A Task Morphing Framework For Energy Consumption and Reliability Tradeoffs

Sathish Gopalakrishnan
Department of Electrical & Computer Engineering
The University of British Columbia

***Abstract*—*Morpheus* is a framework that permits tasks to adapt their execution times (and consequently, energy consumption) and reliability guarantees through the use of (developer-supplied) program approximations and compiler-driven synthesis of reliability mechanisms. *Morpheus* parametrizes different execution sequences of a task, and through its runtime system enables the execution of suitable approximations and reliability mechanisms. *Morpheus* supports custom resource allocation policies that are invoked to decide on suitable parameters for a task at runtime. In the context of real-time computing, *Morpheus* allows a task to produce solutions with acceptable error and gains execution time for methods that increase the reliability of the system.**

I. Introduction

Morpheus is a programming language and runtime framework that we propose to enable application developers to specify and make tradeoffs between energy consumption (or execution time), solution quality and reliability.

Timeliness and reliability are two important (non-functional) properties of a computing system. These properties are important because of an emphasis on quality of service in computing. This emphasis, while typical in safety-critical/mission-critical systems with real-time constraints, is spreading to a broader class of computing applications. Reliability and availability of computing platforms are affected by changes in semiconductor device technology. The changes in device technology have led to smaller devices but have had some downsides: small semiconductor devices are more vulnerable to faults than large semiconductor devices and the manufacturing process for smaller devices is also more error-prone. Additionally, these tradeoffs need to be made with joint consideration for energy consumption, which has become a significant constraint for computing systems today.

We present an approach to managing energy consumption and reliability tradeoffs by controlling execution time and solution quality in certain computational tasks. We make the assumption that controlling the execution time of a task is related to controlling its energy consumption. We do not make any assumptions about the relationship between execution time and energy consumption other than that longer execution times imply a higher energy consumption at a given CPU speed.

The premises of our work are two-fold. The first premise is an observation concerning hardware reliability. The second premise relates to quality of service from a solution accuracy viewpoint. We will discuss the two foundational assumptions and then present the guiding principle behind *Morpheus*.

Premise 1. Hardware faults have a variety of causes and conventional wisdom around hardware design revolves around presenting a fault-free hardware platform abstraction to the software layer. For example, some transient or even intermittent faults can be handled through redundancy in the microprocessing units. There is a high cost of development (and deployment) associated with hardware-oriented techniques for fault tolerance. It is also likely that only a small fraction of applications and users require high levels of dependability. Thus specializing hardware solutions for a small user population leads to elevated prices for reliable computing hardware; the higher prices are a significant stumbling block in many application domains. Hardware techniques are often rigid and may not offer a sufficient space for tradeoffs between some of the key metrics in systems design such as performance, correctness and energy consumption.

Premise 2. In many applications, particularly in applications dealing with multimedia content, signal processing and control algorithms, it is possible to sacrifice program correctness or accuracy for savings in task execution time. The savings achieved in this fashion can then be used to integrate software-driven dependability mechanisms.

Software solutions for improving dependability can be flexibly deployed and excluded when not needed. There are several software-based approaches to improving application dependability such as recovery blocks and replicated execution. Some techniques, such as bounds checking for arrays and pointer checking, address more specific software-level problems. None of these techniques provide complete fault coverage and recovery but a judicious mix of techniques can improve reliability significantly. Additionally, it is possible to handle software-related errors through software mechanisms because it is difficult to implement suitable reasoning methods at the hardware level.

In particular, *Morpheus* permits switching between variants of a task, which we also refer to as *morphing*, based on runtime information, wherein an imprecise version of the task, with reduced running time, can be executed – with some manner of redundancy – to improve reliability. The *Morpheus* approach involves some program modifications that can then be exploited at runtime to identify suitable methods for execution. At runtime, *Morpheus* uses monitoring and a selection mechanism to choose appropriate program transformations (morphs). During the compilation phase, *Morpheus* automatically generates some variants for a program using techniques for improving reliability (e.g., over-allocating space for stacks and heaps) that have been described by others in prior work. Experimental evaluation suggests that *Morpheus* as a framework for task morphing that can be realized with tolerable overhead.

The principal contribution of this work is to demonstrate the feasibility of such runtime adaptation by realizing a holistic framework that involves programming language extensions, compilation support and a suitable runtime system, and to demonstrate that the overheads of runtime adaptation need not be high. In addition, we suggest that using user-space schedulers is a suitable approach to enabling system adaptation.

