ABSTRACT

GHATTAS, RONY. Data Allocation with Real-Time Scheduling (DARTS). (Under the direction of Dr. Alexander G. Dean).

The problem of utilizing memory and energy efficiently is common to all computing platforms. Many studies have addressed and investigated various methods to circumvent this problem. Nevertheless, most of these studies do not scale well to real-time embedded systems where resources might be limited and particular assumptions that are valid to general computing platforms do not continue to hold.

First, memory has always been considered a bottleneck of system performance. It is well known that processors have been improving at a rate of about 60% per year, while memory latencies have been improving at less than 10% per year. This leads to a growing gap between processor cycle time and memory access Time. To compensate for this speed mismatch problem it is common to use a memory hierarchy with a fast *cache* that can dynamically allocate frequently used data objects close to the processor. Many embedded systems, however, cannot afford using a cache for many reasons presented later. Those systems opt to use a *cacheless* system which is particularly very popular for real-time embedded applications. Data is allocated at compile time, making memory access latencies deterministic and predictable. Nevertheless, the burden of allocating the data to memory is now the responsibility of the programmer/compiler.

Second, the proliferation of portable and battery-operated devices has made the efficient use of the available energy budget a vital design constraint. This is particularly true since the energy storage technology is also improving at a rather slow pace. Techniques like *dynamic voltage scaling* (DVS) and *dynamic frequency scaling* (DFS) have been proposed to deal with these problems. Still, the applicability of those techniques to resource-constrained real-time system has not been investigated.

In this work we propose techniques to deal with both of the above problems. Our main contribution, the *data allocation with real-time scheduling* (DARTS) framework solves the data allocation and scheduling problems in *cacheless* systems with the main goals of optimizing memory utilization, energy efficiency, and obviously overall system performance. DARTS is a synergistic optimal approach to allocating data objects and scheduling real-time tasks for embedded systems. It optimally allocates data objects to memory through the use of an *integer linear programming* (ILP) formulation, which minimizes the systems *worst-case execution times* WCET resulting in more scheduling *slack*. This additional slack is used by our *preemption threshold scheduler* (PTS) to reduce stack memory requirements while maintaining all hard real-time constraints. The memory reduction of PTS allows these steps to be repeated. The data objects now require less memory, so more can fit into faster memory, further reducing WCET and resulting in more slack time. The increased slack time can be used by PTS to reduce preemptions further, until a fixed point is reached. Using a combination of synthetic and real workloads, we show that the DARTS platform leads to optimal memory utilization and increased energy efficiency.

In addition to our main contribution given by the DARTS platform, we also present several techniques to optimize a systems memory utilization in the absence of a memory hierarchy using PTS, which we enhance and improve. Furthermore, many advanced energy saving techniques like DFS and DVS are investigated as well, and the tradeoffs in their use is presented and analyzed.

Data Allocation with Real-Time Scheduling (DARTS)

By

Rony Ghattas

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the requirements for the Degree
of Doctor of Philosophy in

COMPUTER ENGINEERING

Raleigh, North Carolina, USA
December 2006

APPROVED BY:

Dr. Alexander G. Dean
Chair of Advisory Committee

Dr. Thomas M. Conte

Dr. Eric Rotenberg

Dr. Ralph C. Smith

To my loving mother...

Biography

Rony Ghattas was born and raised in the Egyptian capital Cairo, until the age of 22. He moved to the city of Murray in the United States in 1997 where he attended Murray State University majoring in Engineering Physics. In spring of 2000 he was named the year's outstanding Engineering Physics Senior and graduated the same year at the top of his class. In Fall of 2000 he joined the department of Electrical and Computer Engineering at North Carolina State University as a graduate student. After working at the Center for Power Electronics Research at North Carolina State University for 18 month, he joined the Center for Embedded Systems Research as a research assistant until December of 2006.

Upon completing his Ph.D., Rony is moving to Columbia, South Carolina, to join Intel's chipset validation team as a Component Design Engineer. Rony is a member of the Sigma Pi Sigma National Physics Honor Society, Gamma Beta Phi National Honor Society, and a student member of the IEEE.

Acknowledgements

In this short space I would like to thank the people that have contributed towards the success of this work. Many individuals helped in this work more than can be listed here, but a few people certainly deserve a special mention. First, I would like to thank my advisor, Dr. Alexander G. Dean; his continuous support and guidance has made this work possible. During the course of my studies I have come to know Alex as a mentor and as a friend. I truly value his advice inside and outside the workplace. Since at the timing of this writing Alex has just been blessed by another baby girl, I would like to congratulate him and his wife on their new baby Jacqueline.

Second, I would like to thank Doctors Tom Conte, Eric Rotenberg, and Ralph Smith for taking time out of their hectic schedule to serve as members on my committee. Dr. Tom Conte, to whom I owe my fascination with compilers, deserves special thanks. I would also like to thank Dr. Ralph Smith for his insightful guidance over a period of more than six years during which he was always ready to lend a helping hand when I needed one.

I would also like to thank all members of the Center for Embedded Systems Research (CESR) at North Carolina State University for creating such a wonderful intellectual environment. Special thanks to Ms. Sandra Bronson whom helped me with all the cryptic forms and paper work often needed to be filled by graduate students: Thank you Sandy for taking care of all the administrative work and making my life a bit easier. I would like to thank all members of the Graduate Office at the Electrical and Computer Engineering Department. Special thanks to Ms. Fay Bostic and to Ms. Pascale Toussaint for their patience and their encouragement. Special thanks to Dr. Joel Trussell, who in spite of some disagreements, had always been a mentor to me, ready to give me his genuine advice.

Many thanks to my friends who have always stood by my side in the time of need. I can't help remember the old saying that *A friend in need is a friend indeed.* Though I cannot list all of your names in this single page, be assured that will always be listed in my mind and heart if not on this page. Last but not least, my love and appreciation toward my mother for her support and guidance throughout my academic career (and my whole life for that matter). Mom, I have not always been the best son you can hope for, but I hope dedicating this work to you can help me repay 1% of the unconditional love and support you have always provided me with.

Above all, I would like to thank God, the most gracious and forgiving. *I now know that life is not about staying on your feet. It's about getting up when you fall.* Thank you dear God.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This work develops a synergistic framework for data allocation and real-time scheduling for *cacheless*[1] platforms. To the best of our knowledge this is the first attempt to combine these two design phases into a single optimal framework.

The DARTS (*data allocation with real-time scheduling*) technique is a novel iterative framework that can be incorporated in any optimizing compiler to render globally optimal data allocations while optimizing the real-time scheduling characteristics of the workload. In fact, our framework enables optimizations that are only possible when these two, seemingly independent, design phases are addressed in a unified framework. As will be shown in this work, the DARTS framework enables many critical optimizations that render improved system performance, better memory utilization, and increased energy efficiency, all while maintaining the optimality and validity of the real-time schedulability of our workload.

Static data allocation is a rather complicated design problem since the complexity typically hidden by hardware-controlled caches is now visible to the programmer and/or compiler. Things are even more intricate when the data objects to be allocated are of a real-time nature with timing constraints that cannot be violated. Still, we show in this work that most of the complexity inherent in static allocation of data and its real-time scheduling is handled ef-

[1] A *cacheless* system or platform is any computing platform that utilizes a memory hierarchy without a hardware-controlled cache. As we will show later, there are many examples of such systems especially in the hard real-time domain.

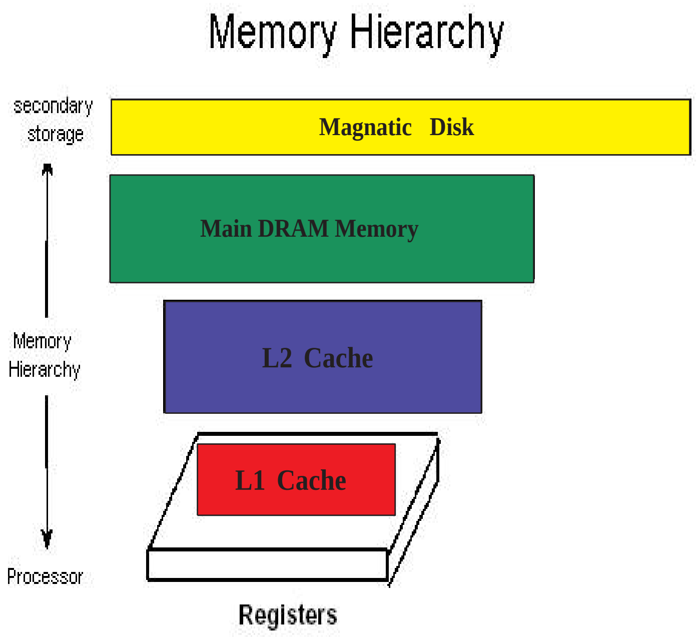


Figure 1.1: Traditional memory hierarchy

ficiently and systematically by DARTS. DARTS works iteratively; repeating each of its phases, until an optimal real-time schedule with optimal memory utilization and improved energy efficiency is obtained.

1.1 Efficient Utilization of Memory

A computer's memory system is the repository for all the information used by and produced by the processor. The phenomenal increase in microprocessor performance places significant demands on the memory system. Indeed, as new processor families and processor cores begin to push the limits of high performance, the traditional processor-memory gap widens and often becomes the dominant bottleneck in achieving high performance [1]. This statement is clearly true for all computing platforms not just the embedded ones.

Since the early ages of computing, people have realized the need for a large amount of fast memory to support the ever increasing complexity of the applications. However, memories are advancing much slower than processors, and fast memories are very expensive. To circumvent this problem a memory hierarchy composed of two or more levels of memory with different latencies and sizes is usually used. An example of a traditional memory hierarchy is shown in Figure 1.1. As can be see, at lower levels of the hierarchy (i.e., closer to the processor) a small and fast, and clearly more expensive *register file* is usually used. On the other end, a larger, slower, and less expensive magnetic disk is used.

With the advances in fabrication technology, it is now possible to combine multiple heterogeneous memory units on the same chip. It is therefore not uncommon anymore to find memory hierarchies composed of multiple heterogeneous units with different sizes, access latencies, as well as bit-widths all on the same chip. Still, the basic property of any memory hierarchy still holds independent of the technology used. That is, each level in the hierarchy is faster and smaller, and usually more expensive, than the one strictly above it. For example, in Figure 1.1, the register file is bound to the processor and most instructions on registers take a single cycle. On the other hand, the magnetic disk at the high end of the hierarchy in the same figure might takes hundreds if not thousands of processor cycles to access. [2].

1.1.1 Hardware Control and Real-Time Predictability

Static analysis[2] of programs is fairly straightforward for simple architectures. The presence of hardware acceleration techniques like pipelines, caches, DMAs, etc. in modern RISC processors, however, has complicated the process considerably. For example, in a pipelined processor the instruction latencies are not constant and depend on pipeline stalls. Caches, on the other hand, introduce unpredictability since it is hard to predict a cache hit or miss from looking to the program alone. DMA controllers, when operating in cycle stealing mode, can contend with the processor for I/O peripherals introducing unpredictability. Recently, however, many techniques have been proposed in the literature to alleviate these problems.

[2]*Static analysis* is the analysis of the application code that is performed without actually running it (as opposed to profiling where the application is executed to infer its dynamic behavior). In the context of real-time applications static analysis is performed to derive bounds on the execution characteristics of the application that can only occur under some worst-case scenario and might not show through profiling.

Since the early days of computing people have realized that applications tend to spend the majority of their execution time executing a small subset of instructions (e.g., loops, certain procedures, etc). In other words, some instructions are much more frequently accessed than others. Moreover, it is usually the case that those frequently executed instructions are spatially and/or temporally close to each other. These facts came to be known later as the principles of *spatial* and *temporal localities* and form the basic operational principles of a cache. Hence, a cache can be defined as a hardware-controlled memory unit that uses the run-time behavior of the workload to *dynamically* predict which data objects have the highest probability of being accessed multiple times. Based on these predictions, it stores these data objects in a fast memory unit close to the processor avoiding the overheads of accessing frequently accessed data object from higher levels of memory every time they are needed. Indeed, it is well known that caches can lead to tremendous performance improvements for general purpose computing platforms as in the desktop and server domains [3].

Hard real-time applications, on the other hand, must meet their deadlines in all situations including the worst-case one; otherwise the safety of the controlled system is jeopardized. Nevertheless, the operation of caches is inherently non-deterministic. In other words, the number of cycles a cache spends accessing data is random and subject to compulsory, capacity, and conflict misses [2]. This introduces a high degree of unpredictability that makes static analysis of real-time applications a daunting task.

