# Improving Dependability using Shared Supplementary Memory and Opportunistic Micro Rejuvenation in Multi-tasking Embedded Systems

Vinaitheerthan Sundaram[§†*] Sandip HomChaudhuri[‡] Sachin Garg[†] Chandra Kintala[†] Saurabh Bagchi[§]

§Purdue University
School of Electrical and Computer Engg.
465 Northwestern Ave., West Lafayette IN 47907

‡Motorola Software Group, †Motorola Labs
Plot#5, 66/1, Bagmane Tech Park,
CV Raman Nagar, Bangalore - 560093, India

Email: {vsundar,sbagchi}@purdue.edu, {sandip.homchaudhuri, sgarg, kintala}@motorola.com

## Abstract

*We propose a comprehensive solution to handle memory-overflow problems in multitasking embedded systems thereby improving their reliability and availability. In particular, we propose two complementary techniques to address two significant causes of memory-overflow problems. The first cause is errors in estimating appropriate stack and heap memory requirement. Our first technique, called Shared Supplementary Memory (SSM), exploits the fact that the probability of multiple tasks requiring more than their estimated amount of memory concurrently is low. Using analytical model and simulations, we show that reliability can be considerably improved when SSM is employed. Furthermore, for the same reliability, SSM reduces total memory requirement by as much as 29.31%. The second cause is the presence of coding Mandelbugs, which can cause abnormal memory requirement. To address this, we propose a novel technique, called Opportunistic Micro-Rejuvenation, which when combined with SSM, provide several advantages: preventing critical-time outage, resource frugality and dependability enhancement.*

***Keywords**: stack overflow, heap overflow, embedded systems, software rejuvenation, resource constrained fault-tolerance*

## 1. Introduction

Embedded systems are becoming increasingly ubiquitous [2] and in recent years, their complexity has grown exponentially. Specifically, the growth in requirements for real-time, networked and multi-tasking applications has led to increased presence of MandelBugs/HeisenBugs [7] in the code. These are rare bugs which are triggered typically by the change in the program run-time environment or the (invalid) input data or due to ageing of the software and are thus not deterministic in nature. It has also made the task of accurately estimating run-time memory requirement (stack and heap) significantly harder. Inaccurate estimation of memory requirements and MandelBugs cause memory overflow or out-of-memory, a grave problem in embedded systems. It leads to unexpected system crash [8] since most embedded systems, as much as 95% according to Middha et al [4], do not use virtual memory. The effect of a system crash can be a slight inconvenience (e.g. crash of set-top boxes), loss of revenue (e.g. dropped call on mobile phones), or even loss of life (e.g. crash of aircraft controller). This has increased the need for designing these systems with fault-tolerance mechanisms to enhance reliability, not just from safety perspective but also from the user-experience standpoint.

In this paper, we propose a comprehensive solution that addresses both important causes of memory overflow, namely, inaccurate estimation of a task's memory requirement, both stack and heap, over its lifetime and the presence of MandelBugs in the code. In addition to complex, inter-dependent and sometime incomplete requirements' specification of real-time, networked and multi-tasking applications, the difficulty in estimation arises from the memory requirements' dependence on input data, certain language features like function-pointers in C, and the complexity of the software, which makes it prohibitively expensive to check all paths of program execution. Inaccuracies in estimation compel the task designers, as frequently seen in industry, to deliberately over-estimate the stack and the heap requirements as a rudimentary form of robustness in the system.

The first technique we propose is a unified approach to solve both stack and heap overflow using Shared Supplementary Memory (SSM). The SSM is formed by combining the supplementary memory allocated to all tasks and making a global/shared use of the large memory so obtained. There are several advantages to the unified approach.

1. It provides improved reliability for the same amount of total memory. In other words, it can reduce total memory requirement for the same

[*] Corresponding Author. This work was done when the corresponding author was interning at Motorola India Research Labs.

reliability. Reduction in memory requirement directly translates to cost savings in ubiquitous embedded systems like mobile phones.
2. It complements the handling of Mandel-Bugs as will be explained later in Section 3.
3. By having independent memory for each task except for the supplementary part, it retains the advantages of fault-containment.
4. The performance overhead in terms of accessing shared supplementary memory is minimal as only supplementary memory is shared and not the estimated memory.

