Dan Tsafrir – CV

Contact

Address (office)	Ross 108, School of Computer Science and Engineering, The Hebrew University – Edmund Safra Campus, Jerusalem 91904, Israel
Phone no. (office)	+972-2-6585766
Address (home)	21-a Ha'Banai Street, Jerusalem 96264, Israel
Phone no. (home)	+972-2-6537265
Email	dants@cs.huji.ac.il
Homepage	http://www.cs.huji.ac.il/~dants

Research Interests

Parallel and Distributed Systems, Operating Systems, Performance Evaluation, Modeling, Scheduling

Objective

To obtain a post doctoral position in a leading research institute, in an environment that will allow me to expand my knowledge on computer systems, interact with leading scholars, and conduct high quality research.

Publications

[all publications are also available through my homepage, as specified above]

***Refereed*:**

[1] Y. Etsion, D. Tsafrir, and D. G. Feitelson, "Process Prioritization Using Output Production: Scheduling for Multimedia". In *ACM Trans. on Multimedia Computing, Communications & Applications (**TOMCCAP**), 2(4), Nov 2006. To appear*

[2] D. Tsafrir and D. G. Feitelson, "Instability in Parallel Job Scheduling Simulation: The Role of Workload Flurries". In *IEEE International Parallel & Distributed Processing Symposium (**IPDPS**), Apr 2006. To appear*

[3] D. G. Feitelson and D. Tsafrir, "Workload Sanitation for Performance Evaluation". In *IEEE International Symposium on Performance Analysis of Systems & Software (**ISPASS**), Mar 2006.*

[4] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, "System Noise, OS Clock Ticks, and Fine-Grained Parallel Applications". In *ACM International Conference on Supercomputing (**ICS**), pp. 303-312, Jun 2005*

[5] D. Tsafrir, Y. Etsion, and D. G. Feitelson, "Modeling User Runtime Estimates". In *Job Scheduling Strategies for Parallel Processing (**JSSPP**), pp. 1-35, Springer-Verlag, Jun 2005. Lecture Notes in Computer Science Vol. 3834*

[6] Y. Etsion, D. Tsafrir, and D. G. Feitelson, "Desktop Scheduling: How Can We Know What the User Wants?". In *ACM International Workshop on Network & Operating Systems Support for Digital Audio & Video (**NOSSDAV**), pp. 110-115, Feb 2004*

[7] Y. Etsion, D. Tsafrir, and D. G. Feitelson, "Effects of Clock Resolution on the Scheduling of Interactive and Soft Real-Time Processes". In *ACM International Conference on Measurement & Modeling of Computer Systems (**SIGMETRICS**), pp. 172-183, Jun 2003*

[8] D. Tsafrir and D. G. Feitelson, "Barrier Synchronization on a Loaded SMP Using Two-Phase Waiting Algorithms". In *IEEE International Parallel & Distributed Processing Symposium (**IPDPS**), pp. 80, Apr 2002*

Submitted for publication:

[9] D. Tsafrir and D. G. Feitelson, "The Dynamics of Backfilling". *Jun 2006, Submitted to IEEE International Symposium on Workload Characterization (**IISWC**)*

[10] D. Tsafrir, Y. Etsion and D. G. Feitelson, "Is Your PC Secretly Running Nuclear Simulations?". *Apr 2006, Submitted to USENIX Symp. on Operating System Design & Implementation (**OSDI**)*

[11] D. Tsafrir, Y. Etsion and D. G. Feitelson, "Backfilling Using Runtime Predictions Rather Than User Estimates". *Feb 2005, Submitted to IEEE Trans. on Parallel & Distributed Systems (**TPDS**), pending revision*

[12] D. Tsafrir, Y. Etsion, and D. G. Feitelson, "General Purpose Timing: The Failure of Periodic Timers". *Feb 2005, to be submitted to IEEE **Computer***

