

Fault-Tolerant Techniques for Ambient Intelligent Distributed Systems*

Diana Marculescu, Nicholas H. Zamora, Phillip Stanley-Marbell and Radu Marculescu

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213-3890

{dianam, nhz, pstanley, radum}@ece.cmu.edu

ABSTRACT

Ambient Intelligent Systems provide an unexplored hardware platform for executing distributed applications under strict energy constraints. These systems must respond quickly to changes in user behavior or environmental conditions and must provide high availability and fault-tolerance under given quality constraints. These systems will necessitate fault-tolerance to be built into applications. One way to provide such fault-tolerance is to employ the use of redundancy. Hundreds of computational devices will be available in deeply networked ambient intelligent systems, providing opportunities to exploit node redundancy to increase application lifetime or improve quality of results if it drops below a threshold. *Pre-copying with remote execution* is proposed as a novel, alternative technique of code migration to enhance system lifetime for ambient intelligent systems. Self-management of the system is considered in two different scenarios: applications that tolerate *graceful quality degradation* and applications with *single-point failures*. The proposed technique can be part of a design methodology for prolonging the lifetime of a wide range of applications under various types of faults, despite scarce energy resources.

1. INTRODUCTION

Ambient intelligent systems are future electronic systems that will bring truly ubiquitous smart environments into daily life by adapting and responding to human actions and environmental conditions.

Fields of research such as networks of sensors [4, 5] and wearable computing [15] have focused on the design of hardware and software systems with the primary intent of providing accessibility, security, communication and intelligence while remaining mostly invisible to the users. As technology advances, devices become smaller and cheaper, making it possible to envision and build highly distributed and fault-tolerant ambient intelligent systems. Many of these ambient intelligent systems will be energy constrained and will make use of batteries to supply the required energy. These platforms will employ large numbers of physically minute devices and permit their true embedding into everyday environments.

Instead of today's rigid programming paradigm – *design, build, compile, and run* – intelligent ambients [2, 8] offer a defect-tolerant programming paradigm – *design, build, compile, run, and monitor*. In such smart spaces, computational elements are inconspicuous in their environments and offer a wide area network for increased defect tolerant computational power. As specific examples, audio-visual remote controls can be incorporated into cushions. Interior environmental conditions can be changed by touching curtains or wall coverings, while sensors could be used for adaptive thermal management of rooms.

*This research was supported in part by Defense Advanced Research Projects Agency under Contract No. F33615-02-1-4004, by Semiconductor Research Corporation under Grant No. 2002-RJ-1052G, and by a Semiconductor Research Corporation Ph.D. Fellowship for Nicholas H. Zamora.

Given an environment consisting of large numbers of failure-prone devices, many challenges exist in designing reliable systems, including programming large networks of energy constrained, error-prone devices, re-programming devices after deployment and exploiting redundancy to improve system lifetime and fault-tolerance. Some characteristics of these and other emerging platforms include:

- *Large Numbers of Devices per Application*: Applications can utilize hundreds of computational nodes, sensors, actuators and communication links. Numerous nodes imply that most applications will utilize multiple processing units, and these applications must be partitioned to make effective use of available hardware resources. Although applications can run on a single node, nodes may still be coordinated in a larger scope [4]. This contrasts sharply with wearable computing where there is typically a single central computing device [15].
- *Energy Constraints and Self-Management*: Given limited energy resources, it will be imperative to design applications in such a way as to allow their continued functionality despite scarce energy resources. This relies on achieving self-management via code migration and on-the-fly handling of error conditions. The system should be designed in such a way to allow continued functionality despite depleting energy resources and failing computational nodes and communication links.
- *Synchronized and Coordinated Behavior*: Each node will need to participate in a coordinated effort to successfully run the distributed application. This requires carefully designed control logic to keep this coordination deterministic and effective.

In summary, mobile ambient intelligent embedded systems will be energy constrained and will require fault-tolerance which can be achieved through self management and code mobility.

1.1 Related Work

The use of *code migration* has been successfully applied in the field of mobile agents [6, 10, 16]. Mobile agents are an evolution of the general ideas of process migration [9]. They can be thought of as autonomous entities that determine their traversal through the network, moving their code as well as state as they traverse. Process migration has traditionally been employed in distributed systems of servers and workstations, primarily for load distribution and fault-tolerance [9]. Unlike the traditional implementations of process migration, the soon to be introduced technique employed in this work is of *significantly lighter weight*, taking into consideration the special properties of the target application class.

The use of *remote execution* to reduce the power consumption in mobile systems has previously been investigated in [7, 13, 12]. The goal in these efforts was to reduce power consumption by offloading

tasks from an energy constrained system to a server without constraints in energy consumption. The tradeoff involved determining when it was worthwhile to transfer data to the remote server. All approaches involve the use of remote execution, but not code migration. The systems investigated were operating within a fast, reliable wireless network. Such an environment is in strict contrast to the *ultra low power, failure-prone* wired-network sensors with low-end computing capabilities and possibly *faulty communication and computation* which are of interest in this paper.

1.2 Contributions of the Paper

This paper introduces techniques for the *robust* execution of applications in distributed, *embedded failure prone* environments such as *ambient intelligent systems*. The proposed techniques, based on the idea of code migration, enable the remapping of failing applications in environments with redundantly deployed hardware.

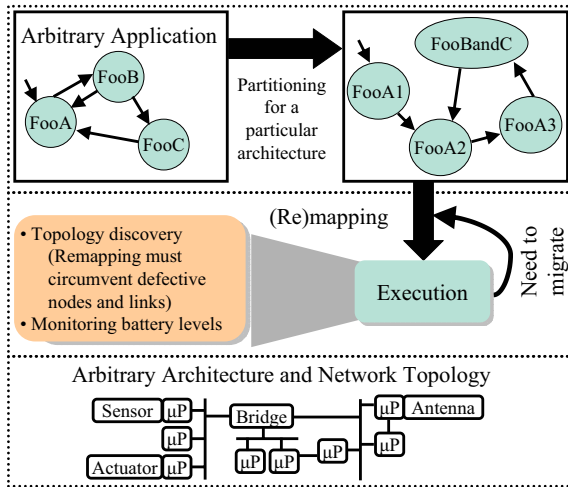


Figure 1: Mapping and remapping applications under energy, performance and fault-tolerance constraints.