The second technique we propose is opportunistic micro-rejuvenation (OMR) to handle Mandel Bugs. Note that there is a difference between MandelBugs and inaccurate estimation and a separate technique is needed to tackle memory overflow due to MandelBugs. A memory leak or invalid input that causes non-terminated recursion might use up the entire available memory over time and hence, has to be handled differently from estimation error. MandelBugs usually occur rarely and it is hard to reproduce such bugs. Software Rejuvenation [1] or rebooting the system at random instants in time has been shown to be an effective technique to combat MandelBugs [1,7] in continuously running servers (e.g. telecom billing). Micro-reboot/Micro-rejuvenation [3] is a fine-grained rejuvenation technique that reboots the individual components instead of the machine (hardware reboot) and thus provides increased availability and continuous service. We propose OMR that is inspired by both techniques described above and adapted to resource-constrained environments like embedded systems. The key idea is that instead of using software rejuvenation for predictive maintenance, we rejuvenate tasks at opportune instants. Also, we do not rejuvenate the entire system but just the tasks that are most likely to cause an overflow in the near future. Since we do fine-grained rejuvenation at opportune instants, we call our technique opportunistic micro-rejuvenation. The rejuvenation is done when the SSM usage is greater than some threshold and the task with most memory usage, in the SSM, is rejuvenated. Since SSM crossing the threshold is rare, the number of times we rejuvenate is minimal. The main advantage of this technique is that by rejuvenating a task that is likely to cause an overflow and its dependents, without rejuvenating other independent tasks in the system, reliability and availability of the system is significantly improved. Another advantage of OMR is its inherent fault-containment property. Moreover, in critical embedded systems, critical tasks are usually comprehensively tested as opposed to non-critical tasks. OMR ensures that a Mandelbug in a non-critical task doesn't affect the critical task in an embedded system.

We have evaluated our solution using analysis and simulation. We derived the reliability of three approaches of memory layout for stack overflow assuming normal distribution for the memory requirement of a task. Since a closed form solution was not possible in the case of SSM, we resorted to simulation. We show that our theoretical results and simulation results match for the cases for which closed form is possible. We show that SSM approach can reduce the total amount of memory up to 29.31% for same amount of reliability. We have created a Stochastic Activity Network (SAN) model to analyse the availability of opportunistic rejuvenation and show slight (0.012%) improvement in availability.

The rest of the paper is organized as follows: Section 2 describes the related work; Section 3 describes the design of the two proposed techniques, SSM and OMR; Section 4 discusses the analysis and simulation methodology; Section 5 concludes the paper.

## 2. Related Work

There have been several approaches to tackle different aspects of the memory overflow problem in embedded systems [4, 11, 12]. In the literature, stack and heap overflow are handled separately. To handle the stack overflow problems, Naganuma et.al [12] suggested the possibility of keeping an unused auxiliary memory unit in the system which would serve to handle the overflowing memory needs of a faulty task. This auxiliary memory unit, however, comes at an additional cost and may not be suitable for a cost-sensitive system. Moreover, no performance study was conducted on the proposal. Biswas et.al [11] proposed handling a stack overflow by relocating the overflowing memory into the various dead global variables, unused array locations and unused heap of the faulty task; Data compression was suggested as a viable technique to free up space in these areas. However, in practice, such reclaimed space is often too less to fulfil the needs of an overflowing stack. Their results also indicate severe performance degradation due to compression. Middha et.al [4] proposed sharing free stack space from other tasks of the system in the event of an overflow in one task and this technique is referred as Multi-Task Stack Sharing, or MTSS. Evaluation of MTSS indicates significant memory saving in the stack segment to achieve the same level of reliability. However, it is performance intensive, owing to multiple levels of stack page sharing. All the above schemes were designed to address the first cause of memory overflow in the system, namely, inaccurate estimation, and would not suffice to handle the overflows caused by MandelBugs. Furthermore, none of the above solutions are generic enough to extend to heap-overflow, as they are dependent on the regularity of the stack growth and shrink model at each function activation; quite unlike a heap, where the tenure of memory occupancy is un-predictable and driven by external inputs. Accordingly, the main contribution in this paper is our solution (SSM), which handles stack and heap overflow together and the solution (OMR), which handles memory overflow due to MandelBugs in the context of an embedded system.

Software Rejuvenation [1] is a well known fault-tolerance technique to increase the system reliability by proactively restarting an ageing software system. In embedded systems, however, the processing overhead of a health-monitoring module as well as full system restart is not desirable. Micro-Reboot [3] is preferred with low overhead of component based restart. However, the primary premise of [3] was that a component restart in such applications is orders of magnitude faster than the complete reboot process, need necessarily not hold true in these systems.