Many studies, however, have proposed techniques for accurate timing analysis in the presence of hardware-managed caches. One of the techniques proposed to make behavior of cache predictable is static cache simulation [4]. In this scheme, each basic block in the program is simulated and the state of the cache at the entry and exit points are saved. This cache state is then used to divide instructions into categories like always hit, always miss, first hit, etc. Straight forward static analysis of the program code can then be performed. The use of *integer linear programming* (ILP) to bound worst-case behavior of a workload in the presence of a cache was suggested by Li et al. [5] through program path analysis and micro-architecture modeling. Program path analysis is used to eliminate paths whose execution time definitely does not correspond to worst-case path. Micro-architecture modeling models the underlying architecture using simplifying assumptions. The problem of finding the program's worst-case path is then expressed in an ILP framework with feasible paths and architecture characteristics expressed as constraints.

Analyzing the unpredictability introduced by data caches is even more difficult since the address of the data reference is not known at compile time. One of the important works in bounding cache access is from White et al. [6]. They provide a tool which will give tight WCET of a program in presence of data caches. They use the static cache simulation by Mueller [4] to categorize instructions into first miss, first hit, always miss and always hit. Using the instruction categorization and program control flow graph, timing analyzer (an iterative algorithm) will give the worst-case behavior of the program. Extensions of this analysis to set-associative caches was also address by Mueller in his later work [7].

Many other architectural mechanisms have been used to hide memory latencies. For example, *direct memory access* (DMA) hardware is used in many systems to relieve the processor from waiting on the slow memory or I/O peripherals. The processor can therefore continue working until the memory transfer is ready and an interrupt is generated by the DMA controller. Yet DMA controllers can introduce other unpredictability factors. When operating in cycle-stealing mode, the DMA controller owns the I/O bus as long as the processor does not need it. As soon as the processor needs the bus, its ownership should be transferred to the processor. However, this ownership transfer does not take place immediately which causes the processor to miss some cycles that would not have been missed in the absence of the DMA hardware [8]. Huang et al. [8] have given an algorithm which calculates the worst-case behavior of a sequence of instructions when executed concurrently with DMA operations. The key idea behind the algorithm is that execution cycles dont overlap among instructions. The time delay due to cycle-stealing operation of the DMA controller can be analyzed for each instruction individually, which results in a very simple algorithm.

In other frameworks, memory latencies have been addressed in different ways. For example, *multitasking* (also known as *multithreading*) *processors* hide the latencies due to memory stalls by switching execution between tasks (managed in hardware) at long latency instructions like memory transfers. This dynamic hardware control, however, introduces a factor of unpredictability that prohibits static analysis of real-time workloads. El-Haj Mahmoud [9], however, proposed a method for exploiting *simultaneous multithreading* (SMT) on a super-scalar processor in a safety-critical real-time system. To this end, El-Haj Mahmoud proposed a new architecture model of a multithreading super-scalar processor as a collection of virtual processors. Each virtual processor is a partition enabling a single task to execute without interference from any other task, and hence, can be analyzed statically. This enables real-time scheduling of safety-critical systems while rendering optimized performance since memory stalls are hidden by the overlapped execution of other tasks.

1.2 Efficient Utilization of Energy

Battery powered electronic systems, and the integrated circuits within them, account for a large and rapidly growing revenue segment for the computer, electronics, and semiconductor industries. For instance, the revenue from wireless voice/data handsets is expected to exceed that from PCs in the near future [10]. As was mentioned earlier, processor technology is reaching a new frontier every day, unfortunately, projected improvements in battery technology are much slower than what is needed to support the tremendous advances in the systems they power. Hence, in an analogous manner, there exists some kind of a processor-energy gap similar to that of the processor-memory gap.

We need to emphasize that the energy efficiency problem is much less significant in the desktop and server domains. For these platforms, there is usually no need for energy storage to begin with, and power dissipation is a substantially more important problem due to heat dissipation issues [11]. Minimizing power dissipation, rather than energy, is crucial in those cases since it implies less expenses spent on heat sinks, cooling systems, and other precautionary measures.

A completely different problem arises in the embedded domain being that of energy (rather than power) efficiency. With the proliferation of portable and battery operated systems, it is crucial that embedded systems can sustain their operation for extended periods of time on the limited energy budget available. Minimizing power alone in this case is useless since the system will require a longer period of time perform the same amount of work. Hence, without a good energy management policy, major problems arise.

Energy Efficiency in a Memory Hierarchy

As was mentioned earlier, modern architectures allow memory hierarchies of multiple *heterogeneous*[3] memory units. Moreover, the energy dissipation characteristics of these memory units are dictated by the particular structure and fabrication technology used for each memory unit. For example, caches in general tend to consume more energy than other memory architectures [12]. Even in cacheless architectures, there is a great variation between the energy consumed per access from one memory unit to the next. For example, energy consumption strongly depends on the physical size of the memory unit (both the number of words and the number of bits per word). Hence, a general rule of thumb is that smaller memories are more energy efficient than larger ones. Second, the larger the memory hierarchy is (i.e., the more memory units in the hierarchy), the more energy need be spent due to the addressing complexity. Third, the further away the memory units are from the processor (distance wise) the increased energy dissipation due to the wiring overheads.

1.3 Motivation

Our work was motivated by the problems discussed in the previous Sections. Though none of these problems is new or unique to any particular platform, the tools and solutions available are. In other words, the tools available to deal with most of the problems discussed above including the memory utilization problem and the energy efficiency problems do not necessarily apply to our special category of real-time embedded applications.

For example, we have already mentioned that caches can be used to render tremendous per-

[3]We will say two memory units are heterogeneous if they have different access latencies, sizes, and/or bitwidths.

formance improvements in the desktop and server domains. Those improvements, however, are usually in the average-case performance of the system. But improvements in the average-case performance do not necessarily imply improvements in the worst-case performances. In fact, it has been shown many times that improving the average-case performance can actually deteriorate the worst-case performance. Hence, other custom tailored solutions are needed.

Second, as we have also already mentioned, scaling down the power dissipation of a system by a factor of two using some of the popular techniques only implies that it will take our system double the amount of time to finish its work. Clearly this does not improve energy efficiency. Again, this just implies that special solutions and tools are needed to handle those special systems and applications. In this work, it is our main aim to develop and analyze such custom optimization techniques and make sure they are applicable to our category of systems where many other optimizations simply fail to be of any use.

In this work we address these problems in a unique way that is applicable to that special category of computing platforms. We develop and analyze new techniques that we apply when other tools fail to provide the needed functionality. Moreover, we present analytical as well as experimental results that prove and support our proposed techniques and methodologies.

1.4 Thesis Contributions

In this work we present the *data allocation with real-time scheduling* (DARTS) framework as a novel approach to software-managed data allocation and real-time scheduling. In addition, we present several techniques to optimize memory utilization for single-memory systems using an extended and improved version of *preemption threshold scheduling* (PTS). Furthermore, a quantitative analysis of the most popular energy management techniques for embedded systems is presented and analyzed. The detailed contributions can be listed as follows (contributions 1 through 4 are the main contribution of this work):

1. In Section 5.4 we present a novel data allocation framework that is applicable to multi-tasking real-time systems.

2. In Section 5.4.3 we show how our *optimal multitasking memory allocation* (OMMA) technique is combined with PTS to render the DARTS framework enabling optimal data allocation and scheduling of real-time embedded applications.

3. In Section 2.4 we present an overview of the complete DARTS tool chain that was specifically developed to support our framework.

4. In Section 5.5 we present multiple simulations and experiments that show how the DARTS framework leads to optimal static allocations along with significantly improved real-time schedules. The improvements in the memory traffic, the WCET, and the energy utilization of the system are all shown to benefit significantly from the DARTS framework.

5. In Section 3.2 a unified framework enabling the use of PTS with both dynamic- as well as fixed-priority schemes in the presence of shared resources is presented. Our framework guarantees schedulability for both schemes and maintains system concurrency.

6. We analytically prove in Section 3.3 that PTS with an algorithm known as the *maximal preemption threshold assignment algorithm* (MPTAA) is stack space optimal in the sense that no other preemption limiting framework can result in smaller memory requirements without changing the system model. These results are shown to hold for both priority-driven scheduling schemes by extending the MPTAA to support dynamic-priority systems.

7. In Section 3.4.1 We analytically prove that PTS results in schedules that are robust to uncertainties in the workload's WCETs.

8. We develop in Section 3.4.2 a backtracking strategy that can be used with the MPTAA to reduce the tasks WCETs by up to 50%.

9. In Chapter 4.2 a survey of the most popular energy and power minimizing techniques is presented along with the tradeoffs involved in the use of each.

1.5 Thesis Outline

This thesis is divided into 6 main chapters. In Chapter II we present the related work and tools used throughout this study. In Chapter III we present a unified framework for using PTS with both priority-driven schemes and prove some of its more important properties used in later chapters. In Chapter IV we present a survey of the most popular energy and power management techniques along with the tradeoffs involved in their use. Chapter V presents our main contributions: the DARTS framework, along with its formulation and experimental results collected to show how it can be used to improve data allocation as well as real-time scheduling of real-time applications. Chapter 6 presents a summary of this thesis along with an outline of our future work. Finally, an appendix is added to present some of the mathematical derivations used in this work.

Chapter 2

Related Work and Tools

In this chapter we try to briefly overview and present some of the studies related to this work. As we have already mentioned, we are mainly interested in embedded platforms that, in addition to other design constraints, have deadlines that must be met. A missed deadline is simply a useless result even if logically correct in the case of *soft real-time systems*[1], or a serious problem as in the case with safety-critical *hard real-time systems*[2] . The area of computer engineering dealing with such systems is known as *real-time systems theory*. This theory is the accumulation of many years of work by many researchers and can take entire books to just describe its fundamentals. Hence, in Section 2.1 we will in no way try to comprehensively cover this extensive subject, just the very few concepts needed to understand our work.

We then move to the field of digital integrated circuits design in Section 2.2, to state some of the main results and findings that are used in this work including the energy and power dissipation models for CMOS-based integrated circuits as well as some of the energy models describing the energy dissipation of memory hierarchies.

Finally, in Section 2.4 we briefly explain and present the tool chain that was developed for the DARTS framework. We briefly discuss its major components as will as their principles of operations. More tool details can be found in their documentation..

[1]A *soft* real-time system is defined as a real-time system where some of the deadlines can be missed without causing serious consequences

[2]As opposed to soft real-time systems, *hard* real-time system are mostly safety-critical ones where a missed deadline can lead to major problems and some times tragic consequences.

2.1 Real-Time Systems Theory

2.1.1 Real-Time System Model and Terminology

A real-time *task*, denoted by T, is an independent thread of execution that competes with other tasks for processor time and other resources. A task is invoked infinitely many times and each invocation results in a single execution which we call an *instance* or a *job*. We will denote the i^{th} task in the workload by T_i and denote the j^{th} job of the i^{th} task by J_{ij}. Every job J_{ij} is characterized by a release time r_{ij} (i.e. the instant of time at which the job becomes ready for execution), a computation time c_{ij} (i.e. the amount of time required to complete execution), and an absolute deadline d_{ij} (i.e. the instant of time by which a job is required to be completed). Moreover, associated with every job J_{ij} is an amount of memory space s_{ij} (referred to as the job's stack space) used by the job for its local variables, nested function calls, return addresses, as well as its own context if necessary for preemptions[3]. We also associate with each job J_{ij} a unique priority $\pi_{ij} \in \{1, 2, \ldots, N\}$ such that contention for resources is resolved in favor of the job with the highest priority that is ready to run as will be explained in the next section. Throughout this study we assume time is discrete and clock ticks are indexed by the natural numbers. Job arrivals, and executions begin at clock ticks, and each of the attributes r_{ij}, c_{ij}, and d_{ij} is expressed as a multiple of (the interval between) clock ticks.

Tasks can be *periodic* or *sporadic*. If T_i is periodic, the period P_i specifies a constant interval between arrival times of any two consecutive jobs, and if it is sporadic, P_i specifies a minimum interval between job arrivals. We say task T_i has a *worst-case execution time* (WCET) of C_i to denote that the maximum execution time of any of the task's jobs is given by C_i (i.e $C_i = \max_j[c_{ij}]$). We say task T_i has a maximum stack space requirement of S_i units if the maximum memory space required by any of the task's jobs for its stack is given by S_i memory units (i.e. $S_i = \max_j[s_{ij}]$). Moreover, we say that T_i has a *relative deadline* of D_i time units if all that task's jobs must complete execution by no more than D_i time units after their arrival to meet their deadline (i.e. $D_i = \min_j[d_{ij} - r_{ij}]$). Finally, we need to point out that the worst-case invocation

[3]The maximum memory required by each task can be computed by many static analysis tools as explained in the next Chapter.

pattern for a sporadic task (worst in the sense requiring the most processor time) occurs if it is released every exactly P_i time units. For this reason, we will not usually need to treat sporadic tasks separately in this work.

