Lifetime-Aware Task Mapping for Embedded Chip Multiprocessors

Adam S. Hartman

December 15, 2015

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy in
Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Abstract

The International Technology Roadmap for Semiconductors has identified reliability as a growing challenge for users and designers of all types of integrated circuits. In particular, the occurrence of wearout faults is expected to increase exponentially as manufacturing processes scale below 65nm. By acknowledging the importance of these faults and the resulting failures, designers can take steps to improve the expected lifetime of the system. Several system-level techniques, such as communication architecture design and slack allocation, are capable of mitigating the effects of wearout faults and improving system lifetime. Task mapping optimization is another system-level technique that can be applied at both design time and runtime to enhance system lifetime and has several advantages over other lifetime optimization techniques.

The first advantage that task mapping has over other system-level techniques is that it is more flexible. We show that task mapping can positively impact system lifetime in a number of scenarios and does not rely on redundancy or complex reconfiguration mechanisms, although both of those provide additional benefit. The second advantage of using task mapping to improve system lifetime is a lower cost compared to other techniques. Other lifetime improvement techniques seek to augment systems in a cost-effective way to mitigate the effects of wearout faults while task mapping does not necessarily require additional investment in hardware to achieve similar effects. The final, and perhaps most significant, advantage of task mapping is its ability to dynamically manage lifetime as the system is running. While decisions made by other system- and circuit-level techniques must be finalized before the system is manufactured, the task mapping can continue to change to account for the actual state of the system in real time.

We propose two distinct task mapping techniques to be used at the two different times during which optimization can occur. At design time, we take advantage of abundant computational re-

sources to perform an intelligent search of the initial task mapping solution space using ant colony optimization. At runtime, we leverage information from hardware sensors to quickly select good task mappings using a meta-heuristic. Our two techniques can be used together or in isolation depending on the use case and design requirements of the system.

This thesis makes the following intellectual contributions:

- Lifetime-aware design-time task mapping - Ours is the first approach to search for initial task mappings that directly optimize system lifetime rather than optimizing other metrics which only influence system lifetime, like temperature and power. Because this technique is meant for use at design time, we employ a powerful search algorithm called ant colony optimization, which takes advantage of a designer's computational resources to find a near-optimal task mapping. Our lifetime-aware design-time task mapping improves system lifetime by an average of 32.3% compared to a lifetime-agnostic approach across a range of real-world benchmarks.

- Lifetime-aware runtime task mapping - Ours is the first approach to dynamically manage the lifetime of embedded chip multiprocessors at runtime through the use of task mapping. By leveraging data from hardware sensors and information about the system state, our meta-heuristic approach is able to find high-quality task mappings which extend system lifetime without performing a costly search of the solution space. Our lifetime-aware runtime task mapping improves system lifetime by an average of 7.1% compared to a runtime temperature-aware task mapping approach, and in the best case, system lifetime was improved by 17.4%. Our approach also improved the amount of time until the first component failure by 14.6% on average and 33.9% in the best case.

- Evaluation of lifetime-aware task mapping - We measure the improvement in system lifetime resulting from our task mapping techniques across a range of benchmarks. We also compare our lifetime-aware techniques to others which attempt to indirectly optimize system lifetime to show that direct optimization is the only way to achieve maximum lifetime. For example, we show that task mappings that are near optimal in terms of average initial component temperature can result in a range of system lifetimes that is up to 53.2% of the optimal lifetime; clearly, low temperature does not imply long lifetime.

- Co-optimization of competing lifetime metrics - The wide range of use cases for embedded chip multiprocessors means that different systems will have different design goals. We consider how the pertinent measure of lifetime changes in different use cases, and analyze the degree to which these competing lifetime metrics can be co-optimized.

- Best practices for a system lifetime simulator - We created a simulator which estimates the lifetime of an embedded chip multiprocessor executing one or more applications. The simulator is detailed enough to capture the effects of various system-level design techniques on lifetime, and thus, it is valuable to the field of lifetime optimization research even outside the context of task mapping.

In summary, lifetime optimization for embedded chip multiprocessors is required so that cutting-edge manufacturing processes can continue to be used for a wide range of systems. Our research mitigates the problem of increasingly common wearout faults by proposing and evaluating a pair of design- and runtime task mapping techniques that enhance system lifetime across a broad range of use cases.

Acknowledgments

My deepest thanks go to Donald E. Thomas, my academic advisor. Working with Don has been very rewarding, and the things I've learned from him about how to validate ideas and communicate them to others are invaluable. This thesis would not have been possible without his experience, effort, and patience.

Next, I would like to thank the members of my thesis committee: Diana Marculescu, Shawn Blanton, and Jim Holt. The thoughts and suggestions that they shared helped me to polish the ideas I originally shared at my thesis proposal to create this final product.

Thank you to my other reviewer, Nikil Mehta, who provided a number of insightful comments that helped me to improve the quality of this thesis.

I would also like to thank Brett Meyer for his collaboration, friendship, and for being an outstanding sounding board. Working with Brett was motivating and enlightening, and he never failed to provide a thoughtful response to any of the questions I asked him.

There are a number of other former and current graduate students from Carnegie Mellon University who deserve my thanks. Thank you to Peter Milder, Peter Klemperer, Kristen Dorsey, and Gabe Weisz for their daily company at lunch and coffee, which always provided interesting and enjoyable conversation. Further thanks go to the Electrical and Computer Engineering softball team, The Gigahurtz, who I had the pleasure of "managing" for five years. The fact that my past teammates are scattered across the country pursuing a wide range of professions speaks volumes about the quality of the people I was fortunate enough to interact with.

Of all the people I met in Pittsburgh, Caitlin Moyer is the most important. When I went to Pittsburgh to attend graduate school, I never imagined I would end up leaving with a best friend. Now my wife, Caitlin was responsible for keeping me sane as we navigated graduate school together, and

for providing unwavering motivation when I sometimes lacked it. Thank you Caitlin, I could not have done this without you.

Finally, I would like to thank my family, and specifically my parents, for their support. My parents, Howard and Cheryl Hartman, have always supported my interests, and that is one of the biggest reasons that I find myself where I am today.

The work in this thesis was supported by the Semiconductor Research Corporation through contract 2008-HJ-1795 and by the National Science Foundation through grant CCF-1116856.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Continually shrinking transistor sizes allow integrated circuits to increase in performance and capability while area and power costs are reduced. Both designers and users have relied on these manufacturing process improvements for over 50 years to increase the functionality and usability of new integrated circuits. More recently, the level of integration has risen to a point where multiple processors, memory, their interconnect, and other peripherials can all be implemented in a single package. These packages are small enough in size that they can be used in a vast number of applications. So called embedded chip multiprocessors are integral to cellular phones, cars, home appliances, communications infrastructure, and numerous other areas of technology that we interact with on a daily basis.

However, there are several drawbacks to process improvements, and degradation of system lifetime is one of the most significant. *System lifetime* can be measured as the amount of time between when the system is powered up for the first time and when the system is no longer able to execute its intended application(s). If all other variables are held constant, a system implemented in a process with a smaller geometry will fail sooner than the same system implemented in a process with a larger geometry. The negative effect of decreases in process geometry size on system lifetime is significant enough that the International Technology Roadmap for Semiconductors has specifically identified it as a problem [1].

The cause of system lifetime degradation is well known. As a system operates, the physical

properties of its transistors and wires will deteriorate until some part of the system is no longer able to meet its original timing constraints. A system's functionality can be reduced to an unacceptable level or become wholly incorrect once a timing constraint is violated. When a system fails due to this phenomenon, it is said to have experienced a *wearout fault*. The rate at which wearout faults occur is dependent on manufacturing process parameters, system usage, and operating conditions. Prior research has shown that the occurence of wearout faults increases exponentially as process geometry decreases past 65nm [2]. As state-of-the-art foundries push their manufacturing processes beyond 14nm, the industry is squarely in the position where wearout faults cannot be ignored.

Both producers and consumers of embedded chip multiprocessors will be affected if nothing is done to mitigate wearout faults. At the very least, a wearout fault will cause an interruption in whatever activity the system is performing, and the system will have to be replaced at some cost. There are also scenarios in which the system cannot be replaced, and so system lifetime must meet established goals for a particular product to be usable at all. The perceived quality of the product can be lowered or guarantees made to end users or system integrators may be violated if the system fails sooner than expected.

While the problem of system lifetime degradation has been acknowledged by many, the existing techniques for addressing it have disadvantages. Wearout faults can be mitigated during system-level design by adding redundant or over-provisioned computation, memory, and interconnect resources. Such techniques require the designer to make predictions about how the system will age in order to allocate additional resources in a cost effective manner. Another issue with these techniques is that they increase the complexity of the system, which usually results in increased design and verification times. Additional resources may also increase the cost of the system beyond the minimum required to implement the desired functionality.

As an alternative to adding resources at the system level, guardband can be added during circuit-level design in order to mitigate wearout faults. Adding guardband involves over-engineering specific circuit features (e.g, widening wires beyond their minimum dimensions) in an effort to improve lifetime. The amounts and types of guardband to be added can be prescribed by the foundry based on an internal evaluation of its manufacturing process. Usually, the foundry requires that this guardband be implemented across the entire design for lifetime guarantees to be made, meaning that cost will increase even in locations of the system which do not require additional protection against

wearout faults. Perhaps the largest disadvantage of this technique, and also the system-level design technique, is that the runtime characteristics of the system usually cannot be taken into account. As mentioned earlier, system usage impacts the rate of wearout faults, and the benefits of these design-time techniques will be limited by the accuracy of their predictions of system usage. While some systems may be used in a tightly controlled manner, many modern embedded chip multiprocessors experience widely varying workloads in an array of operating conditions.

1.2 Thesis Overview

This thesis addresses the problems described in the previous section by proposing and evaluating two techniques for mitigating wearout faults with lifetime-aware task mapping. Task mapping affects system lifetime since it directly controls the power dissipation of each component in the system, which in turn impacts the temperature of the components and the rate at which wearout faults occur. Our solution is comprised of a design-time technique for finding an initial task mapping and a runtime technique for changing the task mapping as the system operates. The individual techniques are constructed to take advantage of the different types of system information that are available at design time and runtime. Similarly, the design of the techniques accounts for the different sets of computational resources that are available to compute a task mapping at design time and runtime. The main advantages that task mapping has over system- and circuit-level design techniques are that it can be altered at runtime to adapt to variations in the system or workload, and it does not impact the cost of the system as severely.

Figure 1.1 shows a qualitative view of how the framing of the task mapping problem changes over time. The x-axis represents time, the red line (right y-axis) represents the amount of known system information, and the blue line (left y-axis) represents the amount of resources available for task mapping computation. A designer may have information about the application(s) to be run on the system at design time, but only sensors in an actual system can provide detailed feedback about how components are accumulating wear. Since the amounts of wear which cause component failure can differ from chip to chip, the quality of the information about the system increases drastically from design time to runtime. What the designer lacks in information at design time can be made up for in computational resources, since it is not unreasonable to assume that a large collection of

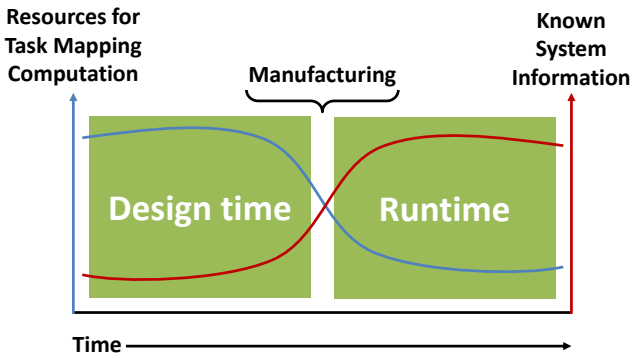

Figure 1.1: Qualitative evaluation of resources for task mapping computation and system information over time

servers would be available for design space exploration. In contrast, a running system only has its own computational resources to rely on for configuring itself, and in the case of an embedded chip multiprocessor, those resources may be meager. Manufacturing is shown in the figure as the step between design time and runtime for completeness, but an exploration of manufacturing techniques which improve lifetime is outside the scope of this thesis.

The first element of our solution is a technique for design-time task mapping. Since the task mapping solution space is too large to explore exhaustively, an optimization algorithm is required to approximate the best solution. We use ant colony optimization (ACO) in this context to search for the initial task mapping which results in the longest system lifetime. ACO is a powerful search strategy in which the problem is represented as a graph that is traversed by simulated ants to build candidate solutions. Solution quality is iteratively improved through communication between past and future ants by annotating the graph in a way that echoes the chemical pheromones real ants use to mark paths to food sources. ACO takes advantage of the non-trivial amounts of time and computation resources that are available to an engineer designing a system to find a near-optimal solution for an initial task mapping.

As a system runs, it is likely that the task mapping should be updated to account for variations in the workload and the system itself. While there are less significant resources available to compute a near-optimal task mapping than at design time, there exists a wealth of information about the system that isn't available during the search for an initial task mapping. We use this additional information, in the form of values reported from wear and temperature sensors, to inform a customized meta-heuristic which periodically changes the task mapping at runtime. This meta-heuristic is the second

element of our solution for mitigating wearout faults and picks up where ACO left off to continue optimizing system lifetime after manufacturing.

We use a system lifetime simulator to evaluate our techniques and compare them to alternative approaches. System lifetime is estimated using statistical models of common types of wearout faults in conjunction with a Monte Carlo simulation. Monte Carlo simulation is required because the specific type of statistical models being used, while accurate, preclude an analytic solution. The simulation compounds the already difficult task mapping problem to create a large solution space where the evaluation of a single solution is computationally expensive. The statistical models are driven by manufacturing process parameters and runtime information about the system being simulated, such as temperature. Arbitrary hardware architectures and streaming application task graphs can be described to the simulator for testing with all relevant task mapping techniques.

This thesis includes comparisons of our proposed techniques to competing approaches. Due to the strong dependence of wearout faults on temperature, it has been suggested that systems which are optimized for temperature will also be optimized for lifetime. We provide a detailed analysis of how well temperature-aware task mapping optimizes for lifetime and vice-versa. Because power dissipation directly impacts temperature, a similar argument can be made about power-aware task mapping being able to optimize lifetime. Our results show that using temperature and power as proxy optimization targets for system lifetime results in task mappings that lead to sub-optimal system lifetime when compared to pure lifetime-aware techniques like the ones we propose.

Lifetime is important in a broad range of applications, including those where system complexity needs to be minimized, and the relevant definition of lifetime can change in these cases. Instead of defining lifetime as the amount of time until the system no longer has sufficient resources to execute its application(s), lifetime may mean the amount of time until a single component in the system fails. This thesis also explores the extent to which these two definitions of lifetime can be co-optimized in the context of our proposed solution. By showing that our techniques positively affect lifetime in these situations, we increase the number of use cases in which they are applicable.

1.3 Contributions

A summary of the contributions of this thesis is as follows:

- a lifetime-aware design-time task mapping technique based on ACO that takes advantage of computational resources to find near-optimal initial task mappings,
- a lifetime-aware runtime task mapping technique using data from wear and temperature sensors to dynamically manage system lifetime,
- an evaluation of lifetime-aware task mapping using a system lifetime simulator with comparisons to techniques that attempt to optimize lifetime indirectly,
- an exploration of how various definitions of lifetime can be co-optimized in complexity-constrained systems using lifetime-aware task mapping, and
- a detailed simulator that uses Monte Carlo simulation to estimate how design decisions, task mapping in particular, affect the rate of wearout faults in components and impact overall system lifetime.

1.4 Organization

The remainder of this thesis is structured as follows. Chapter 2 discusses background information about the relationships between lifetime, task mapping, temperature, and power along with theory about the benefits of lifetime-aware task mapping. Next, Chapter 3 explains the system lifetime simulator that is used to evaluate our proposed techniques. Then, Chapter 4 details the use of ACO to search for near-optimal initial task mappings. Chapter 5 presents our runtime task mapping meta-heuristic for dynamically managing system lifetime. Chapter 6 explores the co-optimization of different lifetime goals using lifetime-aware task mapping. Chapter 7 provides a discussion of related work and Chapter 8 offers directions for future work and our conclusions.

Chapter 2

Task Mapping and System Lifetime

Before describing the details of our design-time and runtime task mapping approaches, it is important to understand the theory about why different task mappings can result in different system lifetime. While it may be clear to the reader how adding hardware redundancy to a system architecture or adding guardband during circuit-level design can increase the expected lifetime of a system, the mechanisms through which task mapping affects system lifetime are not as straightforward. The degree to which task mapping affects system lifetime is dependent on many things, including when the task mapping is changed, for what metric the task mapping is optimized, the structure of the system, and even the particular definition of lifetime.

Systems structured as embedded chip multiprocessors are are becoming increasingly prevalent due to the broad range of features they support and the fact that numerous configurations are readily available from intellectual property vendors. A product designer can leverage an existing embedded chip multiprocessor to perform several functions rather than designing an ASIC from scratch or integrating a set of existing ASIC designs. While a fully custom hardware design may have performance and power advantages over an embedded chip multiprocessor, the use of an embedded chip multiprocessor allows for significantly reduced design costs and a faster time-to-market for the product. In fact, the cost and time savings realized when using an embedded chip multiprocessor are so large they they enable some products to be created that would not be economically feasible otherwise. Cellular phones, tablet computers, smart televisions, digital media streaming devices, and automobile "infotainment" systems are examples of high volume products which are built around embedded chip multiprocessors.

As with any other integrated circuit being manufactured using a cutting-edge process, lifetime is a concern for embedded chip multiprocessors. Because one of the major advantages of using an embedded chip multiprocessor is their low cost, methods of improving lifetime that have little or no impact on cost are ideal. With that requirement in mind, our work focuses on using task mapping as a way to mitigate wearout faults because it avoids the increased costs associated with over-provisioning at the system and circuit levels. Task mapping controls the amount of work being done by each component and is directly responsible for the power dissipation of each component. Component power dissipation has a strong effect on component temperature, and component temperature has a similarly strong effect on how quickly a component accumulates wear. Thus, task mapping affects the times at which the components in the system fail and, consequently, the overall lifetime of the system. We assert that careful manipulation of the task mapping can significantly improve a system's lifetime.

This chapter begins with an overview of the type of system targeted by our task mapping approaches; namely, embedded chip multiprocessors. Following our definition of an embedded chip multiprocessor, we give a formal definition of the task mapping problem and its complexity. Next, we explain the different ways in which the lifetime of a system can be measured and the concept of slack. The definitions of system lifetime and slack are then used to make a high-level comparison of two general approaches to the task mapping problem. Finally, we discuss the advantages of lifetime-aware task mapping over other task mapping approaches which can theoretically improve system lifetime indirectly.

2.1 System Overview

Embedded chip multiprocessors are single-package systems that contain a collection of individual processors that can communicate with each other directly or through shared memory. While the "embedded" moniker has a broad range of connotations, we mean for it to imply that the system is responsible for executing a pre-defined set of applications and is comprised of processors and memories which tend to be more focused on power-efficiency than performance when compared to the corresponding pieces of a desktop computer. We define a system *component* to be any of the individual processors or memories in the embedded chip multiprocessor. Various mixes of

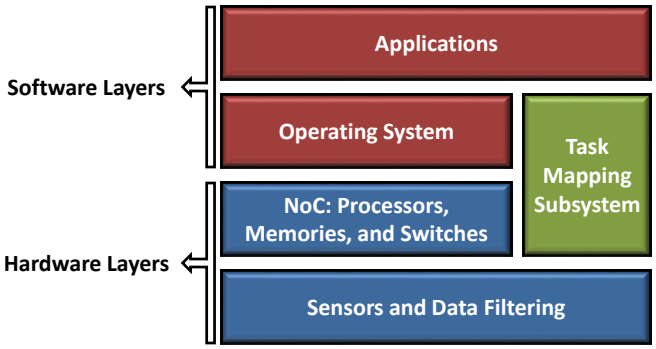

Figure 2.1: NoC-based embedded chip multiprocessor system, layered view

components with different tradeoffs between performance, power, and cost can be used to optimize the embedded chip multiprocessor for a specific set of applications.

There are several design paradigms that can be used to organize an embedded chip multiprocessor at the top level and allow communication between the components. Historically, these types of systems have been bus-based. In this type of architecture, all components in the system are connected to a central bus, which allows for any pair of components to communicate directly with each other. The central bus in this type of architecture has the following drawbacks: it can be a performance bottleneck, it can act as a single point of failure, it can consume large amounts of power, and it can have a high area overhead resulting in increased system cost.

Network-on-chip (NoC) architectures have been introduced as an improvement over bus-based architectures [3]. Components in an NoC send data to each other over a communication architecture that is built with a set of network switches connected in a particular topology. A component can place data on the communication architecture through one of the network switches, and the network switches then forward the data such that it arrives at the desired destination component. NoC architectures improve upon all of the drawbacks of bus-based architectured listed above. NoCs allow for higher performance through a more flexible communication scheme, are more resilient because they can be structured to avoid having a single point of failure, and are more power- and area-efficient because they do not necessarily provide a direct connection between every pair of components in the system. Due to these benefits and industry trends toward the use of these architectures, our work focuses on embedded chip multiprocessors that are implemented as NoCs. Thus, we also consider the network switches that make up the NoC communication architecture to be system components.

Figure 2.1 shows how a generalized task mapping subsystem fits into the overall design of

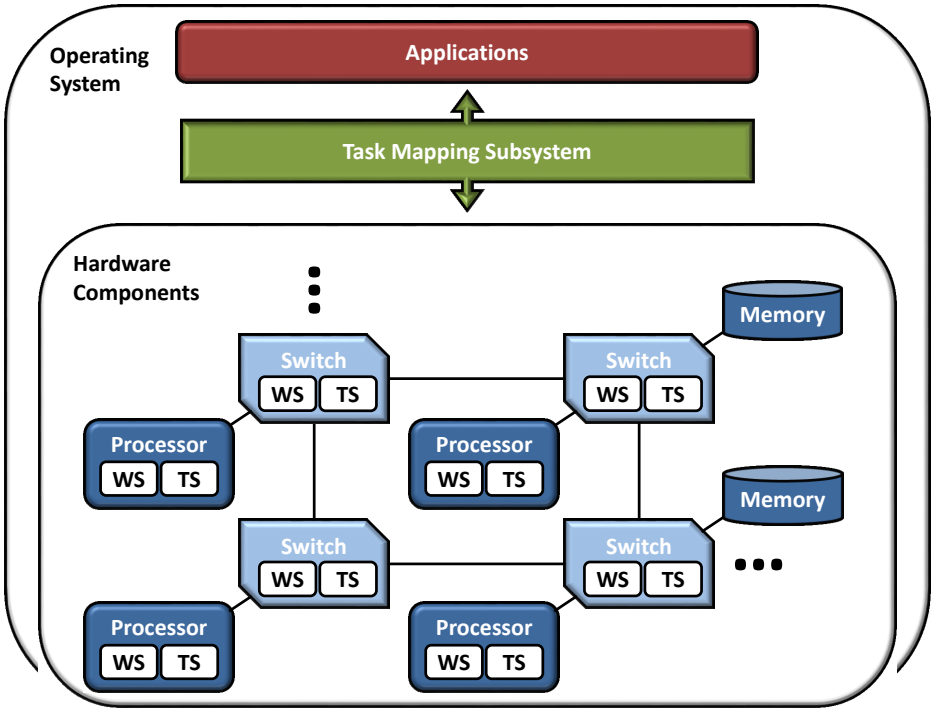

Figure 2.2: NoC-based embedded chip multiprocessor system, topological view

an NoC-based embedded chip multiprocessor. The top half of the figure shows software layers (red) while the bottom half of the figure shows hardware layers (blue), and parts of the system which communicate with each other are adjacent. Software layers consist of the applications being executed on the hardware and the operating system which controls the applications' access to the hardware. The upper hardware layer contains the a set of execution resources that are collected in an NoC framework and exposed to the operating system. The lower hardware layer is made up of the temperature and wear sensors inside each of the execution resources and hardware that filters the data from those sensors prior to its output. The task mapping subsystem (green) spans both hardware and software layers since it can be implemented in hardware, software, or a combination of the two. The task mapping subsystem is responsible for collecting information about the applications, through the operating system, and about the hardware, through the sensors, to create a mapping of software tasks to execution resources. Each of the proposed approaches for lifetime-aware task mapping described in Chapters 4 and 5 represents a potential implementation for the task mapping subsystem.

Figure 2.2 shows a topological view of the system described in Figure 2.1. The dark blue boxes

represent processors, the dark blue cylinders represent memories, and the light blue polygons represent switches. Though Figure 2.2 shows four processors and two memories connected by a simple mesh network, our methods are compatible with arbitrary network sizes and topologies. We assume that each processor and each switch has a wear sensor similar to the delay-based design for measuring time-dependent dielectric breakdown in [4]; these are depicted as white boxes labeled "WS" inside those components. Because memories can be protected from failure inexpensively by adding row/column redundancy and a self-repair circuit [5], our work assumes that they cannot fail, and so they have no need for wear sensors. All processors and switches are also fitted with temperature sensors which are represented by the white boxes labeled "TS" inside those components. The wear and temperature sensors are responsible for providing real-time information to the run-time task mapping meta-heuristic described in Chapter 5. Similar to Figure 2.1, Figure 2.2 does not define where the task mapping subsystem is implemented, although it does imply communication with both hardware and software.

2.2 Task Mapping Problem Definition and Complexity

In order for the systems described in the previous section to do any useful work, one or more software applications must be mapped to the processors and memories in the system. Finding solutions to this problem that have a positive impact on system lifetime is the primary purpose of the work presented in this thesis. Once each component has a set of tasks assigned to it, those tasks must be scheduled. We consider task scheduling to be the process of choosing the order in which the set of tasks mapped to a component is executed such that performance constraints are satisfied. Because task scheduling optimization would be applied in addition to task mapping optimization and not in place of it, we assert that the problem is orthogonal to task mapping, and thus, outside the scope of this thesis.

We assume an application is described via a directed graph where each node represents an individual processing or memory task and each edge represents communication between two nodes. The remainder of this thesis refers to these types of graphs as *task graphs*. Processing tasks are synonymous with computational kernels and are annotated with the amount of computational resources, measured in millions of instructions per second (MIPS), they need to meet performance

requirements. Memory tasks correspond to data arrays and are annotated with the amount of space, measured in kilobytes (KB), that they will occupy in a physical memory. Edges in the task graph are annotated with the rate, measured in KB/s, at which data is transferred between a source task and a destination task.

A *task mapping* is an assignment of the tasks in a task graph to the components in a system. Given a set of *n* tasks, Equation 2.1 shows that each task has a requirement that is a positive integer. The requirements for both processing and memory tasks, measured in either MIPS or KB, are represented in this single array. Each task requirement is a constant value (i.e, it is not dependent on input data), which means that our formulation only considers streaming applications. Each task also has a type, where type "0" indicates a processing task and type "1" indicates a memory task, as shown by Equation 2.2.

$$req_i \in \mathbb{N} \quad \text{for} \quad i \in \mathbb{N}_{\leq n} \tag{2.1}$$

$$tType_i \in \{0, 1\} \quad \text{for} \quad i \in \mathbb{N}_{\leq n} \tag{2.2}$$

A set of *m* components is represented in a similar way. Equation 2.3 defines a capacity, measured in MIPS for processors and KB for memories, for each component. The type of each component is defined in Equation 2.4 where the 0/1 convention matches that for tasks.

$$cap_j \in \mathbb{N} \quad \text{for} \quad j \in \mathbb{N}_{\leq m} \tag{2.3}$$

$$cType_j \in \{0, 1\} \quad \text{for} \quad j \in \mathbb{N}_{\leq m} \tag{2.4}$$

A task mapping which maps a set of *n* tasks to a set of *m* components can be described as a matrix in which there is a row for each task and a column for each component. An entry in the matrix will be "1" if the task represented by that entry's row is mapped to the component represented by that entry's column. All other entries in the matrix will be "0". This formulation of a task mapping is shown in Equation 2.5.

$$TM_{ij} \in \{0, 1\} \quad \text{for} \quad i \in \mathbb{N}_{\leq n}, j \in \mathbb{N}_{\leq m} \tag{2.5}$$

For a task mapping to be considered valid, it must satisfy four constraints. The first two constraints ensure that the task mapping is functionally correct, and the second two constraints ensure

that the task mapping meets the performance requirements of the system. We do not require that a task mapping meet any constraints, such as temperature or power, other than the four described below.