In [5], the authors defined the concept of “Process Resurrection” as a fast reactive recovery mechanism to prevent an embedded system crash due to a single task failure. It was shown that the hardware exceptions, viz. segmentation fault etc, can be intercepted and used as a trigger for resurrecting the faulty process. However, they did not address the memory overflow problem in the stack and the heap.

From the preceding discourse, it is clear that a unified approach to solve stack and heap overflow has not been proposed and memory overflow due to MandelBugs has not been tackled yet.

## 3. Design and Implementation

In this section, we first provide a brief background on memory organizations in embedded systems and the assumptions we have made. Next, we present the detailed description of our Shared Supplementary Memory (SSM) and Opportunistic Micro-Rejuvenation (OMR) techniques along with implementation issues of these two techniques.

### 3.1 Memory Organization and Assumptions

The typical memory organization in an embedded system is shown in Figure 1a. The physical memory is divided into stack and heap areas and each of these areas are further subdivided. Each task pool, set of dependent tasks, has its own stack and heap allocated respectively in stack and heap areas. For this paper’s discussion, we can consider entire task pool as a single task without affecting the results. This memory organization is different from general-purpose

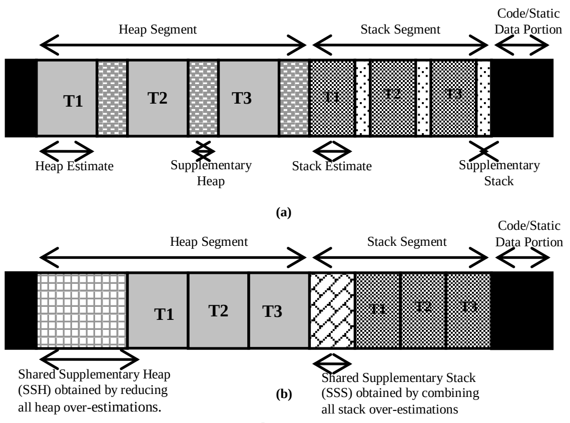

Figure 1. Memory layout or organization in embedded systems (a) A typical organization of memory in embedded systems. (b) Proposed memory organization with shared supplementary memory pool

computers such as desktop/laptop systems, which use hardware-assisted virtual memory to give the illusion that a process can use as much memory as it needs. The hardware-assisted virtual memory swaps pages/frames into secondary memory when there is no space in the physical memory, thus handling memory overflow. Most embedded chips, as much as 95% according to Middha et al. [4], do not provide hardware support for virtual memory as the performance hit every memory access incur in terms of energy can be prohibitively expensive [10]. Since there is no swap device available in embedded systems, if a task’s memory requirement even exceeds by a byte over its lifetime would cause a memory overflow, which leads to system crash. The trend of not having hardware-support for virtual memory in embedded processors will not change due to high energy cost and potential for missing real-time guarantees [4]. Therefore, we assume that most embedded processor do not use virtual memory.

Each embedded system task designer carefully estimates the stack requirement by adopting any of the known techniques [6]. However, estimating correct stack size is complicated by several factors including recursive functions of unbounded length, language features like virtual functions and function pointers, compiler features like allowing stack arrays to be of run-time dependent size and interrupt handling. A detailed list of reasons can be found in [4]. Hence, the task designers intentionally over-estimate the stack size i.e., add some supplementary to the best estimate found by their analysis.

Compile-time heap estimation is inherently difficult as actual heap usage depends on run-time input values. The industrial approach, therefore, is to run a series of test cases to evaluate the worst-case heap usage of the applications under various scenarios [4, 9]. As this method is not exhaustive, heap, similar to stack, is deliberately over-estimated and the supplementary heap added to the best heap estimate is very high. Since heap segment is usually much larger than stack segment for any task, even a slight over-estimation for each task can result in large amount of memory being used as supplementary memory for all tasks. Therefore, we assume that accurate measurement of stack and heap requirement is very difficult and in current systems, each task is provided with supplementary memory for stack as well as heap to the best estimate obtained.

### 3.2 Shared Supplementary Memory

We propose using shared supplementary memory (SSM) for handling stack and heap overflow. The technique works as follows. The supplementary memory from each task's stack (/heap) is removed and combined to form a large chunk of shared supplementary memory that can be accessed by all tasks This is pictorially shown in Figure 1b. When a task's stack (/heap) overflows, the stack is allowed to grow in SSM. We show by analysis and simulations that having a SSM can improve reliability considerably. The intuition is that since the probability of multiple tasks overflowing the stack or heap is very low, SSM essentially provides more supplementary memory to the overflowing task than the non-SSM case and thus the probability of overflow, caused by overflowing the SSM, is reduced significantly. The exact analysis of the probability of multiple tasks overflowing their stack or heap memory is given in Section 4.1.

