International Journal of Machine Consciousness
© World Scientific Publishing Company

1. EVALUATING THE POTENTIAL FOR USING AFFECT-INSPIRED TECHNIQUES TO MANAGE REAL-TIME SYSTEMS

W. SCOTT NEAL REILLY, GERALD FRY, SEAN GUARINO, AND MICHAEL REPOSA

Charles River Analytics,
625 Mount Auburn St., Cambridge, MA 02140, USA
snealreilly@cra.com; gfry@cra.com; sguarino@cra.con; mreposa@cra.com

RICHARD WEST

Boston University, Computer Science Department
111 Cummington St., Boston, MA 02115, USA
richwest@cs.bu.edu

RALPH COSTANTINI AND JOSH JOHNSTON

SAIC
8209 Southpark Circle #100, Littleton, CO 80120, USA
Ralph.J.Costantini@saic.com; Joshua.M.Johnston@saic.com

We describe a novel affect-inspired mechanism to improve the performance of computational systems operating in dynamic environments. In particular, we designed a mechanism that is based on aspects of the fear response in humans to dynamically reallocate operating system-level central processing unit (CPU) resources to processes as they are needed to deal with time-critical events. We evaluated this system in the MINIX® and Linux® operating systems and in three different testing environments (two simulated, one live). We found the affect-based system was not only able to react more rapidly to time-critical events as intended, but since the dynamic processes for handling these events did not need to use significant CPU when they were not in time-critical situations, our simulated unmanned aerial vehicle (UAV) was able to perform even non-emergency tasks at a higher level of efficiency and reactivity than was possible in the standard implementation.

2. Introduction

Many modern computational systems, such as operating systems for real-time applications, need to operate effectively in complex, dynamic environments. Current systems are limited in their ability to dynamically modify their behavior to suit the changing environment and user needs. Humans and other biological creatures, while far from perfect, are considerably better at adapting to dynamic environments than are computational systems. Therefore, we believe the study of human adaptation can provide insights into mechanisms to improve the performance of computational systems.

While it might not be obvious at first why adaptation mechanisms that work for humans should be suitable for computational systems, we believe there are enough architectural similarities that such a transfer of effective adaptation mechanisms is plausible. For instance, they both have multiple, concurrent threads/goals competing for time/attention resources; both are limited in terms of resources, time to respond, and ability to perceive the environment; both have limited short- and long-term memory; and both act in a social/networked environment. We, therefore, believe that computational systems, such as modern operating systems being used for real-time applications, provide a useful application area for biologically inspired control architectures.

Self-Aware Computing provides a conceptual, metacognitive approach for generating dynamic capabilities for computational systems inspired by the adaptive, introspective capabilities of biological systems [Agarwal & Harrod, 2006; Ganek & Corbi, 2003]. These metacognitive approaches are designed to observe and adapt their own processing, as well as other processing within the self-aware system, in an effort to achieve their specified goals even in dynamic environments. Agarwal and Harrod [2006] identify a number of desirable properties for self-aware systems, each of which is associated with the key goal of making the processing system adaptive.

Current metacognitive approaches (e.g., [Anderson & Perlis, 2005; Patterson et al., 2002; Rhea et al., 2003] and IBM's autonomic computing program) have shown promising results in this area, but none of these approaches make any effort to utilize the rich conceptual resources available to humans through *affective* processes. While some might think of emotions and moods as being irrational and counterproductive to effective behavior, psychologists and neuroscientists have largely come to believe that affect is an evolutionarily adaptive aspect of human behavior that is useful for living in dynamic, resource-bounded, dangerous, social environments (e.g., [Damasio, 1994]). Computational scientists and philosophers have come to believe that computational systems with multiple, competing motivations, operating with limited resources (including time, memory, and computational power), and operating in complex, dynamic, and social environments will *require* affect-like mechanisms to be effective [Minsky, 2006; Pfeifer, 1993; Sloman & Croucher, 1981; Toda, 1962].

3. Approach

Human affect moderates attentional, cognitive, and physical resources based on the state of current goals and the relevant aspects of the environment. For instance, fear results in attentional/perceptual focusing that filters out elements of the environment that would normally be attended to by resources that are needed by the threatened goal.