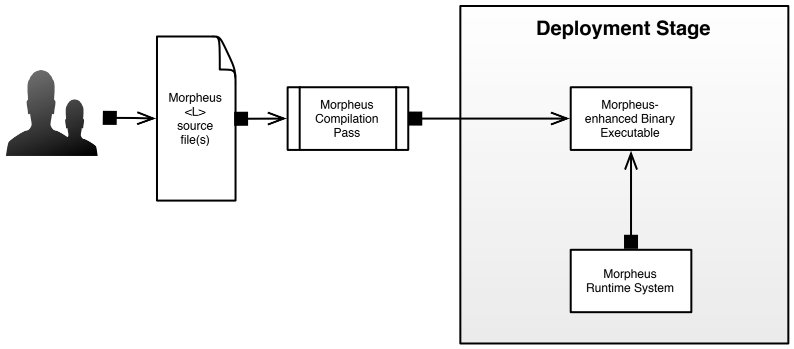

Morpheus <L> source file(s)
Morpheus Compilation Pass
Deployment Stage
Morpheus-enhanced Binary Executable
Morpheus Runtime System

Fig. 1. An overview of the *Morpheus* system: The developer supplies source code in language L with Morpheus annotations. The source code, post-compilation, results in an executable that includes several alternative implementations of the same code block. The alternatives are a result of approximations and additional dependability-enhancing code. During execution, using parameter values that are set via the runtime system, appropriate code blocks are selected.

II. The *Morpheus* System

A. Overview

In *Morpheus* (Figure 1), an application developer can specify multiple versions of certain methods, each with varying levels of approximation. For example, when implementing the `Discrete Fourier Transform` one can achieve different degrees of approximation through approximations of the `sin` and `cos` functions. These approximations can yield significant reductions in task execution time. The program, with all the approximations in place, can then be compiled and certain transformations can be applied during this phase to increase fault resilience. An example, in this phase, would be the allocation of padding space to array declarations or the introduction of additional code to handle reads to `null` pointers. Finally, during program execution, based on operating conditions such as system utilization, the appropriate version of a function can be invoked. With the `Discrete Fourier Transform`, for instance, when utilization is high but reliability is important then an approximated `sin` function can be utilized with additional reliability mechanisms woven in.

Morpheus, as outlined in the example above, has three aspects. The first aspect is the set of annotations to the programming language that permit a developer to specify the different variants for the same functionality as well as to suggest valid/invalid reliability transformations. The second aspect is the modification to the compilation phase that results in automatic generation of reliability enhancements. The last aspect is the runtime system that enables situation-aware adaptation.

We now proceed to detail each aspect of the *Morpheus* system. Our implementation targets applications developed in C and C++. We use the `clang/LLVM` compilation infrastructure [9] and the Minix 3 operating system [21] for our implementation although the approach is applicable to any programming language and operating system. In this introduction to *Morpheus*, we will discuss some of the design choices and then present some of the implementation issues that are specific to the chosen platforms.

B. Programming Language Extensions

1) Design: The goal of this part of the effort is to enable programmers to specify potential approximations that reduce execution time with some (tolerable) reduction in solution quality. We envision two methods of expressing approximations. The first method is through the exposure of specific parameters such as loop bounds, loop increments and, as may be the case with standard polynomial time approximation schemes, parameters that affect the approximation ratio. The second method is the specification of alternative code blocks that represent variations of the same functionality. At runtime a set of parameter values will have to be chosen in the first method and in the second method suitable variations should be chosen.

The primary purpose of the programming language extensions is to expose specific parameters concerning program approximations. These parameters are then set at runtime. Similarly, such parametrization permits a compiler to generate specified reliability mechanisms. The syntax we chose for expressing alternatives is simplistic and one can easily construct more sophisticated solutions to the programming language design problem.

2) Implementation: The programming language extensions have been implemented for C/C++ and preprocessing is performed using `clang/LLVM` [9] to generate standard C/C++ code needed for the additional blocks.

C. Code Generation

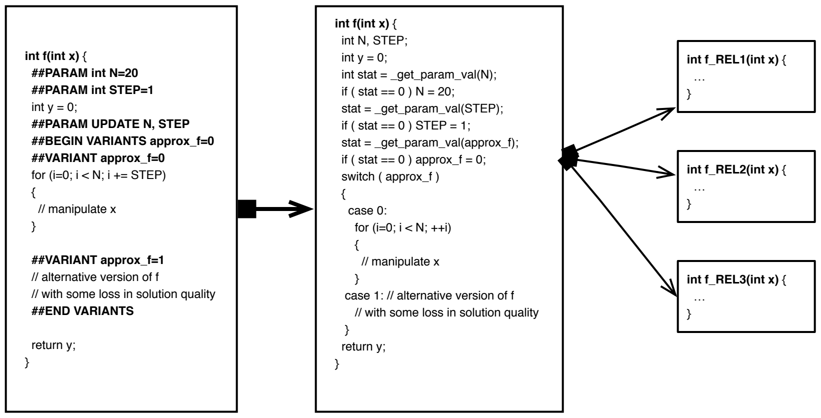
1) Design: Morpheus takes developer-supplied source code and produces an executable program that includes the different approximations as well as additional methods to tolerate errors (arising in software and hardware).