2.1.2 Real-Time Scheduling Policies

A scheduling policy dictates the order in which different tasks use the processor time and how different requests should be serviced. *Priority-driven* scheduling algorithms refer to a large class of scheduling algorithms that never leave any resource idle intentionally and are used by most real-time kernels. Scheduling decisions in a priority-driven system are made when events such as releases and completions of jobs occur. Hence, a priority-driven algorithm is sometime viewed as an *event-driven* scheduling policy.

Let $\mathcal{T} = \{T_i | i = 1, 2, \ldots, N\}$ denote some real-time workload with N tasks. At any time instant t_n, a priority-driven scheduling policy can be thought of as a mapping $\Pi(t_k) : \mathcal{T} \to \{1, 2, 3, \ldots, N\}$ governing the execution of jobs such that contention for resources is always resolved in favor of the job with the highest priority that is ready to run[4]. Many scheduling policies exist and have been used. Examples include the *rate monotonic* (RM) policy, the *deadline monotonic* (DM) policy, the *earliest deadline first* (EDF) policy, etc.

Fixed-Priority Scheduling Policies

In a fixed-priority scheduling policy, priorities are assigned offline and remain fixed throughout the operational time of the system. In a fixed-priority policy, the same priority is assigned to all jobs in each task (i.e. $\pi_{ij} = \pi_{ik} = \pi_i$ for all j and k). In other words, the priority of each task is fixed relative to other tasks. Hence, in a fixed-priority scheme we can speak of a ”task's” priority rather than of a ”job's” priority. Classical examples of *fixed-priority* scheduling policies include the RM and DM policies, as well as Audsley's optimal priority assignment algorithms [13, 14]. We will call a real-time workload along with some priority mapping that has been assigned statically a *static real-time system* and denote such system by the tuple $(\mathcal{T}, \Pi)$.

[4]In this thesis we maintain the natural ordering of numbers such that saying that J_{mn} has a higher (lower) priority than J_{ij} implies that $\pi_{mn} > \pi_{ij} (\pi_{mn} < \pi_{ij})$ respectively.

Dynamic-Priority Scheduling Policies

Unlike fixed-priority scheduling, in a dynamic-priority scheme different jobs of the same task can be assigned different priorities dynamically (i.e. at run-time). The most popular dynamic priority scheduling policy is the *earliest deadline first* (EDF) algorithm [13]. It has been shown in the literature that the EDF algorithm is optimal in the sense that if any dynamic-priority algorithm can schedule a particular workload, so can the EDF algorithm [15]. Hence, we will be mainly referring to the EDF algorithm in this study whenever we refer to a dynamic-priority scheduling algorithm. In EDF scheduling, the priority of each job is assigned dynamically to be inversely proportional to job's absolute deadline. Since different jobs of the same task might have different absolute deadlines, they will also have different priorities. Again, a real-time workload along with some priority mapping that is assigned dynamically will be referred to as a *dynamic real-time system* and denoted by the tuple $(\mathcal{T}, \Pi(t_n))$.

Preemptive versus Non-Preemptive Scheduling Policies

An essential property of a scheduling policy is *preemptability*. We say a scheduling policy is *fully-preemptive* (FP) if a currently executing task can be preempted by a ready-to-run higher priority task. On the other hand, we say a scheduling policy is *fully-non-preemptive* (FNP) if the ready-to-run higher priority task has to wait for the currently executing task to voluntarily release the processor.

A fundamental question naturally arises. When is FP scheduling better than FNP scheduling and vice versa? Unfortunately, there is no general answer to this question. However, in many special cases it has been shown that FP schedulers can make better system utilization, increased system responsiveness, etc [13]. On the other hand, FNP schedulers are easier to implement, have no preemption overheads, and do not have the shared resources problem explained later. As we will show in this study, *preemption threshold scheduling* (PTS) gives the system designer the best of the two worlds.

The term preemption threshold scheduling was first coined by Wang and Saksena [16, 17] after they investigated the concept of preemption thresholds introduced by Express Logic Inc.

in their real-time kernel ThreadX [18]. With PTS, a static real-time system $(\mathcal{T}, \Pi)$ is assigned an additional mapping $\Gamma : (\mathcal{T}, \Pi) \rightarrow \{1, 2, \ldots, N\}$ (i.e. each task is assigned a preemption threshold, denoted by γ_i in addition to its priority π_i with the essential property that $\gamma_i \geq \pi_i$). When a job begins execution, its priority is raised to its preemption threshold. In this way, all jobs with priorities less than or equal to the preemption threshold of the executing task cannot preempt it.

It is easily seen that FP and FNP scheduling policies are special boundary cases of PTS. By assigning the preemption threshold of each task equal to its priority, PTS becomes an FP scheduling policy. By assigning the preemption threshold of each task equal to the system's highest priority, PTS simplifies to an FNP policy. It has been shown by Wang et al. [16] that PTS presents a more general framework than either pure FP or FNP scheduling policies. In fact, in their work, they showed that some workloads that cannot be scheduled neither in a FP manner, nor in a FNP manner, can be scheduled using PTS.

Real-Time Schedulability

A real-time task T_i is said to be *schedulable* if every one of its jobs can complete by or before its deadline. If we denote the completion time of job J_{ij} by f_{ij}, then we can define the job's response time as $R(J_{ij}) = f_{ij} - r_{ij}$. Furthermore, we define the task's *worst-case response time* (WCRT) as $R(T_i) = \max_j[R(J_{ij})]$. It should be clear that requiring a task to be schedulable is equivalent to requiring that $R(T_i) \leq D_i$. This WCRT of a task is assumed to occur under some worst-case scenario that might or might not occur. Clearly knowing a task's WCRT solves the schedulability problem. Unfortunately, however, computing the WCRT is only possible for statically scheduled (i.e. fixed-priority) real-time workload. For this reason, assessing the schedulability in a dynamic-priority scheme cannot use the WCRT criteria and other conditions are used as will be shown later. Finally, if there exists a priority assignment (fixed or dynamic) such that all tasks composing a workload are schedulable, we say that the system $(\mathcal{T}, \Pi)$ (or $(\mathcal{T}, \Pi(t_n))$ in a dynamic-priority scheme) is schedulable.

2.1.3 The Schedulability Problem

A fundamental problem in real-time scheduling theory is that of determining if a particular workload and a particular priority mapping render a schedulable system as defined in the previous section. Stated differently: Given a set of tasks, a set of resources available to the tasks, a scheduling algorithm, and a resource sharing protocol to be used to allocate resources to tasks, determine whether all jobs can meet their deadlines. In this section we present some known schedulability conditions.

Fixed-Priority FP Schedulability

We are given a static real-time system $(\mathcal{T}, \Pi)$ where the fixed-priority mapping has been assigned using some algorithm (e.g. RM, DM, etc). We would like to know if this system is schedulable. In a fixed-priority setting, schedulability can be analyzed using *level-i busy period analysis* [19, 20]. To this end, the WCRT of each task, defined in Section 2.1.2, is computed as a bound on the response times of all jobs by essentially simulating some worst-case scenario that jobs can experience. If the WCRT for each task is no larger than its respective relative deadline, the system is schedulable.

Consider the static real-time system $(\mathcal{T}, \Pi)$, and let $T_i \in \mathcal{T}$ be any task. Since this is a fixed-priority scheme, we can speak of task priorities rather than task job priorities. Let $\mathcal{HP}(T_i)$ denote the subset of all tasks belonging to $\mathcal{T}$ with priorities larger than that of T_i (i.e. $\mathcal{HP}(T_i) = \{T_j \in \mathcal{T} | \pi_j > \pi_i\} \subset \mathcal{T}$). Similarly, let $\mathcal{LP}(T_i)$ denote the subset of all tasks belonging to $\mathcal{T}$ with priorities smaller than that of T_i. The WCRT of a task will occur if one of its jobs is released at the same time with a job from every higher priority task [13]. Hence, the WCRT of a task can be obtained by considering the response-time of a single job that is released under this worst-case scenario. This WCRT can be computed using the following equation [20]:

$$R(T_i) = \min_{q \in \{0,1,2,\ldots\}} w_i(q) \tag{2.1a}$$

Where $w_i(q)$ denotes the length of a busy period for task T_i with q jobs (instances) of T_i included in the busy period and can be solved for using the following recursive equation.

$$w_i(q) = q \cdot C_i + \sum_{T_j \in \mathcal{HP}(T_i)} \left\lceil \frac{w_i(q)}{P_j} \right\rceil C_j \tag{2.1b}$$

This iteration over increasing values of q stops if $w_i(q) \leq q \cdot P_i$. A task set is schedulable fully-preemptively if and only if $R(T_i) \leq D_i$ for all $i \in [1, N]$. Hence, equations (2.1a) and (2.1b) present a necessary and sufficient condition for schedulability using in a fixed-priority setting.

Dynamic-Priority FP Schedulability

Processor demand analysis [15] can be used to obtain sufficient and necessary conditions for the schedulability of a dynamic real-time system $(\mathcal{T}, \Pi(t_n))$. However, in this study we only address sufficient conditions and leave necessary conditions for future work. To this end, given the tuple $(\mathcal{T}, \Pi(t_n))$, a sufficient condition for the schedulability of a real-time system is given by the following *EDF schedulability test* [13]:

$$\sum_{i=1}^{N} \frac{C_i}{\min(D_i, P_i)} \leq 1 \tag{2.2}$$

If the above condition is satisfied, we conclude that our workload is schedulable with the EDF scheme. We note, however, that if it is not satisfied, then the conclusion we may draw depends on the relative deadline of the tasks. If $D_i \geq P_i$ for all i, then equation (2.2) reduces to the following well known *EDF utilization test* which is both a necessary and sufficient condition:

$$\sum_{i=1}^{N} \frac{C_i}{P_i} \leq 1 \tag{2.3}$$

On the other hand, if $D_i < P_i$ for some i, equation (2.2) is only sufficient not satisfying it can only imply that system may not be schedulable.

2.1.4 Shared Resources

A task may need some resources in addition to the processor to make progress. We will let $\mathcal{R} = \{\rho^1, \rho^2, ..., \rho^K\}$ be the set of such resources. These resources are typically granted to tasks on a non-preemptive basis and used in a *mutually exclusive* manner. In other words, when a resource ρ^k is granted to a task, this resource is no longer available to other tasks until the task frees it. Examples of such resources are semaphores, mutexes, reader / writer locks, files, connection sockets, external devices, etc. Throughout this study we assume that a *lock-based* concurrency control mechanism is used to enforce mutually exclusive access of tasks to resources. We call the segment of a task's job that begins at a lock and ends at a matching unlock a *critical section*. The critical section of job J_{ij} on the k^{th} resource will be denoted by ξ_{ij}^k. Moreover, for any of the schedulability conditions to be of any use, the duration of such a critical section has to be bounded. Hence, we will denote the maximum time duration of critical section ξ_{ij}^k by ω_{ij}^k.

Two tasks requiring the same resource are said to have a *resource conflict*. When the scheduler does not grant a resource ρ^k to the task requesting it, the task is said to be *blocked*[5]. If this is the case, a *priority inversion*[6] can occur. More seriously, without a proper resource sharing protocol, the duration of a priority inversion can be unbounded [13]. Furthermore, resource conflicts can also lead to *deadlocks*[7]. Clearly resource conflicts affect the schedulability of our real-time system. We would like to emphasize, however, that resource sharing can only cause a problem if preemption is allowed. In FNP schemes, on the other hand, mutual exclusion is not a problem since a task will never be interrupted while accessing a resource. Below we discuss some of the *resource sharing protocols* that have been proposed in the literature to deal with the above problems.

The Priority Inheritance Protocol

The *priority inheritance protocol* (PIP) [21] prescribes that if a higher priority task becomes blocked by a lower priority one, the task's job that is causing the blocking should execute with a priority which is the maximum of its own priority and the highest priority of the job(s) that is(are) currently waiting on the resource (i.e. it should inherit the higher priority). The PIP is a very

[5]The term *blocked* will be used many times throughout this study to denote a task that is prevented from executing by a lower priority one.

[6]We say a *priority inversion* occurs whenever a lower priority task executes while some higher priority task waits.

[7]A *deadlock* occurs if two tasks A and B require two resources X and Y. Nevertheless, A keeps holding X and requesting Y, while B keeps holding Y and requesting X. In this case the conflict can never be resolved and we say a deadlock has occurred.

simple protocol that works with all priority-driven scheduling algorithms if there are no deadlocks. However, the PIP cannot prevent deadlocks, and a task might be blocked multiple times by each lower priority task that it shares a resource with under the PIP [21].

