (like clicks, amount of time that the window reminded focused, etc) or by explicitly asking the users for their wishes.

Using this new information, combined with other measurements such as liveness and responsiveness, it is possible to make semantical driven decision based on the state of the operating system, instead of the existing heuristics that use a single uncompromising rule of action.

Paper organization. The system settings are described in Section 2. A special case where the state space is bounded is presented in Section 3 with an Optimal algorithm and greedy competitive algorithms for that case. Ways to exploit the process dependency graph is studied in Section 4. Finally concluding remarks are presented in Section 5.

II. System Settings and Goals

Our system consists of the following components:

- A set of n processes, $P = \{p_1, p_2, .., p_n\}$
- A set of r states, $S = \{s_1, s_2, .., s_r\}$
- A utility function $\mu : P \times S \to \mathbb{N}$ which assigns a value to each process and state.
- A transition function $\delta : P \times S \to S$.

A tuple $\langle P, S, \mu, \delta, s_{start} \rangle$ defines an instance to our problem, where $s_{start} \in S$ is an initial state of our system. We wish to find an ordering of the elements of P that leads to *utility maximization*: $\vec{p} \in \pi(P)$ (which when combined with the initial state s_{start}, immediately defines $\vec{s}$ as the vector of states), such that the sum:

$$\sum_{p_i \in \vec{p}, s_i \in \vec{s}} \mu(p_i, s_i) \tag{1}$$

is maximized.

Theorem 2.1: Any algorithm that solves the utility maximization problem, for which $r \geq n!/2^n$, has to consider any ordering, thus its time complexity is $\Omega(n!)$.

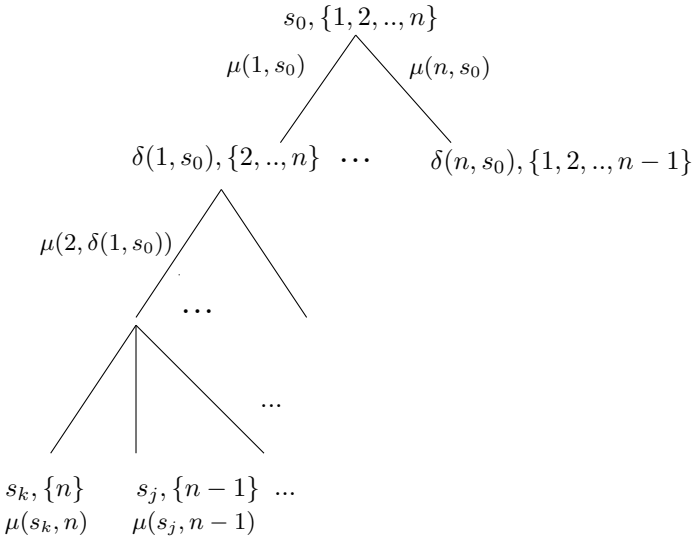

Fig. 1. Permutation tree

First, let us consider the tree that represents all possible ways to order the elements of $P = \{1, 2, .., n\}$ (Figure 1). At its root we have the vertex labeled $\langle s_0, \{1, 2, .., n\} \rangle$. Each vertex $\langle U, s \rangle$ is labeled by a set $U \subseteq P$ and a state $s \in S$. Each such vertex (except the leaves) has an out going edge e_i (for each $i \in U$, represents the selection of i) incident to $\langle W, s' \rangle$ and is labeled by $\mu(i, s)$ (the utility gained by selecting the item i). Moreover $W = U \setminus \{i\}$ and $s' = \delta(i, s)$.

For example, a vertex $\langle \{3, 9\}, s_{14} \rangle$ has an edge e_3 (and e_4) which is labeled by $\mu(3, s_{14})$ and leads to the leaf labeled $\langle \{9\}, \delta(3, s_{14}) \rangle$.