	[13] Y. Etsion, D. Tsafrir, S. Kirkpatrick and D. G. Feitelson, "Fine Grained Kernel Logging with KLogger: Experience & Insights". *Feb 2005, Submitted to Software Practice & Experience (**SPE**), pending revision* [14] D. Tsafrir and Z. Balshai, "The Type of STL Iterators should be Independent of Containers' Comparators and Allocators". [Paper outline]. *Dec 2005, Submitted to Dr. Dobb's Journal (**DDJ**)* ***Technical Reports*:** [15] D. Talby, D. Tsafrir, Z. Goldberg, and D. G. Feitelson, "Session-Based, Estimation-less, and Information-less Runtime Prediction Algorithms for Parallel and Grid Job Scheduling". *Oct 2005* [16] Y. Etsion and D. Tsafrir, "A Short Survey of Commercial Cluster Batch Schedulers". *Technical Report 2005-6, School of Computer Science & Engineering, The Hebrew University of Jerusalem, Feb 2005* [17] D. Tsafrir and D. G. Feitelson, "Workload Flurries". *Technical Report 2003-85, School of Computer Science & Engineering, The Hebrew University of Jerusalem, Nov 2003* [18] D. Tsafrir, "Barrier Synchronization on a Loaded SMP Using Two-Phase Waiting Algorithms". *Master's Thesis, School of Computer Science & Engineering, The Hebrew University of Jerusalem, Sep 2001*
Patents	"System and method for backfilling with system-generated predictions rather than user runtime estimates". *Patent Application PCT/IL2006/000199*, *Feb 2006*
Academic Studies	2003 – present **Ph.D.** Student, Computer Science, The Hebrew University of Jerusalem. *Average grade: 97.6%*. Title: "Topics in runtime support and performance evaluation of computer systems". 1998 – 2002 **M.Sc.** Student, Computer Science, The Hebrew University of Jerusalem. *Average grade: 97.8%*. 1995 – 1997 **B.Sc.** Computer Science and Mathematics, The Hebrew University of Jerusalem. ***Magna Cum laude.*** *Average grade: 93.1%*.
Academic Honors and Awards	Intel - dean prize, 2004. Dean's list, Hebrew University, 1995, 1996 and 1997.
Academic Positions	2004 – present **Hebrew University of Jerusalem.** Participate in **advising M.Sc. students** in lab projects and research. 1998 – present **Hebrew University of Jerusalem.** **Teaching assistant**. Teaching mandatory courses towards a B.Sc. degree (Introduction to Computer Science, Programming Laboratory). 1998 – 2002 **Hadassah College of Technology, Jerusalem.** **Lecturer**. Teaching mandatory courses towards a B.Sc. degree (Introduction to Computer Science, Introduction to Scripting Languages in the Unix Environment).
Short Research Summary and Future Directions	***1) In Relation to Operating Systems Periodicity [4][10][12]**:* All general purpose operating systems (GPOSs) such as UNIX and Windows use periodic clock interrupts, called *ticks,* to maintain control and measure the passage of time. On each tick the kernel performs administrative tasks like accounting the CPU time used by the current process, designating it for preemption if needed, waking processes with pending signals, etc. This mechanism has been in use since the birth of GPOSs. However, due to the rapidly growing applicability of GPOSs (ranging from as little as mobile phones, cameras, and PDAs to as large as supercomputers), drawbacks of periodic timing are accumulating into a critical mass, suggesting it's time for a change. As part of a general effort to turn the GPOS from being based on polling to being event-driven, we've identified several mainstream system domains that inherently conflict with this polling design, and suggest a unified solution that eliminates the conflicts [12]:

1. **Mobile and embedded devices waste power** on unnecessary ticks that happen even if the machine is otherwise idle; power is also wasted as tasks run longer than necessary due to ticks (see section 5 below). We show that an idle crippled laptop that is disconnected from its screen and hard disk consumes at least 4W due to ticks, and more, for increased tick rates. This is the result of ticks continuously preventing the processor from maintaining a power save mode [12].
2. **Virtual machine (VM) settings suffer from excessive overhead**. The overhead generated by ticks is intensified when a hypervisor layer is positioned between the ticking OS and the hardware. Further, VM servers can be overwhelmed by the ticks of VM instances. One reported example is an S/390 mainframe for which servicing clock interrupts of multiple *idle* VMs (running Linux) led to 100% utilization of the physical processor [12].
3. Ticks constitute a serious **security breach, allowing CPU servers to be monopolized** by unprivileged applications **without being discovered**: CPU accounting is done by sampling, that is, an application that happens to run while a tick takes place is billed for the entire tick. We show that processes can take advantage of this to monopolize the machine. Further, monitoring applications (such as the UNIX 'top' utility) might report these processes as consuming 0% CPU making the attack very hard to detect. We show that Linux, FreeBSD, and Windows are vulnerable for such an attack [10].
4. **Parallel applications suffer from** "**noise**" (OS activity like ticks that is unrelated to the app.), where one late process holds up thousands of peers with which it synchronizes, leaving the entire supercomputer idle until the late process catches up. We analytically quantify the effect and empirically show ticks noise is a major source of degraded performance [4].
5. **Desktop applications** might suffer from up to 8% **slowdown** due to indirect overheads of kernel-user context switching that is triggered by ticks [4].
6. **Soft realtime** tasks suffer from **limited timing services**, as alarm signals are only be delivered as part of the tick handler, and not any sooner [12].
7. **Hard realtime systems** further **experience difficulties in predicting deterministic timing behavior**, as ticks may occur while tasks are running. Further, we show that periodic work is susceptible to significant variance in its time of execution [4], which may be dependent for example on the number of processes present in the system.

The source of the above problems is periodic ticks that, as mentioned above, turn the OS into a polling-based system. The alternative is to go event-based by leveraging the fast "one-shot timers" mechanism (timers that are set only for specific needs) made available by modern hardware. However, this allows for too many timer events that might overwhelm the system. We therefore suggest an alternative mechanism we call **"smart timers"**, which combine one-shot timers with a settable minimal interval in the interest of bounding the overhead [12]. The next step is implementing this mechanism within the context of a GPOS, and proving its feasibility by quantifying its pros and cons in all of the domains listed above. It is expected that the pros will far outweigh the cons. After concluding the case against ticks, it is suggested to try and generalize the argument by identifying other GPOS kernel subsystems that employ polling (such as networking), modifying them to be event-driven, and evaluating the benefits.

***2) In Relation to High Performance Computing [5][9][11][15][16]**:*

The most commonly used scheduling algorithm [16] for parallel supercomputers (used by IBM's LoadLeveler, Sun's GridEngine, Maui/Moab, and LSF) is FCFS with *backfilling*, where short jobs are allowed to run ahead of their time provided they do not delay the first queued job. To make this possible, users are required to provide *estimates* of how long jobs will run: both to be able to foresee when the first queued job will run, and to only backfill jobs that will terminate before that time. Jobs that violate their estimates are killed by the system so as not to delay the first queued job.

Indeed, backfilling is largely based on the assumption that users would be motivated to provide accurate estimates, as jobs would have a better chance to backfill if the estimates are tight, but would be killed if the estimates are too short. Surprisingly, the quality of estimates is usually poor. This, along with the equally surprising claim made by several studies that inaccuracy is actually *good* for performance, caused estimates to be perceived as "unimportant". Thus, the common method to artificially generate estimates for the sake of performance evaluation has been to simply multiply the actual runtime by some factor $F \geq 1$, whereas modeling of runtimes received a lot of attention.

We begin by noticing that all existing estimate models don't deliver, as we get unrealistically improved performance when simulating real logs in which real user estimates have been replaced by artificial ones. Thus, they fail to correctly model the "badness" of reality, which is significant because simulations are the main tool with which research is conducted within this domain. Possibly related and much more interesting is the mystery associated with inaccurate user runtime estimates that

several researchers have reported as improving performance if artificially made even *less* accurate.