Generating code for the different approximations is easily achieved because the approximations are clearly parametrized and developer-supplied. On the other hand, to enhance reliability we rely on a set of techniques that have been proposed in prior work (e.g., failure-oblivious computing, redundant execution).

The motivation behind *Morpheus* is not to develop new fault detection and recovery techniques but to enable flexible tradeoffs between solution quality and reliability. In principle new reliability mechanisms may be added.

We implement automatic transformations for the following techniques that have been proposed to target different errors.

- Increase memory allocation (D-IMA) [14]: In this method, we increase stack allocations and heap space allocations to deal with overflow issues.
- Failure-oblivious computing (D-FOC) [18]: This is an implementation of a subset of techniques from the work on failure-oblivious computing. In this method, we deal with two types of pointer access problems. Reads to `null`



```
int f(int x) {
  ##PARAM int N=20
  ##PARAM int STEP=1
  int y = 0;
  ##PARAM UPDATE N, STEP
  ##BEGIN VARIANTS approx_f=0
  ##VARIANT approx_f=0
  for (i=0; i < N; i += STEP)
  {
    // manipulate x
  }

  ##VARIANT approx_f=1
  // alternative version of f
  // with some loss in solution quality
  ##END VARIANTS

  return y;
}
```

```
int f(int x) {
  int N, STEP;
  int y = 0;
  int stat = _get_param_val(N);
  if ( stat == 0 ) N = 20;
  stat = _get_param_val(STEP);
  if ( stat == 0 ) STEP = 1;
  stat = _get_param_val(approx_f);
  if ( stat == 0 ) approx_f = 0;
  switch ( approx_f )
  {
    case 0:
      for (i=0; i < N; ++i)
      {
        // manipulate x
      }
   case 1: // alternative version of f
      // with some loss in solution quality
   }
  return y;
}
```

```
int f_REL1(int x) {
  ...
}
```

```
int f_REL2(int x) {
  ...
}
```

```
int f_REL3(int x) {
  ...
}
```

Fig. 2. In this example, we illustrate the two types of approximations that a developer may want to expose. The parameter N represents the number of loop iterations and one can perform fewer iterations if needed. Similarly, the function $f()$ has two variants that are parametrized; one of these variants will be chosen at runtime. The developer supplies the source code – in C with Morpheus annotations – on the left and on the right we illustrate the equivalent code that is generated after transformations. This example uses C code to show the transformation although an executable binary is generated by our implementation. Reliability mechanisms – not shown as code – are automatically added, leading to one version (e.g., $f\_REL1()$) per reliability mechanism.

pointers are ignored and a 0 is returned as the default value. Writes to `null` pointers are ignored.
- Selective memory replication (D-SMR) [13]: This method uses a modified memory allocator to maintain replicas of critical variables. This permits detection and recovery from some memory errors (and memory corruption attacks).
- Software-controlled fault tolerance (D-SFT) [17]: This method uses triple redundant code execution and majority voting before critical instructions to detect and tolerate faults.

2) Implementation: Code generation is performed using the LLVM compilation framework. We use LLVM to obtain an intermediate representation of the source code, and then we systematically add additional checks and actions. Of the methods that we implemented, D-SMR and D-SFT are the most sophisticated. D-SMR requires a modified memory allocator but is backward-compatible with standard dynamic memory allocation techniques.

In this phase, we also add, at appropriate developer-specified points, calls that retrieve parameter values from a system service. These values are set at runtime.

The number of versions of a task that are generated is a product of the number of program alternatives and the number of reliability mechanisms. At any given time instant (during runtime) only one of the reliability mechanisms will be active. This choice is specific to our current implementation. It may be possible to combine some of the reliability mechanisms automatically or based on developer directives in a future implementation.

D. Runtime System

1) Design: The *Morpheus* runtime system can be divided into three components (Figure 3). The first component is a monitor that determines the current level of resource usage and tracks other execution-time events such as task failures. Data collected by the monitor is then utilized by the second component, the decision engine. The decision engine applies developer-supplied rules to the data obtained from the monitor and emits different parameter settings. The third component of the runtime system is a parameter management system that accepts settings from the decision engine and supplies

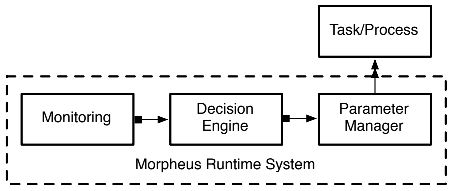


Fig. 3. The *Morpheus* Runtime System. At runtime, statistics are gathered regarding system and task performance. These are then used by a decision manager to set code-level task parameters that guide the approximations and the reliability mechanisms. A parameter manager permits a process to access parameters at runtime.