The Priority Ceiling Protocol

The *priority ceiling protocol* (PCP) extends the PIP to prevent deadlocks and multiple blocking times [22]. In doing so, it assumes that all task priorities are fixed (clearly a situation that only applies to fixed-priority schemes). To this end, the PCP adds a new rule to the PIP: associate each resource ρ^k a priority, called the *priority ceiling* of the resource and denoted by $ceil(\rho^k)$. The priority ceiling is an upper bound on the priority of any task that may lock the resource (which again need be known *a priori*). That is, the priority ceiling of a resource ρ^k is defined as follows:

$$ceil(\rho^k) = \max_i(\pi_i \, | T_i \; can \; request \; \rho^k) \tag{2.4}$$

In addition, at any time instant t_n, the PCP defines a *system ceiling* $\overline{ceil}(t_n)$ to equal the highest priority ceiling of any resource that is in use (i.e. locked) at that time instant, other wise it is set to zero (implying the lowest priority possible).

$$\overline{ceil}(t_n) = \max_k [\{0\} \cup \{ \, ceil(\rho^k) \mid \rho^k \; locked \; at \; t_n\}] \tag{2.5}$$

The PCP rule then states that a task is not allowed to start execution until its priority is highest among the ready or active tasks as well as greater than the system ceiling $\overline{ceil}(t_n)$.

It can be shown that the maximum blocking that a task T_i can experience due to a shared resource under the PCP is given by the following expression:

$$B_i^{rc} = \max_{T_j \in \mathcal{LP}(T_i), \forall h} \{\omega_{jh}^k | \pi_i \leq ceil(\rho^k)\} \tag{2.6}$$

Again, the PCP was shown to bound priority inversion, prevent deadlocks, and a task can be blocked at most once due to a shared resource since no task is allowed to enter its critical section unless its priority is higher than all the priority ceilings currently locked by other jobs (i.e. the system ceiling $\overline{ceil}(t_n)$). However, the PCP applies only to fixed-priority schemes, and therefore a more general resource sharing protocol was needed. such a protocol is presented next.

The Stack Resource Policy

Similar to the PCP, the *stack resource policy* (SRP) developed by Baker [23] is meant to enable real-time jobs to share mutually exclusive system resources while bounding priority inversions and preventing deadlocks. In contrast to the PCP, the SRP can be used with dynamic-priority schemes as well as fixed-priority ones. To this end, Baker noticed that the potential of resource conflicts in a dynamic-priority scheme does not change with time, just as in fixed-priority systems, and hence can be analyzed statically. They key observation here is that a resource conflict can happen between two jobs only if one of them can preempt the other. Baker also noticed that even in a dynamic-priority scheme, it is possible to determine *a priori* the possibility that jobs in each periodic task will preempt the jobs in other tasks. For example, in EDF scheduling, a job of a particular task can only be preempted by jobs of other tasks with smaller relative deadlines.

According to Baker, the possibility that a job of task T_i can preempt another job of any other task can be captured by its *preemption level* λ_i of T_i. In fact, the following essential property is true for all valid preemption level assignments (validly is described next):

Property 2.1.1. *A task T_i can never preempt a task T_j unless $\lambda_i > \lambda_j$*

Under the RM, the DM, and the EDF scheduling policies, it was shown by Baker that a valid preemption level assignment that guarantees that the above property holds is obtained if preemption levels are assigned to be inversely proportional to a task's period:

$$\forall\, T_i,\ \lambda_i \propto \frac{1}{P_i} \tag{2.7}$$

Hence, the preemption level concept defines a static mapping $\Lambda : \mathcal{T} \rightarrow [1, 2, \ldots, N]$ that can be used to statically analyze both static and dynamic priority systems. When the SRP is used together with the RM and DM scheduling policies, each task is assigned a static priority that is inversely proportional to its period. Hence, under the RM and DM scheduling policies, the preemption level mapping is identical to the priority mapping (i.e. $\Lambda \equiv \Pi$). However, when the SRP is used with the EDF, in addition to the static preemption level, λ_i, each job has a dynamic priority that is inversely proportional to its absolute deadline. *We emphasize again, however, that even if we do not know the priorities of jobs in advance as in dynamic-priority schemes, the preemption level mapping provides us with all the needed preemption relations between tasks.*

Similar to the PCP, the SRP assigns for every shared resource ρ^k a ceiling defined as follows:

$$ceil(\rho^k) = \max_i [\{ \lambda_i \mid T_i \; uses \; \rho^k \}] \tag{2.8}$$

Moreover, a dynamic *system ceiling* is defined as follows at each time instant t_n:

$$\overline{ceil}(t_n) = \max_k [\{0\} \cup \{ \, ceil(\rho^k) \mid \rho^k \; locked \}] \tag{2.9}$$

The SRP scheduling rule then states that a task is not allowed to start execution until its priority is highest among the ready or active tasks and its priority level is greater than the system ceiling $\overline{ceil}$.

The blocking that a task can experience due to a resource contention under the SRP can be represented in a very similar form to equation (2.6) with the task's preemption level λ now in place of its priority π. Hence, it can be shown that the blocking time a task T_i can experience due to a shared resource (in either a fixed-priority or a dynamic-priority scheme) with a lower *preemption level* task T_j is given by length of the longest critical section ξ_{jh}^k of T_j on the shared resource ρ^k as follows:

$$B_i^{rc} = \max_{T_j \in \mathcal{T}, \forall h} \{ \omega_{jh}^k | \lambda_i > \lambda_j \wedge \lambda_i \leq ceil(\rho^k) \} \tag{2.10}$$

The SRP ensures that once a task starts execution, it cannot be blocked until completion. The Stack Resource Policy has several interesting properties. It prevents deadlock, bounds the maximum blocking times of tasks, and reduces the number of context switches. From an implementation point of view, it allows tasks to share a unique stack. In fact, a task never blocks its execution; it simply cannot start executing if its preemption level is not high enough.

2.1.5 Schedulability with Shared Resource

It should have been evident that in the presence of shared resources some tasks can miss their deadlines due to blocking. Hence, the schedulability conditions presented in Section 2.1.3 for fixed-priority and dynamic-priority schemes need be modified to account for the possibility of such blocking. In this section we present those extensions assuming that the PCP or the SRP are used (note that the PCP and SRP are equivalent in a fixed-priority scheme).

Fixed-Priority Schedulability with Shared Resources

A task T_i in a fixed-priority system $(\mathcal{T}, \Pi)$ might experience a different kind of worst-case scenario than that of Section 2.1.3 in the presence of shared resource. That is, in addition to interference from higher priority tasks in $\mathcal{HP}(T_i)$, a task T_i might also be blocked by a lower priority task $T_j \in \mathcal{LP}(T_i)$ that is holding a shared resource.

To this end, let ρ^k be a resource required by T_i as well as some other task $T_j \in \mathcal{LP}(T_i)$. Moreover, let us assume that T_j is the task with the longest critical section ξ_{ij}^k on ρ^k of all tasks in $\mathcal{LP}(T_i)$. In this case, the WCRT of T_i will occur if T_j has just locked ρ^k before T_i was released. This WCRT can be computed in terms of the task's (i.e. T_i) worst-case start time and worst-case finish time as given by the following [24]:

$$\mathcal{S}_i(q) = B_i + q \cdot C_i + \sum_{T_j \in \mathcal{HP}(T_i)} \left(1 + \left\lfloor \frac{\mathcal{S}_i(q)}{P_j} \right\rfloor\right) C_j \tag{2.11a}$$

$$\mathcal{F}_i(q) = \mathcal{S}_i(q) + C_i + \sum_{\substack{T_j \in \mathcal{T} \\ \pi_i > ceil(\rho^k)}} \left(\left\lceil \frac{\mathcal{F}_i(q)}{P_j} \right\rceil - \left(1 + \left\lfloor \frac{\mathcal{S}_i(q)}{P_j} \right\rfloor\right)\right) C_j \tag{2.11b}$$

$$R(T_i) = \max_{q \in \{0,1,\ldots,\lfloor L_i/P_i \rfloor\}} (\mathcal{F}_i(q) - q \cdot P_i) \tag{2.11c}$$

Where L_i is the longest level-i busy period and is given by the following:

$$L_i = B_i + \sum_{T_j \in \mathcal{HP}(T_i)} \left\lceil \frac{L_i}{P_j} \right\rceil C_j \tag{2.11d}$$

while B_i denotes the worst-case blocking T_i can experience due to tasks in $T_i \in \mathcal{LP}(T_i)$ sharing the resource ρ^k and is given by equations (2.6) or (2.10). Note that under the PCP as well as the SRP T_i can only be blocked once by task T_j with the longest critical section on resource ρ^k.

Dynamic-Priority Schedulability with Shared Resources

As was mentioned previously, the WCRT of a task cannot be computed for dynamic-priority schemes. Nevertheless, a sufficient condition for the schedulability of a dynamic-priority system $(\mathcal{T}, \Pi(t_n))$ was developed by Baker [23] if the system is transformed into a *statically-analyzable* system through the preemption level mapping explained in Section 2.1.4. To this end, given the system $(\mathcal{T}, \Lambda)$, we proceed as follows.

Let us re-order our system $\mathcal{T}$ such that for any T_i and T_j having $i < j$ implies that $\lambda_i > \lambda_j$. Now let T_i be any task in $\mathcal{T}$ that requires the shared resource ρ^k to make progress. Moreover, let $T_j \in \mathcal{T}$ be the task with the longest critical section ξ_{ij}^k on the resource ρ^k such that $\lambda_j \le \lambda_i$. The maximum blocking that T_i will experience due to T_j holding the resource ρ^k is given by (2.10). A sufficient condition for the schedulability of T_i is then given by [23]:

$$\frac{B_i^{rc}}{D_i} + \sum_{j=1}^{i} \frac{C_j}{D_j} \le 1 \tag{2.12}$$

Where B^{rc} is again the maximum blocking given by (2.10). We say that the entire system is schedulable if the above condition holds for every $T_i \in \mathcal{T}$.

2.2 Integrated Circuits Energy and Power Dissipation

We now present background and related work on energy and power dissipation. We first examine CMOS-based circuit power and energy requirements, memory hierarchy energy use, power and energy management techniques, and power schedulers.

2.2.1 Modeling Power and Energy Dissipation

The total power dissipation in a CMOS-based circuit has been modeled as the sum of two components as given by the following [11]:

$$P_{CMOS} = P_{dyn} + P_{static} \tag{2.13}$$

The first component in (2.13) is known as the *dynamic-power* dissipation component. This component arises from: (1) the switching current from charging and discharging paratactic capacitances, and (2) the short circuit current resulting from *n*-channel and *p*-channel transistors being momentarily ON at the same time. An expression that accounts for both of these sources is given by the following [11]:

$$P_{dyn} = C_P \; f_{CLK} \; V_{CC}^2 + \frac{\beta \; f_{CLK} \; t_{rf}}{12}(V_{CC} - 2V_{tn})^3 \tag{2.14}$$

where f_{CLK} and V_{CC} are the clock frequency and the operational voltage respectively, β is a constant that depends on the transistor's geometry and fabrication technology, V_{tn} is the transistor's threshold voltage, t_{rf} is the rise and fall time of the input signal (assumed equal), and C_P is the gates parasitic capacitance in units of Farads. However, it has been shown that the second component in (2.14), namely the dynamic-power due to short-circuit currents, is typically less than 15% of the dynamic-power dissipation and can safely be ignored [25]. The resulting expression for the dynamic-power dissipation component is thus given by the following:

$$P_{dyn} = C_P \; f_{CLK} \; V_{CC}^2 \tag{2.15}$$

By choosing an infinitesimal period of time Δt such that the clock frequency, f_{CLK}, and the operational voltage V_{CC} are constant over that period of time, the *dynamic-energy* component can now be expressed as follows:

$$E_{dyn} = \int C_P \; f_{CLK} \; V_{CC}^2 \; dt = C_P \; f_{CLK} \; V_{CC}^2 \Delta t \tag{2.16}$$

Moving our attention to the second component in (2.13), namely P_{stat}, which is known as the *static-power* component and is independent of the switching behavior of the circuit. This component arises due to subthreshold leakage currents as given by the following expression:

$$P_{stat} = I_{lkg} V_{CC} \tag{2.17}$$

where V_{CC} is again the operational voltage, and I_{lkg} is the leakage current given by the following expression [26]:

$$I_{lkg} = \mu C_{ox} (W/L) V_t^2 e^{\frac{V_{CC} - V_{th}}{n\, V_t}} \left(1 - e^{-\frac{V_{CC}}{V_t}}\right) \tag{2.18}$$

where μ is the electron carrier mobility, C_{ox} is the gate capacitance per unit area, W and L and the channel width and height respectively, V_t is the thermal voltage, V_{th} is the threshold voltage, and n is the subthreshold swing coefficient.