These observations raise a few questions: Can a more realistic estimate model be found? Can the mystery be solved? And can the solutions of these questions be used to devise a better scheduler? It turns out the answer to all these questions is yes, as will be describe next.

A detailed realistic model of user estimates is developed in [5]. A key finding is that estimates are inherently modal, because human beings tend to repeatedly use the same "round values" e.g. 15 minutes, one hour etc. Further, users seem to work in bursts of very similar jobs that usually have the same estimate. The combination of modality and temporal repetitiveness is traced to be the source of the “badness” of user estimates, as jobs populating the wait-queue usually look the *same*, making it hard for the scheduler to utilize *different* "holes" in the schedule for backfilling.

The solution to the "inaccuracy helps" mystery is given in [9]. In a nutshell, increased inaccuracy (e.g. by increasing the *F* parameter above) means the scheduler thinks the first queued job will start later than it should, thereby enlarging the backfilling window. But this effect seems canceled out by backfill candidates that also appear longer. Nevertheless, We’ve shown that short jobs can exploit the larger window, keeping it open one step at a time in a kind of “heel and toe” pattern. The larger the window is, the more short jobs can enjoy it. And so increased *F* nudges the system to shift from FCFS backfilling towards Shortest Job First (SJF), explaining the improved performance.

The above can be exploited for better scheduling [11]. Attributing performance gains to inaccuracy is wrong, as it's actually the "SJFness". If this is the case, why not obtain this effect outright? Non-backfilled jobs would still run FCFS, in accordance with users’ typical perception of fairness. But backfilling would be explicitly done in Shortest-Job-*Backfilled*-First (SJBF) order. Once implemented, it became clear that better accuracy does in fact translate to improved performance. This has motivated searching for a way to improve the poor quality of user estimates.

Empirical studies have repeatedly shown that system-generated *predictions* based on users’ history jobs can be significantly more accurate than user estimates (as HPC users’ work is highly repetitive). Surprisingly, predictions were never incorporated into production schedulers, partially due to the "inaccuracy helps" misconception, but mainly because jobs are killed when exceeding their estimates (by the backfilling rules) and users will not tolerate this behavior only because system predictions come up too short. We solve this problem by divorcing kill-time from runtime prediction, and adaptively increasing predictions as needed (when proven too short). The end result is a surprisingly simple scheduler, which requires minimal deviations from current practices (e.g. using FCFS as the basis) and behaves exactly the same as far as users are concerned; nevertheless, it achieves significant improvements in performance, predictability, accuracy, and fairness [11]. This work is part of a pending patent.

There are many promising future research projects in the HPC domain that can be based on the above observations. In the interest of being short, only one is described here. It turns out that the techniques suggested above can be utilized to rid users of the (decade old) annoying need to provide runtime estimates altogether. The system can generate better predictions when appropriate user history is available, or fall back on an arbitrary value if this is not the case (and let the prediction correction mechanism kick in if this value is proven too short).

Initial results show that this is a very promising direction [15]. But further research is required: User estimates serve as kill times and therefore eliminating them will result in some jobs running longer, thereby increasing the load on the machine and degrading performance. On the other hand, allowing jobs to complete (instead of killing them) may reduce the number of re-executed jobs due to failure and cause the opposite effect. To truly evaluate the impact of eliminating estimates, algorithms like “Kaplan Meier” may be used to approximate how long killed jobs would have actually ran had they not been killed, and surveys among supercomputer users should be conducted to verify the validity of this approach. The maximal administrative runtime values may be used to derive upper bounds on performance degradation.

3) In Relation to Performance Evaluation [2][3][17]:

Computer systems’ performance depends strongly on their workload. Thus systems evaluation and ranking are often done using recorded log files as workload, assuming these are representative and reliable. Alternatively, recorded workloads serve as the basis of artificial workload models that later serve as the representative workloads.

Targeting this methodology, we show that real workloads often contain anomalies that make them non-representative and unreliable. We argue and demonstrate that, even though unpopular, recorded workloads should be sanitized before being used by removing such anomalies, and that refraining from doing so is actually the action that should be justified [17][3].