First, Equation 2.6 requires that the task mapping be one-to-one, which means that a task may not be split among multiple components, but each component may have multiple tasks mapped to it.

$$\sum_{j=1}^{m} TM_{ij} = 1 \quad \text{for} \quad i \in \mathbb{N}_{\leq n} \tag{2.6}$$

Second, if a task is mapped to a component, then the type of the task must match the type of the component, and this is shown by Equation 2.7.

$$\sum_{j=1}^{m} TM_{ij}(tType_i + cType_j) \in \{0, 2\} \quad \text{for} \quad i \in \mathbb{N}_{\leq n} \tag{2.7}$$

Third, Equation 2.8 says that the sum of the requirements of all tasks mapped to a component cannot exceed the capacity of that component.

$$\sum_{i=1}^{n} TM_{ij} req_i \leq cap_j \quad \text{for} \quad j \in \mathbb{N}_{\leq m} \tag{2.8}$$

Fourth, communication between the tasks must be routed between their host components through the system such that none of the bandwidth capacities of the physical links between components are exceeded. The routing algorithms we describe in Section 3.6 attempt to avoid congestion in the NoC communication fabric when searching for a solution. However, it is possible that we may not be able to find a corresponding routing for a particular task mapping or that a valid routing simply does not exist for some task mappings. If any of the four constraints is violated, the task mapping is considered invalid, and the system will not function when such a task mapping is applied.

Finding a task mapping that is optimal for a given metric (e.g, system lifetime) is an instance of the generalized assignment problem. The generalized assignment problem is a traditional optimization problem that has been shown to be NP-hard, which effectively means that there is no efficient way to locate the optimal solution and that an exhaustive search of the solution space is not feasible for non-trivial problem sizes. The number of ways to map the application(s) to the system increases very quickly with the number of tasks in the application(s) and the number of components in the

system. Further, it is computationally expensive to evalute a task mapping with respect to system lifetime as will be shown in Chapter 3.

$$O(|taskMappings|) = procs^{procTasks} * mems^{memTasks} \tag{2.9}$$

A bound on the upper limit of the number of task mappings is given by Equation 2.9. In Equation 2.9, *procs* and *mems* represent the number of processors and memories in the system, respectively. The variables *procTasks* and *memTasks* represent the total number of processing tasks and data arrays in the application(s) being mapped to the target system, respectively. Thus, the first term in Equation 2.9 gives the number of ways that the processing tasks can be mapped to processors while the second term gives the number of ways that the data arrays can be mapped to memories. The product of these two terms bounds the number of task mappings for the system as a whole because processing tasks are never mapped to memories and vice-versa. However, not all of the task mappings as counted by Equation 2.9, will be feasible; some of these mappings will violate one or more of the constraints described above. As benchmarks are introduced later in this thesis, we will refer back to this equation and show concrete examples of just how quickly the solution space grows with problem size.

2.3 Definitions of System Lifetime and Slack

This thesis is primarily concerned with finding task mappings which optimize the system lifetime of the NoC-based embedded chip multiprocessors described in Section 2.1. System lifetime is a general term, and it can be interpreted in several ways depending on the context for which the system has been designed. Two common interpretations of system lifetime, and the ones which are pertinent to this thesis, are *total system lifetime* (t_{sys}) and *time to first failure* (t_{first}). Definitions of t_{sys} and t_{first} are as follows:

- t_{sys}– the amount of time between when a system is powered on for the first time and when the system is no longer able to execute its intended application(s) due to component failures.
- t_{first}– the amount of time between when a system is powered on for the first time and when the system experiences its first component failure.

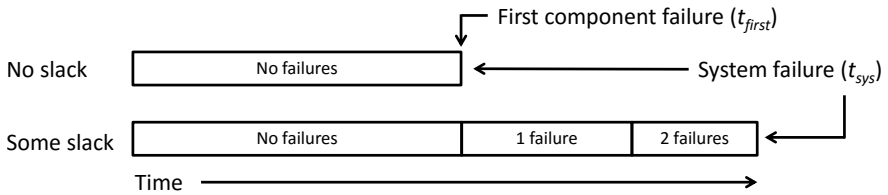

Figure 2.3: Graphical representation of t_{sys} and t_{first}

Both t_{sys} and t_{first} are typically measured in terms of years. Chapter 6 provides a more detailed discussion of the different scenarios in which t_{sys} and t_{first} are important and why a designer may favor one over the other.

Another principle that requires definition to understand the work presented here is slack. *Slack* is defined as the amount of resources in the system, whether computational or memory, in excess of the minimum amount of resources required to execute the application(s) intended for the system. Computational slack is measured in MIPS and memoy slack is measured in KB. For example, a system is said to have 100 MIPS of computational slack if the combined performance of all processors in the system is 300 MIPS and the applications running on the system require a total of 200 MIPS to meet their performance requirements. Given enough slack, a system may be able to continue to execute its intended application(s) even after one or more components has failed. Existing work [6] has presented a detailed exploration of how slack can be allocated cost-effectively to improve system lifetime. All of the work in this thesis is applicable to systems regardless of how much slack they have, but most of our experiments assume that some slack has been added to the system.

Figure 2.3 shows a graphical representation of t_{sys} and t_{first}. The upper bar shows how t_{sys} and t_{first} are measured in the case of a system with no slack. The left edge of the bar represents the point in time that the system was first powered on, and time advances to the right of the figure. The section of the bar labeled "No failures" shows the amount of time that the system runs without experiencing a component failure. The right edge of this bar denotes when the first component in this system fails, and therefore also represents t_{first} for the system. Since the system has no slack, the loss of the first component means that the system as a whole will no longer have the resources required to execute its application(s). Thus, the right edge of the bar also denotes the point in time at which the system fails, or t_{sys}.

The lower bar in Figure 2.3 shows the same measurements for a system with non-zero slack. In this particular case, the system has enough slack to be able to survive the failure of two components before the remaining components no longer satisfy the performance requirements. Assuming all else is equal, this system runs for the same amount of time without component failures as the system with no slack. The first component fails at the same time it did in the system with no slack, and so the t_{first} measurements for the two systems are equal. However, this system then continues to run for some amount of time after the first component has failed ("1 failure") and for an additional amount of time after the second component has failed ("2 failures"). A third component will eventually fail and cause the system to fail as a whole, and this is shown as the rightmost edge of the lower bar. In summary, the presence of slack can increase the t_{sys} of a system but does not necessarily affect t_{first}.

2.4 Comparing Reactive and Proactive Task Mapping

Another aspect of task mapping which can change the effect it has on system lifetime is the choice about when to compute a new task mapping and apply it to the system. Task mappings can either be computed reactively or proactively depending on the tradeoffs between lifetime and cost being targeted by the designer.

In reactive task mapping, the process of computing a new task mapping is triggered only when a component in the system fails. Because the task mapping is never changed during normal operation in this strategy, there is no performance or downtime penalty incurred due to a change in the task mapping. The need for wear or temperature sensors in a system to inform the task mapping process is obviated when reactive task mapping is employed since the biggest change in the state of the system is the fact that a component failed. Reactive task mapping is the strategy in place for the design time task mapping optimization we describe in Chapter 4.

Proactive task mapping involves the computation of new task mappings at a defined time interval. For example, a new task mapping may be computed and applied to the system every week, every 30 days, every year, etc. The purpose of proactive task mapping is to account for more fine grained changes in system state, such as the accumulation of wear on components, than reactive task mapping is able to consider. One drawback of proactive task mapping is that it may lead to

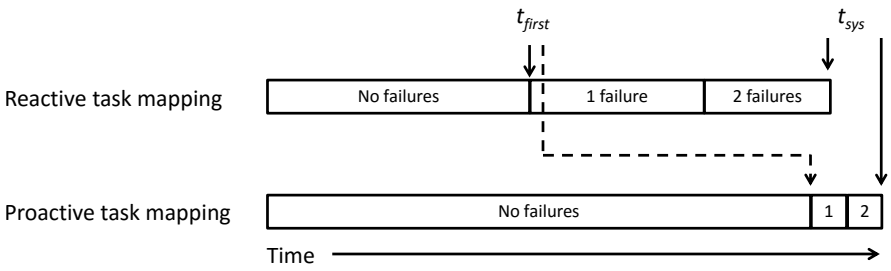

Figure 2.4: Comparison of reactive and proactive task mapping in the presence of slack

increased hardware costs to enable tasks to be moved as the system is running. Also, proactive task mapping may cause the system to be unusable while the task mapping is being changed, but we believe this issue to be negligible in most cases because of the length of time between remappings that we are proposing and the fact that the remapping schedule could be made to take advantage of system idle time. The notion of proactive task mapping is central to our approach for runtime task mapping optimization that is covered in Chapter 5.

Figure 2.4 shows illustrates the effects of proactive task mapping (lower bar) on t_{sys} and t_{first} compared to reactive task mapping (upper bar). In this example, the system has enough slack to surive two component failures. Both bars are divided into three sections which show the amounts of time that the system runs under different numbers of component failures. The most significant change as a result of moving to a proactive task mapping strategy is that t_{first} increases significantly, and the figure depicts this effect as a longer fraction of time spent in the "No failures" state in the lower bar. The reason for this change is that proactive task mapping is able to take advantage of the slack in the system to delay component failures by distributing the wear being accumulated by the system as a whole across all components. Reactive task mapping, on the other hand, uses the slack in the system only to recover from component failures and is unable to change the rate at which the components accumulate wear in between failures. Proactive task mapping can have some positive effect on t_{sys} as depicted by the fact the the lower bar in Figure 2.4 is longer than the upper bar. In theory, proactive task mapping causes all components in a system to accumulate wear almost to the point of failure by the time the first component in the system fails. The system as a whole fails soon after the first component failure since each remaining component is also near failure.

Table 2.1 summarizes the effects of reactive and proactive task mapping on t_{sys} and t_{first} in systems with and without slack. All comparisons in the table are done relative to the case where

Table 2.1: Summary of the effects of task mapping on system lifetime

Slack Amount	0	> 0	
Lifetime Metric	$t_{sys} = t_{first}$	t_{sys}	t_{first}
Reactive Task Mapping	baseline	+	no change
Proactive Task Mapping	+	+	+

reactive task mapping is used in a system with no slack, and so that entry is labeled "baseline". In the case of a system without slack, the entire system will fail as soon as a single component fails, and so t_{sys} and t_{first} will always be the same in these cases. Using proactive task mapping even without any slack present allows t_{sys}/t_{first} to be improved because the task mapping can be adjusted between when the system is first powered on and when the first component fails. When slack is present, reactive task mapping can improve t_{sys} since it will change the task mapping after each component failure, but it has no effect on t_{first} since the task mapping is not changed before the first component failure. Proactive task mapping in the presence of slack can improve both t_{sys} and t_{first} as was shown in Figure 2.4. The trends shown in Table 2.1 are supported by the data gathered from experiments in Chapters 4 and 5.

2.5 Advantages of Lifetime-Aware Task Mapping

At this point, we have shown the different ways in which task mapping can affect system lifetime, but we have not yet addressed the fact that existing techniques may already mitigate the problem of decreasing system lifetime. Temperature is an important metric to minimize in any system because it will reduce the cost and size of the cooling mechanism required to prevent the system from experiencing thermal failure. In embedded systems, temperature is particularly important because the physical space in which the system operates is typically quite small and allows for little or nothing in the way of active cooling. Several pieces of work in the literature propose temperature-aware techniques, including task mapping, to deal with these problems. Because of the strong dependence of the rate of wearout faults on temperature, it is reasonable to suggest that techniques which optimize system temperature will also optimize system lifetime.

The typical goal of temperature-aware task mapping is to improve system reliability by distributing a workload in time and space to minimize either the peak system temperature or the av-

erage system temperature [7]. Other work has discussed temperature-aware thread migration [8] and thread assignment [9]. The authors of [10] present an approach for optimizing the schedule of an embedded application such that system temperature stays within a defined limit. In general, temperature-aware task mappings improve system lifetime, because most, if not all, important failure mechanisms are temperature dependent. As a result, techniques which reduce the temperatures of components in a system also tend to extend component lifetime, and therefore, system lifetime can be extended as well.

However, temperature-aware task mapping fails to capture at least three important classes of factors that also influence system lifetime. The first class of factors involves core properties of the design, such as power density, supply voltage, and circuit geometry. While temperature is primarily related to power density, and many failuire mechanisms are exponentially dependent on temperature, different failure mechanisms are also dependent on a variety of other physical quantities. For example, electromigration is dependent not only on current density, but also circuit geometry. Also, time-dependent dielectric breakdown is dependent on supply voltage, and thermal cycling is dependent on ambient temperature. Further details about these failure mechanisms and how we use them to model system lifetime can be found in Section 3.2. Though many of these parameters change in the same directions as task mappings change because of changes in component utilization (e.g, power density increases as supply voltage or current density increases) they don't change in the same proportion.

The second class of factors deals with system architecture concepts, such as the topology of the communication architecture, the distribution of slack, and the physical floorplan of the system. In systems that allocate execution and storage slack to survive component failure, lifetime is directly related to system architecture. Temperature-aware task mapping is agnostic of which tasks would be the most difficult to remap in the event of a component failure, or alternatively, which components are most important to the longevity of the system. Although there is clearly a relationship between temperature and system architecture via the influence of floorplanning and communication patterns, temperature-aware task mapping does not directly account for these effects. We observe that temperature isn't always a good proxy for lifetime and that temperature-aware task mapping can result in a range of potential lifetimes when optimizing a given temperature metric.

The third class of factors includes those related to manufacutring variability. It is impossible

for all instances of an integrated circuit to be perfectly identical when manufactured on a modern process. Unavoidable die-to-die and chip-to-chip variation mean that each part will have a slightly different set of physical properties, and some of these physical properties will impact how much wear the components in a system can sustain before they fail. But, these changes in physical properties may not lead to measurable differences in temperature, and so the effects of manufacturing variability on system lifetime cannot be captured by temperature-aware optimizations. Instead, task mapping techniques that directly measure component wear and use this information to inform the task mapping decision are required.

Directly optimizing lifetime requires not only that temperatures be generally minimized, but that the inevitable (and at times, advantageous) irregularity in component temperatures be distributed such that the lifetime of important resources is extended at the carefully calculated expense of less important resources. In other words, minimizing a system's peak temperature is important for extending lifetime in general, but system lifetime further depends on the particular component which experiences that peak temperature. Only lifetime-aware task mapping can expose the relationships of both physical parameters and manufacturing variability with component failure, as well as the relationship between component failure and system lifetime, to find task mappings that optimize lifetime. The advantages of lifetime-aware task mapping over temperature-aware task mapping are supported by the experimental data found in Chapters 4 and 5.

2.6 Summary

This chapter summarized many of the foundational concepts for the work in this thesis. We provided definitions of the embedded chip multiprocessors and lifetime metrics that are targeted by our proposed task mapping techniques. We also gave a formal definition of the task mapping problem and its complexity in order to provide context for the choices of algorithms in our proposed approaches. The remainder of this thesis details our approaches for solving the problems defined in this chapter and contains experiments and analyses which serve to illustrate the effects and advantages of lifetime-aware task mapping that are described above.

Chapter 3

System Lifetime Simulator

In order to evaluate the effects of different task mapping techniques, we need a way to measure the lifetime of a given system to which one or more applications are mapped through a particular task mapping technique. Of course, actually building such systems and observing them until they experince a failure would be both cost and time prohibitive, so we have created a system lifetime simulator. The simulator models the possible sequences of wearout faults that a system can experience over the course of its lifetime. Each sequence of wearout faults will result in a different sequence of component failures, which in turn will result in a different system lifetime. We use Monte Carlo simulation to model sufficiently large subsets of wearout fault sequences and estimate an average value for system lifetime.

The system lifetime simulator explained in this chapter is common to all of our experiments in the remainder of this thesis. There is no inherent bias toward a particular task mapping techique in the simulator itself, so it serves as a platform that allows for meaningful comparisons of task mapping approaches relative to each other. Further, the simulator is detailed enough to capture the effects of a broad range of system-level optimization techniques, which speaks for its utility in the field of lifetime optimization research outside the context of this thesis. While the absolute accuracy of the simulator cannot be validated against actual hardware without significant time and resources, we have taken steps to ensure that it produces reasonable results. Prior work has validated the models for wearout faults, temperature, and power that are used in our simulator, and various parts of our simulator have been calibrated such that inputs to those models match published values for similar systems. Given correct models and correct inputs to those models, we assert that our simulator has

sufficient relative accuracy to compare the effects of different task mapping approaches on system lifetime. Additionally, the data presented in the following chapters imply that the simulator is self-consistent in several ways (e.g, a system with more slack tends to have longer lifetime than a system with less slack when all other things are equal).

The remainder of this chapter will explain use the of Monte Carlo simulation, the failure mechanisms that are modeled by the simulator, how the accumulation of wear on components due to those failure mechanisms is tracked, how different task mapping approaches are plugged into the simulator, the technique used to route communication between tasks, and how the simulator can be used in practice. The work described in this chapter was presented in part in [11], [12], and [6].

3.1 Monte Carlo Simulation

Occurrences of the types of wearout faults addressed by our task mapping approaches cannot be directly predicted by any mathematical equation, and instead, are better represented as statistical distributions. Without a direct way of computing the exact points in time at which wearout faults occur for a given system, it is impossible to directly compute the expected lifetime for that system. Thus, we need to use an some indirect method that estimates system lifetime given a set of statistical distributions which describe the wearout faults.

Monte Carlo simulation is a generally accepted method for estimating overall properties of a physical system whose parameters are defined by probability distributions. A Monte Carlo simulation is composed of a number of samples in which each sample represents one particular instance of the system being modeled where the parameters have been set by randomly choosing values according to their statistical distributions. Defining values for the parameters allows some property of the sample system to be measured. Once a sufficient number of samples systems have been evaluated, the measured values of the property of interest from each sample are averaged together to provide an estimate of the actual value of that property.

The remainder of this section describes the assumptions used in building the system lifetime simulator and details the process we use to measure t_{sys} and t_{first} for a sample system.

3.1.1 Assumptions

We must make several assumptions about how components in the system fail and when these failures cause the overall system to fail in order to better define our problem. Individual processors and switches within a system may fail over the course of its lifetime due to wearout faults caused by one of the three failure mechanisms explain in Section 3.2. We do not consider scenarios where the individual memories in a system fail due to wearout faults since architectural techniques are already commonly used to mitigate memory failure. However, a memory may become inaccessible by the rest of the system if the switch to which it was connected fails. While such a memory is still capable of functioning, the fact that no other components can communicate with it means that it is indistinguisable from a memory that failed due to a wearout fault.

We assume that some systems can automatically detect the failure of any component (e.g, using [13]), at which point the operating system signals the task mapping process to begin searching for a solution which does not rely on any failed components. Certain switches may also need to be reconfigured, independently of the task mapping process, when a component fails to avoid attempts to route to failed portions of the communication architecture. While it is feasible to implement these recovery and reconfiguration mechanisms, we recognize that their design and validation cost may not make sense in some applications. For systems which eschew recovery and reconfiguration, our simulator simply halts after the first component failure and reports that time as both t_{sys} and t_{first} for the system. It is important to understand that our task mapping approaches do not rely on the presence of recovery and reconfiguration mechanisms to improve system lifetime. But, our task mapping approaches have increased impact in systems that implement recovery and reconfiguration mechanisms because they allow the system to survive component failures and give task mapping a longer amount of time in which to perform optimization.

In addition to this method of computing task mappings reactively when components fail, our task mapping process can also be triggered at pre-defined time intervals in an effort to proactively address system lifetime. In this thesis, we use reactive task mapping to evaluate our design-time task mapping approach in Chapter 4 and proactive task mapping to evaluate our runtime task mapping approach in Chapter 5. Regardless of when it is invoked, the task mapping process is responsible for remapping tasks and data from failed resources to those with slack and re-routing the affected

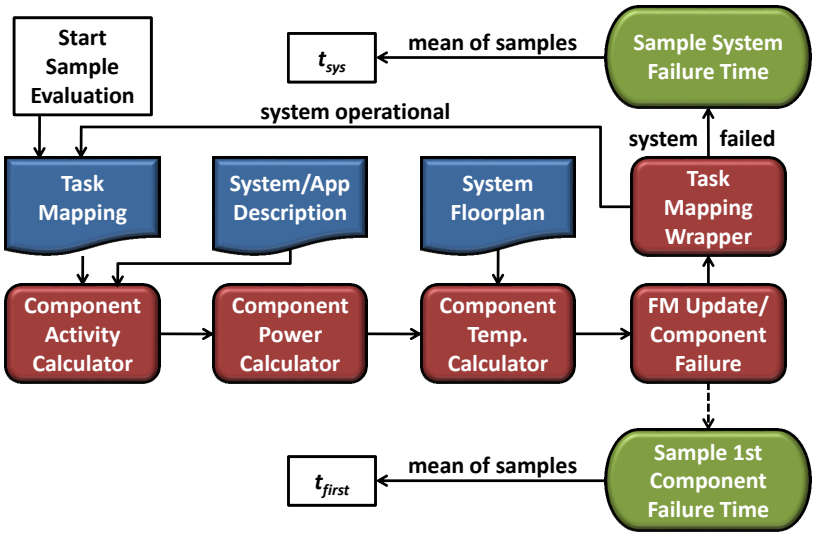

Figure 3.1: System lifetime evaluation process for a single Monte Carlo sample

traffic. We assume that if a valid task mapping exists according to the constraints in Section 2.2, then the system can meet its performance requirements and continue to function. This assumption implies that the operating system is able to create a task scheduling for each component that satisfies the performance requirements of the application(s) given a set of tasks for that component where the sum of the task requirements is less than or equal to the capacity of the component. The problem of finding that task scheduling, and potentially optimizing it for system lifetime, is orthogonal to the work presented in this thesis.

3.1.2 Sample Evaluation

Figure 3.1 gives an overview of our lifetime evaluation process for a single Monte Carlo sample. First, an initial task mapping is applied to the system. If we are performing design-time task mapping optimization, as in Chapter 4, this initial task mapping is one of those being evaluated by the search. When testing our runtime task mapping optimization, as in Chapter 5, then an initial task mapping which minimizes power dissipation is used for a short time (0.01 years). After this short initial period, the proactive task mapping process is automatically triggered to apply the task mapping approach being tested. This initial task mapping is required in this case because task mapping approaches requiring information from wear or temperature sensors would be impossible to compute when the system is first powered on due to the fact that those sensors do not have valid values

at that instant.

Given an initial task mapping, or one that has been computed reactively or proactively, the utilization of each component is calculated based on given information about the system and the application (Component Activity Calculator block in Figure 3.1). Following the notation used in Section 2.2, Equation 3.1 shows how processor and memory utilizations are computed.

$$util_j = \frac{\sum_{i=1}^{n} TM_{ij} req_i}{cap_j} \quad \text{for} \quad j \in \mathbb{N}_{\leq m} \tag{3.1}$$

According to Equation 3.1, utilization is the ratio of the sum of the requirements of the tasks mapped to that component to the capacity of that component for processors and memories. For a switch, utilization is based on the amount of data passing through the switch, and in order to compute this value, all of the communication between the tasks must be routed to the system's communication network. Details about the process we use to route communication between tasks are described in Section 3.6.

Next, component utilization values are used to calculate the amount of power being dissipated by each component (Component Power Calculator block in Figure 3.1). Processor power dissipation is computed by multiplying the utilization fraction for that processor by the maximum power dissipation for that processor type according to manufacturer data sheets. Our library of processors includes the Cortex-M3, ARM9, and ARM11 architectures created by ARM Ltd. [14]. Memory power dissipation is computed by passing the memory's utilization fraction to CACTI, which is a piece of software which can compute various characteristics of user-defined memory configurations [15]. We assume a library of SRAM memories optimized for low standby power with sizes ranging from 64KB to 2MB. Switch power dissipation is computed by sending the data rate being routed by the switch to ORION, a research tool which models power and performance for interconnect networks [16]. We assume that switches can have a crossbar size of 3x3, 4x4, or 5x5 and that they are equivalent to the Alpha 21364's on-chip router. All processors, memories, and switches are assumed to be implemented in a 90nm manufacturing process. Finally, when a processor or switch has zero utilization, we assume that it switches to a low power state where its power dissipation was 1/3 of its maximum power dissipation.

Component power dissipation values can be used to compute steady-state temperatures for each component, but a system floorplan is also required to do this. For each system architecture being tested, we create a floorplan using BloBB [17]. The floorplan and per-component power dissipation data are sent to HotSpot [8], which returns one temperature value for each component in the system (Component Temperature Calculator block in Figure 3.1). HotSpot uses information from the floorplan to create a resistor network which models the ways in which heat can flow in the system. The power dissipation values are applied to this network and the tool can then solve for the temperature of each component. Our temperature modeling assumptions about range and average value are designed to match previously published temperature modeling assumptions for the same types of systems [18]. These component temperatures are then used to shape the failure distribution for each failure mechanism in that component.

Given temperatures for all components, a statistical distribution is created for each failure mechanism in each component. Section 3.2 provides details about how component temperatures are used to compute the statistical distributions that model failure mechanisms. Once the statistical distributions are computed, we initialize or update the failure times for each component in the system (FM Update/Component Failure block in Figure 3.1). When entering this block for the first time in a sample, a failiure time is selected from each statistical distribution, and each component is given a failure time equal to the soonest failure time from all of its failure mechanisms. In subsequent iterations of this block, we update the amount of wear accumulated by each failure mechanism in each component since the last iteration, and this process is detailed in Section 3.3. In addition to their place in the simulation, we use the time to failure and accumulated wear values that are calculated in this step as "outputs" from the wear sensors we assume to exist on chip, and these values are used as input to our wear-based task mapping heuristics in Chapters 5 and 6.

After failure times have been updated, we determine which component in the system has the next earliest failure time. We mark this component as failed and proceed to the task mapping process (Task Mapping Wrapper block in Figure 3.1). A high-level explanation of the task mapping process can be found in Section 3.4. When marking the first failed component in the sample system, we also record the current simulation time as t^i_{first}. If we are able to find a valid task mapping, the "system operational" path is taken, and the simulation loop begins another iteration. If a valid task mapping does not exist, the "system failed" path is taken, and we record the current simulation time as the

failure time for the sample system, t^i_{sys}.

Once a sufficient number of samples have been completed, we compute the sample mean $\overline{t^i_{sys}}$ to estimate t_{sys}. Similarly, the sample mean $\overline{t^i_{first}}$ estimates t_{first}.

3.2 Failure Mechanisms

Each wearout fault that causes a component failure in a system is a result of one of several physical phenomena known as *failure mechanisms*. Failure mechanisms represent the different ways in which the structure of an integrated circuit can break down over time. For the purposes of the experiments in this thesis, we have selected three important failure mechanisms to model: electromigration, time-dependent dielectric breakdown, and thermal cycling. These three failure mechanisms represent the most common causes of wearout faults [19], but several other failure mechanisms exist. Statistical distributions for failure mechanisms are an input to our system lifetime simulator, and as a result, our simulator provides a generalized framework for modeling the effects of failure mechanisms. The generalized framework enables future work to focus on the set of failure mechanisms that is most relevant to the experiments being performed.

The first failure mechanism we chose to model is electromigration (EM), which causes wearout of the metal wires in an integrated circuit over time. As electrons move through metal wires, some of their momentum is transferred to the metal atoms which causes the the atoms themselves to move over time. It is possible for the metal atoms to move and separate in such a way that voids are created in the wire, and these void degrade the performance of the wire from its original specification. If the voids become large enough or cause the wire to separate completely, the wire will not be able to transmit any information, and the component which includes the wire will fail.

Time-dependent dielectric breakdown (TDDB) is the second failure mechanism modeled by our system lifetime simulator. TDDB affects the transistors in an integrated circuit rather than the wire. The presence of electric fields in a transistor eventually causes the dielectric (i.e, the gate oxide) to lose its insulating properties, and the transistor will not function correctly as a result. In any component, a critical set of transistors will fail over time due to TDDB such that the component is unable to function correctly.

The final failure mechanism we choose to model is thermal cycling (TC). While EM and TDDB

are specific to certain parts of an integrated circuit, TC affects the system as a whole. The different substances used to build an integrated circuit can experience different degrees of physical stress under the application of heat due to their varying physical properties. This type of physical stress is highest at the interfaces of different materials, usually between the die and package, and temperature fluctuations can cause these junctions to break. It is especially important that our work models TC because it captures the effects of temperature changes due to infrequent events such as changes in the task mapping and component failures.