We believe computational systems can benefit from a similar mechanism. For instance, operating systems, which are responsible for allocating resources (e.g., central processing unit (CPU) processor time, memory, network access) to processes, can more effectively allocate these resources if they are aware when the processes are being threatened due to a lack of those resources. For instance, an unmanned aerial vehicle's (UAV's) navigation process is threatened when the UAV is in danger of colliding with another object, but it does not normally require a large number of resources for normal flight. Obviously, the success of such a mechanism relies on a level of self-awareness on the part of the processes to recognize threats, but one of the results of this effort was to demonstrate that it is possible to create such self-aware processes and that the overhead of such self-awareness more than pays for itself in efficiency improvements.

Current real-time operating systems attempt to provide scheduling functionality that allows for hard real-time tasks (i.e., tasks that must be achieved within a set amount of time) to meet their deadlines. However, systems currently in use are typically designed to support a single application in which all tasks and their worst case execution times are known a priori. Such systems fail to support applications that consist of a dynamic set of hard real-time, soft real-time (i.e., tasks with set deadlines, but without critical results if they are not met), and non-real-time tasks. Also, in the absence of adaptation to environmental conditions, such as flying near obstacles or not, time-critical tasks may utilize more resources than necessary, therefore preventing efficient use of resources for computation involving soft real-time or best-effort tasks. Our approach uses affective scheduling to adapt the priorities of tasks based on their likelihoods of success or failure. Such adaptation is driven by an analog to human emotions such as *fear* and *hope*.

4. System Overview

We designed and prototyped this concept as the *Affective Process Management Module* (APMM) within the Marvin Affective Architecture, illustrated in Figure 1. The basic Marvin architecture is adapted from Neal Reilly [Neal Reilly, 1996], where it was used to model the generation and influence of affect in software agents for games. The architecture is, in turn, based on the cognitive-appraisal theory of emotion put forth by Ortony et al. [Ortony, Clore, & Collins, 1988].

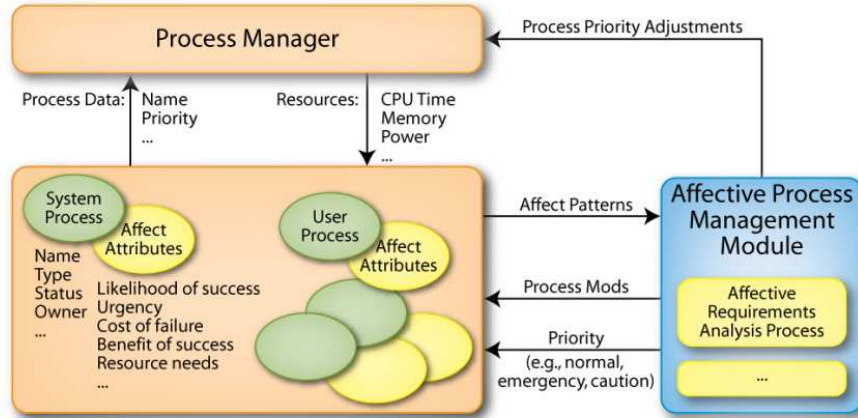

Figure 1. Marvin and APMM Conceptual Design

In this approach, S*ystem Processes* (e.g., operating system threads that correspond to software applications) are extended to support A*ffect Attributes* (e.g., likelihood of success) in addition to standard annotation (e.g., name, status). The *Affective Process Management Module (APMM)* is notified when there are certain kinds of changes we refer to as *Affect Patterns*, which are inspired by the affective literature. For instance, the APMM is updated when the likelihood of success for a high-importance process changes. The patterns correspond to process-specific "emotions." These emotions have a type (e.g., fear), an intensity (which in the case of fear is based on the importance of the process not failing and the likelihood that it will if not tended to), and a cause of the emotion (e.g., fear of failing due to lack of CPU resources or network bandwidth). The latter can be used to provide more specific and appropriate emotional responses. For instance, in humans fear of failing a test and fear of a mugger produce different responses that are suited to the situation [Neal Reilly, 2006]; similarly, in our system, fear of failure due to lack of CPU resources or due to lack of network bandwidth can be handled differently.