With large numbers of devices per application, it will be possible to take advantage of the multiplicity of identical elements to adapt to the constraints imposed by *high failure rates* and *limited energy resources*. It thus becomes necessary to efficiently *map* applications onto hardware by partitioning them to match the underlying architecture. This mapping might further be iterated over the lifetime of a system to match changing system constraints.

The proposed paradigm to be used during the lifetime of an ambient intelligent systems is shown in Figure 1. As shown, the application (the top layer in Figure 1) is partitioned so as to match closely the available underlying computing substrate (the bottom layer). The computing substrate can be a set of deeply networked embedded systems, which are usually characterized by drastic energy constraints and thus may affect the overall quality of the system by their finite battery lifetime. The application is initially mapped onto the computing substrate, or remapped depending on the application needs or changes in the operating conditions (e.g., local energy sources running low, too many failures, etc.). Such services are provided by the middle layer that, in addition to mapping and remapping the application, provides support for battery level monitoring, routing around faulty interconnect or on-the-fly re-partitioning or remapping the application under a dynamically changing topology.

This paper opens a new venue of research in self-management support for ambient intelligent systems. To our knowledge, this work is the first attempt at using an application remapping process to increase availability and fault-tolerance of ambient intelligent systems.

The applications employed in evaluating the proposed techniques are *beamforming* and *software radio*. These applications were partitioned for use on distributed ambient intelligent systems and their executions simulated using a cycle-accurate simulator. Beamforming is a highly symmetric application, simple to partition for execution on multiple processing units. Software radio, on the other hand, is highly asymmetric and slightly more challenging to partition in the way needed to match a given architecture.

The remainder of this paper is organized as follows. A new technique for performing code pre-copying as well as a brief overview of the known techniques for performing code remapping is given in Section 2. Section 3 describes a theoretical framework for pre-copying scheduling for a given set of nodes connected through a shared communication bus. A description of the beamforming and software radio driver applications is given in Section 4. Section 5 describes the simulation infrastructure and details the setup employed in the investigations. Section 6 presents the experimental results and analysis. The paper is concluded with a summary of the contributions of this work and some directions for the future.

2. APPLICATION REMAPPING

In the presence of exceptional conditions, such as critically low levels of energy resources or increasingly rampant intermittent failures, it is desirable to re-map application execution from one device (s) to another. Deciding *if* and *when* to perform remapping involves tradeoffs. For example, with low energy resources, remapping should occur early enough so as to have sufficient energy to complete. Acting too early can lead to unused energy resources going to waste, while acting too late may result in permanent failure.

One possibility for remapping is the use of *baseline code migration*, which remaps executing code to available nodes when the battery depletes past a predetermined threshold. This energy resource threshold must be set conservatively so as to ensure migration even in the presence of intermittent link failures. To provide more flexibility over this baseline code migration scheme, this paper proposes a new mechanism for performing migration which permits the staging of the migration process so that effectively lower thresholds may be used that still guarantee successful migration.

2.1 Code Migration

The code migration scheme employed in this work is a compromise between implementation complexity and flexibility. In the ideal case, migrating an executing application will mean the application itself can be unaware of this move, and can continue execution on the destination host without any change in behavior. This transparent migration requires that *the entire* state of the executing application (code, data, machine state and state pertinent to any underlying system software) must be migrated faithfully. Such migration is however often too costly in terms of data that must be transmitted to the target host and in terms of implementation complexity.

A desirable compromise is to implement applications in a manner in which they can be *asynchronously restarted* while maintaining state *persistence*. Figure 2 illustrates such a solution. The sensor node consists of a processor and memory. The memory space is partitioned between the application and the device's firmware. The memory region in which the application runs is occupied by the different parts of the running application: program code (`text`), initialized data (`data`), uninitialized data (`bss`), stack (grows downwards from the top of the memory region) and heap (grows upwards from `bss`), as illustrated by the blow-up in Figure 2. By placing information that must be persistent across restarts in the `data` and `bss` segments of the application, it is possible to maintain state

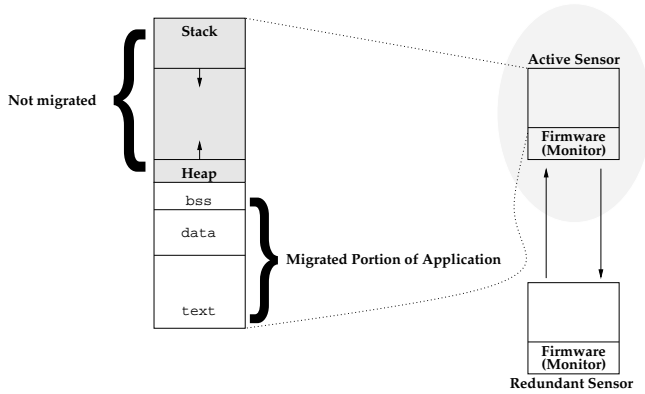


Figure 2: Lightweight migration of application code and state. Only program code, data and uninitialized data are transferred during migration.

across migration while only transferring the text, data and bss segments.

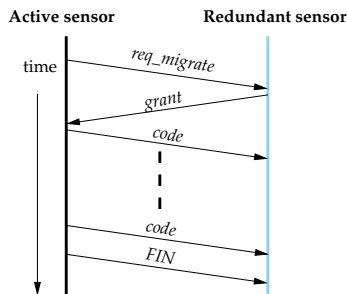


Figure 3: Message exchange during migration of application between a dying node and a redundantly deployed one.

Each node has pre-loaded into it firmware (also called *monitor*) which is responsible for receiving applications and loading them into memory. Each application can transmit its text, data and bss segments, and when these are loaded by a remote monitor, the migration is complete. The sequence of messages that are exchanged between a node attempting to migrate and a redundant one that receives it is illustrated in Figure 3.