Assuming again that we choose an infinitesimal period of time Δt such that the operational voltage and leakage current and almost constant over that period of time, then the *static-energy* dissipation component in a CMOS circuit is given by the following:

$$E_{stat} = \int I_{lkg}\, V_{CC}\; dt = I_{lkg}\, V_{CC} \Delta t \tag{2.19}$$

A final relation that we will be using relates the maximum operating frequency of a CMOS circuit to its operational voltage. This relation is given by the following [27, 11, 28]:

$$\max[f_{CLK}] = \frac{K_p\, (V_{CC} - V_{th})^{1.8}}{V_{CC}} \tag{2.20}$$

where K_P is another constant of proportionality that depends on the circuit's gate delay.

2.3 Energy Dissipation in a Memory Hierarchy

Before we move on to presenting the most popular energy and power management techniques, we would like to present some of the models developed to model energy dissipation of memory subsystems in a memory hierarchy architecture. In this special case there are many factors that have to be accounted for like the size of the memory units, the distance between the various memory modules, and the fabrication technology used to fabricate the various memory modules.

Without delving in the derivation of the individual models, it was shown that the amount of energy required per access is given by a simplified model in terms of the energy required per read e_r and the energy required per write e_w which are both given by the following equations [29]:

$$e_r = \eta_r \cdot \sqrt{b \cdot N_{word}} + \kappa_r \; [pJ/cycle] \tag{2.21a}$$

$$e_w = \eta_w \cdot \sqrt{b \cdot N_{word}} + \kappa_w \; [pJ/cycle] \tag{2.21b}$$

Where η_r, η_w, κ_r, and κ_w are technology dependent with the basic property that $\eta_r \leq \eta_w$, and $\kappa_r \leq \kappa_w$ for all technologies.

Using the models given by equations (2.21a) and (2.21b), we will be able to analyze the energy efficiency attainable by DARTS in Chapter V. It is important to emphasize, however, that those are only simplified models and other more accurate models that account for many other factors like the wiring overheads, the energy cost of each additional memory unit added to the hierarchy exist but are not required at this stage of our analysis.

2.3.1 Energy and Power Management Techniques

In this section we present a brief survey of the most popular energy and power management techniques. We start by presenting some of the less sophisticated techniques, followed by more complex and more sophisticated ones.

Power Down Modes

Every processor designed in the last two decades usually includes a set of additional states known as *power-down modes* (PDM). Basically, PDMs are a set of additional processor states where some modules of the system has the power removed (i.e. the instruction sequencer, some peripherals like on-chip ADCs, etc). PDMs usually include several variations depending on the degree of power (energy) that need be minimized (e.g. idle mode, sleep mode, etc).

To avoid confusion, we will only be interested in PDMs where most of the chip peripherals have been disabled leading to the smallest power dissipation possible. In addition, we will only be concerned with the available PDMs that can be controlled internally in software. That is, only PDMs that can be disabled as well as enabled in software will be considered. This is necessary since, as will be explained later, the power and energy management of the system is usually handled by the *power scheduler*, typically part of the operating system, which has to be able to control those PDMs directly in software.

Dynamic Frequency Scaling

Dynamic frequency scaling (DFS) is a technique in which the processor clock is scaled down to minimize the power dissipated linearly as can be seen from (2.15). We need to emphasize, however, that DFS alone can only lead to minimizing the power and not the energy dissipation of a system. The reason is that at a lower frequency it will take the system more time to process the same workload, and hence, no energy can be saved. It will be shown later, however, that in combination with other power and energy management techniques, DFS does indeed lead to energy savings for *under utilized* systems.

If the processor does not already support DFS internally, implementing DFS using external hardware is simple and cost efficient. Two techniques are possible; the first uses inexpensive simple counter(s) to divide the frequency by an integral value, while the second (clock throttling) uses gates to disable the clock signal periodically and is more complex [30]. It should also be clear that a power scheduler is needed to calculate the frequency levels required to execute the various jobs (task instances).

Dynamic Voltage Scaling

Dynamic voltage scaling (DVS) reduces the power and energy dissipated by a processor through scaling down the operating voltage, and in turn the clock frequency (see (2.20)). In contrast to DFS, DVS also minimizes the static power (and energy) component which is only a function of the operational voltage (2.19). A disadvantage of DVS, on the other hand, is its complexity and expensive implementation.

Implementing an effective DVS system has several requirements, as pointed out by Burd et al. [25]: (1) A variable power supply capable of generating the required voltage levels with a high voltage transition rate, minimal transition energy losses, and a good voltage transient response, (2) a wide operational voltage range for the circuit to be powered, (3) and a power scheduler that can intelligently compute the appropriate frequency and voltage levels needed to execute the various jobs is also required.

The power supply is an essential component of a DVS system as it enables the voltage scaling mechanism. Two important parameters of any power supply are the *transition time* (also known as the *tracking time*), and the *transition energy* (also known as the *tracking energy*). The transition time is simply the time it takes to change the output voltage from one stable state to another, while the transition energy is just the amount of switching losses incurred during that voltage transition. An equivalent metric to the transition time is the *voltage transition rate* (or *tracking rate*), which is just the difference in the output voltage levels between the two states divided by the transition time.

The main objective of any power supply is to change the output voltage from one stable state to another within a defined time, and with the minimal transition energy possible. Unfortunately, various physical and cost constraints exist that limit the achievable minimum transition time and energy. For example, as explained by Burd et al. [25], any processor produces large current spikes which the converter's output capacitor must filter. Hence, a large output capacitor is desirable to filter and stabilize the output voltage. Nevertheless, it was shown by Burd that the transition time and transition energy are both directly proportional to the size of this capacitor. Hence, a large output capacitor implies a large time constant and more energy losses (i.e. less efficiency). Trade-offs exist in several other design dimensions, and hence, a faster converter with minimal energy losses will usually imply higher costs which might not be adequate especially for a one dollar microcontroller unit [25].

Two categories of variable voltage supplies with high transition rates, low transition energy dissipation, and good transients have been used for DVS. The ideal and most efficient approach uses custom-designed hardware (on-chip when possible). Two such designs were reported by Burd et al. [25], and another by Gutnic et al. [31]. Both achieve low transition energy dissipation, excellent transients, and Burd's has a voltage transition rate on the order of 50 volts per sec. The second category of variable voltage supplies are commercial DC-DC converters designed for DVS. For example, the TPS62300 high-frequency buck converter renders high efficiency as well as good transients. However, its voltage transition rate is much smaller (about 20 millivolts per sec). Still, both of these methods will not be adequate for many systems where the cost is important. In fact, many low-end embedded microcontrollers can run for under one dollar [32], making these dedicated power supplies an extremely expensive option.

A final important point for implementing DVS is the state of the processor during the transition between voltage levels. Ideally, one would like to keep the processor running during such transitions. Nevertheless, as pointed out by Qu [33], any practical variable voltage system will have to stop instruction execution during voltage transitions until a stable steady state has been reached. This is very significant since the longer the transition takes (i.e. long transition times or equivalently small voltage transition rates), the less time the processor has to finish executing its workload. This, in fact, is the main drawback of dynamic voltage scaling since DVS systems with long transition times are very susceptible to the workload's granularity. The more jobs, the more transitions, and the more time spent switching, reducing the time available for actual workload execution.

2.3.2 Real-Time Scheduling for Energy Management

A scheduler is a crucial component in implementing any power or energy management technique. A scheduler is responsible for: (1) deciding when the processor can reduce its power consumption in a way that does not affect its overall performance, (2) and by how much should the processor reduce its power consumption, again without affecting the overall performance. Various scheduling policies and algorithms have been proposed for both non real-time and real-time systems. These assume that a FP RTOS is present into which the power scheduler can be incorporated. We note, however, that this might not be the case for low-end computing platforms where memory is scarce, and application software is much less sophisticated. In fact, several low-end embedded applications simply operate on a foreground-background (interrupt-driven) policy. Hence the use of a power scheduler for such systems becomes more complicated since no *central* program knows what the other applications are doing. Moreover, the scheduling policy will actually set the quality of the power savings. An investigation of this topic will be presented later in this study. For now, however, we present a brief overview of some of the power scheduling policies used.

Power Scheduling in a Non-Real-Time Environment

For non-real-time systems, most scheduling policies try to minimize the power dissipation of the system while maintaining a constant throughput. Individual task deadlines are not accounted for since no real-time constraints exist. An operating system is usually present and most scheduling policies just incorporate the scheduler into it. Most schedulers targeting non-real-time systems execute two main steps [34], the first step is known as the *prediction step*, while the second step is known as the *speed setting step*. The prediction step predicts how busy the CPU will be during some future interval by predicting its future utilization. Some of the most popular prediction algorithms are the PAST algorithm (where the algorithm predicts that the upcoming interval's utilization is the same as the last interval utilization), the AGED-a algorithm (predicts that the upcoming interval's utilization will be the average of all last a intervals), and the $FLAT - \mu$ algorithm (predicts that the upcoming interval's utilization will be equal to some constant $\mu \leq 1$). The speed setting step then uses this information to scale the supply voltage and clock frequency accordingly. There are several popular speed-setting algorithms. In the Weiser-style algorithm, if utilization was high during the previous interval ($U > 70\%$) then increase the speed by 20% of the maximum for the next interval. If U was low during the previous interval ($U < 50\%$) then decrease the speed by $60 - U\%$ of the maximum speed for the next interval. In the Peg algorithm, if $U > 98\%$ for the previous interval, set the speed to its maximum for the next interval. If $U < 93\%$ for the previous interval, decrease speed to its minimum for the next interval. Finally, in the Chan-style algorithm, multiply the maximum speed by U of the previous interval to get the speed of the upcoming interval [34].

Power Scheduling in a Real-Time Environment

For real-time systems, maintaining a constant throughput does not guarantee that individual tasks will meet their deadlines [35]. Hence, scheduling for real-time systems is much more critical. To this end, the scheduler usually is integrated into a preemptive RTOS to be able to provide the needed power savings while preserving deadlines guarantees. Several policies and algorithms have been proposed in the literature for real-time systems as well [28, 36, 37, 30].

2.4 The DARTS Tool Chain

The DARTS (*data allocation and real-time scheduling* tool chain was developed during working on this thesis to test and simulate many of the strategies investigated in this thesis. A block diagram showing the different components of our DARTS tool chain is shown in figure 2.1. A brief overview of these components and the main concepts they are based upon is presented in this section.

2.4.1 Tool Chain Overview

Figure 2.1 presents a block diagram of the tools used in this work. As a whole, the tool chain presents a real-time scheduling, analysis, and visualization tool chain. Though the tool chain components can be used separately, we present in this chapter an overview of how they can be used together for analyzing, real-time scheduling, and allocating the data objects of a user's application to a cacheless platform[8].

[8]The tool chain was developed targeting the AVR architecture [38]. Nevertheless, most of the tool chain components can be easily ported to support other architectures as well.