The emotions are used by the APMM to reallocate resources (e.g., CPU) via the standard *Process Manager* for the operating system (OS). We also have put in place mechanisms to bypass the process manager when we find we want to achieve effects (*Process Mods*) that are not directly supported by the OS. For instance, we can communicate a general *Priority* state that the processes can use to affect their behavior; if the APMM indicates that the system is in an *emergency* state, then processes might

switch into a new mode where most processes shut down and others perform shutdown, emergency behaviors.

Self-aware applications in our system are designed for use with the APMM, which enables them to update their affective attributes as needed to improve performance at key junctures. For instance, the self-aware UAV will know that the success of its avoid-obstacles objective is threatened when an obstacle is seen to be in its flight path, and the urgency of this threat is based on the distance to this obstacle. Such application-specific knowledge is most easily encoded in the applications themselves. Those applications that do not use any affective attribute management will execute under their standard priority with no modifications made by the APMM. When an affect-enhanced application updates its affective attributes, however, through a library the APMM provides, the APMM analyzes those updates and determines changes in the application's priority and resource requirements. Based on these changes, the APMM will direct adjustments to the process's priority through the OS process manager.

Consider the practical example of an "affective" UAV gathering information in a particular region. While "content," it distributes resources based on standard process priorities, just as a normal UAV. However, when this affective UAV notices a potential collision, its affective processes react to ensure high-priority goals are achieved. In this case, the key goals are to avoid the collision and to ensure that all data collected is downloaded to a network resource. Thus, the affective navigation might report a dramatic increase in urgency that is combined with the high cost of failure associated with this objective not being met. This combination of factors would cause the APMM to significantly increase the priority of the navigation process, which, in turn, would cause the UAV to shift additional resources to that process. Should the system, however, get to the point where a crash was imminent, it could switch into an emergency mode where any critical information about the situation could be communicated back to the ground station or, in the case of a military UAV, critical data could be deleted before the crash.

5. Results

To evaluate the basic feasibility and usefulness of such an affect-inspired approach to improving the performance of computational systems, we performed a number of experiments relating to the effectiveness of our approach in various environments and system implementations with a focus on fear-based processes for the purpose of these evaluations.

Our experimental systems assign an *importance of not failing* attribute to each process based on the overall criticality of the failure of such a task. For instance, in the

UAV example, the navigation process is given the highest importance as the UAV crashes if this process fails. Then, each process is adjusted to update its own *likelihood of failure* attribute as appropriate for that process. For instance, the navigation process used the distance to the nearest potential obstacle to drive updates to its likelihood of failure attribute. These importance and likelihood attributes were both assigned values in the range [0,1] and the fear associated with each process is computed by multiplying the importance and likelihood attribute values. This fear value is then used to adjust the OS priority for each process, using methods appropriate to the OS being used.

We evaluated this fear-based mechanism in four, increasingly complex systems. First, we implemented it on a MINIX®-based OS and a low-fidelity UAV. Then, we implemented the same system on a Linux OS. Next, we implemented a Linux®-based version of a high-fidelity ground robot system. Finally, we instantiated the APMM on a physical ground robot. In this section, we review these experiments and the associated results.

5.1. *Low-Fidelity simulated UAV experiments*

We implemented the APMM in both MINIX (an educational micro-kernel operating system that let us build a rapid prototype) and then in a full Linux system. We created a basic UAV simulation environment that enables a variety of time-critical system events (e.g., navigation obstacle avoidance, communications requirements, processing requirements, sensor tasks provided by simulated ground agents) to be defined and played back in the specified order to a set of processes. Each event requires a system response within a specified period of time before it fails. Failed navigation-responses lead to crashes; other failed events, such as communication and sensor responses, are logged but not considered system failures. In cases of system failures, the simulation is continued at that point to continue gathering data for the remainder of the scenario events.

The simulated UAV runs its processes as either affective or non-affective, dispatches the time-critical events to the processes, and collects profiling information. When started as affective processes, each process registers an initial affective process profile and then, during execution, uses information from the time-critical events it handles to update the values of its affective profile properties. For instance, the navigation process updates its likelihood of failure to be higher the closer it is to an obstacle (and, therefore, the less time there is to respond before failing).