Each node attempts to migrate its application code when the available energy left falls below a certain threshold B_{low} . This threshold must be chosen conservatively so as to ensure that each node can successfully migrate in the presence of bus collisions and link errors *before* they completely deplete their energy resource and die.

2.2 Remote Execution

Remote execution is an alternative to code migration. In a system employing remote execution, each application that will be run on any node in the system must be duplicated on *every* node in the system. Using remote execution, applications simply need to transfer their application state and subsequently transfer control of execution to the destination node at a desired point in time to complete migration. Transfer of state is inexpensive compared to full application copying, as the text segment need *not* be transmitted. This is particularly useful when communication is prohibitively expensive, as is the case in wireless networks [12]. The disadvantage is the requirement of ample memory on each device to store every application.

2.3 Pre-Copying with Remote Execution (PCRE)

Pre-Copying with Remote Execution (PCRE) is proposed in this work as a means of improving code migration and remote execution by borrowing ideas from both. The primary goal is to distribute the

time at which migration of application code occurs without actually transferring execution of applications until energy resources fall to a critical threshold. At this point, a final message is sent to the destination where the application code was previously pre-copied, and execution is transferred. This enables the threshold for migration to be set much lower, resulting in more complete usage of the energy resource. It also has a beneficial impact on the case of catastrophic failures, in which case the spare node could restart execution if the hand-off signal never arrives.

The *lifetime of a system* is defined as the amount of time that the system remains functioning. Assume that N is the minimum number of nodes required to keep the system alive and a total of P processing nodes are available. If each node in the system contains an equal amount of limited battery energy, the system lifetime can be increased by, at most, P/N times if the $P-N$ nodes are used as spares for migrating code. When the number of all available processing nodes P is large, this potential increase in system lifetime can be considerable and is worthwhile for investigation. While a similar impact would be achieved by just increasing the battery capacity by P/N times, we note that the size of such a local power source may be prohibitive for the type of applications under consideration.

Consider a processing node with an average time to live T_{SL} and migration time T_M assuming no communication link collisions or errors. Each of the N active nodes is identified by a unique number, i , ranging between 0 and $N-1$. A *migration slot* beginning at time T_{P_i} is assigned to each processing node:

$$T_{P_i} = i \cdot W \cdot \left(\frac{T_{SL}}{N} \right)$$

W is a factor ranging from 0 to 1 that is proportional to the time needed for pre-copying T_M . An example with $N = 14$ is shown in Figure 4. In this case, for $W = 0.7$, the 14 slots are equally divided between $t = 0$ and $W \cdot T_{SL}$.

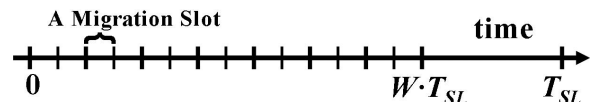


Figure 4: Determining the migration slots for $N = 14$.

After pre-copying, the node resumes executing the application until its battery falls below a low threshold, B_{vlow} . At this time, the final step to complete migration is to have that active node send a final network packet to the redundant node, indicating to it that it must resume execution of the re-mapped application.

When migration occurs over a shared bus, the condition:

$$T_M \leq W \cdot \left(\frac{T_{SL}}{N} \right) \quad (1)$$

must hold for bus collisions to be avoided. Of course, when the communication graph is complete (i.e., there is a communication link between every pair of nodes), then this condition is not necessary. However, assuming the time to pre-copy the application is negligible compared to the node lifetime or, more precisely, if:

$$N \cdot T_M \ll T_{SL} \quad (2)$$

then using this pre-copying scheme for migration is effective regardless of the network topology of the distributed system.

As an example, if $T_{SL} = 5$ hours, $T_M = 20$ seconds, and $N = 30$ then it is safe to make W as small as $\frac{1}{30}$ and it would still be possible to use PCRE over a single, shared migration bus.

3. OPTIMAL REMAPPING SCHEDULING

While PCRE as described so far offers a feasible alternative to baseline code migration, it assumes migrated code size is identical

and thus, it does not guarantee optimality in terms of overall system lifetime in the most general case. To address this issue, this section considers the general case of pre-copying for applications with variable workload characteristics. The goal is to find qualitative guidelines for scheduling the pre-copying process depending on the code size or the average computational load for each migrated application.

To model the battery characteristics for a variable workload characterized by the *compute-migrate-compute-hand-off* paradigm, the generalized battery model with nonlinear effects found in [11] is used. Consider the following notation, consistent with the one in Section 2:

T_{SL} = time to live

T_M = time for pre-copying the code

T_P = time to begin pre-copying the code

S = battery capacity

β = nature of battery nonlinearity = $\frac{\pi\sqrt{D}}{\omega}$ where

D = diffusion coefficient,

ω = length of diffusion region,

($\beta = 0.637$ for a rechargeable 2.2 Watt-hour Li-ion battery)

I_C = current drawn during computation

I_M = current drawn during migration

The battery capacity required to achieve a particular lifetime for a *compute-migrate-compute-hand-off* cycle can be predicted by the equation [11]:

$$S(T_{SL}, T_M, T_P, I_C, I_M) = I_C \cdot \left(T_P + 2 \cdot \sum_{m=1}^{\infty} \frac{e^{-\beta^2 m^2 (T_{SL} - T_P)} - e^{-\beta^2 m^2 T_{SL}}}{\beta^2 m^2} \right) + I_M \cdot \left(T_M + 2 \cdot \sum_{m=1}^{\infty} \frac{e^{-\beta^2 m^2 (T_{SL} - T_P - T_M)} - e^{-\beta^2 m^2 (T_{SL} - T_P)}}{\beta^2 m^2} \right) + I_C \cdot \left(T_{SL} - T_P - T_M + 2 \cdot \sum_{m=1}^{\infty} \frac{1 - e^{-\beta^2 m^2 (T_{SL} - T_P - T_M)}}{\beta^2 m^2} \right)$$

In the following, theoretical results that support the slot scheduling process are presented. As it will be shown, tasks which draw more current during computation, or involve larger costs during the migration process, should be scheduled earlier for pre-copying to maximize overall system lifetime.