Equations 3.3, 3.4, and 3.5 are mathematical definitions of the EM, TDDB, and TC failure mechanisms [20, 21]. The values computed by these equations, $MTTF_{\{EM,TDDB,TC\}}$, are related to the typical mean time to failure (MTTF) of a component due to a particular failure mechanism. More specifically, the result of each equation is used to calculate the mean of a lognormal distribution that represents the probability with which a component will fail due to a given failure mechanism at a given time [19, 22]. Lognormal distributions are used because they model the behavior of wearout faults more accurately than other probability distributions. Monte Carlo simulation is required to estimate t_{sys} and t_{first} from component failure rates because there are no analytic methods to combine a set of lognormal distributions into a single lognormal distribution. Equation 3.2 shows the calculation of the mean of a lognormal distribution, μ, given an MTTF value.

$$\mu = ln(MTTF) - \frac{\sigma^2}{2} \tag{3.2}$$

The term σ in Equation 3.2 represents the standard deviation of the lognormal distribution, for which we use a value of 0.5. Our Monte Carlo simulation samples component failure times directly from these distributions as described in Section 3.1 and they are also used to keep track of component wear across temperature changes as discussed in Section 3.3.

$$MTTF_{EM} = A_{EM}(J)^{-n} e^{\frac{E_{aEM}}{kT}} \tag{3.3}$$

$$MTTF_{TDDB} = A_{TDDB} \left(\frac{1}{V}\right)^{a-bT} e^{\frac{X+Y/T+ZT}{kT}} \tag{3.4}$$

$$MTTF_{TC} = A_{TC}\left(\frac{1}{T - T_{ambient}}\right)^{q} \tag{3.5}$$

The first term in each equation, $A_{\{EM,TDDB,TC\}}$, is a scaling factor used to control the effective strength of each failure mechanism. We solve for values of these scaling factors such that at a constant characterization temperature of 345K, the MTTF of a component due to each failure mechanism is 30 years [23]. The effect of setting the scaling factors in this way is that each of the three failure mechanisms will have an equal effect on component lifetime at the nominal temperature. These equations also imply that the effects of one failure mechanism on a component are independent of all other failure mechanisms.

In all three equations, the term T represents the temperature of a component which is computed using the process described in Section 3.1.2. In Equation 3.3, J represents current density, n is a parameter dependent on manufacturing process, E_{aEM} represents the activation energy of electromigration, and k represents Boltzmann's constant. All of these terms are constant values except for current density, which is dependent on the amount of work being done by the component and the area of the component. In Equation 3.4, V represents the operating voltage of the component, a, b, X, Y, and Z are all constant fitting parameters dependent on the manufacturing process, and k is again Boltzmann's constant. The value of V may change depending on the current power state of the component. In Equation 3.5, $T_{ambient}$ is a constant value representing the temperature of the environment in which the system is operating, and q represents the Coffin-Manson exponent.

3.3 Wear Update Process

After a new task mapping is applied to the system for any reason, it is almost certain that the temperatures of the components will change. Changes in component temperature cause changes in the shapes of the statistical distributions that represent the failure mechanisms as described in Section 3.2. When the statistical distributions change, we need to translate the amount of wear that was accumulated in the previous distribution to the new distribution. Then, using the newly updated amount of wear in the new statistical distribution, we can compute the time at which the component fails due to that failure mechanisms based on the new task mapping.

The statistical distribution for each failure mechanism is, in fact, a probability density func-

tion (PDF). The PDF can be interpreted as a function that describes the probability with which a component will fail versus time. The cumulative distribution function (CDF) derived from a failure mechanism's PDF then represents the amount of wear that has been accumulated due to that failure mechanism versus time. We select a failure time for each failure mechanism in each component using the PDFs, and then we use the CDFs to map the selected time to a critical amount of wear (i.e, the amount of wear at which the component will fail). CDFs can be used to find the amount of time a component spends at one temperature to accumulate a certain amount of wear.

Equations 3.6 and 3.7 precisely define the wear update process. In these equations, $T1$ is the temperature of the component under the previous task mapping and $T2$ is the temperature of the component under the new task mapping. Given these temperatures, we can compute $CDF_{T1}()$, the wear versus time function for the previous temperature, $CDF_{T1}^{-1}()$, the time versus wear function for the previous temperature, and $CDF_{T2}^{-1}()$, the time versus wear function for the new temperature. In Equation 3.6, $curWear$ is a real number between 0 and 1 which represents the amount of wear accumulated due to the failure mechanism prior to this update, $curTime$ represents the current time in the simulator in years, and $lastTime$ represents the last simulation time at which the wear was updated for this failure mechanism. Subtracting $lastTime$ from $currentTime$ results in the amount of time that the component spent at temperature $T1$, and $CDF_{T1}^{-1}(curWear)$ gives the amount of time that would have to be spent at $T1$ to accumulated $curWear$. Adding these two values together tells us the "age" of the component due to a failure mechanism as if it had only ever been operated at temperature $T1$. This age can then be passed to the CDF for $T1$ to compute the amount of wear caused by that failure mechanism; $curWear$ is then updated with this amount.

$$curWear = CDF_{T1}(CDF_{T1}^{-1}(curWear) + (curTime - lastTime)) \tag{3.6}$$

$$timeUntilFailure = CDF_{T2}^{-1}(wearFail) - CDF_{T2}^{-1}(curWear) \tag{3.7}$$

In Equation 3.7, $wearFail$ represents the amount of wear at which the component will fail due to this failure mechanism and is computed as described above. $CDF_{T2}^{-1}(wearFail)$ calculates the time at which the component will fail due to this failure mechanism at the new temperature $T2$. $CDF_{T2}^{-1}(curWear)$ calculates the age of the component at the new temperature given the updated amount of wear from Equation 3.6. The difference of these two values is the amount of time

the component can operate at temperature $T2$ before it will failure due to this failure mechanism, $timeUntilFailure$. After Equations 3.6 and 3.7 are computed, $lastTime$ is set to be equal to $curTime$ to indicate that wear accumulation due to this failure mechanism has been updated. Once a new $timeUntilFailure$ value has been computed for all failure mechanisms in all components, its minimum value across the entire system indicates how much longer the simulation needs to run before the next component fails.

3.4 Task Mapping Process

Regardless of whether a system is using proactive or reactive task mapping, the process starts with a set of common operations. The common operations are encapsulated in a wrapper which provides an interface between the system lifetime simulator and the different task mapping approaches we present in this thesis. The interface simplifies the process of integrating new task mapping approaches into the simulator, and thereby enables the simulator to evaluate a broad range of task mapping approaches in addition to our own.

The common task mapping operations are shown in Figure 3.2, and this figure shows a detailed view of the "Task Mapping Wrapper" block in Figure 3.1. The task mapping process begins by identifying the type of the failed component provided by the "FM Update/Component Failure" block in Figure 3.1. If the failed component is a switch, we assume that all processors and memories connected directly to it become inaccessible and are consequently unusable in a task mapping ("Find Attached Components"). After identifying the set of failed components and removing them from consideration ("Remove Failed Components"), we consider the remainder of the components as candidates for the new task mapping ("Remaining Components"). Our task mapping process considers all tasks in the task graph when it is invoked rather than just the tasks that are orphaned by failed and inaccessible components. Empirical results suggest that mapping all tasks leads to an increase in system lifetimewithout a significant increase in the runtime of the task mapping algorithm.

To short-circuit the task mapping process when possible, we perform two checks to compare the requirements of the application to the remaining resources in the system. The first check sums the requirements of all tasks in the application(s), separated by task type, and compares those total

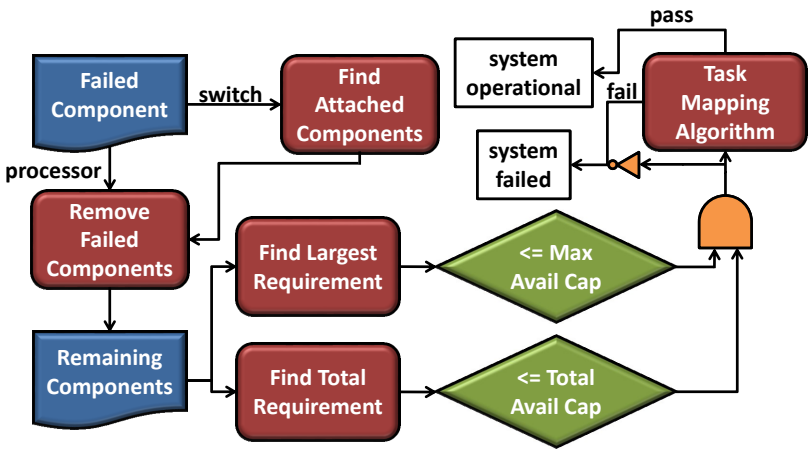

Figure 3.2: Task mapping wrapper process

requirements to the sum of the capacities of all components which have not yet failed ("Find Total Requirement and $\leq$ Total Avail Cap"). The second check finds the largest single task requirement in the application(s) for each task type and compares them to the capacities of the largest remaining components in the system of each type ("Find Largest Requirement and $\leq$ Max Avail Cap"). If either of these simple checks fails, we know that task mapping will be impossible given the current system state, and the sample is ended ("system failed"). If both of the above capacity checks pass, we know that finding a task mapping may be possible and continue on to the actual task mapping algorithm being used for the current experiment. The "system failed" and "system operational" boxes in Figure 3.2 correspond to the paths of the same names in Figure 3.1.

3.5 Communication Groups

In addition to the checks in the task mapping wrapper described in Section 3.4, there is a second technique we use to prevent failures inside the actual task mapping algorithm. Components that have not failed and are available for task mapping can be grouped in a way that reduces the number of invalid task mappings in the solution space. This grouping prevents fragmentation of the task graphs across disjoint sets of components during mapping, which would eventually lead to an invalid task mapping solution. When the system is first powered on and after any switch failures, we analyze the system topology to determine which pairs of switches are connected by at least one routing path. Using this information, we create a list of communication groups. A *communication group* is a

set of components which can communicate with each other but not with any components in other communication groups. To determine if a component should be a candidate for a particular task in a task mapping algorithm, we check whether or not the communication group to which the potential candidate component belongs has enough capacity to accommodate the entire task graph to which the task belongs. Each check that fails generates a new constraint for the task mapping algorithm that disallows the construction of solutions which will eventually become invalid. However, the use of communication groups does not guarantee that a valid task mapping will always be found.

The example shown in Figure 3.3 shows an example in which communication groups prevent the construction of invalid task mappings. The five red squares represent processors, the five blue circles represent switches, and the black circle represents a switch that has failed. Since the switches on the left can no longer communicate with the switches on the right, two communication groups are formed as indicated by the dashed boxes "CG 1" and "CG 2". We assume that a fully connected task graph with three tasks needs to be mapped to this system and that each task completely occupies one processor. When choosing a component for the first task, all five of the processors are candidates based on their available capacity. In the absence of the communication group check, the first task could be mapped to p4, and the task mapping process would eventually be forced to map one of the remaining tasks to one of the processors in communication group 1. In this case, the task mapping process would fail since the task that was mapped to the processor in communication group 1 would not be able to communicate with the task(s) mapped to processors in communication group 2. With the communication group check active, no tasks will be mapped to p4 or p5 since that communication group cannot accommodate the entire task graph. Thus, we prevent situations in which the task mapping process fails due to assigning tasks which must communicate to components which cannot. This communication group check can inherently handle situations in which multiple task graphs must be mapped to a single system.

3.6 Routing Communication Between Tasks

While the nodes in a task graph represent the individual tasks that make up an application and their requirements, the edges define which tasks communicate with each other and the data rates that must be maintained to achieve performance requirements. If two tasks which communicate

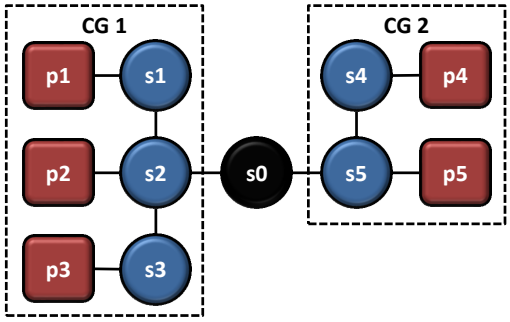

Figure 3.3: Example of communication groups

with each other are mapped to different components in the system, then the switches in the system must be used to transmit the data from the first component to the second. Communication between two tasks that are mapped to the same component does not generate any network traffic because we assume that all tasks mapped to a particular component may freely communicate within that component. Our simulator must determine the exact route used by each pair of communicating tasks in order to compute the utilization of each switch in the system. This traffic routing process is part of the "Component Activity Calculator" block in Figure 3.1.

For each edge in the task graph, we greedily select the route through the system that most closely matches the the requirement of the edge. The routing of an individual edge is performed by Dijkstra's algorithm, a common method for finding the minimum cost path between two nodes in a graph [24]. Dijkstra's algorithm is a variation of a breadth first search in which the graph is explored in the direction of low cost nodes. In the context of our overall routing algorithm, we use Dijkstra's algorithm to find a path between two components in the system where the cost of an edge is the difference between the available capacity of that edge and the requirement of the communication being routed. The capacities of communication links in the system are updated after each task graph edge is routed by subtracting the edge's requirement from the available capacity on each edge used in the routing path. All communication links in the system are assumed to have a maximum bandwidth of 256 MB/s.

Algorithm 1 contains pseudocode for the process we use to route all communication between tasks. The inputs to this algorithm are the set of edges in the task graph, *tgEdges*, the set of edges in the system, *sysEdges*, the set of nodes in the system, *sysNodes*, and the task mapping that is currently applied to the system, *tm*. Lines 1 through 13 show the outer loop of the algorithm which

is responsible for stepping through all edges in the task graph. The call to Dijkstra's algorithm for one edge in the task graph is shown on line 4. Lines 6 through 10 take the path found by Dijkstra's algorithm, *path*, and updates the capacity of each communication link in that path to express that a particular task graph edge has been mapped to it.

An outline of Dijkstra's algorithm is provided by lines 15 through 46 of Algorithm 1. The inputs to Dijkstra's algorithm are the node at which the route is to begin, *sourceNode*, the node at which the route terminates, *destNode*, the edges and nodes in the system graph, and the requirement of the task graph edge being routed, *edgeReq*. Initialization of the data structures used in Dijkstra's algorithm is performed in lines 16 through 25. Here, we set the cost of reaching each node, *cost*, that is not the source node to a large value, and the cost of reaching the source node is 0. All entries in the path to be returned are set to NULL and each node in the system is added to a priority queue, *Q*.

Dijkstra's algorithm then explores the nodes in the priority queue until it reaches the specified destination node. The lowest cost node is removed from the priority queue, and the algorithm stops if the destination node is found (lines 28 through 32). Next, the neighbors of the current lowest cost node are explored to see if they can be reached with a lower cost using a path that goes through the current lowest cost node (lines 34-42). If the cost of reaching a neighbor can be improved by using the current lowest cost node, the cost of the neighbor and the path to the neighbor are updated to reflect the improvement.

If the communication routing process is unable to find a valid path for all edges in the task graph, then the process fails, and this means there is no valid routing for the provided task mapping. Consequently, the task mapping itself is considered invalid according to the constraints in Section 2.2. If a valid routing is found, then a utilization value for each switch in the system can be computed and the lifetime evaluation process can continue to the Component Power Calculator process in Figure 3.1.

3.7 Using the System Lifetime Simulator in Practice

The Monte Carlo-based simulator described in this chapter may require a large number of samples to converge, and this number will increase with the complexity of the system architecture and appli-

Algorithm 1 Communication Routing Algorithm

```
1: function RouteCommunication(tgEdges, sysEdges, sysNodes, tm)
2:     for each e in tgEdges do
3:         if tm[e.source] ≠ tm[e.dest] then
4:             path =Dijkstra(tm[e.source], tm[e.dest], sysEdges, sysNodes, e.req)
5:
6:             tempDest = e.dest
7:             while tempDest ≠ e.source do
8:                 sysEdges[path[tempDest], tempDest].capacity −=e.req
9:                 tempDest = path[tempDest]
10:            end while
11:        end if
12:    end for
13: end function
14:
15: function Dijkstra(sourceNode, destNode, sysEdges, sysNodes, edgeReq)
16:    Q.clear()
17:    for each v in sysNodes do
18:        if v ≠ sourceNode then
19:            cost[v] =MAX_INT
20:        else
21:            cost[v] = 0
22:        end if
23:        path[v] = NULL
24:        Q.pushBack(v)
25:    end for
26:
27:    while !Q.empty() do
28:        u = Q.getMinCostElement()
29:        if u = destNode then
30:            return path
31:        end if
32:        Q.delete(u)
33:
34:        for each v in Q do
35:            if [u, v] ∈ sysEdges && edges[u, v].capacity ≥ req then
36:                newCost = cost[u] + (sysEdges[u, v].capacity−edgeReq)
37:                if newCost < cost[v] then
38:                    cost[v] = newCost
39:                    path[v] = u
40:                end if
41:            end if
42:        end for
43:    end while
44:
45:    return path
46: end function
```

cation(s) being modeled. In addition, the sequence of steps required to compute the sample system lifetime (including several iterations of utilization, routing, power, temperature, statistical distribution, and task mapping processes) is computationally expensive. However, the fact that each sample in a Monte Carlo simulation is independent by definition can be exploited to significantly improve the runtime of the simulator. Each sample can be computed in parallel with all other samples, and only a simple mathematical mean needs to be calculated once all samples are complete to produce the final result.

All of the subsequent experiments in this thesis take advantage of a distributed computing infrastructure to perform the required system lifetime simulations. The distributed compute resources are managed by Condor [25]. Given a batch of jobs, the Condor software will distribute them to any available compute resources, monitor the execution of the jobs, collect the desired output files from the jobs, and return the output files to the location from which the original submission occurred. Thus, Condor can be used to divide a single system lifetime simulation among a collection of resources by assigning a number of samples to each job. Some care must be taken in choosing the number of samples assigned to a single job as there is overhead involved in transferring program and input data to the networked resource and retrieving output from the network resource. Each job must execute a sufficient number of samples such that the time required for the network transfers, when amortized across the number of samples, is inconsequential. The number of samples required for a single system lifetime simulation can be large enough to effectively exercise a collection of a few hundred resources, and it is likely that without such a set of resources, the experiments in this thesis would have been infeasible.

3.8 Summary

This chapter presented our system lifetime simulator that is used to evaluate the task mapping techniques described in the remainder of this thesis. Our simulator uses Monte Carlo simulation to estimate the lifetime of a system given its architecture, the set of applications running on it, and a task mapping technique. Each sample in the Monte Carlo simulation represents an instance of the system with a random set of parameters that defines how quickly each failure mechanism will cause each component in the system to wear out over time. The lifetime of each sample system is mod-

eled at a relatively high level of detail. Our simulation uses realistic statistical distributions to model failure mechanisms, correctly accumulates wear on components even as temperature changes, performs detailed routing to estimate switch power dissipation, and accounts for physical factors like floorplan and component size. By averaging together the failure times of each sample system, we obtain an estimate for the actual lifetime of that system. The architecture of our system lifetime simulator allows arbitrary task mapping techniques to be plugged in so that they can be compared using a common evaluation platform.

During the evaluation of each Monte Carlo sample, we keep track of the amount of wear accumulated by each component due to each failure mechanism. Once a critical amount of wear has been accumulated on a component, that component is considered to be failed and unusable by the remainder of the system. We use a series of mathematical operations on the statistical representation of the failure mechanisms to model how wear is accumulated on each component on the system as its temperature changes. In order to estimate the temperature of all components in the system, we must compute the power dissipation for all components in the system. The power dissipation of network switches is related to the amount of data flowing through them, and we use Dijkstra's algorithm to determine that amount. Finally, we discussed the use of Condor, a system which manages distributed computing resources, as a way to decrease the overall runtime of the system lifetime simulator. The massive set of parallel computing resources provided by Condor contributed significantly to the level of detail possible in our simulator and the overall number of experiments that we were able to complete.

Chapter 4

Design Time Task Mapping Optimization Using Ant Colony Optimization

The first time at which task mapping optimization can be used to enhance system lifetime is when the system is being designed. Our solution for design-time task mapping involves the use of ant colony optimization (ACO) to find the best initial task mapping for a given system and the applications being executed on it. ACO is a generic optimization technique that is applicable to many domains and can be adapted to task mapping in a relatively straightforward manner. In ACO, the task mapping solution space is represented as a graph which is traversed to build candidate task mapping solutions. The solutions are then evaluated, and the graph is annotated with the results in order to improve future traversals.

Given that embedded multiprocessor systems are typically designed over the course of many months to multiple years, there is significant time available to designers to spend on optimization. ACO takes advantage of some of this design time by performing a thorough search of the space of task mappings for near-optimal solutions. This strategy is quite different than one that would be employed to choose task mappings at runtime due to the much tighter time constraints present in that scenario. Design time task mapping optimization is also different from runtime task mapping optimization in that it doesn't require any additional hardware cost in the form of sensors. The

heuristic technique we use for runtime task mapping optimization is described later in Chapter 5.

In addition to being novel on its own, our design-time task mapping approach also affords us the ability to compare direct and indirect lifetime optimization techniques. Due to the dependence of wearout faults on temperature and power, one may suspect that techniques which optimized temperature and/or power will also optimize lifetime. However, we will show that there are other effects at play which prevent these types of indirect lifetime optimization techniques from maximizing lifetime. We perform this comparison between direct and indirect lifetime optimization techniques, as well as the general evaluation of our design-time task mapping approach, using the system lifetime simulator described previously in Chapter 3.

This chapter discusses two major contributions. First, we present a novel strategy based on ACO for performing lifetime-aware task mapping and compare it to lifetime-agnostic (random) and temperature-aware simulated annealing (SA) approaches along with observed optimal task mappings. On average, our ACO implementation significantly improves system lifetime as compared to the lifetime-agnostic approach and performs similarly to the temperature-aware SA approaches, albeit with shorter runtime. Second, we draw on our experimental results to demonstrate that task mappings that optimize system temperature do not necessarily optimize system lifetime. In fact, for a given system temperature, there is significant variation in the lifetime achievable by different candidate task mappings. Further, we show that task mappings which optimize system lifetime also do a good job of optimizing system temperature. Our results clearly demonstrate the importance of performing lifetime-aware task mapping when enhanced system lifetime is a design goal.

The remainder of this chapter will explain how ACO is used to search for optimal task mappings, the benchmark architectures and applications we use for evaluation, how our ACO-based approach compares to a simulated annealing-based approach, and how well indirect lifetime optimization compares to direct lifetime optimization. The work described in this chapter was presented in part in [11].

4.1 ACO-Based Task Mapping

Our approach to task mapping requires a system description which defines the architecture of the system (a list of components, their capacities, and the links between them), a task graph which

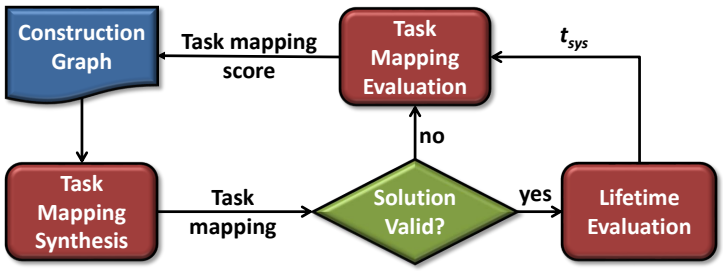

Figure 4.1: Overview of initial task mapping optimization using ACO

defines the properties of the application to be mapped (a list of tasks, their requirements, and their communication rates) and a system floorplan as input. Our goal is to determine the initial mapping of tasks to processors and data arrays to memories which results in the longest system lifetime. We use the term initial because we assume that as components fail, the failure is detected, and the system restarts with a remapping which uses the remaining components. Runtime task mapping optimization is not in the scope of this chapter and is detailed in Chapter 5.

There are a number of reasons for our selection of ACO as a search strategy. First, ACO is capable of dealing with dynamic solution spaces, and while this work only uses ACO to find an optimal initial task mapping, the approach could be extended to find optimal task remappings to be used as components fail. ACO's adaptability to dynamic solution spaces is important in this case since it allows optimal task remappings to be constructed from knowledge gained in searching for an initial task mapping, thereby speeding up the process. Second, the ability to handle dynamic solution spaces makes ACO a good candidate for inclusion in a complete system synthesis flow. For example, as a communication architecture search is synthesizing an optimal solution, an ACO-based task mapping search can build partial task mappings that will become more complete as more of the communication architecture is specified. Third, the ACO algorithm is parallelizable since each ant acts independently. Fourth, our ACO-based approach provides excellent extensibility in that it could be updated to take process variation information into account, and it could make decisions about DVFS settings and task parallelization without significant modification. Finally, ACO has been shown to be an effective solution for the generalized assignment problem, and task mapping is a member of that class of problems [26].

Figure 4.1 gives an overview of our ACO-based approach to task mapping search. Information is taken from the system description and task graph inputs and used to create the construction graph

as described in Section 4.1.1. The construction graph is traversed by simulated ants to synthesize task mappings which are subsequently checked for validity and scored. The task mapping score is fed back into the construction graph via pheromone deposition and affects future task mapping synthesis. As ants traverse the construction graph, the decisions they make about which edges to take are random but weighted by the amounts of pheromones that have been deposited on the edges. Thus, ants will typically follow the path created by a good solution while differing in some decisions, and this allows other solutions in the neighborhood of a good solution to be searched. The fact that the pheromones evaporate over time if not refreshed helps ACO from becoming stuck in local minima. The jobs of a single ant (task mapping synthesis, lifetime/task mapping evaluation, and pheromone deposition) are run to completion before another ant is spawned. Ants are spawned until a pre-determined number of valid task mappings are synthesized.

4.1.1 Task Mapping Synthesis

The basis of any ACO implementation is its construction graph. In the case of task mapping, the ants walk this graph to make decisions about how tasks should be mapped to the components. The nodes in our construction graph consist of the set of all components in the system combined with the set of all tasks in the application. The graph is connected by directed edges from each component to each task, which we call decision edges, and from each task to each component, which we call mapping edges.

Each ant begins its traversal by selecting a decision edge whose endpoint (a task) will be the first task the ant maps. The ant then selects a mapping edge from the endpoint of the selected decision edge to some component. The selection of each mapping edge defines a single task-to-component mapping synthesized by the ant. The ant continues its synthesis by alternating between the selection of decision and mapping edges (i.e, selecting a task and then the component to which it will be mapped) until all tasks are mapped. Edges are selected according to a weighted, random selection where each edge's weight corresponds to the quantity of pheromone on it (described in Section 4.1.3).

Figure 4.2 shows an example of our task mapping synthesis process when completed. Each of the four tasks in the task graph is represented by a different color. The communication architecture for this example is a central switch which connects four processors whose colors indicate the task

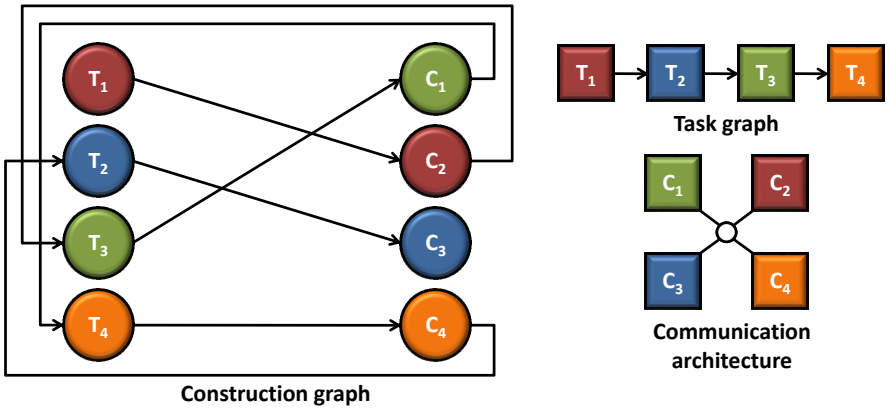

Figure 4.2: Example of task mapping synthesis done by ACO

mapped to them in the synthesized task mapping. An ant begins its traversal of the construction graph at the T1 node in the upper left of the construction graph. The ant then chooses between four mapping edges (one for each component) by weighted, random selection. In this example, the ant moves to node C2. Selecting this edge maps task T1 to component C2, and this selection is reflected by the red color of C2 in the communication architecture diagram. The ant then uses weighted, random selection to choose between three decision edges, each of which leads to one of the remaining tasks to be mapped. Here, the ant moves along the decision edge from C2 to T3, indicating that T3 will be the second task to be mapped. This process continues until all tasks have been mapped.

4.1.2 Task Mapping Scoring

Once all tasks have been mapped, the initial task mapping can be evaluated. The first step in evaluating any task mapping is to determine whether or not any component capacities have been violated. While traversing the construction graph, an ant may place too many tasks on a given component. The evaluation process checks for capacity violations while scoring the ant's solution using information about each component's capacity and the requirements of each task. Processor and memory capacity are measured in MIPS and KB respectively. We obtain the processing requirements of compute tasks, in terms of MIPS, and the storage requirements of data tasks, in terms of KB, through application profiling. We assume that tasks consume the same amount of compute power regardless of which processor they get mapped to. That is, all processors in our component library have the same efficiency; processors and memories differ from one another only in terms of capacity.

The second step in task mapping evaluation is to route the communication traffic between the tasks in the system. We use Dijkstra's Algorithm to route traffic. Given a traffic routing, we can determine if any link bandwidth is violated by summing the communication rates of all pairs of communicating tasks which use a particular link and comparing that sum to our maximum assumed link bandwidth of 256 MB/s. If a solution has neither component capacity nor bandwidth violations, it is said to be valid; otherwise, the solution is invalid.