The scenario was designed to mostly fully occupy the simulated UAV, with transient periods of under-utilization and over-utilization. Thus, our simulation experiment

evaluates the affective and non-affective scheduling policies under a variety of system environments.

The primary metric used for this experiment was whether the system responded within the time-limits allocated for each simulated event. Our preliminary results showed that both the MINIX and Linux APMMs provide dramatic improvements in efficiency over the non-affective versions. This is shown in Table 1. The improvements come both in terms of the ability to respond to non-catastrophic, time-critical events (*Failures* in the table are failures to respond in time to events such as sensor or communications tasks; there were 60 such events in the simulation) and catastrophic, time-critical events (*Crashes* in the table; there were two such events in the simulation).

Table 1. Performance Statistics for APMM

	Affective Linux	Non-affective Linux	% Improvement	Affective MINIX	Non-affective MINIX	% Improvement
Total Processing Time (ms)	28,471	184,135	85%	63,550	156,551	59%
Failures	0	49	100%	23	50	54%
Crashes	0	1	100%	0	1	100%

One concern with affective processing was that it might help in the extreme, emergency cases, but it might not be effective otherwise due to the overhead associated with computing affective priorities. This would imply that some non-emergency tasks that would be handled otherwise would be missed by an affective system. We found, however, that this was not the case. In fact, we found just the opposite; because the emergency-response tasks could be reduced in priority when there were no emergencies, the entire system performed more efficiently and that non-emergency, time-critical event responses were handled successfully more often. In fact, none of the 23 Failures of the affective MINIX system were handled successfully by the non-affective system, so there was never a case where the affective system performed worse than the non-affective system and many cases where it performed better.

5.2. *Linux process-level scheduling analyses*

Using our Linux implementation, we performed additional experiments to quantify the effects of affective scheduling through the use of the UAV simulation. The application

consists of a main process that communicates events to three handler processes. Depending on the type of event, it is passed either to the uavav (navigation control), uavsen (sensor reading), or uavxfr (data transfer) process for handling by a specified deadline. The experiment was run twice—once using the standard SCHED_RR policy in Linux, and another with adaptation of priorities using affective scheduling (built on top of SCHED_RR in Linux). The APMM is implemented as a library in Linux that provides an interface for affective processes to adjust their priorities.

Figure 2 shows the instantaneous requested load on the system at various points in time during the UAV simulation run. At each sampling point, the requested load associated with each outstanding event at that time is calculated as C/D, where C is the remaining time needed to process the event, and D is the event's relative deadline. The sum of these C/D values over all outstanding events is plotted on the vertical axis. The utilization measurements show that, on average, the system is approximately fully loaded, with transient periods of under-utilization and over-utilization. Thus, our simulation experiment evaluates the affective and non-affective scheduling policies under a variety of system environments.

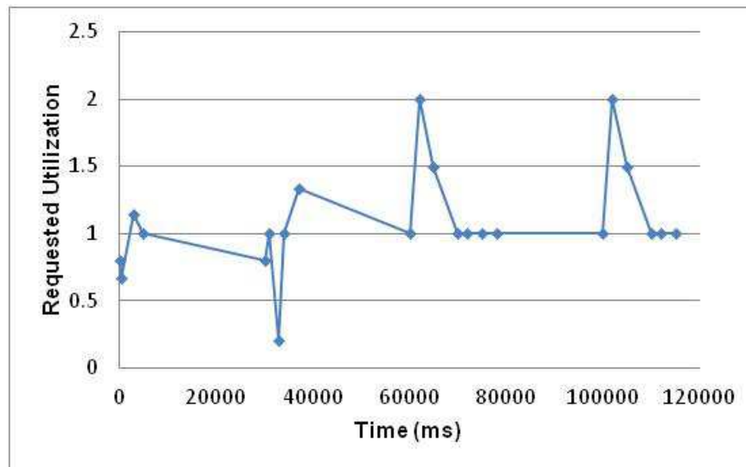

Figure 2. Utilization vs. Time

We measured the percentage of deadlines met for each of the event-handling processes in the simulation. Figure 3 shows the results, comparing the affective scheduling scenario with normal (non-affective) Linux scheduling using SCHED_RR. As shown in the plot, the uavav process achieved all deadlines in the affective case, while only one-third of the events handled by this process completed by their deadlines in