Lemma 1: If $I_M \geq I_C$, S is monotonically increasing with T_P , i.e., $S(T_{SL}, T_M, T'_P, I_C, I_M) \geq S(T_{SL}, T_M, T_P, I_C, I_M)$ iff $T'_P \geq T_P$.

Proof: According to the equation for battery capacity S :

$$S(T_{SL}, T_M, T'_P, I_C, I_M) - S(T_{SL}, T_M, T_P, I_C, I_M) = 2(I_M - I_C)$$

$$\cdot \sum_{m=1}^{\infty} \frac{e^{-\beta^2 m^2 (T_{SL} - T_P)} \cdot (e^{\beta^2 m^2 T_M} - 1) \cdot (e^{\beta^2 m^2 (T'_P - T_P)} - 1)}{\beta^2 m^2}$$

Since $I_M \geq I_C$ and the terms of the sum are positive if $T'_P \geq T_P$, then $S(T_{SL}, T_M, T'_P, I_C, I_M) \geq S(T_{SL}, T_M, T_P, I_C, I_M)$.

Lemma 2: If $I_M \geq I_C$ and I_M is sufficiently small, S is monotonically increasing with T_{SL} , i.e., $S(T'_{SL}, T_M, T_P, I_C, I_M) \geq S(T_{SL}, T_M, T_P, I_C, I_M)$ iff $T'_{SL} \geq T_{SL}$.

Proof: From the equation for battery capacity S :

$$S(T'_{SL}, T_M, T_P, I_C, I_M) - S(T_{SL}, T_M, T_P, I_C, I_M)$$

$$= I_C (T'_{SL} - T_{SL}) + 2 \cdot \sum_{m=1}^{\infty} \frac{(e^{-\beta^2 m^2 T_{SL}} - e^{-\beta^2 m^2 T'_{SL}})}{\beta^2 m^2} \cdot (I_C - (I_M - I_C) e^{\beta^2 m^2 T_P} \cdot (e^{\beta^2 m^2 T_M} - 1))$$

If I_M is sufficiently small, $\frac{I_C}{I_M - I_C} \geq e^{\beta^2 m^2 T_P} \cdot (e^{\beta^2 m^2 T_M} - 1)$ for all values of m relevant for computing S ($m \leq 10$ for all practical purposes [11]).

Thus, $T'_{SL} \geq T_{SL}$ and hence $S(T'_{SL}, T_M, T_P, I_C, I_M) \geq S(T_{SL}, T_M, T_P, I_C, I_M)$.

Theorem 1: In single node migration, migrating earlier (with smaller T_P) provides better overall battery lifetime T_{SL} .

Proof: Consider the diagram in Figure 5. If we put $S'' = S$, then according to Lemma 1 and Lemma 2, a later migration $T'_P \geq T_P$ will trigger a shorter overall lifetime $T'_{SL} \leq T_{SL}$.

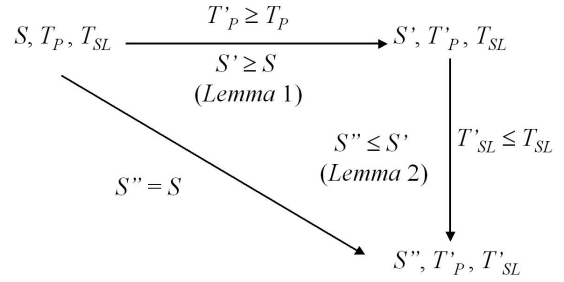


Figure 5: The impact of later precopying on time-to-live T_{SL} for the same battery capacity.

Theorem 1 suggests that instead of employing code migration, which does both code copying and hand-off towards the end of the application lifetime, it is always better to schedule pre-copying as early as possible and hand-off the execution to the redundant node whenever a critical battery level is reached. However, when multiple nodes share the same communication bus, having each node pre-copy its code in the beginning will create collisions and prolong the migration process, thus negating any benefit from early scheduling of pre-copying. Centralized arbitration for relieving congestion would be too expensive to use for the type of applications considered, and thus, it thus makes sense to consider the problem of *scheduling* the pre-copying process. The goal is to maximize the overall lifetime, while allowing each application to pre-copy its code to a spare node in non-overlapping intervals. The following result gives a qualitative set of guidelines for choosing candidate applications for earlier versus later migration on a shared communication bus.

Theorem 2: For a given battery capacity and communication cost I_M , if $I_M \geq I_C$ earlier scheduling of pre-copying for applications with larger I_C (i.e., higher computation workload) generates longer overall system lifetime. Similarly, earlier scheduling of pre-copying for applications with larger T_M (i.e., higher communication overhead during migration) results in longer overall system lifetime.

Proof: S is monotonically increasing with I_C , that is:

$$S(T_{SL}, T_M, T_P, I'_C, I_M) - S(T_{SL}, T_M, T_P, I_C, I_M) = (I'_C - I_C) \cdot \left(2 \cdot \sum_{m=1}^{\infty} \frac{e^{-\beta^2 m^2 (T_{SL} - T_P)} - e^{-\beta^2 m^2 T_{SL}}}{\beta^2 m^2} + (T_{SL} - T_M) + 2 \cdot \sum_{m=1}^{\infty} \frac{1 - e^{-\beta^2 m^2 (T_{SL} - T_P - T_M)}}{\beta^2 m^2} \right) \geq 0$$

if $I'_C \geq I_C$.

Thus, using a similar reasoning as in Theorem 1, one can conclude that to achieve the same lifetime, it is necessary that $T'_P \leq T_P$. Thus, applications with larger I_C should be scheduled for pre-copying earlier. Similarly, S is monotonically increasing with larger T_M :

$$\begin{aligned} & S(T_{SL}, T'_M, T_P, I_C, I_M) - S(T_{SL}, T_M, T_P, I_C, I_M) \\ &= (I_M - I_C) \cdot ((T'_M - T_M) \\ &+ 2 \cdot \sum_{m=1}^{\infty} \frac{e^{-\beta^2 m^2 (T_{SL} - T_P)} \cdot (e^{\beta^2 m^2 T'_M} - e^{\beta^2 m^2 T_M})}{\beta^2 m^2}) \geq 0 \end{aligned}$$

if $I_M \geq I_C$ (which is usually the case) and $T'_M \geq T_M$. Thus, larger time needed for migration implies larger S or shorter lifetime T_{SL} (according to Lemma 2) and thus, for a given S , scheduling pre-copying earlier for applications with larger code size should help minimize the decrease in overall system lifetime.