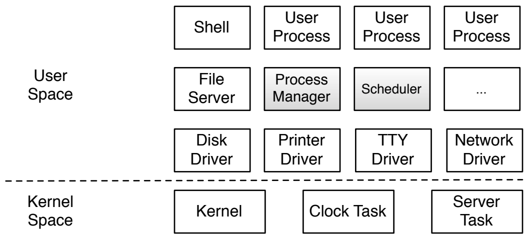


Fig. 4. Minix 3 architecture: The layering of processes in userspace is logical; all processes are treated similarly by the kernel. Synchronous message passing with rendezvous is used for communication between processes and for communication between processes and the kernel.

these parameters, on demand, to executing processes. Whereas monitoring and parameter management can be considered generic services, the decision engine relies on rules that are specific to a task/process that is being controlled. We separate decision making from the actual process being controlled because the rules or policies for controlling a task are separate from the task itself. The separation of policies and tasks allows the two entities to be modified, to a large extent, independently.

2) Implementation: There are several possible implementation tactics for the runtime system. With Minix 3 as the operating system of choice, we were able to use some existing OS features to implement *Morpheus*. Minix 3 is microkernel-based operating system (Figure 4) with an emphasis on reliability [6] and this emphasis aligns with our goal of aiding software developers in building a system with adaptable dependability.

Minix 3 delegates process scheduling to a userspace scheduler [20]. A process is allocated a time slice (quantum) and decisions about priority changes are made in userspace at the end of a timeslice. One can associate a userspace scheduler with each process or a group of processes. If there is no userspace scheduler associated with a process then a default scheduling policy (prioritized scheduling with round-robin allocation among tasks at the same priority level) is applied.

The default implementation of userspace scheduling in Minix 3 involves the userspace scheduler being invoked when a process' scheduling quantum expires. The userspace scheduler can then decide on how priorities should be modified when a quantum expires. (For example, the scheduling policy used in Minix 3 lowers the priority of a task by one level every time it runs out of its quantum and then periodically scans through the list of all processes that have been lowered in priority and increases their priority by one level. This prevents CPU-bound tasks from monopolizing the CPU [22].)

We modified Minix 3 to support periodic processes and the earliest deadline first (EDF) scheduling policy to provide real-time scheduling support to Minix 3. We do not detail these modifications extensively because they have been performed earlier as well [23], [11]. We added a method to enable the kernel to call the userspace scheduler before a context switch was made to a user process, and this event is used to set parameters for the process to be executed. Parameters are set for a process either when it executes for the first time (if the task is not periodic) or at the start of every job instance for periodic tasks.[1] This restriction is achieved by maintaining a bit of information in the process control block to indicate if a process is being scheduled after preemption or whether a new job instance is being scheduled.

When the userspace scheduler is called before a process is executed, some parameters such as a task's execution time averaged over a time window as well as the overall CPU utilization are passed to the scheduler as message contents (because Minix 3 uses message passing). This data is used to adjust the code-level parameters of the process. Thus the *decision engine*, in our implementation, is part of the userspace scheduler although alternative implementations are possible. (We defer discussion regarding the specific resource allocation policies we implemented the section on system evaluation.)

To pass parameters between the userspace scheduler and the process(es) that it controls, we implement a variant of the data store service that Minix 3 provides. The default data store service allows device drivers and other system services to save state (primarily for recovery from failures), and thus only system services can write to and read from the default data store. We implemented a *parameter manager*, which is really a data store, that permits system services to read/write data and user processes can only read data. During the compilation phase, the appropriate system calls are added to a process' code to retrieve parameters from the parameter manager.

We now summarize the timeline of events at runtime. When the highest priority task that is eligible to run is selected, and the process is being chosen for the first time or a new instance of an existing periodic task is chosen, the associated userspace scheduler is sent a message with monitor information. The userspace scheduler uses monitor information and sets parameters via the parameter manager. Then, via the kernel, the process begins execution and it can find new parameter values through a system call (message) to the parameter manager.

[1]We impose this restriction because changing a job instance's flow midway through execution requires that the developer exercise much caution.

III. Design Rationale

Having described the architecture and implementation of *Morpheus*, we now provide a discussion concerning the design choices made. In this section we restrict attention to the choices that have been made. Later we will address possible extensions.

Target systems. In principle, *Morpheus* can be used in hard real-time systems but the effort so far has been towards supporting soft real-time systems. The broad assumption has been that for such systems performance/timeliness is often more important than reliability. Thus, when resources are scarce *Morpheus* attempts to reduce execution times of tasks. When extra resources are available – primarily CPU cycles – then *Morpheus* capitalizes on the extra capacity to add reliability mechanisms. This approach is suited to situations when application developers release new versions of software applications and additional checks for ensuring reliability maybe useful in the early stages of deployment.