The process used to score a task mapping is shown in Algorithm 2. If a solution is valid, its score is equal to the ratio of the resulting t_{sys} as measured by the system lifetime simulator described in Chapter 3 to a baseline t_{sys} for that system (line 3). Baseline t_{sys} values are obtained for our examples using task mappings created by hand and without the intent of optimizing for lifetime. This scoring method favors task mappings which yield longer system lifetime over those which result in shorter system lifetime. If a task mapping is invalid, its score is related to the ratio of the number of incurred violations (line 5) to the number of possible violations (line 6). Invalid scores are further weighted by a fractional penalty which generally makes invalid solutions worth less than valid ones (line 7). Through experimentation, we found that a penalty value of 0.8 works well for the designs we tested.

Algorithm 2 ACO Task Mapping Scoring

```
1:  function ScoreTaskMapping(measuredTsys, validSol, bwsViolated, capsViolated)
2:      if validSol then
3:          score = measuredTsys/baseTsys
4:      else
5:          totalViolations = bwsViolated + capsViolated
6:          possibleViolations = numberOfLinks + numberOfComponents
7:          score = penalty * (1 − (totalViolations/possibleViolations))
8:      end if
9:      return score
10: end function
```

4.1.3 Pheromones

A key aspect of ACO algorithms is the way in which ants share information about their solutions with each other through pheromones. In the real world, ants mark paths to food sources with pheromones so that other ants in the colony can follow the same path to the food source. Over time, the pheromones evaporate.

The purpose of the pheromones is to direct the ants toward areas of the solution space which are known to contain good task mappings while still allowing the ants to explore other, potentially better, task mappings in that area. The amount of pheromones placed on each edge of a particular ant's path through the construction graph is equal to the ant's solution score. An ant only deposits pheromones on the path it has taken if the score of its synthesized solution is the highest found by any ant in the colony up to that point in time. Figure 4.3 shows a construction graph in which some edges have been updated with pheromones according to a good solution. Future ants traversing this construction graph would select the red edges with higher probability than the black edges at each node.

Chemical pheromones evaporate over time so that ants become increasingly disinclined to travel paths which have not been recently shown to lead to a good location. Similarly, the edge weights in a construction graph decay over time to prevent the ACO process from becoming stuck in local minima and to drive the exploration of a larger area of the solution space. In our ACO implementation, edge weights experience decay after an ant has synthesized and scored its solution but before any pheromone deposition occurs. We calculate edge weight after decay simply as a fraction of the original edge weight.

Empirically, we have found that the best value for the evaporation rate parameter is dependent on the fraction of valid task mappings in the solution space. The reason that this fraction changes from design to design is discussed in Section 4.2.2. Generally, lower evaporation rates yield better results when the solution space contains a smaller fraction of valid task mappings. The low evaporation rates force ants to dive deeper into areas of the solution space which have been found to contain near-valid solutions. For solution spaces with higher fractions of valid task mappings, higher evaporation rates tend to produce better results since they allow the ants to explore many areas of the solution space without strongly focusing on a single one. In the future, we would like to automate the selection of the evaporation rate parameter based on the particular system for which we are performing a task mapping search.

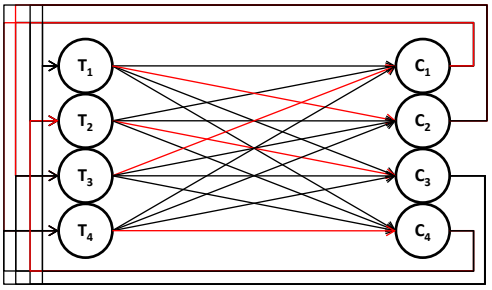

Figure 4.3: Example of how pheromones are used in ACO

4.2 Experimental Setup

To investigate the differences in system lifetime between task mappings which optimize for temperature and lifetime, and to assess the ability of our ACO- and SA-based task mapping approaches to find task mappings resulting in high t_{sys}, we conducted a series of design experiments across several different system architectures. In this section, we present the applications and architectures used in our experimentation, our assumptions about the types of components being used, and a discussion of the complexity of our benchmarks.

4.2.1 Bechmark Descriptions

We use three benchmark applications in the evaluation of our approaches: a synthetic application (synth), multi-window Display (MWD) [27], and an MPEG-4 Core Profile Level 1 (CPL1) decoder [28]. The task graph for the MWD application is shown in Figure 4.4, and the task graph for the CPL1 application is shown in Figure 4.5.

The synthetic application was implemented on a minimal communication architecture consisting of two switches (2-s). We experimented with one communication architecture for MWD, a four switch architecture resembling a ring (4-s), which is shown in Figure 4.6. We also experimented with two communication architectures for MPEG-4 CPL1, a data-flow pipeline of four switches (4-s) and a five switch architecture resembling a ring (5-s) The four and five switch communication architectures for CPL1 are shown in Figures 4.7 and 4.8, respectively.

We define a design point to be a communication architecture that is populated with a particular selection of processor and memory types (i.e, a slack allocation). For example, two different MWD 4-s design points would have the same communication architecture, the one shown in Figure 4.6,

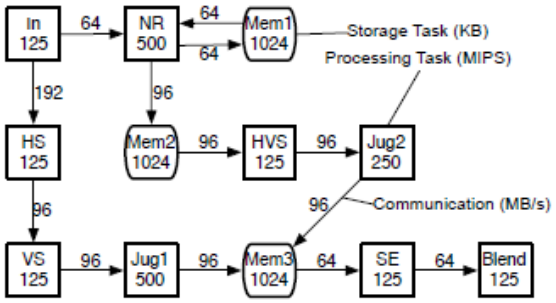

Figure 4.4: Multi-window display task graph

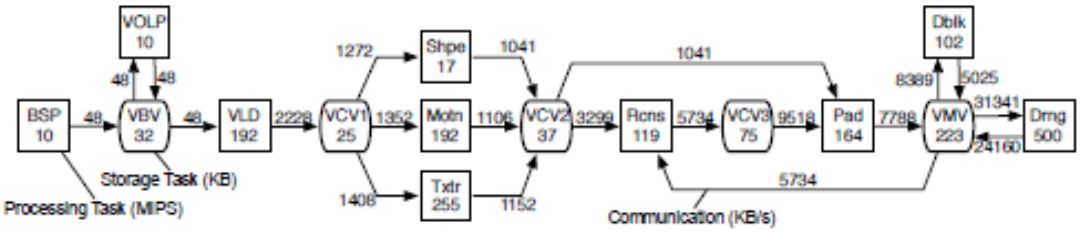

Figure 4.5: MPEG-4 core profile level 1 decoder task graph

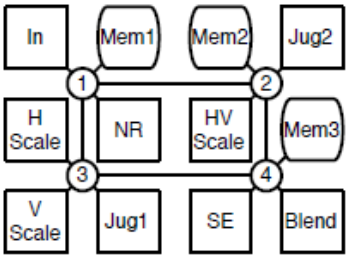

Figure 4.6: 4 switch communication architecture for the multi-window display application

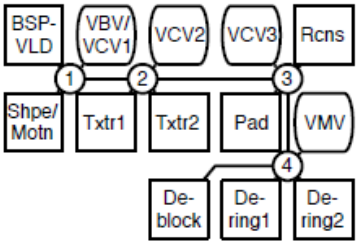

Figure 4.7: 4 switch communication architecture for the CPL1 application

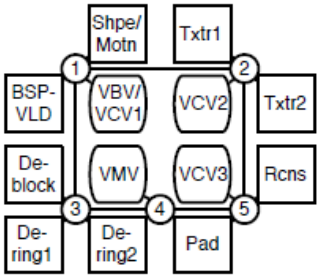

Figure 4.8: 5 switch communication architecture for the CPL1 application

Table 4.1: Summary of benchmark complexity

Benchmark	Design Point	Components	Tasks	Max TMs	% Feasible TMs	Eval Time
synth 2-s	0	6	8	16384	2.2	5s
synth 2-s	5	6	8	16384	12.8	29s
MWD 4-s	0	12	16	6.8e13	1.8e-4	19d
MWD 4-s	5	12	16	6.8e13	0.01	3.2y
CPL1 4/5-s	*	11	22	2.4e17	—	—

but would differ in the processor types chosen to fill the spaces P1-9 and in the memory types chosen to fill the spaces M1-3. Since the selection of components is different in each design point, each design point contains a different amount of slack and is therefore able to survive different sequences of component failures. The slack allocation in each design point is Pareto-optimal in terms of floorplanned area and lifetime as determined by [29] and therefore, the set of design points represents the best possible starting points for the task mapping search. Because each design point has a different slack allocation, the set of feasible task mappings for each design point is different. A number of design points were considered for each communication architecture: 6 for synth, 6 for MWD 4-s, 16 for CPL1 4-s, and 38 for CPL1 5-s.

We constructed all system architectures for each application using Cortex-M3, ARM9 and ARM11 processors, SRAM memories sized from 64KB to 2MB, and network switches. We assume all components are implemented in a 90 nm process. Processor area is based on data sheet values. SRAM area is derived using a low-standby power memory model from CACTI. Finally, all network routers are equivalent to the Alpha 21364's on-chip router and have a maximum crossbar size of 5x5.

4.2.2 Benchmark Complexity

The actual number of feasible task mappings for a given design can be determined through experimentation where each task mapping is enumerated and then tested for component capacity violations. While only testing for component capacity violations does not guarantee that the task mapping is valid, because routing may be impossible, it does bound the size of the solution space more tightly than the maximum number of task mappings. Table 4.1 summarizes the results of such experimentation for some of our design points. The first column shows the benchmark name, and the second column shows the design point within that benchmark being tested. The third and fourth columns show the number of components and tasks for the given benchmark, respectively. The fifth column shows the upper limit on the number of task mappings for the given design point as calculated by Equation 2.9. The sixth column shows the fraction of task mappings that are feasible for a given design point as determined by experimentation. The seventh column gives the amount of time that would be required to evaluate the lifetime of all feasible task mappings for a given design point using our system lifetime simulator. Cells in the table containing hyphens indicate pieces of data which were infeasible to obtain.

In Table 4.1, we see that the percentage of feasible task mappings increases between the two synthetic benchmark design points. This increase is due to an increase in slack between the two design points which allows more freedom in the task mapping process. The same trend is observed when moving between two MWD benchmark design points.

In general, we observe that the number of feasible task mappings increases extremely quickly as the sizes of the application and system increase. This observation leads to three conclusions. First, increased amounts of slack lead to an increased number of feasible task mappings, and some of these may enhance system lifetime. Thus, larger investments in system slack potentially enable greater lifetime enhancements though task mapping search. Second, it is infeasible to evaluate the lifetime resulting from all of the feasible task mappings for each design point. Exhaustive search is intractable for all but the smallest designs, and as a result, we have no verifiably optimal results with which we can compare our approach. Third, because the fraction of feasible task mappings is small, we cannot compare our approach to a truly random approach since randomly generating enough feasible task mappings to obtain high-confidence results would be prohibitively expensive from the

perspective of computational complexity. Thus, we use the benchmarks to compare the performance of our lifetime-aware ACO-based task mapping approach to a directed, random approach that is also based on ACO. Section 4.3.1 describes how the two approaches differ.

4.3 Validation of the ACO-based Tasked Mapping Approach

We created two variants of our ACO-based task mapping approach along with two variants of the SA-based task mapping approach and compared them to observed optimal task mappings for all design points in each of the benchmarks listed in Section 4.2.1. The remainder of this section discusses the variations on our approaches and the evaluation of our lifetime-aware, ACO-based task mapping approach.

4.3.1 Description of ACO and SA Variations

We tested two variations of our general ACO-based task mapping approach. In the first approach, *agnosticAnts*, we set up the ant colony to generate only one valid task mapping before the search was stopped. This strategy uses the ant colony to find a valid task mapping as fast as possible. The task mappings returned by this approach are effectively random since each valid task mapping in the solution space has an equal probability of being the first one found by the ant colony due to the method we use for scoring solutions. Large numbers of task mappings were generated in this manner for each design point to attempt to determine the range of t_{sys} values that different task mappings could produce for a given design point.

The second approach, *lifetimeAnts*, performs lifetime-aware task mapping. In this case, the ant colony is allowed to evaluate 20 valid task mappings for each design point before the search is stopped. Increasing the number of task mappings to be evaluated allows the ants to find a valid task mapping and then attempt to refine it in order to locate a nearby solution that results in higher system t_{sys} while continuing to search other areas of the solution space for high-quality task mappings. The task mapping resulting in the highest t_{sys} out of the 20 valid task mappings was taken to be the result of this approach.

To show that 20 valid task mappings is a reasonable number to choose for our *lifetimeAnts* approach, we ran an experiment on one of our design points to compare the results of the approach

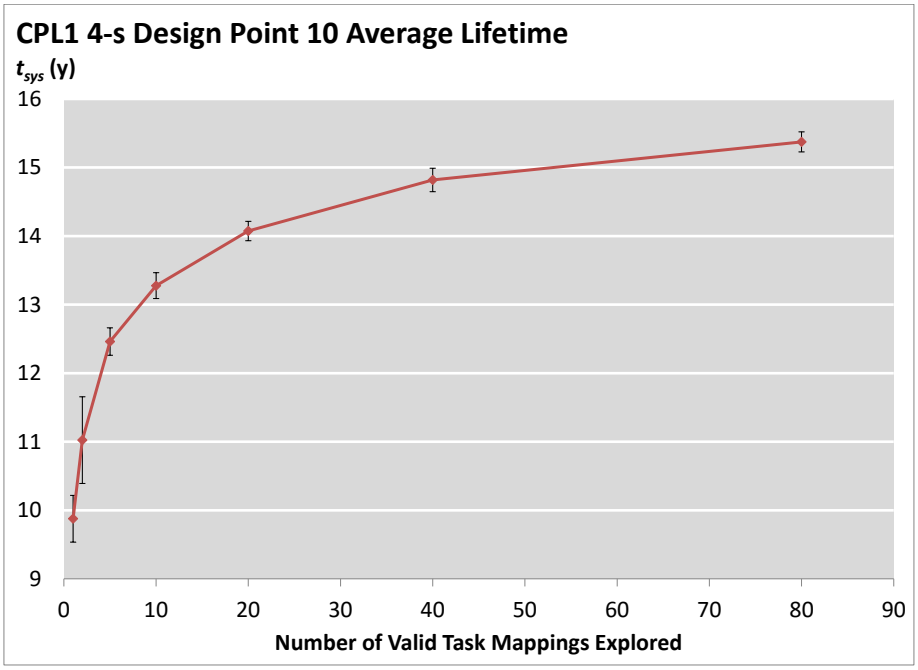

Figure 4.9: Comparison of ACO solution quality as the number of valid task mappings is changed

when allowed to evaluate different numbers of valid task mappings. Figure 4.9 shows the results of this experiment. The x-axis shows the number of valid task mappings that *lifetimeAnts* was allowed to explore before being stopped, and the y-axis shows the average t_{sys} resulting from the task mappings produced by *lifetimeAnts* over a number of runs. The error bars in Figure 4.9 represent 95% confidence intervals for the average t_{sys} based on the number of *lifetimeAnts* runs used to find that average. We observe that the average t_{sys} increases as *lifetimeAnts* is allowed to run longer. However, there are diminishing returns in solution quality as the number of valid task mappings is increased. We selected 20 valid task mappings as the constraint for *lifetimeAnts* for the remainder of our experiments since it represents a good tradeoff between solution quality and runtime. Of course, the number of valid task mappings can be changed to accommodate specific solution quality or runtime constraints.

Since temperature-aware task mapping can target either average [30] or maximum [10] temperature, we implemented two variants of the SA-based approach. In the first, *avgSA*, task mappings which optimize average initial component temperature are found. We define the average initial component temperature resulting from a task mapping as the average temperature of all components in the system at the beginning of the system's lifetime. In the second, *maxSA*, task mappings which

optimize maximum initial component temperature are found. We define the maximum initial component temperature resulting from a task mapping as the temperature of the hottest component in the system at the beginning of the system's lifetime. Both SA-based approaches begin by generating a random, and potentially invalid, task mapping. The annealer temperature is initialized to a value which allows any move to be accepted. We use a single move in the annealer: a task is randomly selected, and then moved to a different, randomly selected component. We perform 100 moves per temperature step and use a cooling rate of 0.9.

Similar to *lifetimeAnts*, the runtime of both SA-based approaches is limited by the number of valid task mappings that are found. We stop the SA-based approaches when they reach 50 valid task mappings, instead of using traditional freezing conditions, to maintain some parity with the *lifetimeAnts* approach. In particular, this limit lets the SA-based approaches achieve average solution quality similar to the *lifetimeAnts* approach at the expense of an increase in runtime.

4.3.2 ACO-based Task Mapping Evaluation

This section covers the evaluation of our ACO-based approach as a method for performing lifetime-aware task mapping. The evaluation is broken into a discussion of our synthetic benchmark results and a discussion of our real world benchmark results.

Synthetic Benchmark Results

The first part of the evaluation of our lifetime-aware, ACO-based task mapping approach involved a comparison with an exhaustive search. In Section 4.2.2, we showed that exhaustive search is infeasible for our real world benchmarks (MWD 4-s, CPL1 4-s, and CPL1 5-s). However, the synthetic benchmark was designed to be small enough to evaluate all feasible task mappings. To perform the exhaustive search, we first generated a complete list of feasible task mappings for each design point in the synthetic benchmark. We then used the Lifetime Evaluation block, from Figure 4.1, in isolation to determine the t_{sys} that resulted from each task mapping.

Comparing the exhaustive data for the synthetic benchmark design points with the results from the *lifetimeAnts* approach allowed us to ensure that our approach performed well in small design spaces. We found that *lifetimeAnts* was able to locate a task mapping resulting in a t_{sys} value

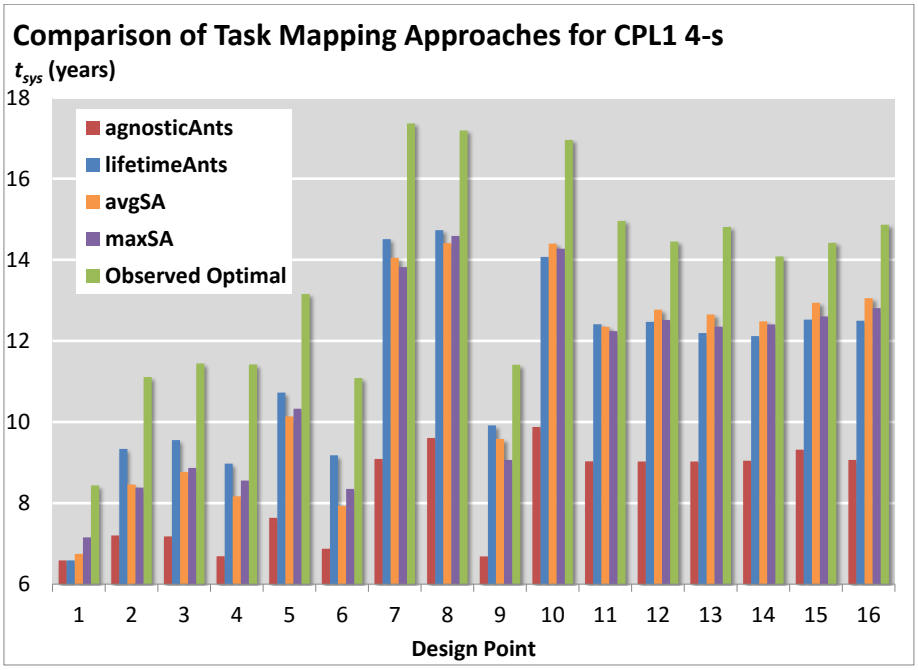

Figure 4.10: Comparison of all approaches to observed optimal task mappings for design points in the CPL1 4-s benchmark

equivalent to that of the best task mapping found by the exhaustive search for all six design points in the synthetic benchmark set.

Real World Benchmark Results: CPL1 4-s

We compared the two ACO-based approaches and the two SA-based approaches, described in Section 4.3.1, with observed optimal task mappings since obtaining the true optimal results is infeasible for the remainder of our benchmarks. For each design point, each of the four approaches was run several hundred times. We define the *observed optimal task mapping* for each design point as the one resulting in the highest system lifetime as found by any iteration of our four approaches. The t_{sys} values resulting from all runs of a particular approach on a particular design point were averaged to obtain an approximation of how well the approach performs on that design point. Figure 4.10 shows the results of this comparison for the design points in the CPL1 4-s benchmark.

Each group of bars in Figure 4.10 represents the results of our approaches for a single design point. The green bars indicate the t_{sys} resulting from the observed optimal task mapping (i.e, the highest t_{sys} resulting from a task mapping found by any single run of one of the four approaches). Each other set of bars represents the average t_{sys} found by all runs of one particular approach. The

Table 4.2: Summary of results for ACO- and SA-based approaches

Benchmark	*agnosticAnts*	*lifetimeAnts*	*avgSA*	*maxSA*	Lifetime Range
MWD 4-s	66.56%	77.3%	83.4%	82.4%	48.7%
CPL1 4-s	61.4%	83.9%	81.8%	81.8%	66.8%
CPL1 5-s	64.0%	85.1%	84.3%	83.1%	68.4%

data represented in Figure 4.10 shows that our *lifetimeAnts* approach consistently outperforms the *agnosticAnts* (random) approach in the CPL 4-s benchmark. When averaging across the 16 design points in this benchmark, the task mappings found by the *lifetimeAnts* approach result in 39.4% higher system lifetime than the task mappings found by the *agnosticAnts* approach. In the best case, design point 7, the average lifetime resulting from the task mappings found by our *lifetimeAnts* approach is 59.7% higher than that of the *agnosticAnts* approach. Also, both SA-based approaches perform about as well as the *lifetimeAnts* approach across these design points. This analysis is expanded to the remainder of our benchmarks in Table 4.2.

Real World Benchmark Results: Generalized

Table 4.2 summarizes the results of comparisons of our ACO- and SA-based approaches to the observed optimal task mappings for all design points in all benchmarks. The percentages in the columns labeled with approach names show how close that particular approach came to the observed optimal task mapping, on average, across all of the design points in a particular benchmark. These results are expressed as a fraction of the observed optimal task mapping's system lifetime, and thus, higher percentages represent better results. For example, our *lifetimeAnts* approach produced task mappings which resulted in lifetimes that were 85.1% of the lifetimes resulting from the observed optimal task mappings when averaged across the 38 design points in the CPL1 5-s benchmark.

We observe that the *lifetimeAnts* approach consistently finds task mappings that are closer to the observed optimal task mappings than those found by the *agnosticAnts* approach. Also, the quality of results achieved by the *lifetimeAnts* approach is generally unaffected by the complexity of the benchmark. Across all benchmarks, we see that *lifetimeAnts* is capable of finding task mappings similar to the observed optimal ones. As mentioned in Section 4.3.1, the SA-based approaches were tuned to produce solutions of similar average to the *lifetimeAnts* approach. Thus, the percentages in Table 4.2 for those three approaches are quite similar. However, similar average solution quality

does not tell the entire story since a user of these tools would not use the average result of several runs; the best result of several runs would be used. This more detailed analysis of the results is presented in Section 4.4.

In 45% of our 60 design points, the observed optimal task mapping was produced by a run of the *lifetimeAnts* approach. The observed optimal task mappings for the other design points were produced by one of the two SA-based approaches (26.7% by *avgSA* and 28.3% by *maxSA*). In the cases in which the *lifetimeAnts* approach did not produce the optimal task mapping, it was able to produce a task mapping resulting in lifetime within 3.9% of the observed optimal lifetime on average.

The approaches have different tradeoffs between solution quality and runtime: for a given design point, a single run of *agnosticAnts* evaluates a single task mapping while a single run of *lifetimeAnts* evaluates 20 task mappings. In practice, the runtime for a single run of *lifetimeAnts* is less than 20 times the runtime of a single run of *agnosticAnts* and is on the order of tens of seconds. For comparison, the observed optimal task mapping for each design point is the result of hundreds of task mapping evaluations.

The rightmost column of Table 4.2 shows the average difference between the best and worst lifetimes resulting from task mappings found by any run of the four approaches across all design points in a particular benchmark. The average lifetime ranges are expressed as a percentage of the lifetime resulting from the observed optimal task mappings in a particular benchmark. For example, when averaging across the 16 design points in the CPL1 4-s benchmark, the difference between the lifetimes resulting from the best and worst task mappings is 9.2 years, or 66.8% of the lifetime resulting from the observed optimal task mappings. It is obvious that system lifetime is greatly affected by the quality of the initial task mapping, and this data punctuates the need for lifetime-aware task mapping.

The data in Table 4.2 clearly show that the lifetime ranges increase with the complexity of the benchmark. Lifetime ranges also differ from one design point to another within a single benchmark. In each of the three benchmarks, we observed that the lifetime range generally increases with the amount of slack in the system; design points with more slack exhibit higher lifetime ranges than those with lower slack. Thus, as the opportunity to improve lifetime is increased through the addition of slack, the importance of lifetime-aware task mapping increases since good initial task

mappings are required to properly take advantage of the slack.

4.4 The Case for Lifetime-Aware Task Mapping

While the analysis in Section 4.3.2 shows that temperature- and lifetime-aware task mapping perform similarly in the average case, it does not explore the ways in which such tools would actually be used. Rather than averaging all results together, this section examines the task mappings found by the approaches that would actually be used (i.e, those found by a temperature-aware approach which result in low temperature and those found by a lifetime-aware approach which result in high lifetime). This comparison allows us to evaluate a temperature-aware task mapping approach's ability to find task mappings resulting in high system lifetime. We can also use the data to make observations about a lifetime-aware task mapping approach's ability to find task mappings resulting in low maximum or average initial component temperature.

4.4.1 Single Design Point Comparison: CPL1 4-s

Figures 4.11 and 4.12 depict the results of our analysis for design point 10 in the CPL1 4-s benchmark. In Figure 4.11, the sets of task mappings found by the *lifetimeAnts* and *avgSA* approaches are combined and plotted. Each point represents a single task mapping where average initial component temperature resulting from the task mapping is encoded by the x-coordinate while lifetime resulting from the task mapping is encoded by the y-coordinate. Figure 4.12 is similar to Figure 4.11 except that the task mappings found by the *lifetimeAnts* approach are combined with the task mappings found by the *maxSA* approach. In Figure 4.12, the x-axis represents the maximum initial component temperature resulting from a task mapping.

In Figure 4.11, we observe that large ranges of lifetimes result from task mappings which yield nearly the same average initial component temperature. To calculate these ranges for a particular design point, we first consider all of the points whose average initial component temperature is within 1% of the lowest average initial component temperature in the dataset. We then compute the difference in lifetime between the points with maximum and minimum lifetime in each set and report the result as a percentage of the maximum lifetime resulting from any task mapping for that design point. For example, the box in Figure 4.11 surrounds the set of points whose average initial

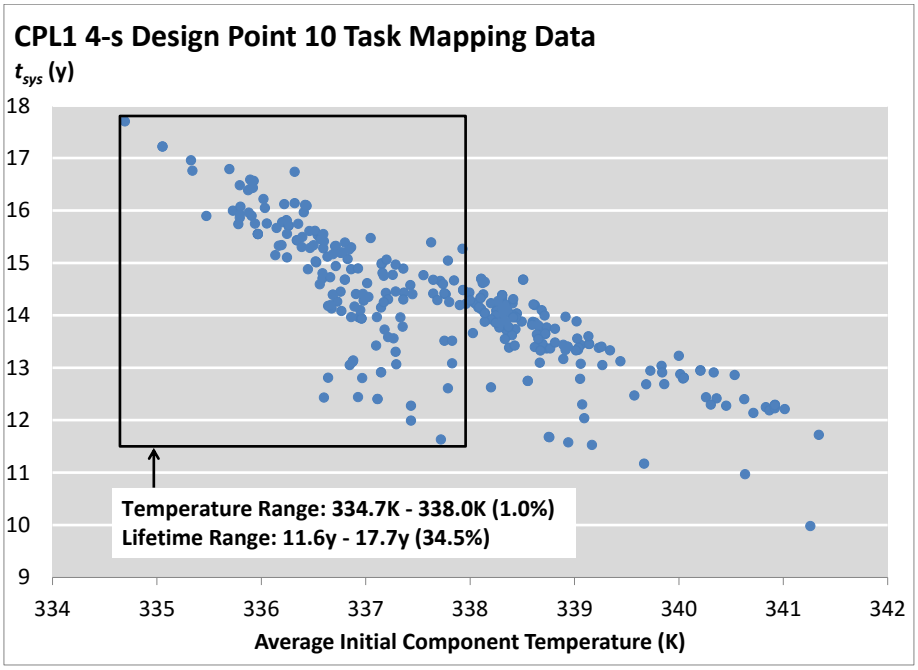

Figure 4.11: Average initial component temperature and lifetime data for CPL1 4-s design point 10 task mappings

component temperatures are within 1% of lowest average initial component temperature resulting from any task mapping in the dataset. The highest lifetime in this set of points is 17.7 years while the lowest is 11.6 years, and the difference between the two is 34.5%. This range indicates that a task mapping which results in a low average initial component temperature will not necessarily be a task mapping resulting in long system lifetime. We expand this analysis to all of our design points in Table 4.3.