To implement this solution, we can leverage the technique proposed in [4] to detect stack overflow. Heap overflow can be easily detected during memory allocation. As memory protection in embedded systems is desired to be absent and any task can access any part in the memory, having a shared segment in memory incurs a small overhead. To manage the SSM, a shared memory manager is required. The primary tasks of shared memory manager are allocating or freeing memory for overflowing task and removing the holes over time. Note that the shared memory manager comes into play only when a task uses the SSM; during normal execution (i.e., when no overflow occurs) the system would operate on a memory manager provided by the OS or by the respective tasks. As the probability of multiple tasks using the shared memory concurrently is rare, the performance overhead in finding a free chunk of memory for a requesting task is always low. Note that compacting holes in SSM can be done as a low-priority task without affecting other tasks, when the system has returned into normal operation and SSM is not in use.

### 3.3 Opportunistic Micro-Rejuvenation

Opportunistic micro-rejuvenation is a fine blend of classical software rejuvenation [1] and micro-reboot [3] obviating the inappropriateness of both the schemes to the resource-constrained multi-tasking embedded systems. The key idea is that we reboot the tasks that are more likely to cause an overflow at opportune instants instead of the entire machine or independent tasks rebooting at random instants for predictive maintenance, as envisaged in [1] and [3] respectively. Since we do fine-grained rejuvenation at opportune instants, we call our technique opportunistic micro-rejuvenation. When OMR is used along with SSM, the rejuvenation is done when the SSM usage is greater than some threshold and the task with most memory usage in the SSM is rejuvenated. If the threshold is sufficiently large, say eighty percent of the size of SSM, the number of times we rejuvenate is minimal.

There are several advantages to using OMR and are outlined in Section 1. The implementation of OMR is straightforward as the rejuvenation tasks can run as a background task and check for the threshold value or memory manager can intimate the rejuvenation task when the memory manager allocates memory above threshold. We do not elaborate on the details in the interest of space.

## 4. Evaluation of SSM

In this section, first, we evaluate the SSM technique using analysis and simulation. Evaluating SSM involves comparing the three techniques to handle inaccurate estimation. First technique, referred to as BASE, is the currently used technique where supplementary memory is added individually to each task. Second technique, referred to as MTSS, is the multi-task stack sharing technique proposed by Middha et al [4], in which tasks lease stack space from other tasks when there is memory overflow. Finally the third technique is the proposed SSM technique, in which, supplementary memory of each task is combined to form a shared supplementary memory. The metric used to compare the three techniques is reliability, which is defined as the probability of no memory overflow.

Intuitively, we expect the performance of SSM to be in between BASE and MTSS because these two represent the two extremes in how supplementary memory is made available to a task upon overflow. When a task overflows its estimated memory, it has small supplementary memory to grow in the BASE case but potentially every task's free memory is available to grow in the case of MTSS. In SSM, since the probability of multiple tasks overflowing their estimate is low, there is a large supplementary memory available to grow and thus probability of overflow is reduced. By analysis and simulation, we confirm the intuition and show that the reliability is improved considerably as it is closer to the MTSS case than the BASE case.

## 4.1. Analysis

Since for the BASE and the SSM case, stack overflow and heap overflow are handled in the same manner and MTSS is proposed only for stack overflow, we discuss the analysis in the context of stack overflow. The same discussion is valid with respect to heap overflow except that MTSS is not applicable.

We make the following assumptions. The amount of memory required by any task is normally distributed with mean $\mu$ and standard deviation $\sigma$. The rationale for this assumption is that in common cases the amount of memory required is more or less the same and only in rare inputs or rare program execution paths the amount of memory required is very high or very low. We use this assumption to estimate the stack and heap usage of a task. We also assume that the amount of memory required by a task is independent from other tasks in the system.

Let there be n tasks in the system and each task be represented using an integer between 1 and n. Let $X_i$ be the Gaussian/Normal random variable with mean $\mu_i$ and standard deviation $\sigma_i$ to denote the amount of stack memory required for a task *i*. Note that $X_i$'s for different values of *i* are independent and identically distributed random variables. Let $\alpha_i$ denote the estimate of the stack memory usage by a task *i*. Let $s_i$ be the amount of supplementary stack memory allocated to a task *i*.