Software-only reliability mechanisms. We have chosen to build a system that supports reliability mechanisms with no additional hardware requirements. This choice is motivated by our assumption about the target systems. By choosing to improve soft real-time applications where the extra operations for reliability are "preferred" but "not required" there is, implicitly, the need to support commercial off-the-shelf hardware components. Such COTS hardware typically does not have additional hardware to increase reliability.

Separation of resource allocation policies and task execution. It is possible to explicitly include all the decision logic that allows a task to adapt to different situations as part of the application. Separating the policies from the application they manage is useful for at least two reasons. The first reason is that one can independently change the resource management policies based on the expectations of solution quality and reliability without rewriting the source code or recompiling to obtain new application binaries. During deployment, one may want to experiment with different policies or change policies when some portion of the application is known to be reliable. A second reason is that this separation permits the system to manage resources across all tasks that are executing concurrently. Hard-coding the adaptation strategy in the application code will limit one's ability to perform whole-system optimization.

Reliability guarantees. Morpheus' reliability guarantees are only as good as the guarantees provided by the mechanisms chosen for implementation. To ensure high reliability it may be desirable to always run tasks with instruction-level redundancy (D-SFT) for detection and recovery but there is a high cost to such redundancy (more than twice the base execution time). *Morpheus* permits for tradeoffs between reliability, solution quality and system utilization. At this point the onus is on system designers to identify suitable operating points and to determine when the system switches from one operating point to another.

Developer effort. Developers do need to specify different approximations that are acceptable for their applications. Whereas some work (such as loop perforation [7]) can automate the generation of approximations, we believe that there are certain approximations that are currently difficult to synthesize mechanically. As regards the program transformations for enhancing reliability, most proposals, including the work that we have selected for implementation, are generic. We, therefore, expect that additional mechanisms can be added to the compilation framework and can then be made available to all applications without any increase in an application developer's effort.

IV. Evaluation

A. Evaluation Metrics

Having indicated the rationale for several choices made in the design of *Morpheus*, we now describe evaluation results for the framework. Our goal is to evaluate *Morpheus*. Neither is it our goal to evaluate techniques for enhancing reliability nor is it our intention to evaluate the quality of approximations. We will present some quantitative results along both these directions to emphasize the motivation for the work but we are primarily interested in the following metrics for assessing *Morpheus*.

- Overhead: What is the additional cost of utilizing *Morpheus* from a scheduling perspective?
- Adaptation: How does *Morpheus* adapt to changes in resource utilization?

It is also important to note that the experimental results are primarily to establish the viability of *Morpheus* as an approach. We believe that it is possible to improve the performance of *Morpheus* by choosing an alternative operating system and by optimizing some of the techniques that we use.

B. Experimental Setup

We used a desktop system with an AMD Athlon 3700+ microprocessor running at 2.4 GHz. The system was provisioned with 2GB RAM. We used Minix 3 (specifically Minix 3.2.0) with the modifications that were needed for *Morpheus* as discussed earlier in the article. We chose this configuration to represent a system that is not significantly resource constrained and at the same time is not as powerful as systems that utilize chip multiprocessors (or multicore processors).

C. Benchmarks

We selected some benchmark applications that offered opportunities for approximating solution quality and could be hardened with reliability mechanisms. It is possible to use *Morpheus* without using any approximations; in this situation, whenever extra utilization is available to a task it can be operated with additional reliability mechanisms. Similarly, it is possible to use *Morpheus* with no reliability enhancements and target only execution time savings (and consequently energy savings) through the judicious application of approximations.

- Discrete Fourier transform (DFT): DFTs are commonplace in signal processing applications. Computing such transforms requires evaluation *sine* and *cosine* functions and these functions can be approximated by varying the decimal precision in the computations. As a QoS metric, we compute the normalized difference in each output sample of DFT between the precise and approximated versions.
- Eon [19]: Eon is a probabilistic ray tracer that is part of the SPEC2000 benchmark suite. It works as follows: A number of 3D lines (rays) are sent into a 3D polygonal model. Intersections between the lines and the polygons are computed, and new lines are generated to compute light incident at these intersection points. The final result of the computation is an image as seen by camera. The computational demands of the program are much like a traditional deterministic ray tracer as described in basic computer graphics texts, but it has less memory coherence because many of the random rays generated in the same part of the code traverse very different parts of 3D space. Kajiya's algorithm, which is one of the probabilistic ray tracers that is part of the Eon benchmark uses n^2 iterations but it is possible to terminate the algorithm early with a loss in output fidelity. To quantify the QoS loss of approximate versions, we compute the average normalized difference of pixel values between the precise and approximate versions.

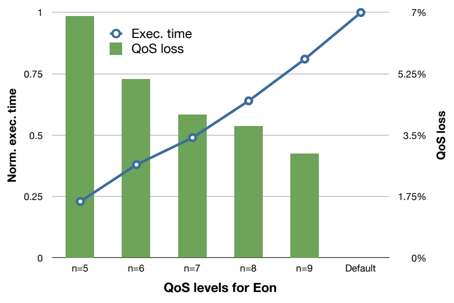


Fig. 5. Eon: QoS Loss vs. Execution Time. The probabilistic ray tracer, Eon, uses n^2 iterations to render an image. Using fewer iterations results in execution time reduction with limit QoS loss.