In particular, an intriguing anomaly we’ve discovered is *workload flurries* [17][2]: surges of repetitive activity caused by a single user, which dominate the log for a very short period. These

extremely rare bursts dramatically disrupt the stability of simulations such that a very short period of time within the log dominates it (e.g. one day within two years). In one occasion we found that shortening the runtime of a *single* program (out of tens of thousands, submitted over a period of two years) by merely a few seconds, yielded an 8% change in average performance of *all* the jobs (!) due to a flurry. Workload modeling is also severely affected by flurries. For example, we show a flurry turning a clearly log-uniform interarrival distribution into being abrupt and modal.

Sanitizing the workload logs solves some, but not all, of the problems. We still often encounter the "variability wall" in which small and insignificant changes in the input overturn the result of the entire evaluation (e.g. when comparing between two system alternatives). In some cases this reflects an inherent property of the evaluated systems, but we believe that in most cases this simply reflects a methodological problem.

The next step is therefore developing a methodology that overcomes this difficulty. A promising new approach, we call "shaking", is systematically inserting small insignificant (artificial) changes to the simulation's input, performing the evaluation a number of times (e.g. run a simulation a hundred times), and analyze the distribution of results. Initial experimentation suggests that this distribution is normal and that is may actually expose the "true performance" of the system being evaluated. However, a lot more work is needed in formalizing the new methodology, deciding what is the "appropriate" amount of shaking, and proving the effectiveness and usefulness of the new approach.

4) In Relation to Barrier Synchronization on SMPs/CMPs [8][18]*:*

This work is done in the context of SMPs/CMPs that are rapidly becoming a commodity. In light of this, it is expect that many types of existing workloads (e.g. of ordinary desktops) will be parallelized to exploit the new hardware. We therefore analyze what happens when running a standard bulk synchronous job (in which each task performs a short computation and then synchronizes with its peers by using a barrier) under a general purpose OS.

We have leaned (and explained why) the threads of a job tend to divide into two disjoint groups that run alternatively (that is, when one is running, the other is blocked-waiting or is ready-to-run). We have shown that by carefully choosing the spin duration that is executed while waiting for the completion of the barrier, the (unmodified) scheduler can be implicitly nudged into unifying the two disjoint groups. Thus, we effectively implicitly "gang schedule" the job, such that all its threads are simultaneously running. This significantly boosts performance as very many context switches are avoided [8][18].

Current initial simulations show that by exposing some minimal kernel state to the application, we can obtain nearly optimal performance. Thus, our future work includes implementing the suggested algorithm on a real system, by adding a new system call. At a later stage, if the approach proves successful, we intend to explore the possibility of adding hardware support for this algorithm.

5) In Relation to User – Operating Systems Interaction [1][6][7][13]*:*

A serious problem with GPOSs is that they fail to prioritize tasks based on what's important to the user, as anyone who ever tried to watch a movie while running a heavy background task would know. The problem is that GPOSs prioritize tasks that consume fewer CPU cycles and sleep a lot. The assumption is that "interactive" processes (that are arguably more important) gain from this approach as they spend most of their time waiting for user input. But this does not apply for modern multimedia applications (games, movie players etc.), which require significant CPU resources. Our attempts to solve this with other metrics that are based on CPU consumption have failed [6][7], as it turns out that for every interactive application we can find a non-interactive counterpart with similar CPU consumption patterns.

We therefore suggest identifying interactive processes by explicitly measuring their interactions with the user, and use this information to prioritize processes for scheduling. This was implemented in Linux, measured using a variety of applications, and found to be a successful approach in several important cases [1]. However, much more research is needed in making the operating system better approximate its users' wishes and accommodating them.

High quality fine-grained measurements of how the vanilla and improved scheduling schemes perform were made possible by "KLogger", a low-overhead fine-grained kernel logging utility we've developed [13]. This utility was also extensively used in other projects, mentioned earlier.