In most cases, applications can be pre-characterized in terms of their workload requirements (I_C) as well as the time needed for migration (T_M). Specifically, T_M is proportional to the code size involved in the migration process as it is a function of the number of frames sent to the spare node. Given these guidelines, one can come up with a schedule for the pre-copying process of different applications and can assign slots in a similar manner to what has been discussed in Section 2.3. Since migration time T_M is usually relatively small compared to the overall application lifetime, we expect that priority in assigning earlier slots should be given to applications exhibiting higher computational workloads (I_C) and breaking the ties based on migration costs (T_M).

4. DRIVER APPLICATIONS

This work employs *beamforming* and *software radio* as its driver applications. Beamforming has the properties of degrading gracefully with failing nodes as well as being trivially partitioned for execution over a collection of devices. Software radio, on the other hand, is dependent on each active node during execution and is also less trivial to partition.

While code migration or PCRE are used primarily to increase application longevity, at the same time, they should not offset their benefits by reducing the quality of application at hand. In the following, we describe two types of applications and an example of each: applications that tolerate graceful degradation in quality, and application with single-point failures for which additional care must be taken.

4.1 Application with graceful degradation: Beamforming

In this case, applications running on a computing substrate of networked embedded systems may tolerate various nodes in the network being down during the migration process. One such example is the beamforming application. A beamformer consists of an array of sensors working in conjunction with a centralized node¹ with the objective of estimating propagating wave signals (either electromagnetic or auditory) in the presence of noise and interference.

The beamforming implementation considered in this work consists of repetitive rounds in which all computational nodes participate in accomplishing the overall goal of signal recovery. At the beginning of a round, the *master* node sends a broadcast message to all *slave* nodes instructing them to begin sampling. Next, each slave

¹These processing elements are henceforth referred to as “slave nodes” and the “master node,” respectively.

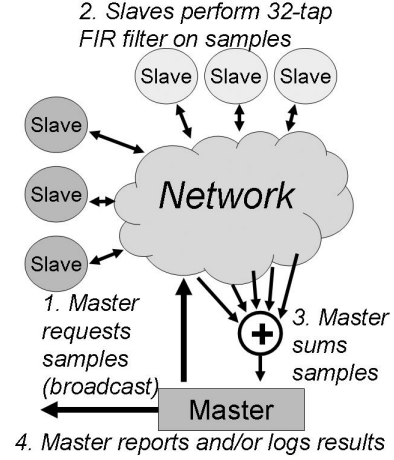


Figure 6: Organization of beamforming application. Application is partitioned across master and slave nodes. The arrows in the Figure indicate communication.

obtains a sample from its attached sensor and performs a K -tap FIR filter operation on this sample. The slave node then waits for its pre-determined time-slot, based on its identification number, to send the result to the master node for analysis. During this analysis step, the master node combines the individual samples to obtain a composite reading for that sampling period. Finally, at the completion of a round, the master node waits for the beginning of the next round to send the next sample request broadcast and continue operation. This work focuses on the slave nodes, the master node, and the network linking together the individual slaves to the master node as shown in Figure 6.

4.2 Application with single-point failures: Software Radio

In this case, applications may be fully connected and thus, a single-point failure (such as the case of a battery running out or a node in the process of migrating) anywhere in the network has catastrophic effects on the overall quality of the system. Such an example is the software radio application which is a common DSP algorithm that takes, as input, a modulated signal real-time streaming signal and outputs the baseband signal. The application can be divided into five main steps as shown in Figure 7. The first step is the acquisition of the modulated signal, perhaps with the use of an antenna. The second step is the low pass filtering of the modulated signal to prevent aliasing due to demodulation. Next is the actual demodulation of the signal, transforming the signal from the carrier frequency down to the baseband frequency. Following demodulation is equalization. The final step is the sink, which simply is the collector of the resulting samples.

The software radio implementation in this work maps each of these five steps onto a single computational node in the distributed architecture, except for the equalization step, which is further partitioned eight ways and mapped onto eight processing nodes as shown in Figure 7. In Figures 6 – 7, redundant nodes are the ones which the active nodes will use to perform code remapping.

5. EXPERIMENTAL SETUP

5.1 Simulation Infrastructure

The *simulator* used in this study is built around a publicly available energy estimating architectural simulator for the Hitachi SuperH embedded processor architecture [14]. It models a network

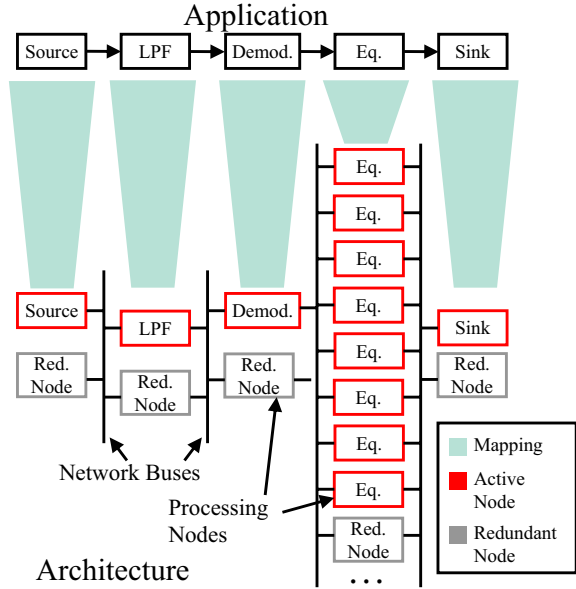


Figure 7: Flow diagram and mapping implementation of software radio application. Application is partitioned into five main stages, with the fourth stage further partitioned into eight stages (LPF = Low-pass filter, Demod. = Demodulation, Eq. = Equalization).

of embedded processors, each consisting of a Hitachi SH3 micro-controller, memory and communication interface. The simulation environment permits the instantiation of a large number of nodes as well as interconnection links (it has been used to simulate networks on the order of 100s of nodes).