the non-affective case. For uavsen, uavxfr, and the total of all events, the affective case scores approximately 20 percent higher, compared to non-affective scheduling. This indicates that not only is the Marvin-based system better able to handle the critical events (those being handled by the uavav process) as intended, but that it performed better on the lower-priority, non-emergency processes as well, largely because the uavav could be adjusted to require both more *or less* CPU time as needed in the current situation.

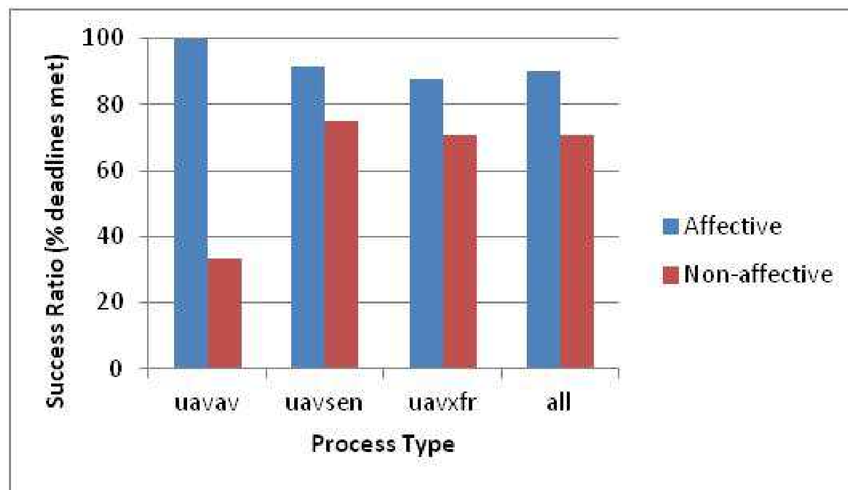

Figure 3. Success Ratio

In addition to success ratios, we record the cumulative response times over all events processed during the simulation run. Figure 4 compares the affective and non-affective cases. The vertical axis displays the total response time incurred for all events completed up to the completion of the event specified on the horizontal axis. As shown, the affective case results in lower total response time in comparison to the non-affective case.

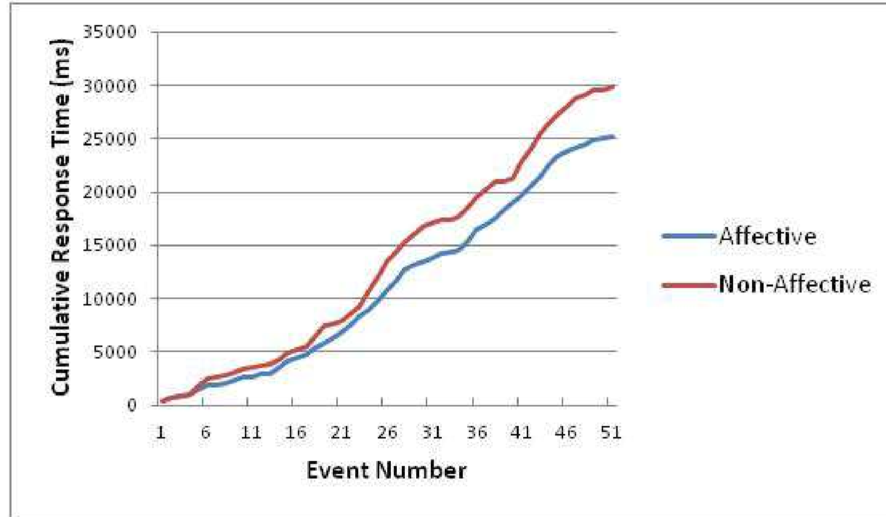

Figure 4. Cumulative Response Time

5.3. *High-fidelity ground robot simulation experiments*

The low-fidelity UAV experiments showed the basic feasibility of this approach. Next, we moved to a higher-fidelity simulator to see if the results held up. For this purpose, we used the SAIC robot simulation that was developed under a previous effort. This software supports creating waypoints to generate a path that the robot will then follow. To test the APMM, we created paths of varying complexity and executed them under increasing speeds until the robot showed signs of losing vehicular control. Another element of performance we tested was how well the robot stays within its operational envelope (i.e., how well it stays on the path) while negotiating the course.