The box drawn in Figure 4.11 also leads to a second important observation. As long as a task mapping is found that results in system lifetime within 34.5% of the optimal, the average initial component temperature resulting from that task mapping will be within 1% of the optimal average initial component temperature. If we consider all of the task mappings explored for this design point, we find that the range of resulting system lifetimes is 43.6% while the range of resulting average initial component temperatures is only 2.0%. This relation of lifetime range to average initial component temperature range shows that task mappings which result in high system lifetime are also likely to result in low average initial component temperature.

The purpose of Figure 4.12 is to show that the analysis described in the two previous paragraphs produces similar results when considering a temperature-aware task mapping algorithm that

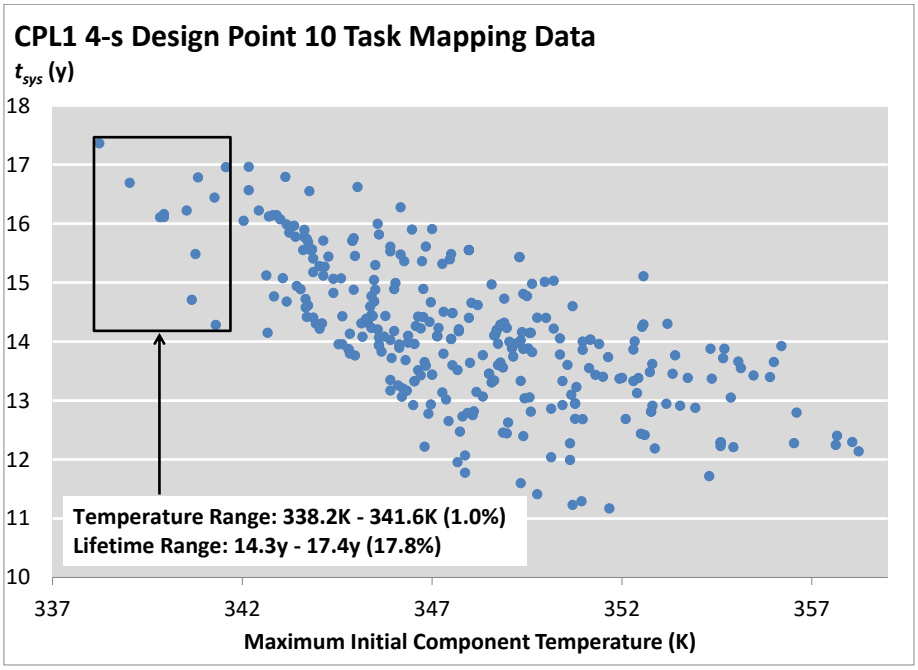

Figure 4.12: Maximum initial component temperature and lifetime data for CPL1 4-s design point 10 task mappings

optimizes maximum initial component temperature (*maxSA*). In this case, the range of lifetimes that results from task mappings within 1% of the lowest maximum initial component temperature is 17.8%. While this range is smaller than the one resulting from the optimization of average initial component temperature, it is still significant and demonstrates that task mappings which are optimized for maximum initial component temperature will not necessarily result in high system lifetimes. The analysis done in Figures 4.11 and 4.12 is expanded to all of our design points in Table 4.3 and is described in Section 4.4.2.

4.4.2 Generalized Comparison

Table 4.3 lists the average and maximum ranges in lifetime resulting from task mappings yielding nearly the same maximum or average initial component temperature across all design points in a benchmark. Before being averaged, the range for each design point is converted to a percentage of the highest lifetime resulting from any task mapping found for that design point. For example, the maximum lifetime range for task mappings which result in an average initial component temperature within 1% of the lowest, across the design points in the MWD 4-s benchmark, is 47.9%. The results

Table 4.3: Comparison of lifetime ranges for task mappings with temperature within 1% of optimal

Benchmark	Max Initial Temp		Avg Initial Temp	
	Avg	Max	Avg	Max
MWD 4-s	27.4%	44.3%	32.3%	47.9%
CPL1 4-s	17.5%	24.5%	33.5%	53.2%
CPL1 5-s	15.3%	23.2%	30.0%	44.0%

in Table 4.3 show that observations made in Figures 4.11 and 4.12 are generally true across all design points.

The lifetime ranges in Table 4.3 lead to another important result. Task mapping is complex enough to make exhaustive search infeasible, thereby requiring the use of intelligent design space searches. These searches cannot guarantee that the optimal answer will be found and instead hope to return a near-optimal result. Here, we assume that a quality temperature-aware task mapping approach will find task mappings that result in temperatures within 1% of the optimal. Our data shows that task mappings in this small range of high-quality solutions, in terms of temperature, are not necessarily high-quality solutions in terms of lifetime. In the case of one design point in the CPL1 4-s benchmark, the range of lifetimes resulting from task mappings within 1% of the lowest average initial component temperature is 53.2% of the lifetime that results from the task mapping with the highest lifetime. The data in Table 4.3 lead us to the conclusion that the high-quality solutions returned by temperature-aware task mapping approaches may be significantly sub-optimal in terms of lifetime.

We also arrive at the closely related conclusion that the high-quality solutions returned by lifetime-aware task mapping approaches are likely to be near-optimal in terms of temperature. Consequently, we assert that temperature-aware task mapping is a subset of the lifetime-aware task mapping problem. Specifically, a temperature-aware task mapping exploration will only find solutions with good temperature, while a lifetime-aware task mapping exploration will find solutions with both good lifetime and good temperature. The analysis of a larger number of task mappings for each design point would likely only strengthen this argument since the ranges we use to draw these conclusions would either stay the same or increase when looking at more task mappings (i.e, the task mappings which currently define the range do not disappear).

4.5 Summary

We presented an approach for lifetime-aware task mapping based on ACO. The approach works by identifying areas of the design space containing good task mappings and communicating that information to future synthesis runs to focus on those areas and find task mappings that yield high system lifetime. By directly optimizing lifetime, our approach considers physical parameters and the interaction of application and architecture that are not easily captured by temperature-aware task mapping.

Our results showed that our lifetime-aware, ACO-based task mapping approach performs well across a range of benchmarks. In a small, synthetic benchmark, our approach is able to locate task mappings equivalent to the best ones found by an exhaustive search. In larger benchmarks, our approach finds task mappings that result in t_{sys} within 17.9% of the observed optimal task mappings on average. Our lifetime-aware approach also outperforms a lifetime-agnostic approach by 32.3% on average. Finally, we make a case for the importance of lifetime-aware task mapping by showing that the difference between lifetimes resulting from good and bad task mappings averages 61.3% of the best lifetime across three real-world benchmarks.

Chapter 5

Runtime Task Mapping Optimization Using a Meta-Heuristic

The second time at which task mapping optimization can be used to enhance system lifetime is while the manufactured system is executing its application(s). Compared to the problem of design-time task mapping discussed in Chpater 4, a decision about how to change the task mapping at runtime must be made very quickly. When it is decided that a system's task mapping should be changed at runtime, either reactively when a component fails or proactively at a set interval, the amount of time used to select the new task mapping must be minimized to avoid system downtime. Thus, a detailed search strategy like the one we used for design-time task mapping is unusable in the context of runtime task mapping due to its computational cost. Instead, we propose the use of a carefully designed meta-heuristic which is able to select good task mappings with significantly reduced computational cost.

Our meta-heuristic combines several individual, component-level heuristics which optimize different system metrics that impact lifetime. Specifically, heuristics for power, temperature, amount of wear, and amount of time until failure are weighted and combined to build our meta-heuristic. While we cannot afford to spend much time searching for good task mappings at runtime, the real-time information from temperature and wear sensors used in the meta-heuristic keeps the quality of the task mappings high. Of course, the ability to account for actual system operating conditions at runtime requires additional hardware cost in the form of the temperature and wear sensors that are

not required for design-time task mapping.

In a wear-based task mapping heuristic, components which have smaller amounts of accumulated wear are favored during the task mapping process. This type of heuristic minimizes the accumulation of wear on components, thereby directly extending their lifetime. In turn, the lifetime of the system as a whole is also extended. Since the current amount of wear on a component is simply the sum of all the wear that was accumulated prior to the current time, this single measurement inherently captures all of the past wear-related state of a component. Thus, wear can be contrasted with component temperature or power dissipation, which only capture the current state of the system.

The main contribution of this chapter is the design of a runtime task mapping subsystem which improves system lifetime through the use of component wear information and its ability to continually adapt to the current state of the system. Ours is the first approach to dynamically manage the lifetime of embedded chip multiprocessors at runtime through the use of task mapping. By definition, runtime task mapping allows us to account for variations in the applications or system that would otherwise be impossible to address at design time. We will demonstrate that the systems being tested have a significantly longer lifetime when using our task mapping subsystem as compared to power- or temperature-based task mapping. Ultimately, we will be able to conclude that runtime wear-based task mapping is a requirement for systems in which lifetime is an important characteristic.

The remainder of this chapter will explain the details of how the individual heuristics are combined to form our meta-heuristic, the set of benchmarks we use to evaluate our runtime task mapping optimization, and how different balancings of the meta-heuristic affect system lifetime. The work described in this chapter was presented in part in [12].

5.1 Task Mapping Heuristics

We use the lifetime evaluation process discussed in Chapter 3 to compare a number of task mapping heuristics. The purpose of the task mapping process is to determine whether or not a component failure can be survived through remapping or if the system is no longer able to satisfy performance constraints. The task mapping process is successful if every task can be assigned to a single compo-

nent without violating any capacity or communication constraints. Tasks cannot be split into pieces, and so they must be mapped to a single component; a single component can accommodate as many tasks as its capacity allows. Communication between tasks is routed through the network and is subject to the communication constraints of the routers. The exception to this rule is when two communicating tasks are mapped to the same resource, and in this case, no network traffic is generated. If a mapping has been created without violating any capacity or communication constraints, we assume that the operating system will be able to create a valid schedule for the tasks mapped to each resource, and this process is outside the scope of our work.

In the case that a task mapping may be possible, we use a weighted scoring function, detailed in Algorithm 3, to map each task to a component. The algorithm is discussed below and corresponds to the "Task Mapping Algorithm" block in Figure 3.2. The tasks are first sorted by their requirements, and are run through the scoring function from largest to smallest (line 3). We performed some experimentation with different task orderings, but found that the order of descending requirements generally produced the best results. An in-depth analysis of the effects of task ordering in our task mapping algorithm is the subject of future research.

Given a task, the scoring function only considers components which currently have the available capacity to execute the task (lines 5-10). The score for a task/component pair is the weighted sum (lines 21-26), where the sum of the weights is always 1.0, of a number of individual scores that are each normalized to 1.0 (normalization values found on lines 14-18). Therefore, the highest total score that any task/component pair can receive is 1.0. The given task is mapped to the component with the highest score, and the available capacity for that component is reduced by an amount equal to the requirement of the task (lines 32-33). We consider each unique set of weights to be a different task mapping heuristic that is used within the larger framework of our task mapping algorithm.

The individual scores in the weighted sum are based on power, available capacity, temperature, time to failure, and wear. The power score for a task/component pair is based on the amount of power dissipated by the component per unit of work, and lower amounts of power dissipation per unit of work receive higher scores (line 21). This definition implies that the power score is dependent only on the characteristics of the component and is independent of the task requirement.

The available capacity score for a task/component pair is based on the difference between the component's available capacity and the requirement of the task, and smaller differences generate

Algorithm 3 Task Mapping Algorithm

```
1: bool taskMapping(tasks, components)
2: mapping.clear()
3: tasks = sort(tasks,largeToSmall)
4: for all t in tasks do
5:     candidates.clear()
6:     for all c in components do
7:         if c.availableCapacity() ≥ t.requirement() then
8:             candidates.add(c)
9:         end if
10:    end for
11:    if candidates.empty() = true then
12:        return usePreviousSolution()
13:    end if
14:    minPow = findMinPow(candidates)
15:    minCap = findMinCap(candidates,t)
16:    minTemp = findMinTemp(candidates)
17:    maxTTF = findMaxTTF(candidates)
18:    minWear = findMinWear(candidates)
19:    maxScore = 0
20:    for all c in candidates do
21:        powScore = powWt * maxPow/c.getPow()
22:        capScore = capWt * minCap/c.getCap()
23:        tempScore = tempWt * minTemp/c.getTemp()
24:        ttfScore = ttfWt * c.getTTF()/maxTTF
25:        wearScore = wearWt * minWear/c.getWear()
26:        totScore = powScore + capScore + tempScore + ttfScore + wearScore
27:        if totScore ≥ maxScore then
28:            maxScore = totScore
29:            bestComp = c
30:        end if
31:    end for
32:    mapping.add(t,bestComp)
33:    bestComp.availableCapacity() -= t.requirement()
34: end for
35: return route(mapping)
```

higher scores (line 22). As an alternative to this definition of the available capacity score, we tested a method that was only based on the available capacity of the component; however, this method produced inferior results.

The temperature score for a task/component pair is based on the current temperature of the component, and higher scores are given to components with lower temperatures (line 23). We assume that each component on the chip has a temperature sensor which reports the maximum temperature observed within the area of that component.

The time to failure (TTF) score is based on the amount of time before the component will fail due to a wearout fault, and higher scores correspond with higher times (line 24). The wear score is based on the wear accumulated by the component as a percentage of the amount of wear which will cause the component to fail (line 25). Components with a lower percentage of accumulated wear are favored by this score, and this wear score offers a slightly different interpretation of the data available from wear sensors than the TTF score. Our assumptions about the capabilities of wear sensors are supported by existing research that is discussed in Chapter 7. Like the power score, the temperature, TTF, and wear scores are all independent of the task requirement.

We implement different task mapping heuristics by changing the weights that are used in lines 21 through 24 of Algorithm 3. We tested numerous combinations of weights, and we chose a subset of the best performing combinations to show in our results. The combinations of weights we used to create different heuristics are shown in Table 5.1. Each column in Table 5.1 corresponds to one of the weights used in Algorithm 3, and each row corresponds to one class of task mapping heuristics. While each heuristic uses a significant power weight, power cannot be used by itself to create a good task mapping heuristic.

We tested a combination of weights in which the power weight was set to 1.0 and all other weights were set to 0.0, and the resulting heuristic did not perform as well as any of the other heuristics we discuss in this chapter. The reason for this result is that many of the components dissipate the same amount of power per unit of work, and some other measurement is needed to intelligently break those ties. The heuristics are named according to the weight that is used in addition to the power weight.

Once all processing tasks have been mapped as described above, we employ a simple, greedy task mapping strategy for memory arrays. This strategy maps memory arrays from largest require-

Table 5.1: Task mapping scoring weights used to create task mapping heuristics

Heur.	powWt	capWt	tempWt	ttfWt	wearWt
Power	0.5	0.5	0.0	0.0	0.0
Temp	0.75	0.0	0.25	0.0	0.0
TTF	0.75	0.0	0.0	0.25	0.0
Wear	0.75	0.0	0.0	0.0	0.25

ment to smallest requirement and favors memories where the available capacity matches that of the array being mapped. We chose not to employ our more sophisticated mapping heuristics for memory arrays due to our other assumption that memories cannot fail, and so the mapping of arrays to memories will not significantly impact system lifetime.

If the task mapping process reaches a point where there are no valid candidate components for a task or memory array, the process temporarily fails (lines 11-13). We found that some combinations of system state (power, available capacity, temperature, TTF, wear) prevent a task mapping from being found by our task mapping process even though one was previously found for the same set of failed components and a different system state. When we encounter these cases, we simply apply the previously found task mapping, even though it was "optimal" for a different system state. If no task mappings exist given the current set of failed components, then our task mapping algorithm fails ("fail" path in Figure 3.2).

The final step in the task mapping algorithm is traffic routing, which we perform via a basic implementation of Dijkstra's algorithm (line 32). After traffic routing, the amount of bandwidth placed on each link is compared to the maximum allowed bandwidth for that link. If any links exceed their maximum allowed bandwidth after the traffic routing process, our task mapping algorithm fails ("fail" path in Figure 3.2). Assuming no link bandwidth violations are found, component utilization and temperature are then re-calculated so that component failure times may be updated accordingly ("pass" path in Figure 3.2).

When there are no feasible task mappings for the current combination of remaining components, we record the time at which the sample system fails. We also record the time of the first component failure in each sample system, although it may be equal to the system failure time depending on the amount of slack in the system. When enough sample systems have been simulated, t_{sys} is estimated by computing the mean of all system failure times, and t_{first} is estimated by computing the mean

of all first component failure times. We found that 10000 samples were enough to estimate both t_{sys} and t_{first} with a 95% confidence interval of 1% or less of the measured t_{sys} or t_{first} value.

5.2 Benchmarks

We have tested all of the heuristics in Section 5.1 using a suite of benchmarks. Each of our benchmarks consists of a hardware architecture and a task graph that must be mapped to it, and remapped as components fail. The components used to construct the architectures include 4- and 5-port switches, three types of ARM processors, and memories of various sizes. The computational capabilities of the processors range from 125 to 500 MIPS, and the memory sizes range from 64KB to 512KB. Each processor or memory must be connected to exactly one switch, and each switch may connect to any combination of processors, memories, or other switches so long as its port limit is observed. We assume that components are connected via links with a maximum bandwidth of 256MB/s. BloBB [17] is used to create floorplans for all of our designs.

Our benchmarks can be divided into two groups. The benchmarks in the first group are related by a common hardware architecture while the benchmarks in the second group are related by a common task graph. Details about these benchmark groups are covered in the following subsections.

5.2.1 Single Architecture, Multiple Applications

For our first set of benchmarks, we selected a hardware architecture similar to one found commonly in academic examples and in industry: a mesh. This architecture contains nine switches (9-s) arranged in a mesh topology, and each switch contains a 5x5 crossbar which allows it to connect to five other components. The ports on each switch that are not used to connect to other switches are connected to processors. Nine M3 processors, six ARM9 processors, and six ARM11 processors are connected to the switches, and their arrangement is detailed in Figure 5.1. Although multiple processors are shown in a single red box with a single link to the switch, each processor in the box is actually independent of the others and has its own link to the switch. This simplification is also used in Figure 5.2.

We used Task Graphs for Free (TGFF) [31] to generate nine random applications for the mesh architecture. The generated applications contain a random number of tasks in the range of 45 to 55,

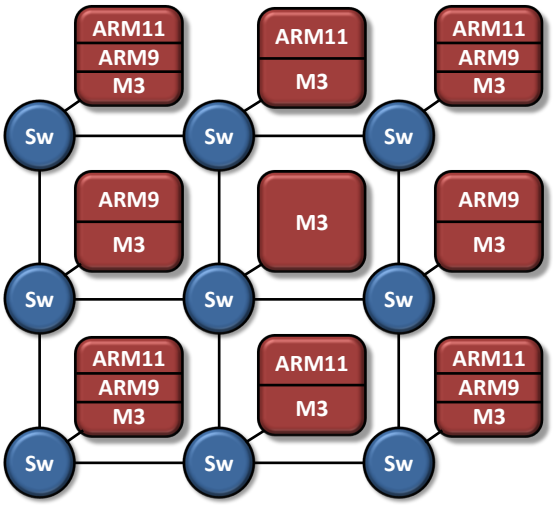

Figure 5.1: 9-s mesh hardware architecture

and the computational requirements for each task are also randomly generated. We parameterized TGFF such that a complete task graph would use about 80% of the resources of the system on average. While communication rates between tasks are also selected randomly by TGFF, we set the parameters for their generation such that they would fall well below the bandwidth limit of the links. Consequently, bandwidth limits are unlikely to restrict the set of feasible task mappings for this set of benchmarks.

To further expand this set of benchmarks, we used the same set of generated applications that were described above in conjunction with a second hardware architecture. The second architecture contains ten 4-port switches (10-s) arranged in a ring. Thus, each switch is connected to two other switches and two processors. A total of ten M3 processors, five ARM9 processors, and five ARM11 processors are used in this architecture, which is detailed in Figure 5.2. The 9-s mesh and 10-s ring architectures do not contain memories.

5.2.2 Single Application, Multiple Architectures

Our second set of benchmarks is based on an implementation of an MPEG Core Profile Level 1 decoder/encoder. The task graph for this application contains 17 processing tasks and five memory arrays. While this task graph is somewhat smaller than the randomly generated task graphs, it provides valuable data about how the various task mapping heuristics deal with a real-world system.

The MPEG task graph is combined with several different hardware architectures to create the

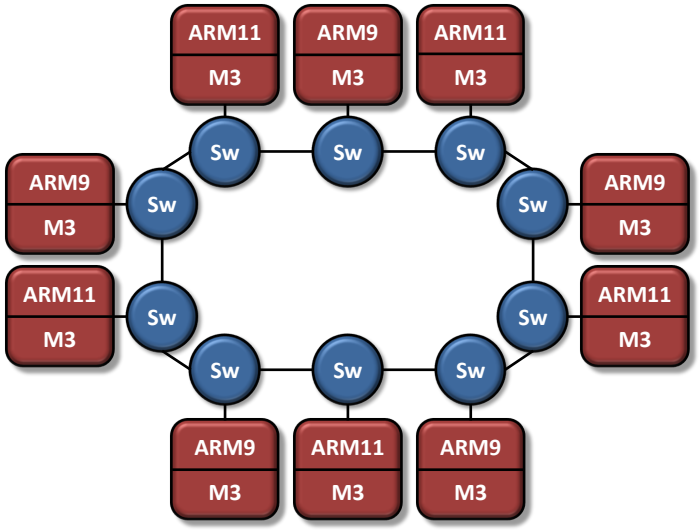

Figure 5.2: 10-s ring hardware architecture

second set of benchmarks, and all of those architectures are built around five switches arranged in a ring. This ring of switches is shown on the left side of Figure 5.3; the figure excludes the processors connected to each switch since their types change as we vary the hardware architecture. We create different hardware architectures by varying the types of processors and memories connected to the switches. The first design point in this set of benchmarks contains just enough resources to execute the task graph, and additional design points are created by replacing the baseline component with those having greater computation or storage abilities. By varying the types of components in the system, we are able to see how the task mapping heuristics use slack to affect the lifetime of the system.

We duplicate the MPEG task graph in a second part of this group of benchmarks in order to create another large-scale example and to see how the task mapping heuristics handle the mapping of two, independent task graphs. The hardware architectures are doubled in size, by doubling the number of components, to accommodate the addition of the second instance of the task graph. The doubled hardware architectures contain two rings of five switches each which are connected to each other through switches with previously unconnected ports. The switch topology for these hardware architectures is shown on the right side of Figure 5.3. Thus, while the two instances of the task graph do not communicate, the underlying tasks can be mapped to any component in the system since there is a path between any two components in the system. We also assume that components

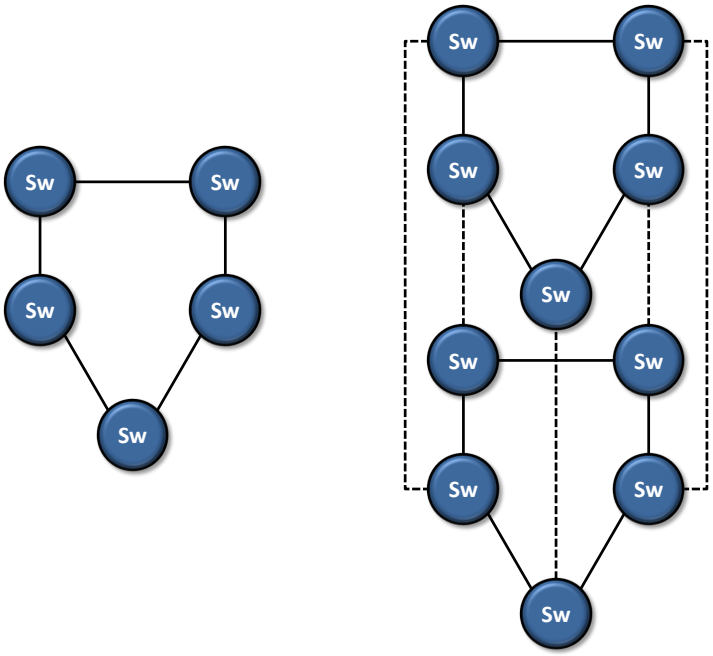

Figure 5.3: 5-s ring hardware architectures; left - single, right - double

are able to simultaneously accommodate tasks from the two task graphs if resource constraints are met.

5.3 Results

We use the benchmarks from Section 5.2 to compare the effects of the task mapping heuristics described in Section 5.1 on t_{sys} and t_{first}. We break down our results by benchmark set in the same manner as Section 5.2.

5.3.1 Single Architecture, Multiple Applications

This section covers the results of the experiments we ran using the benchmarks in Section 5.2.1. Figure 5.4 depicts the system t_{sys} values that were obtained when several task mapping heuristics were applied to the nine task graphs on the 9-s mesh architecture. The x-axis in the figure is a category axis in which each group represents the t_{sys} values resulting from the task mapping heuristics for one of the task graphs. The y-axis shows t_{sys}, as measured in years, when a particular task mapping heuristic is used to map a particular task graph. Each group of bars is organized from left to right as follows: power (red), temperature (blue), TTF (green), and wear (orange). For the

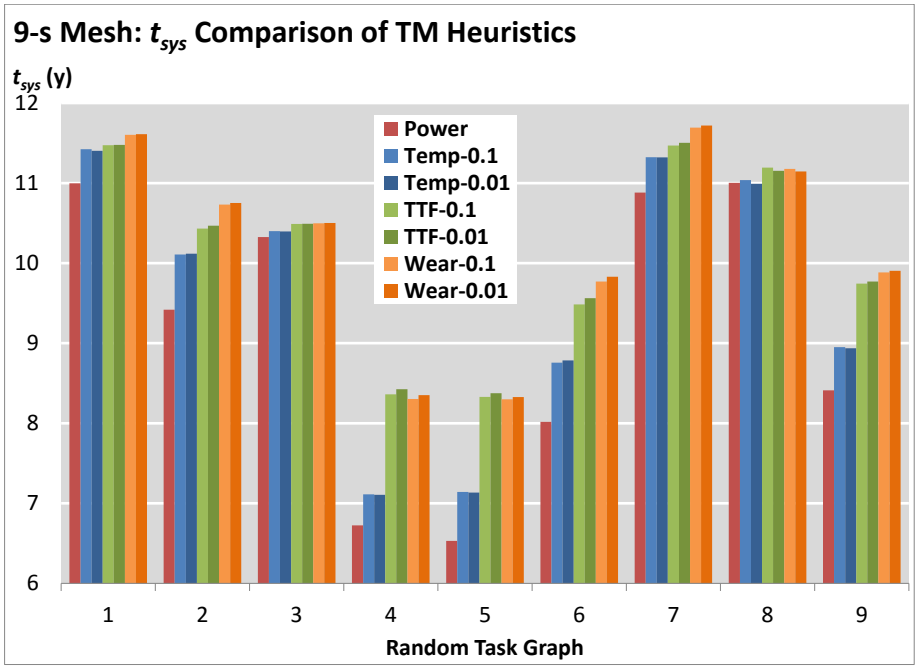

Figure 5.4: 9-s mesh t_{sys} results

temperature-, TTF-, and wear-based heuristics, we show the results when using the heuristic at two different intervals with two different shades of the same color. The color used to represent each heuristic remains consistent throughout the remainder of this chapter. The interval time (i.e, amount of time between triggering of the task mapping subsystem) is prefixed by a "-" in the name of the data set and is measured in years. When the power-based heuristic is used, new task mappings are only generated when a component fails since the amount of power a component dissipates per unit of work does not change over time. Thus, the power-based heuristic is listed without a task mapping interval time in the legend.

In all task graphs, the power-based heuristic yields the lowest t_{sys}. The temperature-based heuristics produce higher t_{sys} than the power-based heuristic, and this is further improved upon by the TTF- and wear-based heuristics. The TTF- and wear-based heuristics produce similar results for many of the task graphs, but there are a few cases in which the wear-based heuristics produce the best results by a measurable margin. Since the task graphs are not listed on the x-axis in any particular order, there is no observable trend in t_{sys} influenced by the characteristics of the task graph. Analyzing the relationship between task graph properties and system t_{sys} is outside of the scope of this thesis and is left for future work.