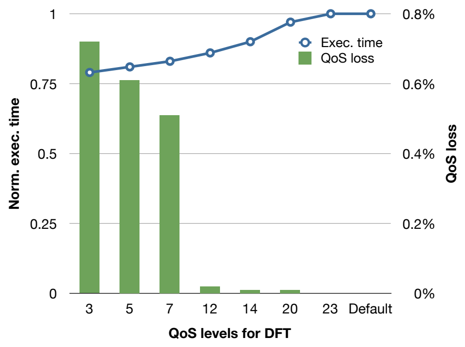


Fig. 6. DFT: QoS Loss vs. Execution Time. For discrete Fourier transforms, changes in the precision of *sine* and *cosine* functions result in execution time savings with marginal QoS degradation. The QoS levels correspond to the number of decimal places used in the computation.

Our benchmark list could have been longer but others have shown, separately, the value of approximations (e.g., [2]) and the value of reliability mechanisms (e.g., [17], [14]).

D. Evaluation Results

The first set of experiments show that it is possible to save on execution time by accepting a drop in solution quality. We demonstrate this with two benchmarks, Eon and DFT. We used different levels of approximation and noted the changes in execution time and solution quality for Eon (Figure 5) and DFT (Figure 6). We used 120 different input data sets for each benchmark.

The second set of experiments (Figure 7) captured the effect of the different reliability mechanisms on the running times of tasks. Taking the first two sets of experiments together, we observe that there is sufficient space for trading off solution quality for enhancing reliability.

The third set of experiments we performed were to measure the runtime overhead of using *Morpheus* in the critical path of switching from one process to another. When switching from one process to another, the additional work involved is

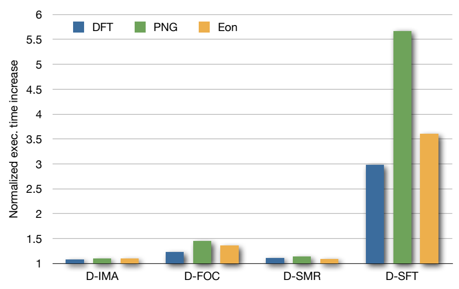


Fig. 7. Overhead of Reliability Mechanisms. The change in execution times for tasks depends on the reliability mechanism and the application.

the invocation of the userspace scheduler at the start and then the actual transfer of control to the appropriate process. Then, there is an extra step of potentially refreshing parameters when the process starts. At the implementation level, in Minix 3, the work involved can be translated to additional messages and context switches. To identify the overhead of this step we performed 10000 runs and used the processor's timestamp counter (TSC) to measure the additional time involved. In this evaluation, the userspace scheduler does not actually make any task adjustments. Any extra time used in deciding on task parameters needs to be considered before choosing the policy. We found that the additional overhead was no more than $1.6\mu s$. (By default, the context switch time with Minix 3 on our platform was no more than $1.1\mu s$. With *Morpheus*, the context switch time was no higher than $2.7\mu s$.) This additional overhead is incurred only once every period for a periodic task in our current implementation.

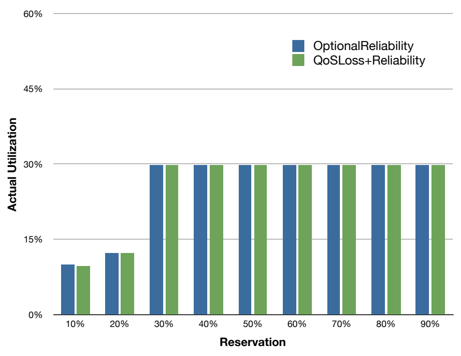
The fourth set of experiments we performed demonstrates how *Morpheus* can enable task morphing at runtime. We evaluated the impact of *Morpheus* on the benchmarks. For Eon and DFT we added approximations. We implemented these tasks as periodic tasks, and each task was provided with a worst-case execution budget and a period. The *Morpheus* runtime system was then allowed to decide on the execution times of tasks using simple policies, and then set the parameters corresponding to the chosen execution times. The execution time budget therefore acts like a resource reservation. The task periods were chosen to be long enough to permit the basic implementation of a task – without any modifications that *Morpheus* requires – to run at 10% utilization. Whenever a task is allocated more than 10% CPU utilization there are spare cycles that could be used for additional reliability instrumentation.

We experimented with two policies for using the allocated reservations. The policies attempt to maximize execution time without violating the reservation.