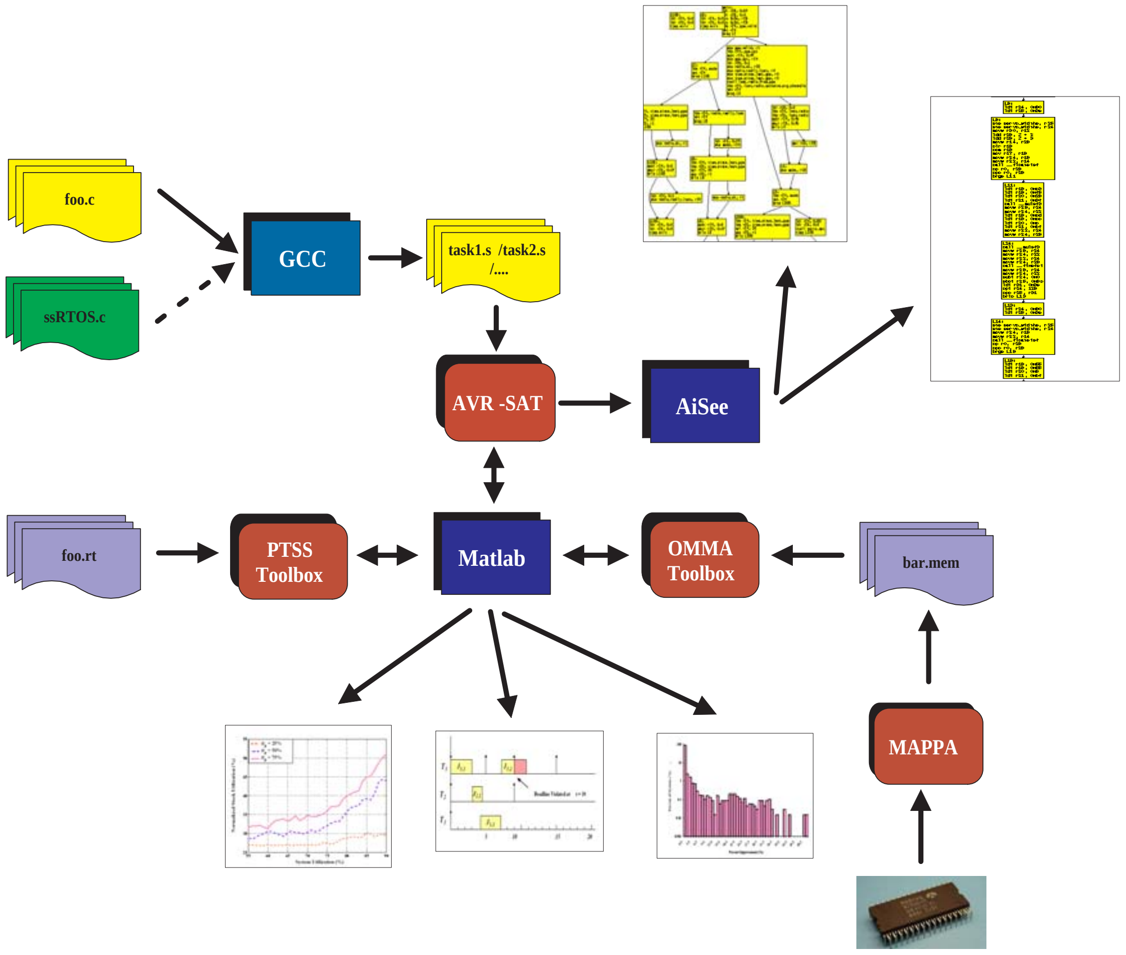


Figure 2.1: The DARTS simulations and analysis tool chain

To this end, a user's source code file(s) (which can be compiled separately or with the source code of a real-time kernel like the *single-stack real-time operating system* (ssRTOS) presented later in this chapter) is(are) compiled into assembly code using AVR-GCC [39]. At this step, it is assumed that the assembly code has been divided into a single assembly file for each of the logical real-time tasks composing the application. The assembly code files (task1.s / task2.s / ...) are processed by AVR-SAT (*AVR-Static Analysis Tool*) which constructs several data structures including the each task's *call graph* (CG) and each procedure's *control flow graph* (CFG), both used to compute each task's *worst-case path* (WCP). The WCP can either be graphically visualized using the AiSee tool [40], or fed to Matlab [38] for further analysis.

Around Matlab, two main tool chain components have been developed: the *preemption* *threshold scheduling and simulation* (PTSS) toolbox, and the *optimal multitasking memory allocation* (OMMA) toolbox. The PTSS toolbox uses the real-time characteristics of the application (i.e. task periods, deadlines, etc.) to compute the optimal preemption threshold assignment (using the MPTAA algorithm presented in later chapters of this thesis), and can simulate the operation of the system and output the results of this simulation in a timing diagram (also known as a Gantt chart). On the other hand, the OMMA toolbox, given the system's memory architecture, uses *integer linear programming* (ILP) to find the optimal memory allocation resulting in the optimal performance. We present below a more detailed discussion of the various tool chain components and their operational concepts.

2.4.2 The AVR Static Analysis Tool

The AVR-SAT have been developed with the main goal of extracting the WCP of from a task's code. Given the WCP of a particular task, we can use it to compute the task's worst-case execution cycles (WCEC). This is used by our real-time scheduler and simulator, the PTSS toolbox. Moreover, the memory traffic (represented by the number of loads and stores) along this WCP is further used by the OMMA toolbox to allocate the various task's data objects to memory optimally.

AVR-SAT Operation

Given the assembly file (taskN.s) for one of the application's tasks, the AVR-SAT tool performs multiple steps to extract the WCP of an application. First, a simple grammar is used to parse the assembly file. The parser was implemented using the traditional Lex and Yacc tool chain [41]. Second, the calling behavior of each procedure is analyzed to construct the task's CG (this CG is used later for constructing the memory allocation constraints). The AVR-SAT then constructs the CFG for each procedure. Once all procedures' CFGs have been constructed, AVR-SAT recursively traverses the call graph to compute the WCP of leaf node procedures (i.e. all procedures that have no calls). Once all the WCPs of all leaf node procedures have been extracted, the parents of these procedures are then traversed and so on.

In computing the WCP of a procedure, another recursive function is used that uses a *breadth first search* (BFS) *like* algorithm. Once the task's WCP has been found, it is traversed to record the number of loads and stores along this path (for both global and stack variables) which are required later by the OMMA toolbox for performing data allocation.

2.4.3 The Preemption Threshold Scheduling and Simulation Toolbox

The PTSS toolbox was developed before any of the other tool chain components presented in this chapter. The main goal behind the PTSS toolbox is to schedule a real-time application in some optimal manner. As will be explained later in this study, PTS can be used limit the preemptions in a real-time application such that it only requires the minimal memory for its data structures. To this end, an algorithm known as the MPTAA (or the maximal preemption threshold assignment algorithm) can be used to perform this optimal scheduling.

The PTSS toolbox constructs several data structures to enable the analysis, scheduling, and simulation of a real-time workload. To this end, the MPTAA (see Section 3.3) is used to assign the optimal preemption thresholds to each task that minimize the stack space of each task while maintaining its schedulability. The schedule can then be plotted, simulated, or used by the OMMA toolbox for further optimizations.

2.4.4 The Optimal Multi-Tasking Allocation Toolbox

The *optimal multi-tasking allocation* (OMMA) toolbox was developed to compute the optimal memory allocation for a set of data objects (analyzed using the AVR-SAT, or the AVR-static analysis tool, described earlier) to a memory hierarchy composed of multiple heterogeneous memory units with different latencies, sizes, and / or bit-widths.

2.4.5 The Single-Stack Real-Time Operating System

The *single-stack real-time operating system* (ssRTOS) was constructed to experiment with many of the scheduling policies and protocols available in the literature. ssRTOS utilizes a single task for all of the system tasks. At this point of the development, the only restriction is that tasks cannot suspend themselves. In other words, once a task gets the processor for execution it can only be preempted by a higher priority task but can never voluntarily suspend itself. This was a major requirement to avoid stack corruption if a task that has suspended itself and resume execution after another task has changed the stack context, stack data corruption can results.

On the other hand, ssRTOS does support mutual exclusion of resources by using semaphores. A task that might request a particular semaphore is not allowed to start execution unless all of the semaphores it might need during execution are available. This is just another way of applying Baker's SRP, where a task can never be blocked once it starts execution since all its resources are available.

In addition to the implemented SRP policy, the ssRTOS also supports PTS. In other words, each task is assigned a preemption threshold in addition to its nominal priority. Once a task starts execution, it can only be preempted by task with higher preemption thresholds. This, in effect, creates groups of task can are mutually non-preemptive and can share the same memory space at run-time. Additional information on ssRTOS is available through its user manual [42].

2.4.6 The Microcontroller Automated Power Analyzer

The *microcontroller automated power analyzer* (MAPA) was developed to evaluate the power and energy characteristics of may microcontrollers. To this end, MAPA was developed to measure and record the supply current utilized by the target microcontroller at different supply voltages and clock frequencies. It uses a simple 8-bit microcontroller, an external oscillator, a few programmable counters, and a few op-amps including a power op-amp.

Chapter 3

Optimizing Memory Utilization with PTS

As application complexity increases, it often becomes necessary to use a *real-time operating system* (RTOS) for modularizing the application into easily manageable segments or modules while ensuring real-time constraints are met. The benefits of using an RTOS are well known and have been heavily documented. However, as was discussed in the introductory chapter, memory (especially RAM) is a very valuable and scarce resource, and an RTOS requires significant memory support. The reason is that most real-time operating systems employ a fully-preemptive scheduling policy which can have excessive preemption overheads and require significant memory space to support the preemption overheads. For example, many low-end RTOS's *statically* allocate a dedicated space in RAM (e.g. uC/OS-II, FreeRTOS, AvrX, etc.) for each task. This memory space is referred to as the tasks stack which it uses for nested function calls, return addresses, and saving its context if it is to be preempted. This memory overhead can be prohibitively expensive for systems with little RAM or many tasks [43,44,45]. Moreover, this stack space must be large enough to accommodate worst-case function call nesting, local variable allocation, and possibly the task control block and context. In addition, many kernels also allocate additional space in each task's stack for servicing interrupts (e.g. uC/OS-II and

FreeRTOS)[1].

Many high-volume embedded systems are built around low-end *commercial off-the-shelf* (COTS) microcontrollers because of their low cost. These devices may have very little RAM. Even in systems with larger amounts of RAM, running out of memory can be a common problem. Embedded systems tend to evolve over time, with each new generation of software accreting a new layer onto the existing code base. Hence, over a long enough periods of time, data memory requirements can easily exceed available memory making RAM a precious resource. This makes the use of real-time kernels for low-end to mid-range embedded platforms particularly hard if not impossible.

These issues and problems inherent in fully-preemptive real-time kernels have led to the emergence of several methods and standards for designing more efficient kernels with reduced preemption overheads. As the main consumer of low-end microcontrollers, the automotive industry was the first to realize the need for more efficient operating systems by developing the OSEK / VDX standard[2]. Several OSEK / VDX-compliant real-time kernels have been developed. Live Devices (Realogy) [46, 47] introduced several variations of their *real-time architect* (RTA) operating system that utilizes the *single-shot execution* (SSX) model which enables tasks to share a single stack while dividing them into *mutually non-preemptive groups*. Many other OSEK / VDX compliant real-time operating systems were also developed to be memory efficient in some way [48, 49]. Outside of the automotive industry, other efficient RTOS design techniques have been developed. One very promising technique is PTS which was briefly presented in Section 2.1.2. PTS tries to minimize preemptions as much as possible while preserving the system's schedulability.

Minimizing preemptions to reduce their overheads and improve the system's memory utilization is an ad-hoc process which has not been adequately examined to date. For example, several questions naturally arise that the existing techniques and tools fail to answer. What are the memory savings attainable through preemption limiting? What are the application char-

[1]Some kernels use more efficient techniques for servicing interrupts like emulating a separate interrupt stack if not supported in hardware like AvrX [45].

[2]The OSEK / VDX standard includes specifications for embedded operating systems, communication subsystems, and embedded network management systems.

acteristics that make a particular preemption limiting technique more suitable than another? Given the maximum stack space that a task can require, is there a way we can in a similar way know the minimum stack requirements that would maintain the system properties? Unfortunately, such questions had no general answers and were only addressed on a case by case basis.

In this chapter we try to answer these and other questions. First, we build a framework for PTS that naturally applies to both fixed-priority and dynamic-priority schemes. Second, we analyze and enhance PTS and show that when used in combination with any of the well known scheduling algorithms (e.g. RM, DM, EDF, etc.) it will result in the smallest possible stack space attainable by any preemption limiting technique without changing the particular task model. Hence, given a particular real-time application, we provide the system developer with a limit on the amount of memory that can be saved by limiting preemptions while maintaining the system schedulability. This feasibility test has a computational complexity of $O(N^2)$ as opposed to ad hoc methods that are usually exponential in the number of tasks. Third, we discuss some characteristics of PTS, including undesired side-effects of preemption limiting in general (e.g. deteriorated system responsiveness and robustness), and show how some can be reduced. Finally, through simulations, we quantify application characteristics (e.g. workload utilization, type of scheduling scheme used, etc.) that affect PTS to help developers of real-time applications assess the design tradeoffs in a timely and systematic manner.

3.1 Related Work

As was explained earlier, PTS was first introduced by Express Logic Inc. in their real-time kernel ThreadX [18]. The real-time analysis of this dual-priority scheme was pioneered by Wang et al. [16, 17] and by Davis et al. [47]. In these studies, preemption between tasks was limited to occur only when necessary to maintain system schedulability. Tasks that run non-preemptively with respect to each other (referred to as a *mutually non-preemptive group*) can be mapped into the same run-time thread and share the same run-time stack minimizing the memory requirements and other preemption overheads. Our work builds directly on Wang's et al. work by proving the optimality of their preemption threshold assignment method as well as extending it to support more general dynamic-priority schemes.

Many resource sharing protocols and real-time synchronization schemes have been adapted to PTS [50, 51]. Kim et al. [51] showed how the priority inheritance and priority ceiling protocols could be used with PTS when shared resources are present. Gai et al. [50] showed how Baker's stack resource policy [23] can be extended to support PTS, resulting in the *stack resource policy with threshold* (SRPT). Similar to Gai et al. [50], in this work we use the SRP to show how shared resources can easily be incorporated in our framework.

Since dynamic-priority task scheduling schemes can be much more efficient than static-priority schemes, it is of particular interest to know if PTS applies to these schemes. Gai's [50] work enables the use of PTS in dynamic-priority schemes through the use of the preemption level concept, also introduced earlier by Baker [23], which enable static analysis of dynamic-priority systems. Another dynamic preemption threshold scheme was presented by He et al. [52] which uses an earliest-deadline-first algorithm to dynamically change the task priorities as well as their preemption thresholds. Our work on PTS for dynamic-priority schemes builds on Gai et al. and Baker's work since, in contrast to He's et al. [52] work, we enable the system developer to analyze and verify a design at design time while avoiding all the additional run-time overheads of dynamically updating preemption thresholds in addition to priorities. To this end, we modify an existing preemption thresholds search algorithm to apply to both fixed- and dynamic-priority schemes. The search can be done offline to obtain the optimal preemption threshold assignment which remains fixed at run-time (incurring no overhead).