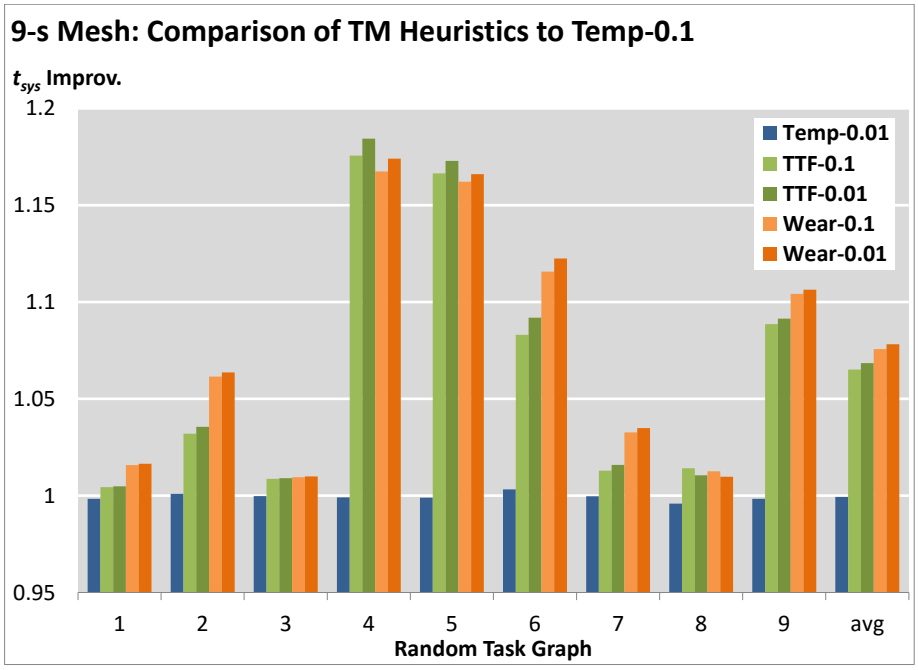

Figure 5.5: 9-s mesh t_{sys} improvement

Figure 5.5 shows the same data as in Figure 5.4, but all data has been normalized to the results of the temperature-based heuristic using an interval of 0.1 years. The x-axis remains a category axis representing the different task graphs, and it includes an "avg" group which shows the average improvement in lifetime across the nine task graphs that each heuristic achieved. The y-axis shows the percent improvement in t_{sys} that a particular heuristic was able to achieve over the Temp-0.1 heuristic. For example, the value of the Wear-0.01 heuristic for task graph 1 is shown to be 1.016, or a 1.6% improvement. We arrive at this value by dividing the t_{sys} which results from using that heuristic (11.61 years) by the t_{sys} which results from using the Temp-0.1 heuristic (11.42 years). The results for the power-based heuristic are not shown in this figure since they are always worse than those of the Temp-0.1 heuristic. This figure shows that the Wear-0.01 heuristic usually produces the greatest improvement over the Temp-0.1 heuristic with an average t_{sys} improvement of 7.8% and a maximum t_{sys} improvement of 17.4%. These t_{sys} improvement results are mirrored in the "Wear-0.01" row of Table 5.2 by the values of 1.078 and 1.174 in the "9-s Mesh/Avg" and "9-s Mesh/Max" columns.

Figure 5.6 is set up in the same manner as Figure 5.4 with the exception that the y-axis shows t_{first} as measured in years. All measurements shown in Figure 5.6 are less than or equal to the

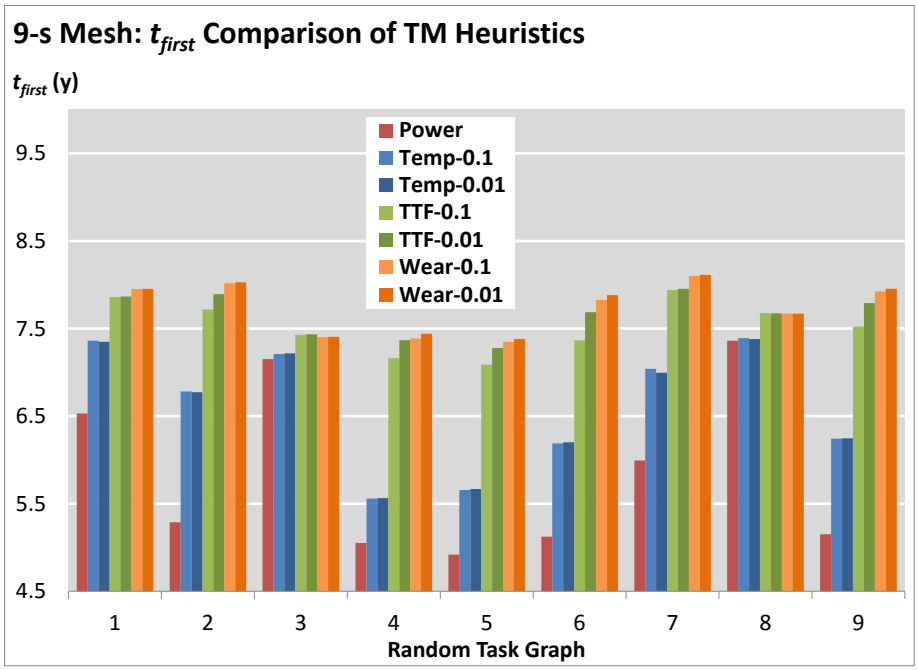

Figure 5.6: 9-s mesh t_{first} results

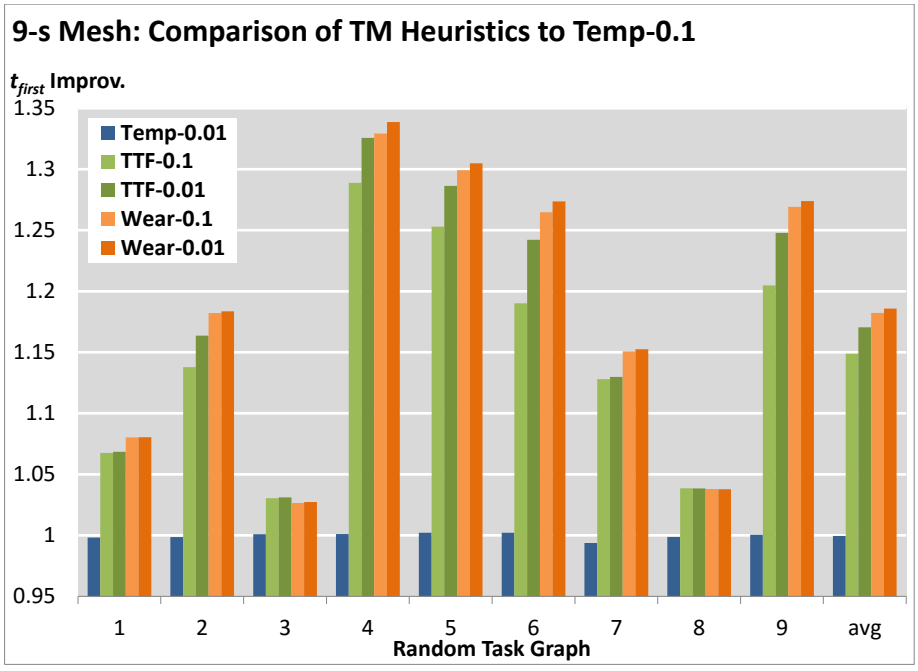

Figure 5.7: 9-s mesh t_{first} improvement

Table 5.2: Improvement in t_{sys} by task mapping heuristic and benchmark set

Heuristic	9-s Mesh		10-s Ring		MPEG		MPEG (double)	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max
Power	0.951	0.997	0.975	1.026	0.947	0.980	0.974	1.014
Temp-1.0	1.000	1.011	0.999	1.005	1.000	1.011	1.002	1.004
Temp-0.1	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Temp-0.01	0.999	1.003	1.002	1.007	1.000	1.007	1.000	1.005
TTF-1.0	1.054	1.151	1.046	1.102	1.041	1.105	1.046	1.136
TTF-0.1	1.065	1.176	1.057	1.127	1.052	1.134	1.050	1.165
TTF-0.01	1.068	1.184	1.061	1.137	1.055	1.134	1.050	1.169
Wear-1.0	1.062	1.141	1.044	1.091	1.051	1.094	1.045	1.121
Wear-0.1	1.076	1.167	1.057	1.113	1.061	1.116	1.053	1.156
Wear-0.01	1.078	1.174	1.060	1.123	1.062	1.115	1.053	1.154

Table 5.3: Improvement in t_{first} by task mapping heuristic and benchmark set

Heuristic	9-s Mesh		10-s Ring		MPEG		MPEG (double)	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max
Power	0.882	0.996	0.913	1.038	0.883	0.971	0.885	1.009
Temp-1.0	1.011	1.033	1.004	1.030	1.001	1.009	1.001	1.010
Temp-0.1	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Temp-0.01	1.000	1.002	1.002	1.007	1.001	1.008	0.998	1.005
TTF-1.0	1.108	1.245	1.071	1.156	1.070	1.105	1.117	1.218
TTF-0.1	1.149	1.289	1.095	1.173	1.100	1.134	1.157	1.261
TTF-0.01	1.170	1.326	1.105	1.190	1.104	1.133	1.177	1.288
Wear-1.0	1.154	1.265	1.089	1.152	1.095	1.130	1.141	1.210
Wear-0.1	1.182	1.329	1.103	1.184	1.109	1.153	1.175	1.275
Wear-0.01	1.186	1.339	1.108	1.193	1.111	1.157	1.177	1.279

corresponding measurements in Figure 5.4 since the systems have enough slack to continue functioning after the first component failure. The same patterns observed in the t_{sys} data also apply to this t_{first} data: the wear-based heuristic performs the best, followed by the TTF-, temperature-, and power-based heuristics. Figure 5.7 shows the t_{first} improvement over the Temp-0.1 heuristic corresponding to the data in Figure 5.6, and it is set up in the same manner as Figure 5.5. Once again, the Wear-0.01 heuristic shows the greatest improvement in t_{first} across all of the task graphs with an average improvement of 18.6% and a maximum improvement of 33.9%. These values for average and maximum improvement can also be found in the "Wear-0.01" row of Table 5.3 as the values 1.186 and 1.339 in the "9-s Mesh/Avg" and "9-s Mesh/Max" columns.

Table 5.2 summarizes the improvements in t_{sys} yielded by each heuristic for all of our bench-

mark sets. Each row in the table corresponds to one of the heuristics we tested in our task mapping subsystem, and each of the four major columns corresponds to one of our four benchmark sets. Under each benchmark set name, there are two subcolumns named "Avg" and "Max" which denote the average and maximum results across all benchmarks in a benchmark set. Each entry in the table represents a factor of improvement that is measured according to the description of Figure 5.5 above. These factors of improvement show how well a particular heuristic does compared to the Temp-0.1 heuristic. Values below 1.0 indicate that a heuristic did not perform as well as the Temp-0.1 heuristic, while values above 1.0 indicate that a heuristic performed better than the Temp-0.1 heuristic. The entire row for the Temp-0.1 heuristic contains entries of 1.0 to show that all results in the table are normalized to this heuristic. Table 5.3 is set up in the same manner as described above, except that it summarizes improvements in t_{first} yielded by each heuristic.

Our second set of experiments involved mapping the nine random task graphs to the 10-s ring architecture, and a summary of these results can be found in the second major column of Tables 5.2 and 5.3. In this set of experiments, the TTF-0.01 heuristic performed as well, on average, as the Wear-0.01 heuristic in regard to t_{sys}. Both heuristics managed to improve t_{sys} by an average of 6% over the Temp-0.1 heuristic. The Wear-0.01 heuristic does show marginal improvement over the TTF-0.01 heuristic in regard to average t_{first} improvement. Thus, the Wear-0.01 heuristic is again the best heuristic of the ones we tested.

5.3.2 Single Application, Multiple Architectures

The results from our experiments with the MPEG decoder/encoder application can be found in the third and fourth major columns of Tables 5.2 and 5.3. The Wear-0.01 heuristic again leads to the greatest average improvement in t_{sys} in both the single- and double-sized MPEG benchmark sets. In the single-sized MPEG benchmark set, the Wear-0.01 heuristic improves lifetime by up to 11.5% over the Temp-0.1 heuristic, and that maximum improvement jumps to 15.4% in the double-sized MPEG benchmark set. Similar to the results in the synthetic benchmark sets, the Wear-0.01 and TTF-0.01 heuristics perform similarly with respect to improvement in t_{first}. On average, the Wear-0.01 heuristic improves t_{first} by 11.1% in the single-sized MPEG benchmark set and by 17.7% in the double-sized MPEG benchmark set. The maximum t_{first} improvements achieved by the Wear-0.01 heuristic are even greater: 15.7% in the case of the single-size MPEG benchmark set and

27.9% in the case of the double-size MPEG benchmark. These results indicate that not only does our wear-based task mapping heuristic perform well across the benchmarks we tested, but also that the improvements in t_{first} scale with the number of tasks in the application.

5.4 Summary

We presented the design for a runtime task mapping subsystem which extends system lifetime by using a wear-based heuristic. Our simulator allows us to assume the existence of on-chip wear sensors and use data from them to modify the task mapping as the system runs to adapt to the changing wear patterns in the system. Basing task mapping decisions on a combination of wear and power data gives us a direct path to system lifetime optimization in contrast with using power by itself, or a combination of temperature and power, as proxies for lifetime. Also, using wear data is elegant and concise since it captures all relevant, previous system state in a single value. Temperature and power information only show the current state of the system by default and would require more complicated heuristics to capture and utilize prior values.

Our results showed that our wear-based task mapping heuristic performed well across a wide range of benchmarks. When testing large, random task graphs on a mesh architecture, we were able to improve t_{first} by an average of 18.6% and t_{sys} by an average of 7.8% over a temperature-based heuristic. We then used the same set of task graphs on a different hardware architecture, a ring, to show that our results were not dependent on a mesh architecture. Our wear-based task mapping heuristic improved t_{first} by 10.8% and t_{sys} by 6.0% over a temperature-based heuristic when the ring architecture was used.

We based a second set of benchmarks on a real-world MPEG decoder/encoder application with a corresponding hardware architecture and a variety of slack allocations. We were able to improve t_{first} and t_{sys} by averages of 11.1% and 6.2%, respectively, over a temperature-based heuristic. In order to see how our heuristics scaled with system size, we composed another set of benchmarks by combining two instances of the MPEG application with double-sized hardware architectures. In these benchmarks, we improve t_{first} by an average of 17.7% and t_{sys} by an average of 5.3% over a temperature-based heuristic.

Our data also led to two interesting secondary results. First, we showed that the addition of

slack to the system can improve both t_{first} and t_{sys} when our wear-based task mapping heuristic is used. Thus, we can conclude that task mapping and slack allocation are important considerations in designs where lifetime is a design goal, even if those systems are not meant to be used after the failure of any components. Second, we found that there are diminishing returns in lifetime improvement with regard to how often task mapping is performed. When decreasing the task mapping interval from one year to 0.1 years in both the wear- and temperature-based heuristic, a reasonable improvement in lifetime is seen in all benchmarks. As the task mapping interval is moved from 0.1 years to 0.01 years, we observed a much smaller improvement in lifetime. This observation means that designers must choose an appropriate task mapping interval depending on the overhead required to change the task mapping to obtain a good tradeoff between that overhead and lifetime improvement. This observation also points to the fact that the time required to compute the task mapping, which is on the order of seconds, is significantly shorter than the amount of time between the computation of task mappings, which is on the order of tens of days. Therefore, the amount of computation required during task mapping is insignificant compared to the amount of computation the system performs while executing its target application.

Chapter 6

Co-optimizing Competing Lifetime Metrics

In the previous chapters about design-time and runtime task mapping optimization, we focused on optimizing t_{sys} and reported resulting t_{first} values as secondary results. However, depending on design requirements and the use case of a system, t_{first} may be a more important figure of merit than t_{sys}. This chapter explores how the previously described task mapping techniques can be used to co-optimize these different lifetime metrics.

The notion of t_{sys} is based on the idea that most systems are over-provisioned to some degree and will therefore be able to withstand the failure of some components at runtime. In theory, system failure is averted as long as the required tasks can be mapped to components in the system which have not failed. The underlying assumption in these situations is that the system contains mechanisms for detecting permanent component failures and logically reconfiguring itself to remove any dependencies on failed components. In general, we define these recovery and reconfiguration mechanisms as any process which helps present a coherent and usable view of the hardware architecture to the operating system. While such mechanisms are certainly feasible, they may not be a realistic option for all use cases.

A system that must meet strict safety or security requirements may not be able to implement recovery and reconfiguration mechanisms because the complexity of these mechanisms is at odds with the strict verification requirements of these systems. Systems which implement recovery and

reconfiguration mechanisms have the potential to operate in scenarios with different combinations of failed components. This number of scenarios grows quickly with the number of components in the system and the amount of hardware over-provisioning. In safety critical systems, each of these scenarios must undergo an extensive verification process which can quickly become untenable. Also, a system which is being targeted for a quick time to market might not include recovery and reconfiguration mechanisms since they would increase the difficulty of design and verification. In both of these cases, where recovery and reconfiguration is not available, system-level techniques which target t_{sys} are ineffective since the system will become inoperable as soon as a single component has failed, even if sufficient resources are still available.

The lifetime metric of importance in these cases is time to first failure (t_{first}); the amount of time between when the system is powered on for the first time and when one component in the system has failed. System-level techniques which target t_{first} are able to make use of any over-provisioned hardware to delay the first component failure for as long as possible. In particular, the runtime task mapping optimization described in Chapter 5 accomplishes this by changing the task mapping as the system is running.

The main contribution of this chapter is an exploration of how various task mapping approaches affect t_{sys} and t_{first}. Our results show that there are scenarios in which no single task mapping approach maximizes t_{sys} and t_{first} simultaneously. We use our task mapping evaluation framework to show that different task mapping approaches are required to properly optimize different lifetime metrics. Simply put, there is no catch-all definition for lifetime, nor is there a task mapping approach which always optimizes lifetime as defined in different ways.

As a secondary contribution, we introduce a floorplan-aware task mapping heuristic. This heuristic combines information about the amount of wear on each component with the spatial relationships between components in the system floorplan. The benefits of this heuristic are twofold. First, the use of our floorplan-aware task mapping heuristic increases the overall number of Pareto-optimal task mapping approaches. This increase is important because it leads to a wider range of worthwhile tradeoffs between t_{sys} and t_{first} from which a designer can select. Second, the use of our floorplan-aware task mapping heuristic results in a modest increase in t_{first} for some designs.

6.1 Effects of System Utilization and Task Mapping on t_{sys} and t_{first}

As mentioned above, there are situations in which no task mapping approach can simultaneously maximize t_{sys} and t_{first}. The situations in which t_{sys} and t_{first} cannot be simultaneously maximized mostly occur when system utilization is less than 50% (detailed results can be found in Section 6.5.1). Situations in which system utilization is less than 50% are common and may even represent the majority of a system's lifetime depending on its use case. For example, cell phone processors likely spend more time idling (e.g, the phone is locked and the display is off) than they do at maximum utilization (e.g, a web browser or game is being used). A system must be designed with enough hardware resources to be able to handle the most demanding software that is planned for use on the system, but the system may not always be operating at maximum utilization. Depending on how system utilization changes over time, the task mapping approach may also need to be varied to ensure the target lifetime metric is maximized. Therefore, designers must consider the relative importance of t_{sys} and t_{first} in conjunction with the expected utilization when choosing a task mapping approach for a system.

6.2 Floorplan-Aware Task Mapping Heuristic

The individual heuristics which are used in the meta-heuristic are based on power, available capacity, temperature, time to failure, wear, and the system floorplan. The first five heuristics are identical to those found in Chapter 5.

The system floorplan heuristic combines floorplan information with TTF information from the wear sensors. This heuristic first computes the Euclidean distances between the candidate component and the two components in the system which have the lowest TTF values (i.e, the two components nearest to failure) based on the floorplan. We experimented with using distances to other numbers of components but found that considering two components worked the best for many of our benchmarks. The distances to these two components are then weighted by the inverses of the TTF values of those components. Thus, distances to components which will fail sooner are weighed more heavily. The two weighted distances are then averaged, and this average represents the floorplan score for a candidate component. By favoring higher floorplan scores, this heuristic

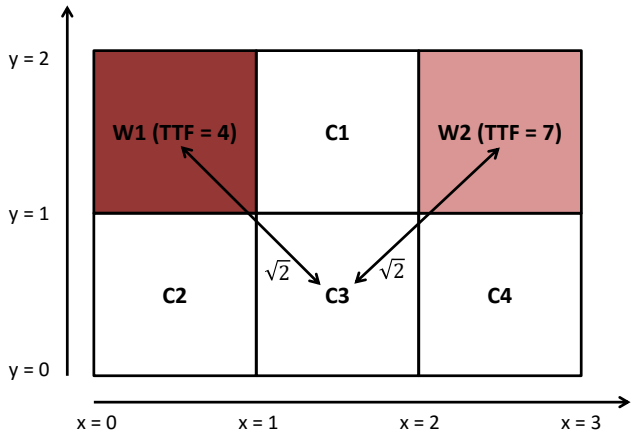

Figure 6.1: Example of the floorplan-aware task mapping heuristic

favors components which are far away from other components that will fail soon.

Figure 6.1 shows an example of how the system floorplan heuristic scores candidate components. The figure shows a floorplan containing six components, and the candidate components for the task mapping are labeled C1 through C4. The components with the two highest amounts of wear are W1 and W2; all components in the floorplan are of size 1x1 (in arbitrary distance units). In this example, W1 has the soonest failure time of any component in the system (TTF = 4) and is shaded dark red, while W2 has the second soonest failure time (TTF = 7) and is shaded light red. The arrows in the floorplan indicate the Euclidean distances from C3 to W1 and W2. The floorplan heuristic score computation for candidate component C3 is shown in Equation 6.1 and evaluated in Equation 6.2.

$$score(C3) = \frac{dist(C3, W1) * TTF(W1)^{-1} + dist(C3, W2) * TTF(W2)^{-1}}{2} \tag{6.1}$$

$$score(C3) = \frac{\sqrt{2} * \frac{1}{4} + \sqrt{2} * \frac{1}{7}}{2} \approx 0.278 \tag{6.2}$$

A score for each candidate component is computed in a similar way, and in this example, C4 has the highest score (0.351). C4 gets a higher score than C3 because even though C3 is equidistant from the components with the highest amounts of wear, C4 is further away from the most critical component in the system. C2 gets a lower score than C3 because it is closer to the critical component, and C1 gets the lowest score because it is the closest component in the floorplan to both W1 and W2.

The combination of these six heuristics into a single meta-heuristic has several benefits. First,

each of the individual heuristics addresses at least one part of the system state that is related to lifetime in some way. The temperature, power, and available capacity heuristics indirectly affect lifetime while the wear, TTF, and floorplan heuristics directly affect lifetime. Thus, the meta-heuristic can be used to improve system lifetime from a wide variety of angles. Second, the combination of heuristics eliminates the weaknesses of any individual heuristic. For example, performing strict wear-based task mapping would ignore the fact that some components may be more power efficient than others. This issue can be overcome by giving non-zero weights to both the wear and power heuristics inside of the meta-heuristic. Third, the meta-heuristic provides a platform that can easily be used to compare combinations of heuristics to individual heuristics. These comparisons allow us to see how our task mapping approaches perform with respect to existing, single-mode (i.e, power, temperature, etc.) approaches.

6.3 Creating Task Mapping Approaches

To create a list of task mapping approaches based on our meta-heuristic, we first created a list of all possible weight combinations. Each weight combination corresponds to one possible task mapping approach, and thus, each approach is a unique blend of the individual heuristics. A weight combination can be defined as a 6-tuple where each value is a weight for one of the six heuristics. We consider a weight combination to represent a valid task mapping approach if the six values sum to 1.0 and each value is a multiple of 0.2. While the number of possible task mapping approaches would increase if the "multiple of 0.2" constraint was reduced to a smaller number, we found that such a change would not significantly impact the results.

As an example, the ith weight combination can be written as shown in Equation 6.3. Each w^* value in Equation 6.3 corresponds to one of the heuristics described in Section 6.2.

$$WC_i = \left\{ w_i^{pow}, w_i^{cap}, w_i^{temp}, w_i^{ttf}, w_i^{wear}, w_i^{fp} \right\} \tag{6.3}$$

The list of weight combinations was constrained by setting a step size for each weight (s in Equation 6.4) and forcing the sum of all weights to be 1 (Equation 6.5).

$$w_i^{\{pow,cap,temp,ttf,wear,fp\}} = n \times s, n \in \mathbb{Z}_{\geq 0}, s \in \mathbb{R}_{\geq 0} \tag{6.4}$$

$$w_i^{pow} + w_i^{cap} + w_i^{temp} + w_i^{ttf} + w_i^{wear} + w_i^{fp} = 1 \tag{6.5}$$

$$WC_{example} = \{0.2, 0.2, 0.4, 0, 0, 0.2\} \tag{6.6}$$

An example weight combination is shown in Equation 6.6; the six weights sum to 1, and each weight is a multiple of a step size of 0.2. While decreasing the step size increases the number of weight combinations, we found that results were not significantly improved by step sizes smaller than 0.2. With a step size of 0.2, the list of weight combinations that we tested contained 252 entries.

While examining some early results, we noticed that some of the 252 weight combinations did not give unique results. That is, there were cases in which two different weight combinations produced exactly the same t_{sys} and t_{first} over the course of many samples. The reason why this can happen is that in a weighted meta-heuristic like ours, results are more dependent on the proportions of the weights to one another than they are on the exact values of the weights. It then follows that some of the weight combinations we enumerated would cause the meta-heuristic to behave in the same way as other weight combinations even thought they are not strictly identical.

We went through a process to eliminate weight combinations which produced identical results since we are only concerned with unique task mappings. For each architecture/task graph combination, we simulated each of the 252 weight combinations for 10 preliminary samples. If the results of all of the preliminary samples were identical for any pair of weight combinations, we removed one of those weight combinations from the list. This process reduced the number of unique weight combinations to between 100 and 150 depending on the architecture and task graph.

6.4 Benchmarks

We tested all of our task mapping approaches on each of several benchmarks. Each of our benchmarks consists of a hardware architecture and a task graph that must be mapped to it, and remapped as components fail. The components used to construct the architectures included ARM9 processors and 3-, 4-, and 5-port switches. The computational capability of the processors is equivalent to 250 MIPS, and the links between the switches have a maximum bandwidth of 256 MB/s. Each processor was connected to exactly one switch, and each switch was connected to one processor and 2, 3, or 4 other switches in square mesh topologies. Three different target hardware architectures

Table 6.1: Summary of benchmark task graphs

Arch.	TG Num.	Num. Tasks	Utilization
9 processors	1	1	0.111
	2	2	0.222
	3	3	0.333
	4	4	0.444
	5	5	0.556
64 processors	1	3	0.047
	2	10	0.156
	3	20	0.313
	4	30	0.469
	5	40	0.625
100 processors	1	15	0.15
	2	25	0.25
	3	35	0.35
	4	44	0.45
	5	55	0.55

were created in this way: a 9 processor (3x3) mesh, a 64 processor (8x8) mesh, and a 100 processor (10x10) mesh. All floorplans were created to match the mesh topologies described above and have no empty space between components.

We created five unique task graphs for each of the three architectures. The five task graphs for any particular architecture cover a range of utilization values between about 5% and 60% of the capacity of that architecture. Details about the task graphs are shown in Table 6.1. The first two columns identify the task graph and the architecture on which it is used. The third column lists the number of tasks in each task graph, and the fourth column lists the fraction of the system that is occupied by each task graph.

6.5 Results

We completed a set of experiments using the benchmarks described in Section 6.4 to compare the effects of the task mapping approaches described in Section 6.3 on t_{sys} and t_{first}. This section breaks down the analysis of those experiments into three sections. First, we examine how well t_{sys} and t_{first} can be simultaneously maximized by our task mapping approaches. Second, we discuss the impact of floorplan-based task mapping on the number of good tradeoffs between t_{sys} and t_{first}.