Remark 2.2: Each vertex $\langle U, s \rangle$ at depth d is labeled by a set U, such that $|U| = n - d$. It follows from the construction, whenever we follow an edge (thus gaining depth) we eliminate one element that will not reappear in the subtree rooted at $\langle U, s \rangle$.

Remark 2.3: Each path from the root to a leaf defines a permutation of P. By the previous remark we know that the depth of the tree is $n - 1$, therefore any path from the root to the leaf contains n vertices. By the previous remark we also know that each consecutive label appearing on the path is missing (exactly) one element, and this defines its place on the permutation.

Lemma 2.4: The number of edges in the permutation tree is $\Theta(n!)$.

Proof: The number of vertices in the permutation tree follows from its branching factor. At the first level a decision has to be made out of n vertices (first elements of P) at the second level a decision has to be made out of $n(n-1)$ vertices. Generally the set of vertices located at height $k \in [1, .., n]$, consist out of $n!/k!$ vertices. Thus taking the sum over the tree's heights we will get $n!(\sum_{k=1}^{n} 1/k!)$ and when $n \to \infty$ we will obtain that the number of vertices is bounded by $n!e$, therefore the number of edges is bounded by $n!e - 1 = \Theta(n!)$. ■

Lemma 2.5: Let μ be a bijection on $\mathbb{N}$, $|S| = r$, $|P| = n$.

Then the number of different weights in the permutation tree is $\Theta(r2^n)$.

Proof: Let us number the $\Theta(n!)$ vertices of the permutation tree $v_1, v_2, ..v_{cn!}$ (for a constant c) as they were discovered by a depth first (and left first) search initiated from the root. Each vertex is labeled with a state and a subset of P, therefore the number of possible different vertex labels is $r2^n$ (out of $\Theta(n!)$ vertices). Consider two subtrees T_1 and T_2 rooted at the vertices v_i and v_j with the same labels, such that $i < j$ (v_i is located to the left of v_j in the tree). Then T_2 will not introduce any new weights, so we delete T_2 and repeat the process. We stop when all the vertex labels will be unique, which means that the remaining tree will consist of $r2^n$ vertices (and $r2^n-1$ edges) which implies $\Theta(r2^n)$ different weights. ■

Remark 2.6: In general case, if r is unbounded ($r = \Omega(n!/2^n)$) and μ is bijection on $\mathbb{N}$ then the number of different weights is $\Omega(n!)$. It follows straight from substituting r in lemma 2.5.

Proof of Theorem 2.1

Let us consider an algorithm working on the input $\langle P, S, \mu, \delta, s_{start}\rangle$. Where $|P| = n$, $|S| = r = \Omega(n!/2^n)$ and μ bijection on $\mathbb{N}$. Assume in the sake of contradiction that this algorithm finds an optimal ordering $\vec{p}$ that maximizes equation 1 and achieves this by performing $k < n!$ steps. The running time of the algorithm which is under consideration, combining with the fact that the number of unique weights is $\Omega(n!)$ (remark 2.6) implies that at least one weight is not considered by the algorithm while obtaining the ordering $\vec{p}$. Meaning the algorithm does not consider a transition from some state s' to state s'' while choosing some $p \in P$. So we will modify $\mu(p, s^{'}) = \infty$, therefore achieving higher utility values in contradiction to the assumption that such an algorithm exists. □

Remark 2.7: In case $r = 1$ this problem becomes trivial, since the processes become state indifferent. Thus any ordering will produce the same total utility sum (equation 1).

Remark 2.8: For $r = 2$ this problem becomes exponentially hard. The proof for lemma 2.5 yields that the number of vertices in the decision tree is $\Theta(n!)$ yet the number of the unique vertex labels is $|\{s_1, s_2\} \times (2^P \setminus \phi)| = 2 \cdot (2^n - 1)$. Therefore, it suffices to visit only $O(2^n)$ vertices.

III. Bounded State Space

In the previous section we have shown that in the general case, when the number of states is unbounded ($r \geq n!/2^n$), it is impossible to devise an algorithm with time complexity smaller than $\Omega(n!)$. Yet it is reasonable to assume, for given operating systems, that the number of abstract states is bounded, and in fact is much smaller than $n!/2^n$.