In the BASE case, $\alpha_i + s_i$ is the total amount of stack memory allocated to each task *i*. There is a stack overflow if any of the task's stack requirements grows larger than the amount of stack memory estimated for the task. This can be written as in Equation **(0.1)**.

$$\text{P\{Stack Overflow in BASE\}} = 1 - \prod_{i=1}^{n} \text{P}\{X_i \le \alpha_i + s_i\} \qquad \mathbf{(0.1)}$$

In the MTSS case, the tasks are allowed to share task space from memory and there is stack overflow only if the sum of all tasks stack requirement is greater than the total available stack memory for all the tasks in the system. This can be written as in Equation **(0.2)**.

$$\text{P\{Stack Overflow in MTSS\}} = \text{P}\{\sum_{i=1}^{n} X_i > \sum_{i=1}^{n} (\alpha_i + s_i)\} \qquad \mathbf{(0.2)}$$

In the SSM case, task *i* will be allocated only $\alpha_i$ units of memory and a pool of stack memory called SSM that is shared among all tasks is allocated. Note the task's own space is not shared in SSM. The size of the SSM is the sum of size of all individual supplementary memories and is denoted by *s*. The stack overflow event in this organization is not as straight forward as in the other two cases. The stack overflow happens if space cannot be allocated in task's stack or in the SSM. SSM is only used when the task's stack is exhausted. The SSM can be used by one or more tasks at the same time and hence, stack overflow can be due to a single task or any combination of *n* tasks. Taking into account all possible combinations, the stack overflow event's probability can be written as in Equation **(0.3)**.

$$\text{P\{Stack Overflow in SSM\}} = \sum_{i=1}^{n} P\{X_i > \alpha_i + s \mid X_i > \alpha_i\} * \prod_{\substack{j=1 \\ j\neq i}}^{n} P\{X_j < \alpha_j\} +$$

$$\sum_{r=2}^{n-1} P\{\sum_{k\in\{\binom{n}{r}\}} X_k > \sum_{k} \alpha_k + s \mid \bigcap_{k} X_k > \alpha_k\} * \prod_{\substack{j=1 \\ j\neq k}}^{n} P\{X_j < \alpha_j\} + \qquad \mathbf{(0.3)}$$

$$\text{P}\{\sum_{k=1}^{n} X_k > \sum_{k}^{n} \alpha_k + s \mid \bigcap_{k} X_k > \alpha_k\}$$

The index *k* in the second term of Equation **(0.3)** represents one combination and it is bound for the entire term. For example, when *r* is 2 and *n* is 3, *k* represent any one the following combination: (1, 2), (1, 3), (2, 3). The two boundary terms in Equation **(0.3)** can also be taken inside the summation and the expression can be rewritten as shown in Equation **(0.4)**

P{Stack Overflow in System SSM}=

$$\sum_{r=1}^{n} P\{\sum_{k\in\{\binom{n}{r}\}} X_k > \sum_{k} \alpha_k + s \mid \bigcap_{k} X_k > \alpha_k\} * \prod_{\substack{j=1 \\ j\neq k}}^{n} P\{X_j < \alpha_j\} \qquad \mathbf{(0.4)}$$

Since a closed form for Normal CDF does not exist, obtaining a closed form for the Equation (1.4) is not possible. Therefore, we resort to simulation. We simulated all three cases and found that results from analysis and simulation for the other two cases match.

## 4.2 Simulation and Results

The simulation is done as follows: We generate normal random variables using Box-Muller method. The value of a generated normal random variable represents the memory requirement of a task. Each experiment consists of generating memory requirement of n tasks and checking if any one of the cases would cause an overflow. In the BASE case, an overflow occurs if the any one of the tasks memory requirement is greater than the sum of task's stack memory and supplementary memory. Similarly, for MTSS case, an overflow occurs if the sum of all task's requirement is greater than the total available memory. Note that in practice not all available memory can be used by MTSS due to internal fragmentation and hence, the results obtained is for MTSS ideal case. In SSM case, an overflow occurs if all possible combination of i tasks where i varies from 1 to n task's memory requirement is greater than the sum of tasks own stack memory plus the SSM. Each simulation run repeats the experiments 100,000 times with different random seeds. The probability of overflow is calculated as number of overflows divided by the total number of experiments.