The experiment simulated a path-following scenario involving a ground vehicle that must autonomously follow a predetermined path in the simulated environment at increasingly fast speeds, more rapid accelerations, and on narrower paths. Default Linux scheduling mechanisms were used in the APMM-disabled case. In the APMM-enabled case, affective processing was used to dynamically adjust the priorities of three processes. The vehicle control unit (VCU) process is responsible for movement control, the path pursuit process (PathPursuit) computes the necessary adjustments to keep the robot as close to the path as possible, and the arbiter process (Arbiter) mediates the execution of the other two processes. All runs of the experiment were performed on a dual-core Dell Inc.'s Latitude® laptop running a Linux 2.6.32 kernel,

which we modified with APMM extensions (such extensions were disabled for the APMM-disabled run).

Our first experiment focused on the time to complete the task. Our hypothesis was that by being able to dynamically switch between the VCU and PathPursuit tasks as each task needed CPU resources, we would be able to achieve the overall goal of completing the task quickly while staying on the path more often than in the case where the priorities of the tasks were fixed (as is standard on the MP systems). The basic idea being that the system could adjust the priorities of the various processes on the straight-aways and in the turns to use those processes that were most important at the time. In fact, we found that the APMM-enabled case completed the path-following scenario as much as 12 percent faster than in the APMM-disabled case with no loss of control.

Our second experiment focused on power usage. A key problem with many modern systems (e.g., robotic systems, smart phones), is the limits on power availability due to battery constraints. We had expected to find that the battery would drain less simply because of spending less time moving, but we also wanted to measure if the CPU and other non-navigation systems would require more or less power. Figure 5 plots the results of power usage measurements for the APMM-enabled and APMM-disabled runs of the experiment. The measurements were taken using the *dstat* utility while the system was unplugged, thereby putting a power drain on the battery. The results show a base-line power usage of approximately 33.7 watts, just before the path-following scenario begins. This base-line power usage is a result of running system peripherals such as the display, disk, universal serial bus (USB) devices, network, and so on. Measurements were taken at 1-second intervals and correspond to the instantaneous power usage at each sample. Results show decreased power usage (normalized by the base-line system power usage) of 25 percent in the APMM-enabled case when compared with APMM-disabled case. We believe this result demonstrates significant promise for using these kinds of mechanisms on cell phones and other power-constrained systems.