As an example, let us consider ACPI [7] (Advanced Configuration and Power Interface) ACPI and ACPI enabled OSs allow direct control over the power management of the different hardware components using predefined power state which bear different power consumption levels. Its state space consist of: 4 device states ($E_0, .., E_3$), 5 CPU related states ($C_0, .., C_4$), 7 general system states ($S_0, .., S_6$), thus total of 140 different states.

Moreover, in a system with large state space, a state clustering technique could be used to greatly reduce the states space. Although taking this path will reduce the generality of the problem, by combining similar states under certain similar features, it will provide a more relaxed running time.

Dynamic programming approach. Next we will describe a simple dynamic programming algorithm with memorization, which works in $O(rn2^n)$ time. In order to compute the optimal value that can be obtained by ordering the process $P = \{p_{i_1}, p_{i_2}, .., p_{i_k}\}$ while starting at the state s, we can use the following recursive equation:

$$\mathbf{U}(P, s) = \begin{cases} 0 & |P| = 0 \\ \max_{p \in P}\{\mu(p, s) + \mathbf{U}(P \setminus \{p\}, \delta(p, s))\} & |P| > 0. \end{cases} \quad (2)$$

Each recursive call removes one process and thus determines its place at the permutation. Furthermore, by choosing the process p_i, the problem is reduced to: finding the optimal value obtained by scheduling processes $P \setminus \{p_i\}$ with the initial state $\delta(p_i, s)$. We have $2^n - 1$ subsets of P and each subset has to be considered with any of the r states from S. Keeping memorization in mind, it follows that it's enough to make only $r(2^n-1)$ recursive calls. In each recursive call we either return an already computed result with $O(1)$ cost, or compute the maximum in $O(n)$ time over the results of the recursive calls. Hence the total time complexity of this algorithm is $O(rn2^n)$ and the space complexity is $O(r2^n)$. (See Algorithm 1.)

Algorithm 1: OptUtility

Input: $\langle P = \{p_{i_1}, p_{i_2}, .., p_{i_k}\}, s\rangle$
Output: Optimal utility values obtained from a schedule of P, starting from the state s

```
begin
    if P = φ then
        return 0
    if Already − Computed[P, s] ≠ null then
        return Already − Computed[P, s]
    u = −∞ ;
    foreach p_i ∈ P do
        u = max {u, μ(p_i, s)+ OptUtility(P \ {p_i},
        δ(p_i, s)) }
    Already − Computed[P, s] = u
    return u
```

Greedy approach. For the cases in which the number of abstract states is small, we propose a greedy algorithm which works in $O(rn\log(n))$ time to schedule a given batch of n processes. The algorithm chooses the process that introduces the maximal utility value in the current state. The algorithm will use r priority queues Q_s, $1 \leq s \leq r$ for each state,

which will contain all the processes ordered by $\mu(p_i, s)$. All the priority queues have to be interconnected so that a removal of one process from the queue Q_s will remove this process from all other queues $Q_{s'}$ such that: $s' \neq s$. The algorithm requires an $O(rn)$ preprocessing time in order to build the r priority queues Q_s. At each step of the algorithm which starts from an arbitrary state s (initially $s \leftarrow s_{start}$) we will dequeue the process p out of Q_s (and use the links to remove it from all other queues). Then we will switch to state $s \leftarrow \delta(p, s)$, and the algorithm will repeat this step until no processes are left. Each step of the algorithm requires $O(r \log(n))$ time if implemented efficiently using heaps. Therefore the total runtime of this algorithm is $O(nr \log(n))$ and space is $O(nr)$. (See Algorithm 2).