The modeled processor in each system may be configured to run at different voltages and, hence, different operating frequencies. Each simulated processor can be configured with multiple network interfaces. Each of these network interfaces may be attached to an instantiated communication link. The communication links can be configured for data transmission at different bit rates and link frame sizes. Both the processing nodes and the interconnection links can be configured for *independent* or *correlated runtime failures* with different failure probabilities.

The simulation of both the processor elements and the interconnection network is cycle-accurate. The simulation of computation happens at the instruction set level. The simulation of the interconnection network is at the data-link layer of the Open System Interconnection (OSI) model. The power estimation performed for each individual processor is based on an instruction level power model. The model used within the simulator has been verified to be within 6.5% of measurements from the actual hardware. The power estimation for the communication links assigns fixed costs for both transmitting and receiving. Each individual processor, or groups of processors, may be attached to batteries of configurable capacity. The battery model employed is based on well-known discrete-time battery models [3, 11].

Failures are modeled in the links that interconnect the processing elements. These failures are modeled after intermittent electrical failures, with a parameterizable probability of failure (referred to as the *failure rate* in Section 6). These probabilities of failure are with respect to one simulation time step. For example, a stated failure rate of 1E-8 implies a probability of failure of 1E-8 per simulation step. The correlation coefficient is the likelihood that a link error will cause a node error on a node connected to that link.

Simulation Parameter	Beamforming	Software Radio
Operating Modes (each node)	60 MHz, 3.3V 15 MHz, 0.85V	60 Mhz, 3.3V 15 Mhz, 0.85V
Battery Size (each node)	1 mAh	1 mAh
Trans. and Rec. Power	100 mW	250 mW
Link Speed	200 kbps	10 Mbps
Frame Size	1024 bits	8192 bits
Frame Headers	288 bits	296 bits
Node Failure Probability	1E-8	1E-8
Correlation Coefficient	0.1	0.1
Baseline Migration Thresh., B_{low}	0.6	0.2
PCRE Migration Thresh., B_{vlow}	0.06	-

Table 1: Simulation variables and their associated values for both driver applications

	Topology	Errors?	CPU MHz
1	Dual-Bus (BF)	None	60 MHz
2	Dual-Bus (BF)	L.E.	60 MHz
3	Dual-Bus (BF)	N.E.	60 MHz
4	Dual-Bus (BF)	Ind. L.E. + N.E.	60 MHz
5	Dual-Bus (BF)	Cor. L.E. + N.E.	60 MHz
6	Dedicated Migration Link (BF)	None	60 MHz
7	Dedicated Migration Link (BF)	L.E.	60 MHz
8	Dedicated Migration Link (BF)	N.E.	60 MHz
9	Dedicated Migration Link (BF)	Ind. L.E. + N.E.	60 MHz
10	Dedicated Migration Link (BF)	Cor. L.E. + N.E.	60 MHz
11	Dual-Bus (BF)	None	20 MHz
12	Software Radio	None	60 MHz
13	Software Radio	Cor. L.E. + N.E.	60 MHz

Table 2: Details showing how each experiment was setup (L.E. = Link Errors, N.E. = Node Errors, Ind. = Independent, Cor. = Correlated, BF = beamforming application).

5.2 Simulation Setup

Several relevant simulation variables, and their associated values, are given in Table 1. The software radio application is a much more demanding application, requiring a much higher communication link speed and was configured with a higher transmit and receive power dissipation rate on the communication links.

All the beamforming experiments employ an implementation with 1 master node and 10 slave nodes. In every experiment, half the slave nodes are inactive and used as targets for migration. The software radio application employs 12 nodes, one for each of the source, low-pass filter, demodulator and sink nodes, and 8 nodes for the equalizer stage. This partitioning of the software radio application is illustrated in Figure 7.

As can be seen from Table 1, each node is attached to a battery with a capacity of 1.0 mAh. For the design alternative of using a larger battery per node, each node is attached to a battery with twice the capacity, or 2.0 mAh. These battery capacities are on the order of 10 times smaller than a common wristwatch battery [1]. Such a small battery in today's technology would have a size of 1 mm³ and cost much less than a dollar.² The threshold for PCRE, B_{vlow} in Table 1 is 10 times lower than that for baseline migration in the beamforming experiments because only one frame needs to be transmitted to complete migration.

Eleven experiments are conducted to compare PCRE in the case of the beamforming application with two other design solutions: using

²This size of battery was chosen to limit the simulated system's lifetime and make simulation possible in a reasonable amount of real time.

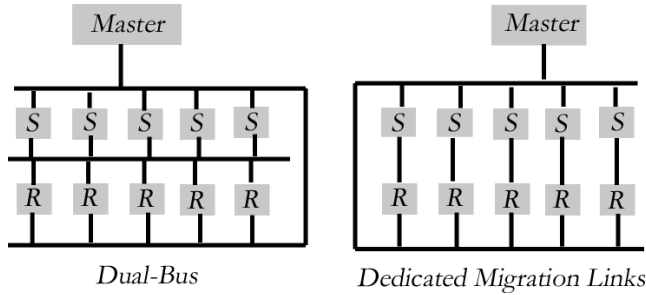


Figure 8: Two topologies considered in experiments: *Dual Bus* and *Dedicated Migration Link* topologies (S = slave node, R = redundant node)

the baseline migration scheme as defined in Section 2.1, and using a larger battery per processing node instead of using code migration at all. PCRE is well suited for remapping the beamforming application because beamforming degrades gracefully with the removal of nodes. Therefore, using non-overlapping migration time-slots as in PCRE permits graceful degradation of performance during migration. Software radio, in contrast, is highly dependent on the functioning of each of its components. Migration for each active node occurring simultaneously provides the best application lifetime possible, and therefore baseline migration is more well-suited for this, and other similar applications.