Many other studies investigated PTS and presented extensions to its basic concepts. Building on Wang's work for fixed-priority schemes, Regehr [24] presented two abstractions that can be used to force the PTS scheduler to group tasks into non-preemptive groups. Regehr also investigated the presence of WCET uncertainty and presented algorithms for finding fault-tolerant PTS schedules. The incorporation of PTS into many real-time design settings was also addressed. Kim et al. [53] showed how PTS can be used with dynamic voltage scaling to render efficient schedules that optimize the energy usage of the system. Saksena et al. [54], and Wang et al. [17] also addressed using PTS in a real-time object oriented framework, showing how object-oriented models of real-time systems can be synthesized automatically into a real-time task set to be implemented using a fixed-priority PTS scheme.

3.2 Unified Schedulability Analysis Framework

The schedulability conditions for fixed-priority and dynamic-priority systems in the absence of shared resources were presented in Sections 2.1.3. If there are shared resources, those schedulability conditions were modified and presented in Section 2.1.5. In this section we extend those conditions to apply to fixed-priority and dynamic-priority schemes when PTS is used. Before we proceed, however, we define in the following subsection the notation used to build our unified framework.

3.2.1 System Notation

As we recall from Section 2.1.2, a real-time workload can be scheduled according to either a fixed-priority or a dynamic-priority policy. A fixed-priority policy is completely characterized by the fixed priority mapping $\Pi : \mathcal{T} \rightarrow [1, 2, \ldots, N]$. Nevertheless, if a dynamic-priority policy is used, the priority mapping is dynamic and cannot be analyzed statically.

To treat both fixed-priority and dynamic-priority schemes in unified scheme, we will use the preemption level mapping $\Lambda : \mathcal{T} \rightarrow [1, 2, \ldots, N]$ for both fixed and dynamic schemes. As was explained in Section 2.1.4, the preemption level mapping is equally applicable to both schemes enabling their static analysis. Hence, whenever the particular scheduling policy used is of no importance (i.e. whether fixed or dynamic) we will denote our real-time system by the tuple $(\mathcal{T}, \Lambda)$ without any mention to the actual nature of the scheduling policy.

In this framework, for any task $T_i \in \mathcal{T}$ we define $\mathcal{HL}(T_i)$ to be the subset of all tasks belonging to $\mathcal{T}$ with preemption levels that are strictly larger than that of T_i (i.e. $\mathcal{HL}(T_i) = \{T_j \in \mathcal{T} | \lambda_j > \lambda_i\}$). Similarly, let $\mathcal{LL}(T_i)$ denote the subset of all tasks belonging to $\mathcal{T}$ with preemption levels strictly smaller than that of T_i. Note that if a fixed-priority scheme is used, then

$\mathcal{HL}(T_i) = \mathcal{HP}(T_i)$ and $\mathcal{LL}(T_i) = \mathcal{LP}(T_i)$ for any $T_i \in \mathcal{T}$. On the other hand, if a dynamic-priority scheme is used, then we cannot make any assumptions about the priority relations between the tasks. However, according to property 2.1.1 presented in Section 2.1.4, we know that no task in $\mathcal{LL}(T_i)$ can *preempt* task T_i. Similarly, only tasks in $\mathcal{HL}(T_i)$ can (but might not) preempt T_i. Based on these observations we can state the following two important properties (which hold for both fixed and dynamic scheduling schemes):

Property 3.2.1. *A task $T_i \in \mathcal{T}$ can only be preempted by tasks in $\mathcal{HL}(T_i)$ independent of the priority relations*

Property 3.2.2. *A task $T_i \in \mathcal{T}$ can never be preempted by tasks in $\mathcal{LL}(T_i)$ independent of the priority relations*

In summary, the preemption level mapping will be used as an alternative to the priority mapping to enable a unified treatment of both priority schemes. This is possible since, as was explained previously, the former maintains all of the properties of the latter. Additionally, we will say that task A has a higher (lower) priority than task B to mean that task A has a higher (lower) preemption level than task B, whenever there is no fear of confusion. Using this notation and definitions, we now proceed to build our unified schedulability analysis framework under PTS.

3.2.2 Total Blocking

As was explained in Section 2.1.4, a task might be blocked by a lower priority task that is locking a shared resource. Yet another reason for blocking is the non-preemptability of a lower priority task when PTS is used. As was mentioned previously, in PTS, we associate an additional mapping $\Gamma : (\mathcal{T}, \Lambda) \rightarrow [1, 2, \ldots, N]$ such that a task T_i cannot preempt a lower priority task $T_j \in \mathcal{LL}(T_i)$ unless $\gamma_i > \gamma_j$.

It was shown by Wang et al. [16] that under PTS, a task can be blocked at most once. Hence, the blocking time that a task T_i can experience due to the non-preemptability of a lower priority task $T_j \in \mathcal{LL}(T_i)$ is given by the following:

$$B_i^{pts} = \max_{T_j \in \mathcal{LL}(T_i)} [C_j - 1] \tag{3.1}$$

Nevertheless, in the presence of shared resources a task T_i might also be blocked by a lower priority task $T_l \in \mathcal{LL}(T_i)$ that is locking some needed resource. The duration of this blocking was given previously and is repeated here for convenience:

$$B_i^{rc} = \max_{T_l \in \mathcal{LL}(T_i), \forall h} \{\omega_{lh}^k | \lambda_i \leq ceil(\rho^k)\} \tag{3.2}$$

Since under the SRP a task can be blocked at most once, the duration of the blocking experienced by a task T_i due to resource contention as well as the non-preemptability of lower priority tasks can be represented by the following *total blocking* expression:

$$B_i^{total} = \max[B_i^{rec}, B_i^{pts}] \tag{3.3}$$

Given that total blocking time a task can experience due to either a shared resource or the use of PTS, we present the modified schedulability conditions in the following section.

3.2.3 Fixed-Priority PTS Schedulability

Schedulability conditions for fixed-priority fully-preemptive systems with and without shared resources were presented earlier. Under PTS, the WCRT of a task T_i happens in one of two cases: (1) if the task $T_j \in \mathcal{LL}(T_i)$ with the longest critical ξ_{jh}^k section on the shared resource ρ^k has just locked it before the release of T_i, or (2) if the task $T_l \in \mathcal{LL}(T_i)$ with the largest WCET C_l and a preemption threshold $\gamma_j \geq \gamma_i$ was released one clock cycle before T_i. In both cases, the WCRT of T_i is given in terms of its worst-case start time and worst-case finish time as follows:

$$R(T_i) = \max_{q \in \{0,1,\ldots,\lfloor L_i/P_i \rfloor\}} (\mathcal{F}_i(q) - q \cdot P_i) \tag{3.4a}$$

$$\mathcal{S}_i(q) = B_i^{total} + q \cdot C_i + \sum_{T_j \in \mathcal{HL}(T_i)} \left(1 + \left\lfloor \frac{\mathcal{S}_i(q)}{P_j} \right\rfloor\right) C_j \tag{3.4b}$$

$$\mathcal{F}_i(q) = \mathcal{S}_i(q) + C_i + \sum_{\substack{T_j \in \mathcal{T} \\ \lambda_i > \gamma_j}} \left(\left\lceil \frac{\mathcal{F}_i(q)}{P_j} \right\rceil - \left(1 + \left\lfloor \frac{\mathcal{S}_i(q)}{P_j} \right\rfloor \right) \right) C_j \tag{3.4c}$$

where L_i is the longest level-i busy period and is given by the following:

$$L_i = B_i^{total} + \sum_{T_j \in \mathcal{HL}(T_i)} \left\lceil \frac{L_i}{P_j} \right\rceil C_j \tag{3.4d}$$

while B_i^{total} denotes the total blocking T_i can experience due to tasks in $\mathcal{LL}(T_i)$ which are either locking a shared resource or cannot be preempted due to their preemption threshold value and is given by equation (3.3). Again, our system is schedulable if and only if $R(T_i) \leq D_i$ for all $T_i \in \mathcal{T}$.

3.2.4 Dynamic-Priority PTS Schedulability

In this section we extend the schedulability analysis under PTS to dynamic-priority schemes. Since the blocking experienced by a task is the same independent of the priority scheduling scheme, the blocking a task T_i can experience in a dynamic-priority scheme is the same as that in a fixed-priority scheme as given by equations (3.1), (3.2), and (3.3). From equations (3.5a) in Section 2.1.5, we know that a dynamic-priority system with blocking due to shared resources is schedulable if the following holds for every $T_i \in \mathcal{T}$:

$$\frac{B_i^{rc}}{D_i} + \sum_{j=1}^{i} \frac{C_j}{D_j} \leq 1 \tag{3.5a}$$

Though the above condition is only sufficient, we can use it to develop another sufficient condition that applies to dynamic-priority systems under PTS. To this end, we can use the above conditions to define the following *maximum blocking* term for any task $T_i \in \mathcal{T}$:

$$B_i^{max} = \left(1 - \sum_{T_j \in \mathcal{HL}(T_i)} \frac{C_j}{P_j} \right) P_i \tag{3.5b}$$

Since the above definition only applies to the special case were $D_i = P_i$ for all $i = [1, 2, \ldots, N]$, we need to use the EDF schedulability conditions to extend the above expression as follows:

$$B_i^{max} = \left(1 - \sum_{T_j \in \mathcal{HL}(T_i)} \frac{C_j}{\min(D_j, P_j)}\right) \min(D_j, P_j) \tag{3.5c}$$

Informally speaking, the maximum blocking defined by the above equation provides us with a limit that if we can guarantee that the total blocking of a task is no larger than that limit, then it is schedulable. In other words, a sufficient condition for a real-time system $\mathcal{T}$ in a dynamic-priority scheme to be schedulable under PTS is given by the following:

$$\forall T_i \in \mathcal{T} : \; B_i^{total} \leq B_i^{max} = \left(1 - \sum_{T_j \in \mathcal{HL}(T_i)} \frac{C_j}{\min(D_j, P_j)}\right) \min(D_j, P_j) \tag{3.6}$$

Where B_i^{total} is the total blocking given by equation (3.3).

3.3 Stack Space Optimality of PTS

Given the real-time system $(\mathcal{T}, \Lambda)$, we would like to find a preemption threshold mapping Γ that is feasible and in some sense optimal. A preemption threshold assignment Γ is *feasible* if and only if the workload is schedulable according to the conditions given by (3.4a), (3.4b), (3.4c), and (3.4d) for a fixed-priority scheme, and by (3.6) for a dynamic-priority scheme. Hereafter, the set of all feasible preemption threshold assignments for the system $(\mathcal{T}, \Lambda)$ will be denoted by $\mathcal{G}(\mathcal{T}, \Lambda)$. A particular feasible assignment of special interest is defined below:

Definition 3.3.1. (Identity Preemption Threshold Assignment) *We define the identity preemption threshold assignment, $\Gamma^I \in \mathcal{G}(\mathcal{T}, \Lambda)$, as the preemption threshold assignment where all tasks have been assigned preemption thresholds that are equal to the tasks' preemption levels (i.e. $\Gamma^I = \Lambda$).*

In this study we are more interested in optimizing existing real-time schedules than developing new ones. Hence, in this study we assume that for a given workload $\mathcal{T}$ there exists some preemption level mapping Λ such that the system is at least schedulable in a fully-preemptive manner (note that we need not differentiate if Λ is associated with fixed-priority or a dynamic-priority mapping). We also assume that this preemption level mapping Λ has been

Algorithm 1 : $\Gamma = MPTAA(\mathcal{T}, \Pi)$

```
for $i = N$ down to 1 do
  while ( $schedulable == TRUE$ and $\gamma_i < N$) do
    $\gamma_i = \gamma_i + 1$;
    /* Let $T_j$ be the task such that $\pi_j = \gamma_i$ */
    $R_j = WCRT(T_j)$;   /* check the schedulability of the affected task $T_j$ */
    if ($R_j > D_j$) then
      $schedulable = FALSE$;
      $\gamma_i = \gamma_i - 1$;
    end if
  end while
  $schedulable = TRUE$;
end for
return $\Gamma$;
```

pre-assigned according to some scheduling algorithm (e.g. RM, DM, EDF, etc), so the identity assignment Γ^I for the real-time system $(\mathcal{T}, \Lambda)$ is always known. Moreover, $\mathcal{G}(\mathcal{T}, \Lambda)$ is never empty since it will at least contain the identity assignment.