Greedy vs optimal analysis. In terms of worst case analysis the ratio between the optimal algorithm's result and the greedy algorithm is unbounded. For example let us consider the simple scenario of two process p_1, p_2, two states s_1, s_2, and the following transition matrix δ (rows are process, columns are states).

$$\begin{pmatrix} s_1 & s_1 \\ s_2 & s_2 \end{pmatrix}$$

And a payoff function μ as follows (rows are process and columns are states).

$$\begin{pmatrix} \epsilon & \infty \\ \frac{\epsilon}{2} & \infty \end{pmatrix}$$

Its clear that when the system starts in s_1, the optimal solution to this instance is $\langle p_2, p_1 \rangle$, since the total utility value is $\frac{\epsilon}{2} + \infty$, while the greedy algorithm produces the solution $\langle p_1, p_2 \rangle$ with the utility value of $\frac{3}{2}\epsilon$.

Parametric greedy variant. The greedy algorithm presented earlier can be naturally extended with a parameter t, that determines the number of steps to look ahead. For instance, in the greedy algorithm presented earlier t was equal to 1, since we were constantly choosing the process p which gains us the maximal utility value under the current state. Another example is when $t = n$, the rule becomes: choose the sequence of n processes which when started at the initial state s_{start} will result in a maximal utility value.

The parametric extension is as follows: find the sequences $P' \subseteq P^t$ (of size t) which produce the maximal utility value in t steps. Clearly, as stated before, that for $t = n$ we will have the exact solution and for $t = 1$ we will have the greedy algorithm presented earlier. The cost in terms of time complexity, is for a fixed $t > 0$, we will have to produce $\binom{n}{t} = O(n^t)$ permutations and select the best one. Hence the runtime and space complexities in this case is $O(n^t)$.

Algorithm 2: GreedyUtility

Input: $\langle P = \{p_1, p_2, .., p_n\}, s_{start} \rangle$
Output: A schedule of P, starting from the state s_{start}
begin
  $Order = \phi$;
  foreach $s \in S$ **do**
    $Q_s \leftarrow$ Build-Max-Heap(P, key: μ_s) ;
  $s \leftarrow s_{start}$;
  $p \leftarrow null$;
  while $Q_s \neq \phi$ **do**
    $p \leftarrow$ Extract-Max[Q_s] ;
    $appendLast$(Order,p) ;
    foreach $s' \neq s \in S$ **do**
      Remove[$Q_{s'}, p$]
    $s \leftarrow \delta(p, s)$;
  return $Order$;

IV. The Use of Dependencies Graph

Interesting aspect of process scheduling is the management of background processes. Services (Services in Windows and Daemons in Unix based systems) are processes that have no or little interaction with the user. Services are used for routine tasks such as handling mounted/unmounted devices, logging or providing functionally via shared memory or any other *inner process communication* (IPC) mechanism to other application and services.

A common problem with services is that over time the amount of services increases implying a significant impact on the system boot time and overall performance. Recently this issue was partially addressed in Microsoft's Windows 7 operating system by introducing triggered services which binds a service to an event (for instance: device mounted) reducing the need to load this service at boot time. We purpose a different approach where we will study the dependencies among services and user applications (foreground); automatically managing the life span of a service.

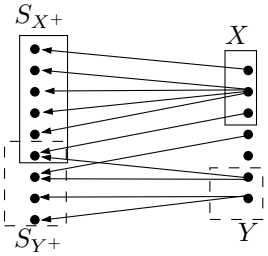

Fig. 2. Dependence graph

Consider two sets of processes: S the set of services and P the set of user applications. We model this as a bipartite directed graph $G = (S \cup P, E)$. E is the set of dependences among user applications and services. If a user application p depends on the service s then $(p, s) \in E$. We also assume

that for each $p \in P$ we have the utility value μ_p. At any given time the system keeps track of: A set X ($X \subseteq P$) of processes that their dependencies must be present (ready set). A set of services that are currently loaded S_{X^+} ($S_{X^+} \subseteq S$) that support the processes in X, in addition to different services that the system decided not to shutdown or to preload for future processes that are anticipated to arrive (hence the X^+ notation). In addition we will assume that the system has a global parameter t (which maybe dynamic with respect to the system load) that bounds the size of S_{X^+} in terms of the number of services that should be loaded simultaneously, t should be large enough to support at least the processes in X (otherwise the system is "trashing"). The question arises upon a new set $Y \subseteq P$ that becomes the ready set. The system has to decide which services to remove from S_{X^+} and which to leave and which new services to add to S_{Y^+} with respect to t. For this purpose we define α_p for each p in P, which is the normalized expected utility of p.