The details of each configured experiment are shown in Table 2. Two topologies for interconnecting the master, slave and redundant nodes are considered for the beamforming application, and are depicted in Figure 8. In one topology, referred to as *Dual-Bus*, shared communication buses are employed, one for exchanging samples between the master and slave nodes and the other for migration of slaves to redundant nodes. In the second topology, the single shared bus for the exchange of samples is maintained, but *Dedicated Migration Links* are employed for migrating the slaves to the redundant nodes. In Figure 8, the redundant nodes are labelled R and the slave nodes labeled S.

To investigate the robustness of the new code migration technique in the presence of faults, simulations are performed with intermittent failures in the slave and redundant nodes, failures in the links, as well as independent and correlated failures in both the nodes and the links. Table 1 shows the node and link failure probabilities per simulation cycle. A $1\text{E-}8$ per 16.6ns cycle time error probability, for a link operating at 200 kbps, translates to an error rate of $3.125\text{E-}6$ per bit transmitted for the beamforming application.

6. RESULTS AND DISCUSSION

The results using baseline migration are presented for use in both beamforming and software radio. A detailed evaluation of PCRE is presented for the beamforming application.

Performance results for the software radio implementation using baseline code migration is given in Figure 9. In this case, the lifetime of the application is extended by 30%. The figure shows data points representing whether or not each particular sample value (spaced 4 milliseconds apart) successfully exited the pipeline or not. Using baseline code migration improved the operational longevity of software radio. Exp. #12 lived 65.1% as long as the case of using a battery with twice the capacity, and Exp. #13 lived only 52.6% as long as this ideal case. However, there is overhead in attempting migration and checking for the right time to do so, and this prevents the results from reaching the ideal value of 100%

Figures 10 and 11 show the number of samples received by the

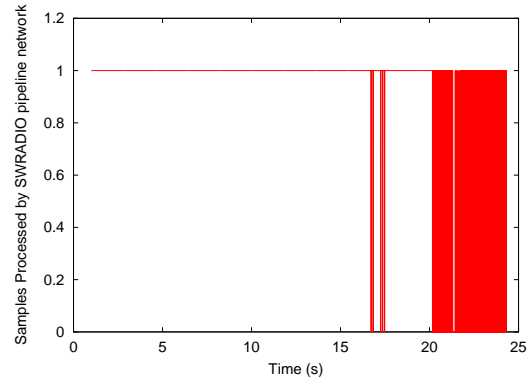


Figure 9: (Exp. #12) Performance of baseline migration scheme for the software radio application. The data points in the graph are spaced at 4 milliseconds apart and indicate whether the samples exit the pipeline successfully “1” or not at all “0”. The graph indicates that migration happens for the equalizer, low-pass filter, demodulation, and source nodes, in this order.

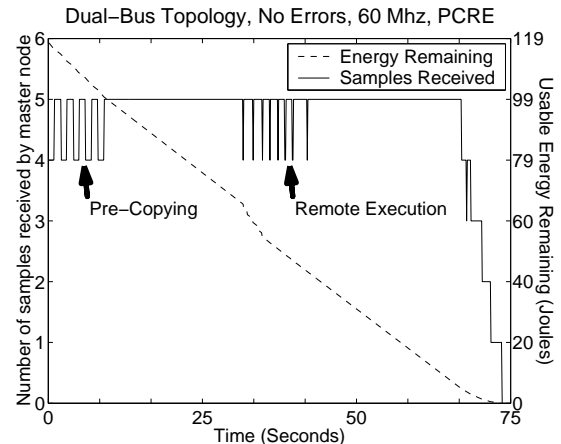


Figure 10: Performance of PCRE in *Dual Bus* Topology, no errors, 60 MHz processors

master node as well as the remaining usable battery energy for the beamforming application as it evolves for Exp. #1. In the case of PCRE, the scheduling algorithm presented in Section 3 has been employed. The solid line of Figure 10 shows that the first 10 seconds is the time in which pre-copying is taking place by each active node, and this unique “sawtooth” shape helps to prevent possible link collisions during pre-copying.

At approximately 35 seconds, all nodes send their final *FIN* frame to the redundant node to complete migration. For the baseline migration implementation, as is apparent in the solid line of Figure 11, nodes begin migration around 15 seconds and completely resume normal operation by 20 seconds. The dotted lines show that, for the baseline migration scheme, the amount of usable battery energy decreases considerably in the region of 15 to 20 seconds because, in this region, each slave node is abandoning what remaining energy it may have left by performing the migration. The amount of usable energy remaining using PCRE also drops during the final migration stage, but *not* as drastically as in the baseline migration case.

In Exp. #11, the processor’s active voltage mode was scaled down until the operating frequency was 20 MHz, and that processor is not computationally strained when handling beamforming with a 100ms sampling time. The system lifetimes in that experiment was 25.4

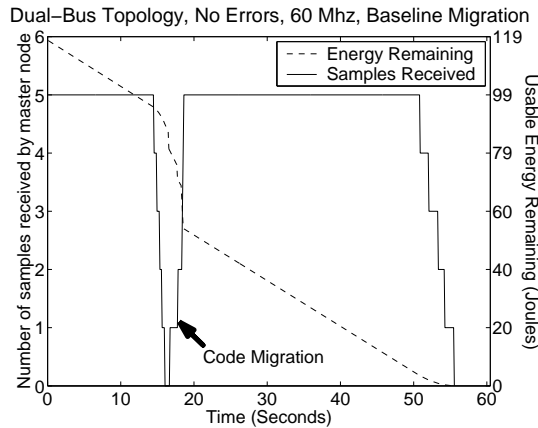


Figure 11: Performance of baseline migration scheme in Dual Bus Topology, no errors, 60 MHz processors

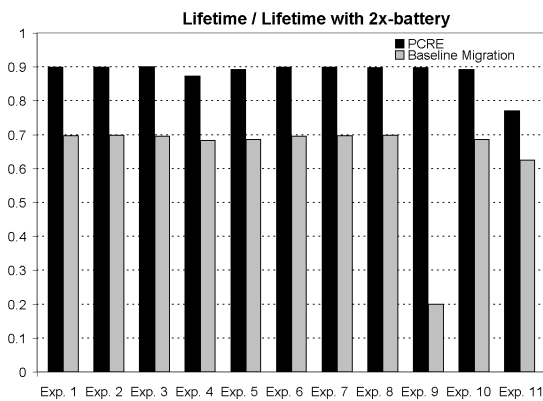


Figure 12: System lifetime for two migration schemes, normalized to the system lifetime resulting by simply using batteries with twice the capacity

minutes for the PCRE migration scheme and 20.6 minutes for the baseline migration scheme. Additionally, these experiments all assume that there is *no means* by which the batteries can recharge after being partially or completely depleted of energy. With rechargeable batteries, these system lifetimes can be extended much further, and the platform able to support multiple, concurrent distributed applications.