**Table 1: Parameters used in SSM simulations Note memory units are in 1k for stack and 50k for heap.**

| **Parameter Name** | **Expr. #1** | **Expr. #2** |
|---|---|---|
| Number of tasks in the system (n) | 8 | 8 |
| Estimate of a task i ( $\alpha_i$ ) | 8.5 | 1.7, 8.5, 42.5 |
| Supplementary Memory of task i ( $s_i$ ) | 1.5 | 0.75, 1.5, 3 |
| Mean of the task's instantaneous memory requirement ( $\mu_i$) | 7 | 1.4, 7, 35 |
| Standard Deviation of task's instantaneous memory requirement ($\sigma_i$) | 1.5 | 0.75, 1.5, 3 |
| Shared Supplementary Memory ( s) | 12 | 15.75 |

The parameters used for the simulation are given in the Table 1. In experiment 1, we assume for the sake of simplicity, that all tasks have equal memory requirement and thus estimated memory and supplementary memory are also equal for all tasks. The unit for memory can be varied for the case of stack and heap as stack segment is usually much smaller than the heap segment. Note that in line with our assumptions that we have made the estimate to be one standard deviation away from the mean. This means that the probability of using supplementary memory is reduced. In experiment 2, we choose one-third of the task to have low memory requirement, another one-third to have medium memory requirements and the one-third to have high memory requirement.

We have plotted the total memory available in the system against the reliability and are shown in Figure 2. Since the estimated memory doesn't vary in all the three cases, we vary the supplementary memory. Therefore, the total memory in the system varies from sum of all estimated memory, that is, no supplementary memory to sum of all estimated memory plus 400% supplementary memory.

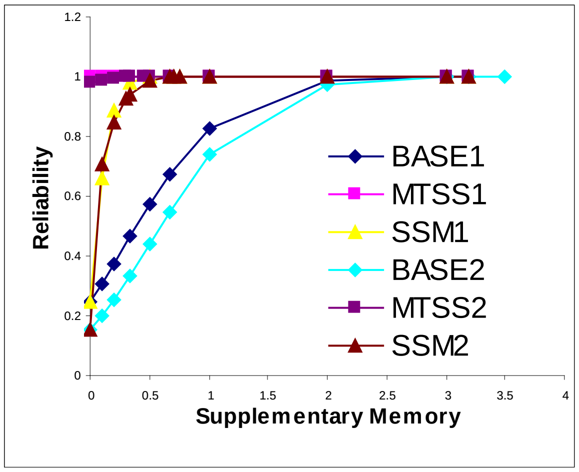

**Figure 2: Variation of reliability in three methods when the supplementary memory is increased from 0% to 400%.**

When there are no supplementary memories, SSM reduces to the BASE case. As the available supplementary memory increases, the reliability increases exponentially in the case of SSM. This is because not all tasks use the SSM concurrently and hence, few tasks have more supplementary memory. However, in the BASE case, the supplementary memory is not combined and task that uses supplementary memory has only its own supplementary memory and it cannot borrow supplementary memory from other tasks.

Note that for the same reliability, SSM requires much less supplementary memory than BASE and MTSS requires the least. In the case of MTSS, the parameters chosen for the simulation provide enough available memory in other tasks' estimated space and the supplementary memory was not needed. From the Figure 2, we see that the results for experiment 2, in which total memory is divided unequally among the tasks, follow the same trend as experiment 1, in which total memory is equally divided among the tasks and thus generalize the advantages explained above.

## 5. Evaluation of OMR

Next, we use stochastic models to evaluate the OMR technique. Since overflow failures and hardware exceptions are rare, a real implementation-based evaluation is not feasible. Hence, we model our system using Stochastic Activity Networks (SAN) [13, 14],which has been successfully used to model non-functional aspects such as reliability and availability of many complex systems [14]. Following the traditional model used for software rejuvenation [1], we model the system using three states, namely, robust, vulnerable, and failed.

The metric that we have used is availability of the system and is defined as the average availability of all the tasks in the system. Let $A_i$ be the availability of the task i. The availability of the system of n tasks, A, is defined as the average of n tasks availabilities. That is, $A = \sum A_i / n$.

We compare OMR to the base case which is the existing system that reboots on a crash. Here we assume that the estimation of memory is accurate and memory overflow is only due to Mandelbugs. We show that since MandelBugs are rare and the recovery on a failure is usually very quick, there is only slight increase in availability. As expected, mean time to crash is very high in OMR as opposed to the base case.