The need for probability and expected utility. To maximize the utility of the system the scheduler should preload services in a way that have the best probability to obtain the highest expected utility. The scheduler should take in account both the probability of an application to be activated and its expected utility. The scheduler may check the expected utility obtained by each preloaded set of services that the system may preloaded in the current state, and choose the best set of services. Accumulating statistical knowledge on the probability of usage of (class of) processes and the utility obtained from this processes can serve as the base for future preloaded decision; possibly, assuming initially that there is an equal probability to use (class of) processes p: $\alpha_p = \mu(p) / \sum_{p' \in P} \mu(p')$. Later after history is accumulated the calculation of α_p also reflects the probability of loading p, having the expected utility divided by the total of expected utilities rather than the pure utility. Note that history statistics maybe stored across boots.

Online setting. Given a processes set X, the support set S_{X^+} and a new set of processes Y. We wish to select a new set S_{Y^+} that maximize the excepted utility of the processes to come. First we evict the process in $Y \setminus X$ and replace them with the process of Y, and we load the services required by Y. At last, the scheduler chooses to preload services according to the possible left quota of services, in a way that will yield the best expected utility.

V. Concluding remarks

The user experience during the boot of a computer is a bold example for the need for semantical cognitive scheduling. In many cases a user (that gives an indication that a frontal presentation should be uploaded) waits for other services to be uploaded (e.g., wireless communication capabilities) which dramatically slows the interactive experience. Thus, new paradigms and methods that learn and react to the needs of a user in a seamless fashion are of great importance. To the best of our knowledge our study is the first to define the semantical progress requirement and to demonstrate ways to achieve such progress. We plan to demonstrate the benefit by prototyping methods that incorporate machine learning techniques.

References

[1] S. Dolev and R. Yagel, "Stabilizing Trust and Reputation for Self-Stabilizing Efficient Hosts in Spite of Byzantine Guests", *Proc. of the 9th International Symposium on Stabilization, Safety and Security*, pp. 266-280, 2007.

[2] J. J. Hanson, *JMX: Java Management Extensions*, Apress, 2004.

[3] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, Wiley, pp. 162-169, 2004.

[4] A. S. Tanenbaum, *Modern Operating Systems* 2nd Edition Prentice Hall, Chapter: 2.5.3 Scheduling in Interactive Systems, March 2001.

[5] L. A. Torrey, J. Coleman and B. P. Miller, "A comparison of interactivity in the Linux 2.6 scheduler and an MLFQ scheduler", *Commun. ACM*, Vol. 43, pp. 99-105, April 2000.

[6] The Completely Fair Scheduler, `http://kerneltrap.org/node/8059`.

[7] Advanced Configuration and Power Interface Specification, `http://www.acpi.info`.

[8] The Linux Kernel, `http://www.kernel.org/`.

[9] A. Negi and K.P. Kishore, "Applying machine learning techniques to improve linux process scheduling", *In TENCON 2005 IEEE*, Region 10, pages 16, Nov. 2005.

[10] J. Dhok, V. Varma, "Using Pattern Classification for Task Assignment in MapReduce", *International Institute of Information Technology, Hyderabad, India*, 2005.

[11] Suranauwarat Sukanya and Taniguchi Hideo, "The design, implementation and initial evaluation of an advanced knowledge-based process scheduler", *SIGOPS Oper. Syst. Rev.*, Vol. 35, Num. 4, pp. 61-68, 2001