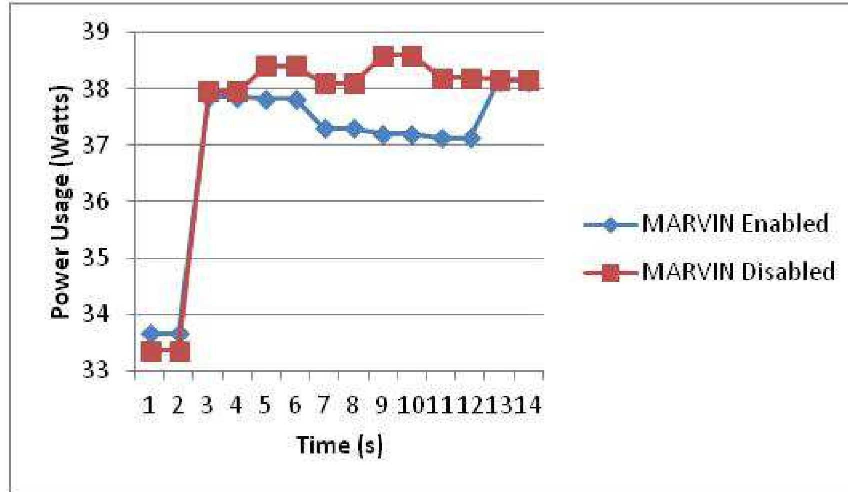

Figure 5: Power Usage Comparison

Finally, we wanted to measure how much other work an APMM-enabled system could potentially get done with the savings provided. To evaluate this, we ran the same path-following experiment for both APMM-enabled and APMM-disabled cases while running a background workload process, to compare the amount of background processing that can be done during the scenario. The background process is a CPU-bound synthetic workload that executes a loop consisting of integer arithmetic instructions. We found that the APMM-enabled system was able to process 56 percent more of these background tasks than with the APMM disabled (Figure 6). This increased capacity is due to restricting CPU bandwidth available to tasks related to path-following when not needed, which provides a larger fraction of the CPU for background tasks when using affective, real-time scheduling schemes.

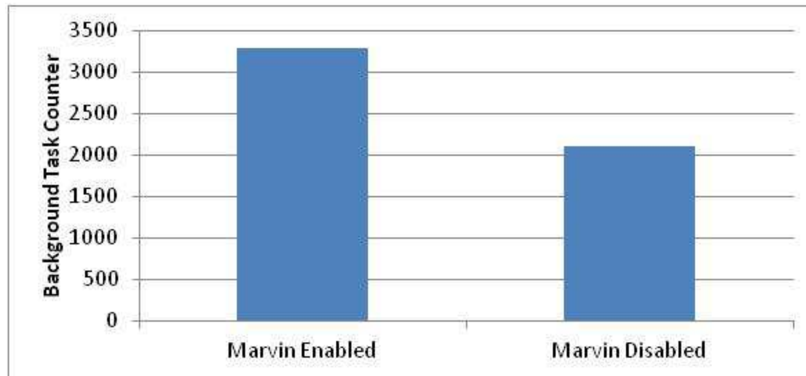

Figure 6: Background Workload Progress

5.4. *Live Robot Demonstration*

The simulation experiments demonstrate the effectiveness of the APMM capability for improving scheduling of applications to improve both emergency and non-emergency performance. We also created a live robot demonstration to show that: (1) the APMM modifications to the Linux kernel and environment require only modest application changes and (2) that they can be deployed on an existing, non-simulated platform. We did not, however, have the time or budget to perform a thorough set of quantitative tests, especially as our tests require pushing the robot to the edge of safe use, so we have not yet demonstrated that the efficiencies found on the simulated systems carry over to the physical system and leave this for future work.

SAIC modified its previously developed robot autonomy software suite for the APMM program. This meant adding approximately 50 lines of code to each of four software applications. Integration also required updating the Linux kernel to an APMM-enhanced kernel. Otherwise, this effort was one of deploying the updated robot software used in the MP simulation environment onto the physical platform.

We used a Segway® RMP 400 robot platform for the demonstration. The robot as it appeared at the demonstration site can be seen in Figure 7.

Figure 7: The SAIC Robot Used in the Demonstration

The robot was given a rectangular path to follow (Figure 8), which is shown by the red line in the figure. The software on the robot was started and stopped using a laptop computer connected via wireless radio, but all processing occurred onboard the robot.