### 5.1 SAN Model

Following the traditional model used for software rejuvenation [1], we model the system using three states, namely, robust, vulnerable, and failed and the transition times between these states are exponentially distributed. The Stochastic Activity Network (SAN) model for the base case and the OMR case are shown in Figure 3 and Figure 4 respectively.

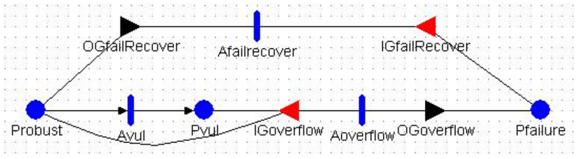

Figure 3: SAN Model used for the base case where there is no rejuvenation.

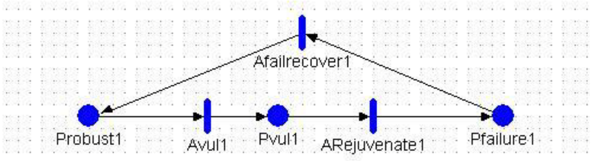

Figure 4: SAN Model used for the OMR case

As the places and transitions are self-explanatory, we mention only the important aspects of the model. All the transitions follow the exponential distribution with rate shown as in Table 2. In Figure 3, when the Aoverflow activity fires, the marks in both places Probust and Pvul are removed and all marks are placed in Pfailure to emulate failure of the system. Similarly when the Afailrecover activity fires, the place Probust is initialized to total marks and marks from failure is removed to emulate a system reboot.

### 5.2 Simulation and Results

We solved the above described SAN models on Mobius tool[13] using simulation. The parameters used in the experiments are shown in the Table 2 and are chosen so that they are close to real-world embedded systems.

Table 2: Parameters used in the SAN simulation.

| **Parameter** | **Base Case** | **OMR** | **OMR_SLOW** |
|---|---|---|---|
| Number of tasks/task pools | 8 | 8 | 8 |
| Vulnerability Rate ( in /hour ) | 0.01 | 0.01 | 0.01 |
| Overflow after vulnerability Rate ( in /hour ) | 0.1 to 1 | - | - |
| Fail Recover Rate ( in /hour ) | 60 | 240 | 60 |
| Rejuvenation Rate ( in /hour ) | - | 0.1 to 1 | 0.1 to 1 |

In an embedded system, we assume that the ratio of restarting a system to rebooting a task to be 6. The vulnerability rate is kept constant at once every 100 hours and the overflow rate is varied from once every 10 hours to once every 1 hour. One order of magnitude difference between vulnerability rate and overflow rate is because in embedded system once vulnerability happens, overflow is imminent due to the fact that amount of memory available in embedded systems is less. Note that an overflow happens only after the system becomes vulnerable and therefore, the average number of hours it takes to cause an overflow since the start of the system is 101 to 110 hours.

Figure 5 shows the steady state system availability as the average time to overflow after the vulnerability is varied. The BASE, in Figure 5, represents the existing system that reboots on a system crash due to memory overflow. The simulation result shows that micro-rejuvenation or task level rejuvenation increases availability slightly compared to the base case. The reason we see only slight increase in availability (0.012%) is because we look exclusively at rarely occurring bugs. In practice, however, more memory overflows occur due to inaccurate estimation. Since we couldn’t include the memory overflows caused by inaccurate estimation into our stochastic model, we are not able to evaluate the combination of OMR and SSM. When OMR combines with SSM, we can show a significant improvement in availability. The reason is that the probability of an overflow is very small (less than 1%) when the threshold is 80% of SSM when

OMR and SSM are used together. However, in the base case the probability of overflow with the same threshold is high (around 30%). This is also the reason why we call these two techniques complementary. Another reason for only slight improvement is that we assume that every task ages which is not the case and thus it is a worst case improvement for OMR.

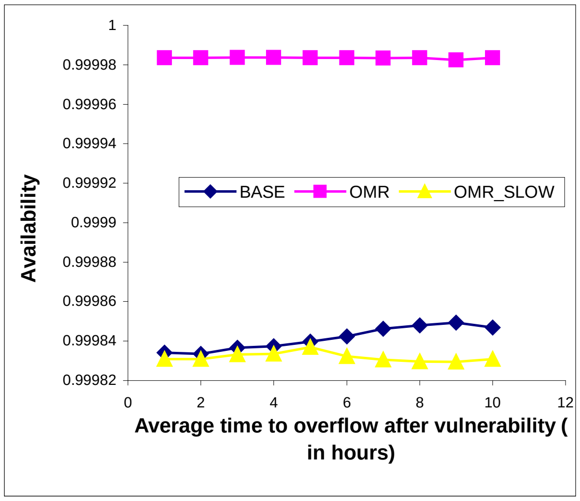

Figure 5: Effect of availability when average time to overflow after vulnerability varies.