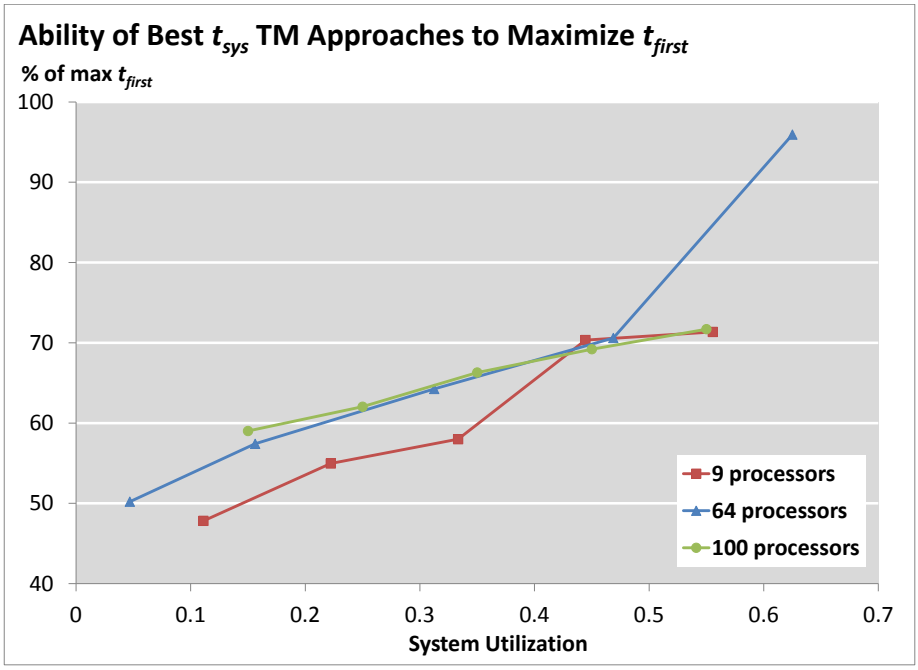

Figure 6.2: t_{first} maximization potential using t_{sys}-optimal task mapping approaches

Finally, we show that floorplan-based task mapping can be used to improve t_{first} in some cases. The t_{sys} and t_{first} values for each benchmark are calculated according to the lifetime evaluation and task mapping processes described in Chapter 3. Each value is the result of 400 Monte Carlo samples, which leads to 90% confidence interval sizes that are 1-5% of the calculated t_{sys} and t_{first} values.

6.5.1 Maximizing t_{sys} and t_{first}

In this section, we examine the possibility of simultaneously maximizing t_{sys} and t_{first} using our task mapping approaches. In other words, we look at what percentage of the maximum t_{first} (or t_{sys}) is achieved by the task mapping approach that results in the maximum t_{sys} (or t_{first}). To do this, we first tested all of the unique task mapping approaches in each benchmark (architecture/task graph pair). We then determined the maximum values of t_{sys} and t_{first} that could be attained in each benchmark from these data.

Figure 6.2 shows how well the task mapping approaches which maximize t_{sys} are able to maximize t_{first}. The x-axis shows system utilization, or the percentage of the system that is occupied at any one time by the task graph being tested. The y-axis shows the percentage of the maximum

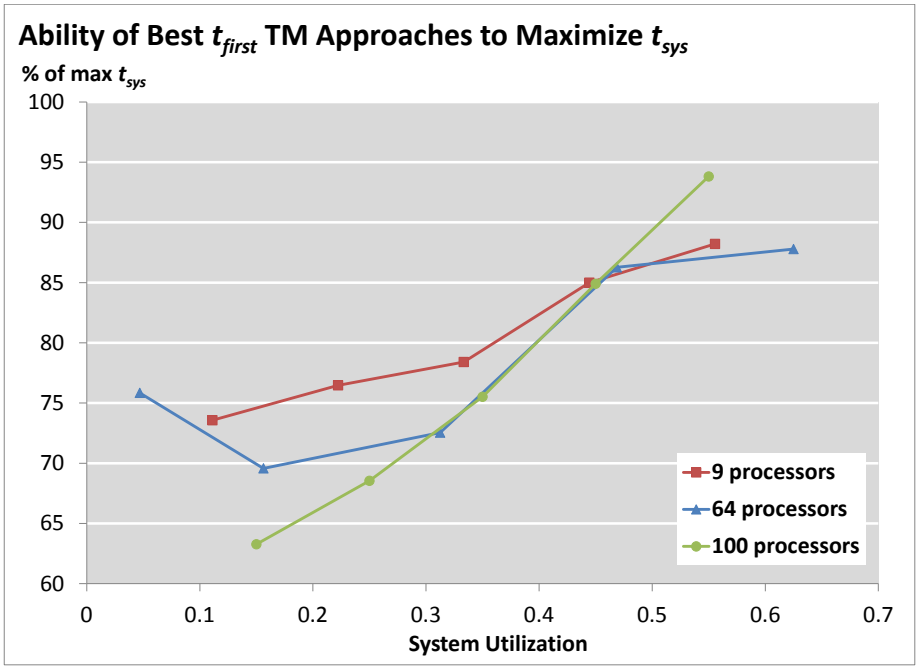

Figure 6.3: t_{sys} maximization potential using t_{first}-optimal task mapping approaches

t_{first} that was achieved by a task mapping approach. The y-values of the data points correspond to the task mapping approaches which resulted in the maximum t_{sys} for each benchmark. Each data point represents the results from one benchmark, and like-colored data points share an architecture and differ by task graph. For example, each of the five red squares represents the result of mapping one of the 9 processor architecture's task graphs to that architecture.

The main takeaway from these data is that, at utilization values below 50%, task mapping approaches that maximize t_{sys} will not maximize t_{first}. At 50% utilization, the t_{sys}-optimal task mapping approaches can only achieve about 70% of the maximum t_{first} in all of the architectures we tested. As utilization falls, these task mapping approaches are decreasingly able to yield good t_{first} results. Above 50% utilization, the fact that the t_{sys}-optimal task mapping approach results in about 96% of the maximum t_{first} means that simultaneous optimization is possible in the 64 processor architecture. However, this result only addresses half of the simultaneous optimization issue, and we still need to analyze the t_{first}-optimal task mapping approaches.

Figure 6.3 is set up in the same way as Figure 6.2 except it shows how well the task mapping approaches that maximize t_{first} are able to maximize t_{sys}. Thus, the y-axis now shows the percentage of the maximum t_{sys} that was achieved by a task mapping approach, and the y-values of the

data points correspond to the task mapping approaches that resulted in the maximum t_{first} for each benchmark. From this figure, it is clear that t_{first}-optimal task mapping approaches are unable to maximize t_{sys} at utilization values below 50%. While the effect is not as severe as in Figure 6.2, the fact that t_{first}-optimal task mapping approaches can only achieve about 86% of the maximum t_{sys} may be at odds with some design requirements. Similar to Figure 6.2, the ability to simultaneously optimize t_{sys} and t_{first} decreases with utilization but is likely acceptable for most designs at utilization values above 50%.

By combining the analysis of Figures 6.2 and 6.3, we conclude that there are no task mapping approaches which can simultaneously maximize t_{sys} and t_{first} at utilization values below 50%. This conclusion means that designers must make a decision about which lifetime metric is more important when the system is deployed since it is impossible to maximize both with a single task mapping approach in some scenarios. While t_{sys} and t_{first} can be simultaneously maximized at utilization values above 50%, the scenario in which utilization is below 50% is likely to be common for many systems as described in Section 6.1.

The reason that simultaneous maximization of t_{sys} and t_{first} is impossible in some cases is related to the fact that the two metrics require the system to be loaded in different ways. In order to maximize t_{sys}, the general strategy is to load the system in such a way that some components are reserved for use only when strictly necessary. t_{first} is generally maximized by loading the system in such a way that work is diverted from components that will wearout soon. It should be noted that these general strategies alone will not maximize t_{sys} and t_{first}, and a number of other factors must be accounted for in some fashion (e.g, using the meta-heuristic described in Chapter 5). These two strategies can produce very different task mappings, which in turn prevent t_{sys} and t_{first} from being simultaneously maximized, depending on the task graph and the architecture to which it is being mapped.

6.5.2 Increasing the Number of Pareto-optimal TM Approaches

Because t_{sys} and t_{first} cannot be simultaneously maximized in some cases, designers will be forced to choose from tradeoffs between the two. Figure 6.4 shows how well each of the task mapping approaches that we tested performs with respect to t_{sys} and t_{first}. The x-axis shows the inverse of t_{sys} (in years), and the y-axis shows the inverse of t_{first} (in years). Each data point on the plot

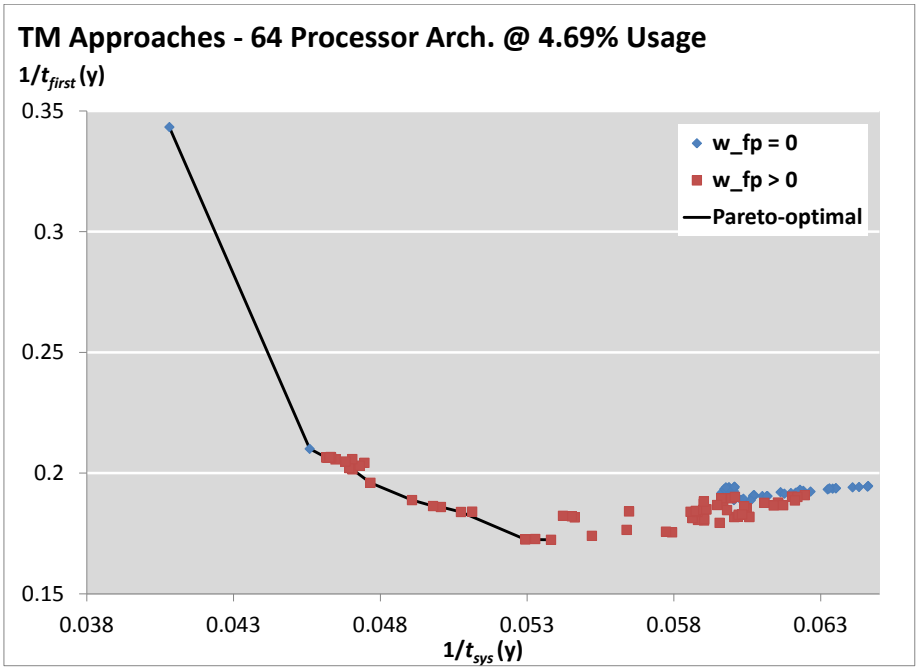

Figure 6.4: Results for all task mapping approaches on the 64 processor architecture with task graph 1

corresponds to one of the task mapping approaches as tested on the 64 processor benchmark using its smallest task graph; the two groups of data points will be discussed in Section 6.5.2. Points closer to the origin have higher t_{sys} and t_{first} values and represent better task mapping approaches. The points in the Pareto-optimal set of task mapping approaches are connected by a black line and represent the best possible tradeoffs between t_{sys} and t_{first}. Before deploying a system, a designer would generate a set of task mapping approaches and simulate them as we have described in this chapter. Then, the designer would be able to examine the range of Pareto-optimal task mapping approaches to determine which one has appropriate tradeoff for the design.

The task mapping approaches shown in Figure 6.4 are split into two groups. The group of blue diamonds represents task mapping approaches where w_{fp} was set to 0. The group of red squares represents task mapping approaches where w_{fp} was greater than 0. This second group of task mapping approaches were the ones that incorporated our floorplan-based task mapping heuristic. Figure 6.4 shows that the number of Pareto-optimal task mapping approaches is increased by the inclusion of our floorplan-based task mapping heuristic. When only considering the task mapping approaches which do not include this new heuristic (i.e, the blue diamonds), there are six Pareto-optimal task mapping approaches. The number of Pareto-optimal task mapping approaches increased to 15, a

factor of 2.5, when considering approaches which incorporated our floorplan-based task mapping heuristic.

We observed similar increases in the number of Pareto-optimal task mapping approaches in most of the other benchmarks. Details about these increases can be found in the last three columns of Table 6.2. Column four shows the number of Pareto-optimal points when approaches which use the floorplan-based heuristic are excluded ($w_{fp} = 0$). Column five shows the number of Pareto-optimal points when all approaches are included ($w_{fp} \geq 0$), and column six shows the factor of improvement. An increased number of Pareto-optimal task mapping approaches is a benefit since it gives a wider range of tradeoffs from which designers can select.

6.5.3 Improvements from Floorplan-based Task Mapping

We draw one final conclusion from our data that is alluded to in Figure 6.4. The group of red squares is closer to the x-axis than the group of blue diamonds, indicating that there are task mapping approaches which use our floorplan-based heuristic that provide better t_{first} than any of the task mapping approaches that do not use the heuristic. In the case of Figure 6.4, the maximum t_{first} given by an approach using the floorplan-based heuristic is 9.5% higher than the maximum t_{first} given by any approaches that do not use it. Across all of our benchmarks, the average improvement in t_{first} is 4.5% when using the floorplan-based heuristic. The exact improvement in t_{first} for each benchmark can be found in column three of Table 6.2.

6.6 Summary

In this chapter, we argued that there are two different definitions for lifetime that are relevant in different situations. Complicated recovery and reconfiguration mechanisms may not be able to be implemented in situations where design time is short or verification requirements, like those for safety-critical systems, are strict. Without these mechanisms, it makes no sense to measure lifetime as the amount of time between when the system is first switched on and when the system no longer has sufficient resources to accommodate the required task graphs (i.e, t_{sys}) since it is impossible to take advantage of any over-provisioned hardware resources. Instead, t_{first} is the lifetime metric of interest for these systems since they are only usable until a single component fails.

Table 6.2: Summary of floorplan-based task mapping heuristic results

Benchmark	Lifetime Inc		Num. Pareto-opt TMs		
(arch-TG)	t_{sys}	t_{first}	$w^{fp} = 0$	$w^{fp} \geq 0$	Inc.
9p-1	0.888	1.053	14	13	0.929
9p-2	0.804	1.058	6	12	2.0
9p-3	0.806	1.044	8	14	1.75
9p-4	0.878	1.034	5	16	3.2
9p-5	0.969	1.054	7	23	3.286
64p-1	0.884	1.095	6	15	2.5
64p-2	0.73	1.021	4	4	1.0
64p-3	0.733	1.066	6	7	1.167
64p-4	0.886	1.042	15	25	1.667
64p-5	0.99	1.041	7	20	2.857
100p-1	0.703	0.994	4	8	2.0
100p-2	0.701	1.056	5	7	1.4
100p-3	0.756	1.045	7	6	0.857
100p-4	0.869	1.039	12	17	1.417
100p-5	0.972	1.033	5	11	2.2

In light of there being two valid lifetime metrics, we explored how task mapping is able to maximize them. We tested a wide range of task mapping approaches that we created using a meta-heuristic that can take power, temperature, component capacity, wear, and system floorplan into account. Each task mapping approach was tested on 15 different benchmarks, which were created by applying 15 different task graphs to one of three hardware architectures.

Our results showed that when system utilization is under 50%, no single task mapping approach can simultaneously optimize t_{sys} and t_{first}; this result was replicated in all of our benchmarks. When system utilization is greater than 50%, some task mapping approaches are capable of providing maximum, or near-maximum, values for both lifetime metrics. However, task mapping approaches that maximize one of the lifetime metrics are increasingly unable to maximize the other as system utilization decreases. Scenarios in which system utilization is 50% or smaller can be common and may represent the majority of the lifetime of some systems. Thus, the selection of a task mapping approach is dependent on which lifetime metric is of interest, which is related to whether or not the system includes recovery and reconfiguration mechanisms, and the typical utilization of the system. In scenarios where t_{sys} and t_{first} cannot be simultaneously maximized, we show that there are a subset of approaches which are Pareto-optimal. These Pareto-optimal task

mapping approaches could be presented to a designer who would then choose one that resulted in an appropriate tradeoff between t_{sys} and t_{first} for the design.

We also presented a floorplan-based task mapping heuristic. This heuristic combines component wear information with spatial information from the system floorplan to map tasks to components thqt are far away from the parts of a system that will fail the soonest. We incorporated this heuristic into our overarching meta-heuristic and compared approaches which did and did not use the floorplan-based heuristic. Adding the floorplan-based heuristic increased the number of Pareto-optimal task mapping approaches in many of our benchmarks, which effectively increases the range of tradeoffs from which a designer can select. Finally, we found that task mapping approaches that use the floorplan-based heuristic can increase t_{first} by up to 10% when compared to task mapping approaches that didn't.

Chapter 7

Related Work

This chapter reviews two areas of existing work in the literature: research from which we draw foundational concepts and research that studies system lifetime optimization.

7.1 System Lifetime Simulation

Our system lifetime simulator is fundamental to the evaluation of our task mapping techniques, and its design relies on prior work in a wide range of subjects outside the scope of this thesis. This section discusses the published results and methods that we have incorporated into our simulator to give it a high level of detail.

7.1.1 Failure Mechanism Modeling

All of the work in this thesis relies on mathematical models of failure mechanisms developed in the literature. The models for electromigration, time-dependent dielectric breakdown, and thermal cycling were originally described in [20] and [21]. The work in [22] and [19] provides information about how those models can be used in the context of a system-level simulation. Specifically, this work presents the idea of using the mathematical models to create lognormal distributions, which are then sampled by a Monte Carlo simulation.

7.1.2 Power Modeling

In order for the failure mechanism models to produce useful results, we must be able to compute component power dissipation and temperature accurately. Although our work focuses on optimizing system-level metrics (i.e, t_{sys} and t_{first}), we use type-specific component-level power models to increase the fidelity of our simulation. We rely on existing work for these power models because the details of memory and network switch architecture are not relevant to the work presented here.

[15] describes CACTI, a tool developed to model various characteristics of caches and memories. The models in CACTI are validated against actual manufactured devices to prove their accuracy. We use CACTI as the source of our power models and floorplan areas for memory components in our benchmark systems.

ORION is a specialized tool for modeling the power and area characteristics of communications components in NoC architectures [16]. It covers a range of manufacturing process technologies and operating conditions for network links and switches. We integrated ORION's source code into our system lifetime simulator and used the provided interface to give us direct and fast access to the models.

7.1.3 Floorplanning

Along with component power dissipation values, Hotspot takes the floorplan of the system as input. We again turn to existing work to create floorplans for our benchmarks because it it outside the scope of our work to optimize the floorplan for lifetime.

Over the course of our work, we have used two different tools to create system floorplans. Initially, we used a simulated annealing-based tool called Parquet to build the floorplans for our benchmarks [32]. Later, we turned to BloBB for floorplanning [17]. Because BloBB is a constructive floorplanner, it has a much shorter runtime than Parquet and is still able to produce compact floorplans. And, because the number of components in our designs is relatively small in the context of floorplanning problems, we could obtain sufficient results from BloBB without incurring the runtime cost of Parquet. Both tools accept the dimensions of each component as input, and produce a block-level layout of the system.

There is a slight disconnect between the floorplans produced by Parquet/BloBB and the floor-

plans accepted by Hotspot. The outputs of the floorplanning tools may have sections of empty space in them where components did not fit together perfectly, and Hotspot expects a floorplan without any empty space. To solve this problem, we created a greedy algorithm which fills any empty space in the floorplan with a set of rectangular components. All of these space-filling components have zero power dissipation, and therefore, they do not contribute any heat to the system.

7.1.4 Condor

As mentioned earlier in this thesis, the level of detail in our system lifetime simulator and the range of experiments that we ran would not have been possible without the use of distributed computing. In our case, Condor was used to manage the distributed computing resources provided by the department [25]. A Condor environment is composed of a central computing resource that acts as the manager and numerous other computing resources that act as helpers. The manager and helpers must all be able to communicate with clients through a network.

The Condor manager is a continuously running process that listens for job submissions from clients. A job submission usually includes the compiled binary of the program to be run, a set of arguments to pass to the program, the set of input files required by the program, the set of output files to be collected when the program terminates, and a description of the type of computing resource that should be used to execute the program. A client can submit a large number of jobs, each of which performs some portion of an expensive computation, by changing only the arguments or input files. The Condor manager will then dispatch individual jobs to computing resources on the network based on the requirements set by the client and the amount of work the client has submitted recently. The computing resources are shared evenly, or unevenly according to a pre-defined policy, across all clients by measuring the amount of time being spent on each job.

Each computing resource runs a Condor helper process which accepts jobs sent to it from the Condor manager. Given a job to execute, the helper process sets up an environment for the specified program, executes it, and returns the requested output files to the client. The helper process is also responsible for communicating the state of the computing resource to the manger so that the manager has a complete view of all computing resources on the network.

7.2 Ant Colony Optimization

Our design-time task mapping optimization uses ACO to find near-optimal task mappings, but we are not responsible for the creation of this algorithm. ACO was first introduced as a generalized optimization technique in [33]. Since then, a significant body of work has been published in which ACO is applied to different types of problems. While the applications of ACO are too numerous to list here, there are cases where it has been applied to solving problems similar in nature to ours. For example, the authors use ACO to solve the task mapping and scheduling problem while optimizing for performance in [34]. [35] also solves the performance-aware task mapping problem with ACO. Finally, [36] uses ACO to solve the task mapping problem but with a focus on energy instead of performance.

7.3 Wear Sensors

Some of the work we present in this thesis relies on the existence of on-chip wear sensors to provide data to the task mapping approaches. The design and performance of wear sensors is discussed in several pieces of existing work. [4] presents a method for creating a wear sensor which tracks TDDB via critical path delay and is able to accurately predict when microarchitectural components fail with a small overhead cost. [37] discusses wear sensors which can be placed inside of flip-flops to track system wear due to negative bias temperature instability (NBTI), and although we do not model the effects of NBTI in this paper, our framework allows us to include it in the future. [38] details a slightly different method for sensors which track wear caused by NBTI. Finally, [39] explains how circuitry can be added to SRAM cells in order to track accumulated wear in memories.

7.4 Task Mapping for Non-Lifetime Metrics

Task mapping optimization can be formulated to focus on a number of different metrics. In this section, task mapping techniques that do not directly optimize system lifetime are discussed. These techniques are further classified into those which optimize metrics unrelated to system lifetime and those which optimize system lifetime indirectly through a related metric.

7.4.1 Metrics Unrelated to Lifetime

In Chapters 5 and 6, one of the comparison approaches we use is power-aware task mapping, and this topic is explored extensively in the literature. [40] proposes a task mapping and scheduling technique to minimize power consumption due to leakage effects. [41] explains a runtime power-aware task mapping technique where a set of task mappings that minimize average power consumption is precomputed at design time. Then, the best task mapping is chosen from the set at runtime according to current workload on the system. [42] builds on the concept of power-aware task mapping to solve energy optimal task scheduling with integer linear programming. In a special case of energy-aware task mapping, [43] and [44] optimize task mapping and scheduling to maximize battery life in sensor networks and distributed mobile applications, respectively.

In heterogeneous chip multiprocessors, changing the task mapping and scheduling can also impact the performance of the system. In [45], the authors develop a method to create a number of static schedules at design time that satisfy performance constraints, each of which is tailored to a pattern of manufacturing variation. The variation of the manufactured systems can be measured when they are produced, and an appropriate task scheduling can be applied to each individual system. The problems of task mapping and task scheduling are solved simultaneously in [46]. An integer programming forumulation is used to perform communication/performance-aware task mapping while a constraint programming formulation is applied to task scheduling. The two solvers communicate infeasible solutions to each other as a way to converge on the optimal combination of solutions. Using a completely different strategy, [47] improves performance via a task mapping approach designed to avoid permanent, transient, and intermittent faults. The three types of faults in this work are combined into a single, general model for component faults.

A few researchers have chosen to investigate how task mapping can be used to improve system availability by decreasing the rate at which transient faults occur. [48] builds on a commercial tool to create a simulator used to measure system reliability due to transient faults, and performs some exploration of how a task mapping approach can be evaluated using the simulator. Transient faults are addressed by co-optimizing the task mapping and system redundancy allocation via a multi-objective genetic algorithm in [49]. [50] points to communication energy and the number of tasks being migrated as causes of transient faults and seeks to reduce both using task mapping.

Task mapping optimization can also be used to control the temperature of a chip multiprocessor. Given a set of repeating tasks, [51] finds the sequence that minimizes system temperature. The authors extend their approach by exploring how voltage scaling settings can be altered to provide additional benefit to the system temperature. [30] proposes heuristics for temperature-aware task mapping and scheduling alongside an approach to temperature-aware floorplanning. The combined approaches are used to show that temperature cannot be indirectly optimized using power-aware techniques. Using an extended temperature model, [52] performs temperature-aware scheduling for system manufactured in a 3-dimensional (i.e, layered) process. [53] also examines temperature-aware task mapping in 3-dimensional ICs while specifically focusing on the tradeoffs between using intra-layer wires and inter-layer wires (i.e, through-silicon-vias) for communication. The difference between these temperature-aware task mapping techniques and those in the next section is that these make no claims about impact on system lifetime.

7.4.2 Metrics that Indirectly Optimize Lifetime

A number of task mapping techniques claim to optimize lifetime indirectly via some more easily computed metric. While the experimental results we present in this thesis dispute such claims, the related work is discussed here for reference.

Temperature optimization is claimed to be a proxy for system lifetime optimization by several works in the literature. [18] details a runtime task mapping technique that improves temperature profiles and claims that system reliability will be improved as a result, but no lifetime data are provided. [54] improves upon the previous work by modeling system lifetime so that the effect of temperature-aware task mapping on system lifetime can be measured, but the overall approach still optimizes lifetime indirectly via temperature. [55] determines a temperature profile that will maximize lifetime at runtime and then changes the task mapping and scheduling to achieve that temperature profile. The authors also present a method to adaptively control how often the task mapping is changed at runtime to reduce any performace impact. In [56], task migration is directed to reduce average temperature and smooth thermal gradients without sacrificing performance, and many details about the mechanics of task migration are provided, which can be used to show that our task mapping techniques are feasible. Similar to other work, system lifetime improvment is claimed by the authors without any measurement. In addition to another temperature-aware task mapping

technique, [57] also describe an accurate and fast steady-state dynamic temperature analysis as an alternative to Hotspot. The authors examine the effects of their approach on system lifetime, but thermal cycling is the only modeled failure mechanism.

Another collection of work proposes that system lifetime can be optimized by controlling the utilization of components. [58] employs a control theoretic approach to achieve utilization targets that result in optimal system lifetime for homogeneous soft-real-time systems. Instead of controlling component utilization via the task mapping, the authors manage the DVFS settings in the system as a way to change apparent utilization. [59] searches for task mappings that create uniform system utilization in an effort to reduce hotspots and improve lifetime, but system lifetime is never evaluated. Another attempt to improve lifetime in [60] focuses on reducing the load on the NoC communication fabric, thereby implying that it is the bottleneck for system lifetime. The approach was found to decrease the amount of time required to complete tasks, and the authors claim that system lifetime is inversely proportional to computation time; there is no strong justification for that simplification, however.

7.5 Task Mapping for Lifetime

This thesis contains some of the earliest published work on lifetime-aware task mapping. In the time since those original publications, the volume of work in this research area has grown in step with the significance of the problem of degrading system lifetime. This section summarizes some of the more recent publications related to lifetime-aware task mapping and closely related topics.

Perhaps the most comprehensive work on lifetime-aware task mapping can be found in [61] with the only caveat being that it does not include a design time technique. The authors propose a combination of greedy and genetic algorithms to optimize task mapping and scheduling for lifetime, and detailed models for TDDB and NBTI are used. [62] considers NBTI as the only failure mechanism and co-optimizes power consumption and lifetime in a performance neutral manner. [63] combines a lifetime-aware task mapping approach based on simulated annealing with solution space pruning to improve runtime. [64] suggests that lifetime-optimal task mappings can be selected using convex optimization and includes the possibility of communication link failure due to wearout in the system lifetime model. The last two approaches select a static mapping and scheduling that do

not change at runtime, use Weibull distributions to model failure mechanisms, and do not support lifetime modeling of systems with slack.

There are also several works in the literature that use task mapping to co-optimize system lifetime and one of several additional objectives. [65] presents a runtime task mapping algorithm where the metric being co-optimized is communication energy. [66] effectively improves upon the previous work by using Hotspot to model component temperatures, thereby improving the accuracy of the system lifetime evaluation. The work is further extended in [67] where a genetic algorithm is created to co-optimize transient faults and system lifetime. [68] also targets transient faults and system lifetime for co-optimization, but uses hardware/software partitioning to reach that goal.

To summarize, ours is the only work that accounts for all of the following items:

- Design time and runtime task mapping approaches
- System architectures with and without slack
- Components that experience temperature changes, and subsequent changes in the rate at which they accumulate wear, as the task mapping changes
- Failure mechanisms that are modeled accurately using lognormal distributions instead of less accurate Weibull distributions

As a caveat, our system lifetime simulator uses a more constrained model for applications, in that it is limited to streaming applications, than some of the other work described above.

Other work defines task mapping as a one-to-many mapping of tasks to components, in contrast to the one-to-one definition we use throughout this thesis. A one-to-many mapping implies a system structure in which a task is executed on multiple components and the correct result is selected to guard against problems in a single component (i.e, modular redundancy). [69] describes a novel task mapping approach for use in systems with modular redundancy, which improves both the amount of time during which a system is fault tolerant and total system lifetime. Co-optimization of performance and reliability in the context of modular redundancy is proposed in [70], but only transient failures are modeled.