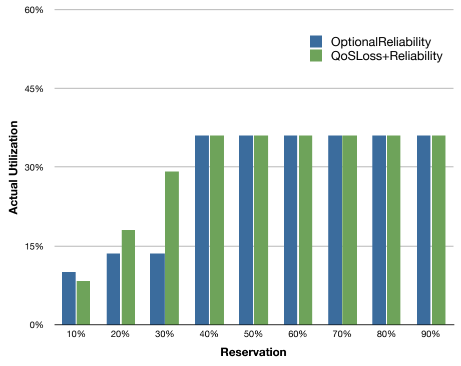
- OptionalReliability: This policy never opted for degraded solution quality but when sufficient CPU reservation was available then it would select the reliability technique that maximized execution time.
- QoSLoss+Reliability: This policy would pick the reliability technique that maximized execution time while also accepting degraded QoS.[2]

Morpheus is able to adapt to the available resource supply easily. As reservations are increased, *Morpheus* uses the additional capacity to either improve QoS or add reliability mechanisms (Figures 8).

[2]We assumed that a developer would never include as a candidate approximation any method that degraded solution quality to unacceptable levels.



(a) DFT



(b) Eon

Fig. 8. Effect of *Morpheus* on Utilization. Given the resource reservation, *Morpheus* selects the best version of a task to execute. This choice depends on whether reliability is optional, in which case there is no drop in QoS, or whether it is acceptable to use a lower QoS version and a reliability technique without exceeding the reservation.

Morpheus is also able to perform the task morphing quickly. In one experiment, we added to the reservation for a task when a job instance was executing. *Morpheus* correctly invoked additional reliability mechanisms for the next invocation of the task to utilize the additional resource allocation.[3]

This initial experimental study of *Morpheus* suggests that it is a feasible approach to making tradeoffs at runtime between different aspects of a real-time system that are often in a constant state of tension: execution time (and also energy consumption), reliability and solution quality.

V. Related Work

Two noteworthy research projects that are closely aligned with the *Morpheus* idea are (i) the Green system [2] for principled use of approximations at the programming language level (Microsoft Research) and (ii) the code perforation project [7] (MIT). Green provides programming language support, in some ways similar to *Morpheus*, for a programmer to specify different approximations and to perform QoS profiling at runtime to select approximations that yield running time reductions (or energy efficiency) with acceptable QoS loss. With code perforation, it is possible to automatically realize specific approximations. A special case of code perforation is loop perforation, which results in omitting some loop iterations because such omissions result in small solution

[3]We decided not to include a timeline graph to illustrate this effect because of the limited value of such a graph.

quality loss. *Morpheus* explicitly tackles reliability issues and uses the spare capacity created as a result of approximations to execute code for fault detection and recovery. Additionally, *Morpheus* explicitly permits the setting of runtime parameters that influence program execution. This can be used to influence per-job execution in the case of periodic tasks. Green does not employ such techniques; the time window over which Green makes changes to an executing task is much longer. The code perforation project has resulted in the PowerDial mechanism [8] that does permit limited adaptation every planning interval. We also note that code perforation has been considered in conjunction with fault recovery but primarily in the context of using spare cycles for task re-execution when an error occurs.

With an eye on building real-time systems, there have been efforts that incorporate aspects of program approximation and fault tolerance primitives with the scheduling of periodic tasks. The imprecise computation model due to Liu et al. [10] used the notion that tasks could be interrupted, if needed, after a mandatory execution time with a loss on solution quality. This idea is related to that of *anytime algorithms* [25], [24] that have been of interest in the design and understanding of control systems and artificial intelligence systems. From a fault tolerance perspective, there has been extensive work on scheduling job re-executions in the event of an error, especially from Melhem, Mossé, et al. [5], [1], [12]. From an implementation perspective, fault-tolerant real-time Mach [4] is one operating system that was designed to support re-scheduling of tasks when failures are encountered. There have been very few other implementation efforts whereas there has been a significant investment in analysis of scheduling policies.

This effort is also related to bridging the gap between fault characterization, diagnosis of hardware faults [16], [3], [15] and recovery; this work represents the next stage where recovery mechanisms are enabled.

There are many other articles of interest in each of the areas that *Morpheus* spans but we do not discuss all such work for the sake of brevity.

VI. Conclusions

We believe that runtime adaptations are important in jointly addressing issues such as timeliness and reliability requirements in long-running embedded systems. To this end, we have presented a system called *Morpheus* that enables such adaptations. For this article we have not carried out a complete study of the reliability enhancements that may be achieved by such an approach. We have also not established strategies to select the best adaptation(s) for a given scenario. This requires further work. We have only established that it is possible, with limited overheads, to achieve cross-layer adaptations through extensions to the programming language and the runtime environment. We make use of user-space scheduling to this end and this design choice may be valuable in other situations as well (for example, enabling support for mixed-criticality scheduling).

Acknowledgements. This work was supported in part by NSERC Canada under the Discovery Grant program, Grant CRDPJ 434659-12 and the ICICS/TELUS People & Planet Friendly Home Initiative at UBC.