The OMR_SLOW, in Figure 5 represents the case when the recovery rate of OMR is reduced to be same as the base case and hence, the tasks recover slower in this case. The OMR_SLOW shows that micro-rejuvenation is not always better than restart because in the OMR case tasks age even when other tasks rejuvenate as opposed to reboot in which all tasks get rejuvenated when rebooting.

## 6. Conclusion and Future Work

In this paper, we identified two main causes of memory overflow, namely, inaccurate estimation and presence of MandelBugs.. We proposed two complementary techniques namely Shared Supplementary Memory (SSM) and Opportunistic Micro Rejuvenation (OMR) to handle inaccurate estimation and Mandelbugs respectively. We showed that considerable increase in reliability can be obtained using SSM and for the same reliability, the total memory requirement can be reduced considerably. Synergistic combination of OMR with SSM can yield significant improvement in reliability and availability.

In future work, we are currently implementing our techniques in ucLinux, an open source embedded operating system. We are also looking into ways of developing stochastic petrinet based models that can handle both identified causes of memory overflow simultaneously.

## 7. References

[1] Yennun Huang, Chandra Kintala, Nick Kolettis and N. Dudley Fulton; Software Rejuvenation: Analysis, Module and Applications. In Proc of the 25th Intl. Symposium on Fault-Tolerance Computing (FCTS-25), Pasadena, CA, June 1995
[2] Dr. Doug Locke; Real-Time & Embedded Systems: Past, Present and Future. In Keynote Talk at 10th IEEE RTAS Conference, Toronto, 2004
[3] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman and Armando Fox; Micro-reboot: A technique for cheap recovery. In Proc of the 6th conference on Symposium on Operating Systems Design & Implementation – Volume 6, CA, 2004
[4] Bhuvan Middha, Matthew Simpson and Rajeev Barua; MTSS: Multi Task Stack Sharing for Embedded Systems. In Proc of the ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), San Francisco, CA, September 25-27, 2005
[5] Kihwal Lee and Lui Sha; Process Resurrection: A Fast Recovery Mechanism for Real-Time Embedded Systems. In Proc of 2005 11th IEEE Internation Conference on Embedded and Real-Time Computing Systems and Applications
[6] John Regehr, Alastair Reid and Kirk Webb: Eliminating stack overflow by abstract interpretation. In Proc of the 3rd International Conf on Embedded Software, Philadelphia, PA, 2003
[7] Michael Grottke and Kishor S. Trivedi. Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate. In IEEE Computer February 2007.
[8] G. V. Neville-Neil. Programming without a net. ACM Queue: Tomorrow's Computing Today, 1(2):16–23,April 2003.
[9] N.D.D. Brylow and J. Palsberg. Stack-size estimation for interrupt-driven microcontrollers. Technical report, Purdue University, June 2000.
[10] P.R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg. Data and memory optimization techniques for embedded systems. ACM Transactions on Design Automation Electronic Systems, 6(2):149–206, 2001.
[11] Surupa Biswas, Thomas Carley, Matthew Simpson, Bhuvan Middha and Rajeev Barua. Memory Overflow Protection for Embedded Systems using Run-time Checks, Reuse and Compression. In *ACM Transactions on Embedded Computing Systems (TECS)*, 5(4), pp 719 - 752, Nov 2006
[12] Naganuma Masayuki, Eto Takeshi. Method and apparatus for controlling stack area in memory space. Published patent application US2003177328
[13] G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. Webster. The Möbius Modeling Tool. Proceedings of the 9th International Workshop on Petri Nets and Performance Models, Aachen, Germany, September 11-14, 2001, pp. 241-250.
[14] W. H. Sanders and J. F. Meyer. Stochastic Activity Networks: Formal Definitions and Concepts. in E. Brinksma, H. Hermanns, and J. P. Katoen (Eds.), Lectures on Formal Methods and Performance Analysis, First EEF/Euro Summer School on Trends in Computer Science, Berg en Dal, The Netherlands, July 3-7, 2000, Revised Lectures, Lecture Notes in Computer Science no. 2090, pp. 315-343. Berlin: Springer, 2001