Summarizing results for all the beamforming experiments appears in Figure 12. On average, PCRE increases the system lifetime over the baseline migration scheme by 57.9%. For baseline migration, Exp. #9 did not successfully migrate mainly due to random node and link errors, so this impressive increase is largely due to the great improvement in lifetime for this experiment. Removing this experiment from consideration results in an average system lifetime improvement of 28.6% for PCRE compared to baseline migration. Figure 12 shows these lifetimes, normalized to the system lifetime of the “ideal” configuration where each node has twice the battery capacity and no migration occurs. As it can be seen, PCRE achieves, in most cases, a system lifetime within 10% of the ideal case while baseline migration is usually 30% worse than the ideal system lifetime.

The availability of the beamformer was improved using PCRE compared to baseline migration. In Exp. #1, PCRE had an average of 4.698 samples received during its lifetime per round, with a standard deviation of 0.795. Baseline migration, meanwhile, suffered somewhat with only 4.498 samples received per round, with a

higher standard deviation at 1.273. Similar arguments can be made for the other experiments, as well.

Next, the *energy efficiency* is also improved using PCRE as opposed to baseline migration. For Exp. #1, PCRE consumes 115.26 Joules of its 118.8 Joule total capacity with a system lifetime of 72.3 seconds, and baseline migration consumes 95.2 Joules of its 118.8 Joule total capacity with a system lifetime of 54.4 seconds. This gives PCRE a ratio of 0.627 seconds lifetime per Joule consumed, and the baseline migration 0.571 seconds lifetime per Joule consumed. Therefore, for Exp. #1, PCRE enjoyed an energy efficiency improvement of 9.8% compared to the baseline migration scheme.

7. CONCLUSION

Ambient intelligent systems are a promising emerging platform that presents unique challenges to communities in both hardware and software design and testing. Environment-driven failures may be common and redundancy in nodes will be required to provide a satisfactory level of application performance. This paper explores various methods for performing code remapping as a form of fault-tolerance and implements these techniques with two driver applications, beamforming and software radio. Our experimental results show 28.6% average improvement in overall beamforming lifetime using PCRE and about 30% improvement in system lifetime for software radio using baseline code migration.

8. REFERENCES

- [1] Panasonic Coin Type Li Ion Battery (Part no. BR1216). Digi-Key Catalog, <http://www.digikey.com>.
- [2] T. Basten, L. Benini, A. Chandrakasan, M. Lindwer, J. Liu, R. Min, and F. Zhao. Scaling into Ambient Intelligence. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE '03)*, pages 76–81, 2003.
- [3] L. Benini, G. Castelli, A. Macii, E. Macii, M. Poncino, and R. Scarsi. A discrete-time battery model for high-level power estimation. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE '00)*, pages 35–39, January 2000.
- [4] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next Century Challenges: Scalable Coordination in Sensor Networks. In *Proceedings of the 5th annual International Conference on Mobile Computing and Networking*, pages 263–270, 1999.
- [5] W. R. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive Protocols for Information Dissemination in Wireless Sensor Networks. In *Proceedings of the 5th annual International Conference on Mobile Computing and Networking*, pages 174–185, 1999.
- [6] D. Johansen, R. van Renesse, and F. Schneider. Operating system support for mobile agents. In *5th IEEE Workshop on Hot Topics in Operating Systems*, 1995.
- [7] U. Kremer, J. Hicks, and J. Rehg. Compiler-Directed Remote Task Execution for Power Management. In *Workshop on Compilers and Operating Systems for Low Power (COLP '00)*, October 2000.
- [8] M. Lindwer, D. Marculescu, T. Basten, R. Zimmermann, R. Marculescu, S. Jung, and E. Cantatore. Ambient Intelligence Visions and Achievements; Linking abstract ideas to real-world concepts. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE '03)*, pages 10–15, 2003.
- [9] D. Milojević, F. Douglass, Y. Paindaveine, R. Wheeler, and S. Zhou. Process Migration. *ACM Computing Surveys*, 32(3):241–299, September 2000.
- [10] D. Milojević, W. LaForge, and D. Chauhan. Mobile objects and agents. In *USENIX Conference on Object-oriented Technologies and Systems*, pages 1–14, 1998.
- [11] D. Rakhmatov, S. Vrudhula, and D. Wallach. Battery Lifetime Prediction for Energy-Aware Computing. In *Proceedings International Symposium on Low Power Electronics and Design*, pages 154–159, 2002.
- [12] P. Rong and M. Pedram. Extending the lifetime of a network of battery-powered mobile devices by remote processing: A markovian decision-based approach. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC '03)*, pages 906–911, June 2003.
- [13] A. Rudenko, P. Reiher, G. J. Popek, and G. H. Kuenning. The Remote Processing Framework for Portable Computer Power Saving. In *ACM Symposium on Applied Computing*, pages 365–372, February 1999.
- [14] P. Stanley-Marbell and M. Hsiao. Fast, flexible, cycle-accurate energy estimation. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 141–146, August 2001.
- [15] S. Thad, S. Mann, B. Rhodes, J. Levine, J. Healey, D. Kirsch, R. Picard, and A. Pentland. Augmented reality through wearable computing, 1997.
- [16] J. White. *Mobile Agents*. J. M. Bradshaw (Ed.), MIT Press, 1997.