7.6 Other System-Level Lifetime Optimizations

In addition to task mapping, a number of other system-level techniques have been leveraged to improve system lifetime. [71] explores ways to cost-effectively add slack to a system in order to increase both its lifetime and its manufacturing yield. [72] proposes a complete system synthesis flow that is designed to be lifetime-aware through temperature optimization. Wear sensors are modeled to drive a core gating algorithm that favors components with less wear in [73]. Routing optimization is addressed in [74], which uses a dynamic programming technique to route communication along the paths that contain network resources with the least amount of wear. Finally, [75] attempts to mitigate the closely related problem of reliability through dynamic voltage scheduling strategies that lower the occurence of single event upsets.

Chapter 8

Conclusions

System lifetime is becoming a primary target for optimization, as performance, cost, temperature, and power have been in the past, because of the rapidly increasing effects of wearout faults on system lifetime due to advances in manufacturing process size. Without mitigation, the decrease in system lifetime due to increases in the rate of wear will be significant enough to prevent cutting-edge manufacturing processes from being used in particular applications. There exist certain manufacturing techniques and guidelines that address these concerns, but they tend to be most effective only when the use case of a system is well understood before it is manufactured, as in the case of ASIC design. However, the increasingly high cost of ASIC design means that many designers are turning to generalized chip multiprocessors for solutions. New techniques for improving system lifetime are required in order for chip multiprocessors to continue to be usable for a wide range of applications.

This thesis addresses the problem of embedded chip multiprocessor lifetime by proposing and evaluating two techniques for lifetime-aware task mapping. Our work is the first to examine the differences between the design time and runtime task mapping problems and to exploit the characteristics of each as a method of optimizing lifetime. While a system is being designed, there are significant computational resources available to the designer that can be used for optimization, but there is potentially little known about what will happen to the system at runtime. For our first contribution, we use ant colony optimization to exercise the time and resources that the designer has budgeted for optimization and search the large space of task mappings for initial solutions that are near-optimal with respect to system lifetime. In contrast to the design-time scenario, there is less time and fewer resources available to compute a task mapping at runtime, but there is a significant

amount of information available about the actual state of the system. Our second contribution is a meta-heuristic which considers data from wear and temperature sensors that are built into the hardware as well as information about the application(s) currently running on the system and quickly optimizes the task mapping given that state.

The overall contribution of this work is that the evaluations of our techniques demonstrate that task mapping has a significant effect on system lifetime and that we are able to choose task mappings that improve system lifetime at both design time and runtime. The advantage our task mapping techniques have over manufacturing-level techniques is that we are able to specifically optimize for the actual application(s) being run on the system. Further, our runtime task mapping technique allows us to dynamically manage system lifetime using real-time feedback from hardware sensors as the system is in use. This ability to do dynamic management is unique to our runtime task mapping technique and cannot be realized through either manufacturing-level optimizations or system-level optimizations that suggest architectural changes.

We also analyze how our lifetime-aware task mapping techniques compare to task mapping techniques that optimize for different, but related, metrics such as temperature and power. This analysis leads to another contribution, which is a series of experiments that shows system lifetime cannot be indirectly optimized by techniques that primarily focus on other metrics. Though system lifetime has strong dependencies on temperature and power, neither temperature- nor power-aware task mapping techniques result in near-optimal system lifetime. We discovered an interesting corollary to this contribution, as well: lifetime-aware task mapping is able to produce results which are near-optimal in terms of temperature. Our studies of lifetime-aware task mapping are supplemented by an investigation of how well different lifetime metrics can be co-optimized, and this study serves as the final contribution of this thesis.

While all of the results presented in this thesis are based on simulations and not measurements of actual systems, it is important that the learnings detailed herein be applicable in the real world. To that point, all of the technology required to implement our techniques is already available, so designers can draw on this work immediately to begin exploring how to leverage task mapping as a way to extend the lifetime of their systems. The fact that our techniques span a broad range of use cases gives designers the flexibility to choose the subset of optimizations that best fits their requirements. Our work enables designers to make suitable tradeoffs between design effort, system cost,

and optimization quality to easily reach the desired lifetime targets for their system. In addition, our system lifetime simulator can be used to evaluate the effects of design choices independent of task mapping on system lifetime. The flexibility of our system lifetime simulator, along with the analysis of lifetime-aware task mapping presented in this thesis, opens the door to new areas of research in task mapping and generalized system-level lifetime optimization. Some potential directions for this future research are discussed in Section 8.2.

8.1 Summary

Chapter 1 of this thesis begins by describing trends in integrated circuit manufacturing which contribute to the degradation of system lifetime. We cite these trends as motivation to solve the problem of system lifetime optimization. This chapter outlines our approach to improving system lifetime via task mapping and the experiments we used to validate that approach.

Then, Chapter 2 covers much of the required background information and theoretical foundations of our work. The purpose of this chapter is to explain to the reader why our proposed task mapping techniques should be successful before discussing the details of those techniques. We give an overview of the type of system that we target with this work and the assumptions we make about such systems. We also provide precise definitions of the lifetime metrics for which we are optimizing and explain how they can be influenced by task mapping and the presence of over-provisioned system resources.

Chapter 3 provides details about the system lifetime simulator that we used to evaluate our task mapping approaches. The simulator is built around a Monte Carlo simulation that models the statistical nature of the failure mechanisms that most significantly affect system lifetime: electromigration, time-dependent dielectric breakdown, and thermal cycling. Each sample in the Monte Carlo simulation represents an instance of the system being examined under a unique set of parameters, which define the exact effect that each failure mechanism will have on each component in the system. The sample is simulated by advancing time until a component fails or until the task mapping needs to be recomputed, updating the amounts of wear accumulated by each component in that time step, and determining whether or not the system can continue to operate given the remaining set of resources. By averaging together the results of all of the samples, we can obtain a precise

estimate of the lifetime of an arbitrary hardware architecture which is executing an arbitrary set of applications under a task mapping approach of interest.

In Chapter 4, we present our approach of design-time task mapping optimization. We use ant colony optimization to search the large space of initial task mappings to find solutions which are near optimal in terms of system lifetime. Ant colony optimization is a graph-based search technique which iteratively improves the solution quality while avoiding local minima in the solution space. We compare our solution to an exhaustive search for sufficiently small designs and to a simulated annealing-based approach for larger designs to prove that our task mappings are high quality. Also, we use this experimental framework to examine how well a more traditional temperature-aware task mapping approach optimizes lifetime compared to our lifetime-aware approach. These results are generalized to show that temperature cannot be used a proxy when optimizing for lifetime, but optimizing for lifetime directly will result in both near-optimal system lifetime and temperature.

Our meta-heuristic for runtime task mapping optimization is described in Chapter 5. Due to the resource and time constraints on finding a task mapping while the system is running, we forego a detailed optimization algorithm in favor of a carefully designed set of heuristics that can be evaluated quickly. Each individual heuristic focuses on a single metric that impacts system lifetime; wear, temperature, power, and component capacity are all represented. A series of experiments is used to determine how to best weight the individual heuristics for use in the overarching meta-heuristic and how often the meta-heuristic should be used to change the task mapping. These experiments are conducted across several applications while the system architecture is held constant and across several system architectures while the application is held constant. We use the results to conclude that our meta-heuristic can significantly improve system lifetime compared to any other runtime task mapping approach which only considers a single metric.

Chapter 6 contains a discussion of how well task mapping is able to co-optimize different lifetime metrics. This is an important topic because the design requirements and use case for a particular system will determine whether it is more important to optimize t_{sys} or t_{first}. While the previous chapters focus on optimizing t_{sys} and report the corresponding t_{first} as secondary information, this chapter explores how our optimization techniques would need to be re-framed in scenarios where t_{first} is more important than t_{sys}. This chapter also introduces floorplan-aware task mapping as an additional element in our runtime task mapping meta-heuristic. This new element jointly con-

siders the amount of wear on a component together with the component's physical location in the system to further improve system lifetime and provide a wider range of task mappings from which a designer can choose.

Finally, Chapter 7 covers related work in the literature regarding lifetime simulation concepts, ant colony optimization, wear sensors, task mapping, and other system-level lifetime optimizations. We include a discussion of the differences in assumptions between our approaches and competing approaches along with a summary of the existing work on which we rely for foundational concepts.

8.2 Directions for Future Work

To conclude this thesis, we summarize extensions of this work that could be interesting research areas in the future.

Machine learning-based runtime task mapping. Our runtime task mapping meta-heuristic is composed of individual heuristics that capture information about the system state that affects lifetime. However, it is possible that system lifetime could be further improved by recognizing a more complex pattern in the system state and choosing the task mapping based on that. Machine learning algorithms are able to find complex patterns in data sets that may not necessarily be recognizable by a human. We could employ a machine learning algorithm in our runtime task mapping technique in two different ways. First, we could create a hybrid approach in which the machine learning algorithm would be responsible for changing the weights in the meta-heurstic as the system is running. This approach would account for the fact that the optimal meta-heuristic weighting may change as the system ages, and machine learning may be uniquely able to determine the best time for any such changes to occur. Second, we could replace the meta-heuristic completely and allow a machine learning algorithm to alter the task mapping directly. In this case, our runtime task mapping may have greater potential to improve system lifetime since the task mappings it selects would not be influenced by the set of individual heuristics we chose to use in the meta-heuristic.

Application of the techniques to place and route for FPGAs. Although the techniques in this thesis were developed to find suitable locations in for software tasks in embedded chip multiprocessors, they could also be extended and used to find suitable locations for synthesized hardware blocks in an FPGA fabric. FPGAs are manufactured using the same types of processes as embed-

ded chip multiprocessors, and so FPGAs will also be increasingly susceptible to wearout faults and shorter lifetime. Our design-time techniques could be used during the floorplanning stage of an FPGA design flow to find arrangements of blocks which maximize lifetime. Similarly, our runtime techiques could be combined with the ability to partially reconfigure an FPGA as it is running to move hardware blocks around the fabric such that the accumulation of wear is distributed and lifetime is increased. The biggest challenge in making this extension would lie in converting our task mapping approaches from working in a discrete solution space (where each task gets mapped to a component in a set location) to working in a continuous solution space (where each hardware block could be placed at any location in the fabric). Alternatively, our system lifetime simulator could be incorporated into current floorplanning and placement algorithms to make them lifetime-aware.

Comparisons to manufacturing-level techniques. In Chapter 1, we state that manufacturing-level techniques that improve lifetime tend to do so by adding guardband across the entire system. It is difficult to quantify the added cost of the guardband because integrated circuits are rarely designed without the suggested guardband. The only way to measure the cost of the guardband would be to perform the physical design of an integrated circuit according to a set of design rules that did not include any rules related to guardband. Then, we could measure the difference in area between a system with guardband and a system without guardband. This cost difference due to guardband could then be compared to the area added by implementing a wear sensor in each component in the system, which is the physical cost of our best runtime task mapping approach. If the cost of the wear sensors in all components turned out to be too high, further experiments could be completed to determine the subset of components to instrument with wear sensors to achieve Pareto-optimal cost/lifetime tradeoffs.

Effects of inaccurate wear sensors. Our work assume that each wear sensor in the system can accurately measure the amount of wear on the component to which it belongs. This assumption is somewhat unrealistic because it is likely that a real wear sensor would only provide an approximation of how much wear a component has accumulated or how soon the component will fail. Our system lifetime simulator could be augmented to model wear sensor inaccuracies by perturbing the calculated wear values before passing them along to the task mapping techniques. With this change, additional experiments could be conducted to determine the degree to which the improvements in system lifetime are affected. Of particular interest would be the new optimal weighting

of our runtime meta-heuristic under inaccurate wear sensors. One would expect to observe a direct relationship between the accuracy of the wear sensors and the weight given to the wear metrics in the meta-heuristic; low accuracy should lead to low weights, and high accuracy should lead to high weights.

System lifetime simulator improvements. While the system lifetime simulator described in this thesis is quite flexible and models many details relevant to system lifetime, there are three facets of the simulator that could be improved. First, the simulator could model applications that start at a given time and continue for a finite amount of time instead of only modeling a set of applications that execute indefinitely. This change would allow more realistic workloads to be modeled for some use cases, like cellular phones. Second, the simulator could use a more realistic DVFS model for processors. Currently, the simulator models only two power states for processors, and the realism of the simulation could be improved becuase state-of-the-art processors typically have a greater number of power states. Third, the simulator could model additional types of components. It is increasingly common for embedded chip multiprocessors to include specialized hardware accelerators that are able to execute a subset of the tasks in an application. Task mapping could be given the choice between executing a task on a general purpose processor or on a compatible hardware accelerator by altering the requirements of the task based on the type of component to which it is mapped.

System lifetime simulator speedup. While we do exploit the inherent parallelism of Monte Carlo simulation, the granularity at which it is exploited is relatively coarse and limited by our use of distributed computing. By re-engineering some parts of the simulator to allow for more finely grained parallelism, the runtime of the simulator could be decreased. One method of doing this involves creating an implementation of the simulator that targets graphics processing units (GPUs) as an execution platform. The hardware architecture of a GPU is designed such that large numbers of threads can run simultaneously. If a thread was created for each Monte Carlo sample, it is possible that hundreds of samples could run in parallel on a single system. In fact, we did create an initial implementation of our system lifetime simulator for GPUs, and the preliminary results were promising. However, it became clear that the use of distributed computing would be easier in the short term, and so further work is required to complete the GPU implementation.

Bibliography

[1] "International technology roadmap for semiconductors, 2013 edition: Process integration, devices, and structures." http://www.itrs.net/ITRS%201999-2014%20Mtgs,%20Presentations%20&%20Links/2013ITRS/2013Chapters/2013PIDS_Summary.pdf.

[2] J. Srinivasan, S. V. Adve, P. Bose, J. Rivers, *et al.*, "The impact of technology scaling on lifetime reliability," in *Dependable Systems and Networks, 2004 International Conference on*, pp. 177–186, IEEE, 2004.

[3] W. J. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," in *Design Automation Conference, 2001. Proceedings*, pp. 684–689, IEEE, 2001.

[4] J. Blome, S. Feng, S. Gupta, and S. Mahlke, "Self-calibrating online wearout detection," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 109–122, IEEE Computer Society, 2007.

[5] H.-c. Kim, D.-s. Yi, J.-y. Park, and C.-h. Cho, "A bisr (built-in self-repair) circuit for embedded memory with multiple redundancies," in *VLSI and CAD, 1999. ICVC'99. 6th International Conference on*, pp. 602–605, IEEE, 1999.

[6] B. H. Meyer, *Cost-effective lifetime and yield optimization for NoC-based MPSoCs*. PhD thesis, Carnegie Mellon University, 2009.

[7] T. Chantem, X. S. Hu, and R. P. Dick, "Temperature-aware scheduling and assignment for hard real-time applications on mpsocs," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 19, no. 10, pp. 1884–1897, 2011.

[8] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware microarchitecture," *ACM SIGARCH Computer Architecture News*, vol. 31, no. 2, pp. 2–13, 2003.

[9] M. Gomaa, M. D. Powell, and T. Vijaykumar, "Heat-and-run: leveraging smt and cmp to manage power density through the operating system," in *ACM SIGARCH Computer Architecture News*, vol. 32, pp. 260–270, ACM, 2004.

[10] S. Zhang and K. S. Chatha, "Approximation algorithm for the temperature-aware scheduling problem," in *Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, pp. 281–288, IEEE Press, 2007.

[11] A. S. Hartman, D. E. Thomas, and B. H. Meyer, "A case for lifetime-aware task mapping in embedded chip multiprocessors," in *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pp. 145–154, IEEE, 2010.

[12] A. S. Hartman and D. E. Thomas, "Lifetime improvement through runtime wear-based task mapping," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pp. 13–22, ACM, 2012.

[13] J. C. Smolens, B. T. Gold, J. C. Hoe, B. Falsafi, and K. Mai, "Detecting emerging wearout faults," in *Proc. of Workshop on SELSE*, 2007.

[14] `http://www.arm.com/products/processors/index.php`.

[15] S. Thoziyoor, N. Muralimanohar, and N. P. Jouppi, "Cacti 5.0," *HP Laboratories, Technical Report*, 2007.

[16] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi, "Orion 2.0: a fast and accurate noc power and area model for early-stage design space exploration," in *Proceedings of the conference on Design, Automation and Test in Europe*, pp. 423–428, European Design and Automation Association, 2009.

[17] H. H. Chan and I. L. Markov, "Practical slicing and non-slicing block-packing without simulated annealing," in *Proceedings of the 14th ACM Great Lakes symposium on VLSI*, pp. 282–287, ACM, 2004.

[18] A. K. Coskun, T. S. Rosing, and K. Whisnant, "Temperature aware task scheduling in mpsocs," in *Proceedings of the conference on Design, automation and test in Europe*, pp. 1659–1664, EDA Consortium, 2007.

[19] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "Exploiting structural duplication for lifetime reliability enhancement," in *ACM SIGARCH Computer Architecture News*, vol. 33, pp. 520–531, IEEE Computer Society, 2005.

[20] J. S. S. T. Association *et al.*, "Failure mechanisms and models for semiconductor devices," *JEDEC Publication JEP122-B*, 2003.

[21] E. Wu, J. Sune, W. Lai, E. Nowak, J. McKenna, A. Vayshenker, and D. Harmon, "Interplay of voltage and temperature acceleration of oxide breakdown for ultra-thin gate oxides," *Solid-State Electronics*, vol. 46, no. 11, pp. 1787–1798, 2002.

[22] J. Srinivasan, *Lifetime reliability aware microprocessors*. PhD thesis, University of Illinois at Urbana-Champaign, 2006.

[23] Z. P. Gu, C. Zhu, L. Shang, and R. P. Dick, "Application-specific mpsoc reliability optimization," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, no. 5, pp. 603–608, 2008.

[24] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[25] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: The condor experience," *Concurrency-Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.

[26] H. Ramalhinho Dias Lourenço and D. Serra, "Adaptive approach heuristics for the generalized assignment problem," *Economics Working Paper*, vol. 288, 1998.

[27] E. G. Jaspers and P. H. De With, "Chip-set for video display of multimedia information," *Consumer Electronics, IEEE Transactions on*, vol. 45, no. 3, pp. 706–715, 1999.

[28] J. Kneip, S. Bauer, J. Vollmer, B. Schmale, P. Kuhn, and M. Reissmann, "The mpeg-4 video coding standard-a vlsi point of view," in *Signal Processing Systems, 1998. SIPS 98. 1998 IEEE Workshop on*, pp. 43–52, IEEE, 1998.

[29] B. H. Meyer, A. S. Hartman, and D. E. Thomas, "Cost-effective slack allocation for lifetime improvement in noc-based mpsocs," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pp. 1596–1601, IEEE, 2010.

[30] Y. Xie and W.-L. Hung, "Temperature-aware task allocation and scheduling for embedded multiprocessor systems-on-chip (mpsoc) design," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 45, no. 3, pp. 177–189, 2006.

[31] R. P. Dick, D. L. Rhodes, and W. Wolf, "Tgff: task graphs for free," in *Proceedings of the 6th international workshop on Hardware/software codesign*, pp. 97–101, IEEE Computer Society, 1998.

[32] S. N. Adya and I. L. Markov, "Fixed-outline floorplanning: Enabling hierarchical design," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 11, no. 6, pp. 1120–1135, 2003.

[33] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: optimization by a colony of cooperating agents," *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, vol. 26, no. 1, pp. 29–41, 1996.

[34] C.-W. Chiang, Y.-C. Lee, C.-N. Lee, and T.-Y. Chou, "Ant colony optimisation for task matching and scheduling," in *Computers and Digital Techniques, IEEE Proceedings on*, vol. 153, pp. 373–380, IET, 2006.

[35] M. Bank, U. Hönig, and W. Schiffmann, "An aco-based approach for scheduling task graphs with communication costs," in *Parallel Processing, 2005. ICPP 2005. International Conference on*, pp. 623–629, IEEE, 2005.

[36] P.-C. Chang, I.-W. Wu, J.-J. Shann, and C.-P. Chung, "Etahm: An energy-aware task allocation algorithm for heterogeneous multiprocessor," in *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pp. 776–779, IEEE, 2008.

[37] M. Agarwal, V. Balakrishnan, A. Bhuyan, K. Kim, B. Paul, W. Wang, B. Yang, Y. Cao, and S. Mitra, "Self-calibrating online wearout detection," in *Proc. Int. Test Conf*, pp. 1–10, 2008.

[38] B. Zandian, W. Dweik, S. H. Kang, T. Punihaole, and M. Annavaram, "Wearmon: Reliability monitoring using adaptive critical path testing," in *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pp. 151–160, IEEE, 2010.

[39] F. Ahmed and L. Milor, "Analysis and on-chip monitoring of gate oxide breakdown in sram cells," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 20, no. 5, pp. 855–864, 2012.

[40] J.-J. Chen, H.-R. Hsu, and T.-W. Kuo, "Leakage-aware energy-efficient scheduling of real-time tasks in multiprocessor systems," in *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pp. 408–417, IEEE, 2006.

[41] A. Schranzhofer, J.-J. Chen, and L. Thiele, "Dynamic power-aware mapping of applications onto heterogeneous mpsoc platforms," *Industrial Informatics, IEEE Transactions on*, vol. 6, no. 4, pp. 692–707, 2010.

[42] M. Goraczko, J. Liu, D. Lymberopoulos, S. Matic, B. Priyantha, and F. Zhao, "Energy-optimal software partitioning in heterogeneous multiprocessor embedded systems," in *Proceedings of the 45th annual design automation conference*, pp. 191–196, ACM, 2008.

[43] S. Abdelhak, C. S. Gurram, S. Ghosh, and M. Bayoumi, "Energy-balancing task allocation on wireless sensor networks for extending the lifetime," in *Circuits and Systems (MWSCAS), 2010 53rd IEEE International Midwest Symposium on*, pp. 781–784, IEEE, 2010.

[44] J. Li, M. Qiu, J.-W. Niu, and T. Chen, "Battery-aware task scheduling in distributed mobile systems with lifetime constraint," in *Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific*, pp. 743–748, IEEE, 2011.

[45] L. Huang and Q. Xu, "Performance yield-driven task allocation and scheduling for mpsocs under process variation," in *Proceedings of the 47th Design Automation Conference*, pp. 326–331, ACM, 2010.

[46] M. Ruggiero, A. Guerri, D. Bertozzi, F. Poletti, and M. Milano, "Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip," in *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pp. 3–8, European Design and Automation Association, 2006.

[47] C.-L. Chou and R. Marculescu, "Farm: Fault-aware resource management in noc-based multiprocessor platforms," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pp. 1–6, IEEE, 2011.

[48] Z. Wang, C. Chen, P. Sharma, and A. Chattopadhyay, "System-level reliability exploration framework for heterogeneous mpsoc," in *Proceedings of the 24th edition of the great lakes symposium on VLSI*, pp. 9–14, ACM, 2014.

[49] J. Huang, A. Raabe, K. Huang, C. Buckl, and A. Knoll, "A framework for reliability-aware design exploration on mpsoc based systems," *Design Automation for Embedded Systems*, vol. 16, no. 4, pp. 189–220, 2012.

[50] A. Das, A. Kumar, and B. Veeravalli, "Communication and migration energy aware design space exploration for multicore systems with intermittent faults," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 1631–1636, EDA Consortium, 2013.

[51] R. Jayaseelan and T. Mitra, "Temperature aware task sequencing and voltage scaling," in *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, pp. 618–623, IEEE Press, 2008.

[52] X. Zhou, Y. Xu, Y. Du, Y. Zhang, and J. Yang, "Thermal management for 3d processors via task scheduling," in *Parallel Processing, 2008. ICPP'08. 37th International Conference on*, pp. 115–122, IEEE, 2008.

[53] Y. Cheng, L. Zhang, Y. Han, and X. Li, "Thermal-constrained task allocation for interconnect energy reduction in 3-d homogeneous mpsocs," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 21, no. 2, pp. 239–249, 2013.

[54] A. K. Coskun, R. Strong, D. M. Tullsen, and T. Simunic Rosing, "Evaluating the impact of job scheduling and power management on processor lifetime for chip multiprocessors," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, pp. 169–180, ACM, 2009.

[55] T. Chantem, Y. Xiang, X. S. Hu, and R. P. Dick, "Enhancing multicore reliability through wear compensation in online assignment and scheduling," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 1373–1378, EDA Consortium, 2013.

[56] D. Cuesta, J. Ayala, J. Hidalgo, D. Atienza, A. Acquaviva, and E. Macii, "Adaptive task migration policies for thermal control in mpsocs," in *VLSI 2010 Annual Symposium*, pp. 83–115, Springer, 2011.

[57] I. Ukhov, M. Bao, P. Eles, and Z. Peng, "Steady-state dynamic temperature analysis and reliability optimization for embedded multiprocessor systems," in *Proceedings of the 49th Annual Design Automation Conference*, pp. 197–204, ACM, 2012.

[58] Y. Ma, T. Chantem, X. S. Hu, and R. P. Dick, "Improving lifetime of multicore soft real-time systems through global utilization control," in *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*, pp. 79–82, ACM, 2015.

[59] M. Mandelli, G. Castilhos, G. Sassatelli, L. Ost, and F. G. Moraes, "A distributed energy-aware task mapping to achieve thermal balancing and improve reliability of many-core systems," in *Proceedings of the 28th Symposium on Integrated Circuits and Systems Design*, p. 13, ACM, 2015.

[60] M. Mandelli, L. Ost, G. Sassatelli, and F. Moraes, "Trading-off system load and communication in mapping heuristics for improving noc-based mpsocs reliability," in *Quality Electronic Design (ISQED), 2015 16th International Symposium on*, pp. 392–396, IEEE, 2015.

[61] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Maestro: Orchestrating lifetime reliability in chip multiprocessors," in *High Performance Embedded Architectures and Compilers*, pp. 186–200, Springer, 2010.

[62] M. Basoglu, M. Orshansky, and M. Erez, "Nbti-aware dvfs: a new approach to saving energy and increasing processor lifetime," in *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, pp. 253–258, ACM, 2010.

[63] L. Huang, F. Yuan, and Q. Xu, "On task allocation and scheduling for lifetime extension of platform-based mpsoc designs," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 12, pp. 2088–2099, 2011.

[64] A. Das, A. Kumar, and B. Veeravalli, "Reliability-driven task mapping for lifetime extension of networks-on-chip based multiprocessor systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 689–694, EDA Consortium, 2013.

[65] C. Bolchini, M. Carminati, A. Miele, A. Das, A. Kumar, and B. Veeravalli, "Run-time mapping for reliable many-cores based on energy/performance trade-offs," in *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2013 IEEE International Symposium on*, pp. 58–64, IEEE, 2013.

[66] A. Das, A. Kumar, and B. Veeravalli, "Temperature aware energy-reliability trade-offs for mapping of throughput-constrained applications on multimedia mpsocs," in *Proceedings of the conference on Design, Automation & Test in Europe*, p. 102, European Design and Automation Association, 2014.

[67] A. Das, A. Kumar, B. Veeravalli, C. Bolchini, and A. Miele, "Combined dvfs and mapping exploration for lifetime and soft-error susceptibility improvement in mpsocs," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pp. 1–6, IEEE, 2014.

[68] A. Das, A. Kumar, and B. Veeravalli, "Aging-aware hardware-software task partitioning for reliable reconfigurable multiprocessor systems," in *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, p. 1, IEEE Press, 2013.

[69] B. Nahar and B. H. Meyer, "Rotr: Rotational redundant task mapping for fail-operational mpsocs," in *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS), 2015 IEEE International Symposium on*, pp. 21–28, IEEE, 2015.

[70] A. Benoit, F. Dufossé, A. Girault, and Y. Robert, "Reliability and performance optimization of pipelined real-time systems," *Journal of Parallel and Distributed Computing*, vol. 73, no. 6, pp. 851–865, 2013.

[71] B. H. Meyer, A. S. Hartman, and D. E. Thomas, "Cost-effective lifetime and yield optimization for noc-based mpsocs," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 19, no. 2, p. 12, 2014.

[72] C. Zhu, Z. Gu, R. P. Dick, and L. Shang, "Reliable multiprocessor system-on-chip synthesis," in *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2007 5th IEEE/ACM/IFIP International Conference on*, pp. 239–244, IEEE, 2007.

[73] A. Simevski, R. Kraemer, and M. Krstic, "Increasing multiprocessor lifetime by youngest-first round-robin core gating patterns," in *Adaptive Hardware and Systems (AHS), 2014 NASA/ESA Conference on*, pp. 233–239, IEEE, 2014.

[74] L. Wang, X. Wang, and T. Mak, "Dynamic programming-based lifetime aware adaptive routing algorithm for network-on-chip," in *Very Large Scale Integration (VLSI-SoC), 2014 22nd International Conference on*, pp. 1–6, IEEE, 2014.

[75] F. Dabiri, N. Amini, M. Rofouei, and M. Sarrafzadeh, "Reliability-aware optimization for dvs-enabled real-time embedded systems," in *Quality Electronic Design, 2008. ISQED 2008. 9th International Symposium on*, pp. 780–783, IEEE, 2008.