References

[1] Aydin, H., Melhem, R., and Mossé, D. Optimal scheduling of imprecise computation tasks in the presence of multiple faults. In *Proceedings of the International Conference on Real-Time Computing Systems and Applications* (December 2000), pp. 289–296.

[2] Baek, W., and Chilimbi, T. M. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 2010), pp. 198–209.

[3] Dadashi, M., Rashid, L., Pattabiraman, K., and Gopalakrishnan, S. Integrated hardware-software diagnosis for intermittent hardware faults. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (2014), pp. 363–374.

[4] Egan, A., Kutz, D., Miklun, D., Melhem, R., and Mossé, D. Fault-tolerant RT-Mach (FT-RT-Mach) and an application to real-time train control. *Software: Practice and Experience 29*, 4 (April 1999), 379–395.

[5] Ghosh, S., Melhem, R., Mossé, D., and Sarma, J. S. Fault-tolerant rate-monotonic scheduling. *Real-Time Systems 15*, 2 (1998), 149–181.

[6] Herder, J. N., Bos, H., Gras, B., Homburg, P., and Tanenbaum, A. S. Minix 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review 40*, 3 (July 2006), 80–89.

[7] Hoffmann, H., Misailovic, S., Sidiroglou, S., Agarwal, A., and Rinard, M. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Tech. Rep. MIT-CSAIL-TR-2009-042, MIT, September 2009.

[8] Hoffmann, H., Sidiroglou, S., Carbin, M., Misailovic, S., Agarwal, A., and Rinard, M. Dynamic knobs for responsive power-aware computing. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (March 2011), pp. 199–212.

[9] Lattner, C. LLVM: An infrastructure for multi-stage optimization. Tech. rep., University of Illinois at Urbana-Champaign, 2002.

[10] Liu, J. W.-S., Shih, W.-K., Lin, K.-J., Bettati, R., and Chung, J.-Y. Imprecise computations. *Proceedings of the IEEE 82*, 1 (January 1994), 83–94.

[11] Mancina, A., Lipari, G., Herder, J. N., Gras, B., and Tanenbaum, A. S. Enhancing a dependable multiserver operating system with temporal protection via resource reservations. In *Proceedings of the Conference on Real-Time Networks and Systems* (October 2008).

[12] Melhem, R., Mossé, D., and Elnozahy, E. The interplay of power management and fault recovery in real-time systems. *IEEE Transactions on Computers 53*, 3 (February 2004), 217–231.

[13] Pattabiraman, K., Grover, V., and Zorn, B. G. Samurai: Protecting critical data in unsafe languages. In *Proceedings of the ACM SIGOPS European Conference on Computer Systems* (Mar./Apr. 2008), pp. 219–232.

[14] Qin, F., Tucek, J., Sundaresan, J., and Zhou, Y. Rx: Treating bugs as allergies – a safe method to survive software failures. *ACM Transactions on Computer Systems 27*, 3 (August 2007), Article 7.

[15] Rashid, L., Pattabiraman, K., and Gopalakrishnan, S. Modeling the propagation of intermittent hardware faults in programs. In *Proceedings of the IEEE Pacific Rim International Symposium on Dependable Computing* (December 2010), pp. 19–26.

[16] Rashid, L., Pattabiraman, K., and Gopalakrishnan, S. Intermittent hardware errors: Modeling and Evaluation. In *Proceedings of the International Conference on Quantitative Evaluation of Systems* (2012), pp. 220–229.

[17] Reis, G. A., Chang, J., and August, D. I. Automatic instruction-level software-only recovery methods. *IEEE Micro 27*, 1 (January 2007), 36–47.

[18] Rinard, M., Cadar, C., Dumitran, D., Roy, D. M., Leu, T., and Jr., W. S. B. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of USENIX Conference on Operating Systems Design and Implementation* (December 2004), pp. 303–316.

[19] SPEC. 252.eon. http://www.spec.org/cpu2000/CINT2000/, October 1999.

[20] Swift, B. P. User mode scheduling in Minix 3. Tech. rep., Vrije University, October 2010.

[21] Tanenbaum, A. S. Minix 3. http://www.minix3.org/, 2010.

[22] Tanenbaum, A. S., and Woodhull, A. S. *Operating Systems Design and Implementation*, 3 ed. Prentice Hall Software Series. Prentice Hall, 2006.

[23] Zandbergen, B. A more real-time Minix 3. http://www.rtminix3.org/, October 2009.

[24] Zilberstein, S. Using anytime algorithms in intelligent systems. *AI Magazine 17*, 3 (March 1996), 73–83.

[25] Zilberstein, S., and Russell, S. J. Aproximate reasoning using anytime algorithms. In *Imprecise and approximate computation*, S. Natarajan, Ed. Kluwer Academic Publishers, 1995.