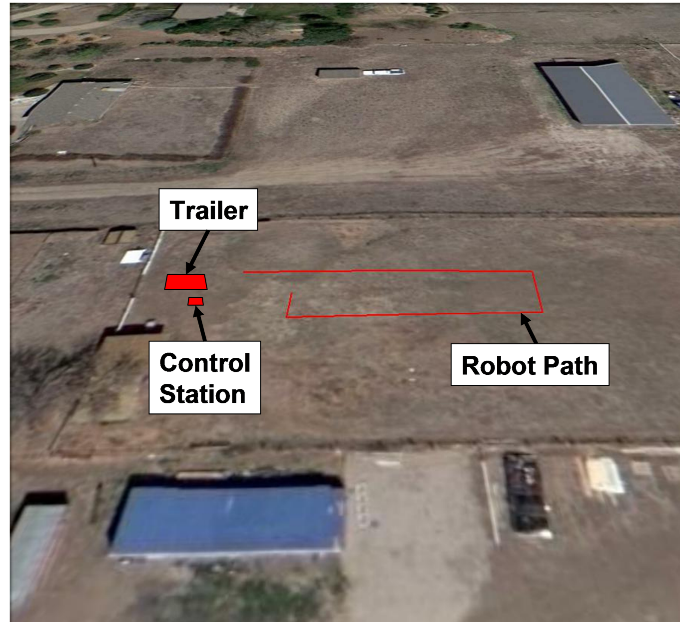

Figure 8: Aerial View of the Demonstration Site

The robot performed several complete and partial runs of the path with and without an APMM. Since the robot was constrained to well within its performance envelope for safety reasons, we did not expect to see any visually noticeable impact from running the APMM, and this was the case. The demonstration did, however, show that the APMM can be applied to an existing platform with minimal modification and no negative impacts to operation and performance, which was the objective for this live exercise.

6. Discussion and Future Work

To date, we have only used the Marvin architecture to implement and evaluate a fear-inspired method of reallocating CPU resources to threads. The point of this initial fear-focused effort was to demonstrate that using ideas from human emotional mechanisms as inspirations for computational system design is a fruitful path for generating effective resource-allocation mechanisms at all, but we believe that there are many other such mechanisms that might be developed using this same motivation. For instance, we already have the infrastructure in place to support "hope," which we believe would tend to provide additional resources to threads that are nearing completion, thus poten-

tially reducing context switching. Similarly, using the existing mechanisms, we could use fear- or hope-based mechanisms to allocate non-CPU resources, such as network bandwidth. Other mechanisms corresponding to, say, anger or pity are also possible and could be used to adjust to threads that regularly over-demand or under-receive CPU resources. We also believe reallocating physical resources, much as emotion-driven adrenaline, which is used to allocate physical resources in humans, could be used to make more power-efficient systems. This could be used to improve the performance of smart phones or other power-limited devices beyond the efficiencies already demonstrated due to reduced CPU usage.

Acknowledgments. This work was performed under DARPA contract number Contract W31P4Q-09-C-0469.

References

Agarwal, A. & Harrod, W. [2006]. Organic Computing. Cambridge, MA: MIT CSAIL / DARPA IPTO.

Ganek, A. G. & Corbi, T. A. [2003]. The Dawning of the Autonomic Computing Era. IBM Systems Journal, 42(1), 5-18.

Rhea, S., Eaton, P., Geels, D., Weatherspoon, H., Zhao, B., & Kubiatowicz, J. [2003]. Pond: the OceanStore Prototype. In Proceedings of 2nd USENIX Conference on File and Storage Technologies (FAST '03). San Francisco, CA.

Patterson, D. A., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J. et al. [2002]. Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. (Rep. No. Technical Report UCB // CSD-02-1175). Berkeley, CA: U.C. Berkeley.

Anderson, M. L. & Perlis, D. R. [2005]. Logic, Self-Awareness, and Self-Improvement: The Metacognitive Loop and the Problem of Brittleness. Journal of Logic and Computation, 15(1), 21-40.

Damasio, A. R. [1994]. Descrates' Error: Emotion, Reason and the Human Brain. New York: G.P. Putnam's Sons.

Sloman, A. & Croucher, M. [1981]. Why Robots Will Have Emotions. In Proceedings of 7th International Joint Conference on Artificial Intelligence. Vancouver.

Toda, M. [1962]. The Design of the Fungus Eater: A Model of Human Behavior in an Unsophisticated Environment. Behavioral Science, 7.

Pfeifer, R. [1993]. The New Age of the Fungus Eater. In Proceedings of Second European Conference on Artificial Life (ECAL). Brussels.

Minsky, M. [2006]. The Emotion Machine. New York: Simon & Schuster.

Neal Reilly, W. S. [1996]. Believable Social and Emotional Agents (PhD Thesis). (Rep. No. CMU-CS-96-138). Pittsburgh, PA: Carnegie Mellon University.

Ortony, A., Clore, G. L., & Collins, A. [1988]. The Cognitive Structure of Emotions. NY: Cambridge University Press.

Neal Reilly, W. S. [2006]. Modeling What Happens Between Emotional Antecedents and Emotional Consequents. In R. Trappl (Ed.), Cybernetics and Systems 2006. Vienna, Austria.