Given the identity assignment Γ^I, other assignments that optimize the system in some sense need to be found. Wang et al. [16] developed an algorithm that can always find a feasible preemption threshold assignment if it exists. We shall refer to Wang's algorithm as the *maximal preemption threshold assignment algorithm* (MPTAA) since it finds a special assignment that is in some sense the largest as will be explained below. This algorithm is shown in algorithm 1. As can be seen, this algorithm uses the WCRT of a task as a measure of schedulability. Clearly, this implies that it can only be used with fixed-priority systems. Indeed, the original PTS framework by Wang et al. [16] only address fixed-priority schemes.

To extend the MPTAA to support both scheduling schemes, a modified MPTAA was developed and shown in algorithm 2. As can be seen, the basic flow of the modified algorithm is the same. However, on line (7) of the extended version we use the function *is_task_schedulable()* to analyze the schedulability of the task. This function depends on the scheduling scheme used. If a fixed-priority scheme is used, this function uses WCRT analysis[3] to analyze the task's schedulability. On the other hand, if a dynamic-priority scheme is used, the function

[3]The WCRT analysis for fixed-priority schemes under PTS was presented by equations (3.4a), (3.4b), (3.4c), and (3.4d).

Algorithm 2 : $\Gamma = MPTAA(\mathcal{T}, \Lambda)$

```
1:  $\Gamma = \Gamma^I$   /* initialize preemption threshold to identity assignment */
2:  for $i = N$ down to 1 do
3:    $j = i + 1$;
4:    while ( $schedulable == TRUE$ and $\gamma_i < N$) do
5:      $\gamma_i = \gamma_i + 1$;
6:      /* check the schedulability of the affected task */
7:      $schedulable = is\_task\_schedulable(T_j)$;
8:      if ($schedulable == FALSE$) then
9:        $\gamma_i = \gamma_i - 1$;
10:     end if
11:     $j = j + 1$;
12:   end while
13:   $schedulable = TRUE$;
14: end for
15: return $\Gamma$;
```

uses the maximum blocking concept[4] to analyze the schedulability of the task.

The modified MPTAA algorithm has a computational complexity of $O\left(N^2 \cdot f(N)\right)$ where $f(N)$ depends on the schedulability test used on line (7). If the test uses the maximal blocking test for dynamic-priority schemes, then $f(N) = N$ and the overall computational complexity of the MPTAA is $O(N^3)$. On the other hand, if WCRT analysis is used, the complexity is not deterministic since a recursion is utilized. Hence, in this case we say that the computational complexity is non-deterministic and can only be addressed on a case by case basis. Nevertheless, empirical studies have shown that a recursion can usually be modeled by a computational complexity of $f(N) = N^r$ for some constant r [55]. Another important definition is now needed:

Definition 3.3.2. (PTS Mapping Size) *Given any two preemption threshold mappings Γ and Γ', we say that Γ is* larger *than Γ', and denote it by $\Gamma \succcurlyeq \Gamma'$, if and only if all preemption thresholds of $(\gamma_1, \gamma_2, \ldots, \gamma_N) \in \Gamma$ are equal to or greater than the corresponding preemption thresholds of $(\gamma'_1, \gamma'_2, \ldots, \gamma'_N) \in \Gamma'$. That is, if and only if $\gamma_i \geq \gamma'_i$ for all $i = [1, 2, \ldots, N]$.*

The largest of all preemption threshold assignments is of particular interest and is defined as follows:

[4]The maximum blocking concept in a dynamic-priority scheme was given by equation (3.6).

Definition 3.3.3. (Maximal Preemption Threshold Assignment) *We define the maximal preemption threshold assignment, denoted by* $\Gamma^{max} = (\gamma_1^{max}, \gamma_2^{max}, \dots, \gamma_N^{max}) \in \mathcal{G}(\mathcal{T}, \Pi)$, *as the largest preemption threshold assignment in* $\mathcal{G}(\mathcal{T}, \Pi)$. *That is,* $\Gamma^{max} \succcurlyeq \Gamma$ *for all* $\Gamma \in \mathcal{G}(\mathcal{T}, \Pi)$.

The original MPTAA was analyzed by Chen et al. [55], and was shown to always find the maximal preemption threshold assignment if one exists[5]. Since in our case we always start with a feasible assignment, namely the identity assignment Γ^I, the maximal preemption threshold assignment always exists (which in the worst-case would simply be the same as the identity assignment).

In the following, we present the main theorem for this section. To prove this theorem, we will use the tuple $(\mathcal{T}, \Lambda, \Gamma)$ to denote a particular real-time workload $\mathcal{T}$, a particular preemption level mapping Λ, and a particular *feasible* preemption threshold mapping Γ. Moreover, we will denote the total stack space required by the real-time system for for its tasks by $\mathcal{S}^{total}(\mathcal{T}, \Lambda, \Gamma)$. We again emphasize that the nature of the scheduling scheme (i.e. whether static or dynamic) is irrelevant to our proof and hence will not be mentioned explicitly.

Theorem 3.3.1. *Given two real-time systems* $(\mathcal{T}, \Pi, \Gamma)$ *and* $(\mathcal{T}, \Pi, \Gamma')$ *with preemption thresholds assignments* Γ *and* Γ' *in* $\mathcal{G}(\mathcal{T}, \Pi)$ *such that* $\Gamma \succcurlyeq \Gamma'$. *The total stack size* $\mathcal{S}^{total}(\mathcal{T}, \Lambda, \Gamma')$ *can be no smaller than* $\mathcal{S}^{total}(\mathcal{T}, \Lambda, \Gamma)$.

Proof. Without loss of generality, let us assume that $\gamma_k = \gamma'_k$ for all $k = 1, 2, \dots, i-1, i+1, \dots, N$ and let $\gamma'_i = \gamma_i - 1 < \gamma_i$ for some arbitrary $i \in \{1, 2, \dots, N\}$. It should be clear that $\Gamma \succcurlyeq \Gamma'$ according to our definition. Now let $T_j \in \mathcal{T}$ be the task belonging to the system with a priority λ_j chosen such that $\lambda_j = \gamma'_i + 1 = \gamma_i$. The existence of T_j is guaranteed by the way we assign preemption thresholds. Now since $\lambda_j = \gamma_i$, then T_j is not allowed to preempt T_i with the larger preemption threshold assignment Γ according to the rules of preemption threshold scheduling. On the other hand, T_j is allowed to preempt T_i with the smaller preemption threshold assignment Γ' since $\gamma'_i < \gamma_i = \lambda_j$). Hence, we have to allocate an additional S_j stack units to accommodate the potential preemption between T_j and T_i if we use the smaller preemption threshold assignment Γ', and therefore:

[5]This was only addressed for fixed-priority schemes, but holds equally for dynamic-priority schemes as well.

$$\mathcal{S}^{total}(\mathcal{T},\Lambda,\Gamma') = \begin{cases} \mathcal{S}^{total}(\mathcal{T},\Lambda,\Gamma) + S_j & \text{if } T_j \text{ preempts } T_i \\ \mathcal{S}^{total}(\mathcal{T},\Lambda,\Gamma) & \text{otherwise} \end{cases}$$

Hence, $\mathcal{S}^{total}(\mathcal{T},\Lambda,\Gamma) \leq \mathcal{S}^{total}(\mathcal{T},\Lambda,\Gamma')$ which completes the proof. □

The above theorem and the definition of the maximal preemption threshold assignment leads us directly to the following important corollary.

Corollary 3.3.2. *The MPTAA finds the preemption threshold assignment with the smallest possible total stack space requirements.*

Proof. Theorem 3 in [55] shows that the MPTAA finds the maximal preemption threshold assignment Γ^{max} with the essential property that $\Gamma^{max} \succcurlyeq \Gamma$ for all $\Gamma \in \mathcal{G}(\mathcal{T},\Pi)$. Combining this with theorem 3.3.1 proves this important corollary. □

Based on theorem 3.3.1 and corollary 3.3.2, it should be clear that the MPTAA provides us with a lower limit on the amount of stack space the system can utilize without violating any of its real-time constraints. Again, though many methods exist to obtain upper bounds on the stack usage of a real-time system, the above framework is the only method that provides the real-time system developer with the lower limits rather than the upper ones. Those limits can then be used with many goals like determining the minimum memory support need to run a particular real-time system without violating its constraints, or as will be shown in the next chapter, can be used to synthesize a data allocation methodology that improves memory utilization and enhance the system performance.

3.4 Extensions to PTS

Minimizing preemptions can have some adverse effects on the real-time system under consideration. That is, as preemptions are minimized, some of its benefits are lost. In this section we address some of these adverse side effects associated with preemption limiting and investigate how those adverse effects affect real-time PTS.

3.4.1 Robustness Properties of PTS

A real-time schedule is said to be *robust* if it remains correct even when some assumptions of the underlying workload model are not valid [13]. Clearly, a robust scheduling algorithm significantly reduces the need to extensive validations since it is guaranteed to work even in the presence of some uncertainty (which is unavoidable in any realistic system). According to Mok [56], for example, a scheduling policy is robust if the system schedulability is not affected by uncertainties in the tasks' execution times. For example, a task that executes for less than its WCET should remain schedulable. Though our intuition might tell us that this has to be the case, unfortunately it is not.

As an example, let us consider the set of real-time task given in table 3.1. If we schedule this system in a fixed-priority fully-non-preemptive manner, the execution timeline of this system if all tasks run for their WCET budget is shown in figure 3.1. As can be seen from that figure, all tasks make their deadlines and no problems occur. Now suppose that job J_{21} of task T_2 completes execution in one clock cycles less than its WCET. The resulting schedule is then given by figure 3.2 where task T_3 misses its deadline at $t_n = 10$. To see why this happens, note the following:

- At $t_n = 0$ all three tasks are release but T_3 gets the processor due to its higher priority.
- At $t_n = 3$ J_{31} of task T_3 completes execution. The second highest priority task T_2 get the processor.
- At $t_n = 4$ J_{21} of task T_2 completes execution, but because T_3 has not been released again for execution, T_1 gets the processor.
- At $t_n = 5$ T_3 is released for execution again, but becasue an FNP policy is used, it cannot preempt the lower priority task T_1 that has the processor.
- At $t_n = 8$ J_{11} of task T_1 finishes execution finally enabling T_3 to start execution.
- At $t_n = 10$ T_3 misses its deadline because by the time it got the processor from T_1, there was not enough time for it to complete by its deadline.

Table 3.1: Robusness exmple system

Task ID	Priority	Period	Deadline	WCET
T3	3	5	5	3
T2	2	10	10	2
T1	1	20	20	4

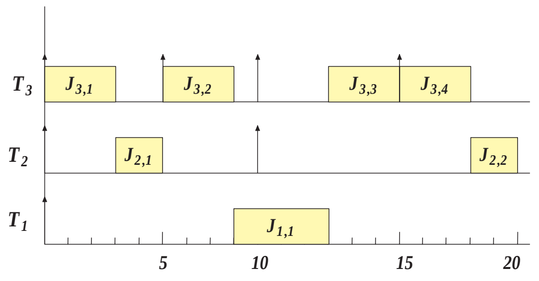


Figure 3.1: Schedulable system with all tasks running to their WCETs

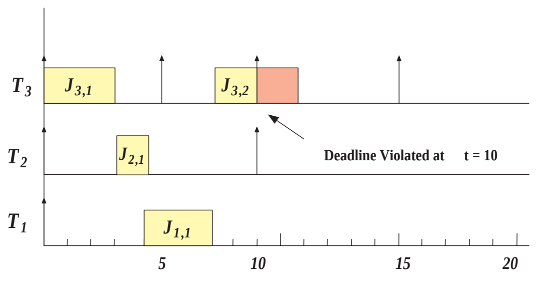


Figure 3.2: Non-schedulable system with some tasks not using their WCET budget

The above example shows the effect of non-preemptability on real-time schedulability. As was shown, non-preemptability of some tasks can lead to other tasks missing their deadline. Mok showed that the RM and DM scheduling policies are robust in the sense that if a system is schedulable with all tasks running to their WCET budgets, the system remain schedulable if any task runs for less than its WCET. Below we show that PTS is also robust in that same sense.

Theorem 3.4.1. *If a real-time system, $(\mathcal{T}, \Lambda, \Gamma)$, is schedulable with PTS for some feasible $\Gamma \in \mathcal{G}(\mathcal{T}, \Lambda)$ with all tasks executing to their WCET, it remains schedulable if any or all tasks execute for less than their WCET.*

Proof. We assume that we are given a system $(\mathcal{T}, \Lambda, \Gamma)$ that is schedulable (i.e. Γ is in $\mathcal{G}(\mathcal{T}, \Lambda)$). Now consider an arbitrary task T_i and suppose it executes for $c_i \leq C_i$ time units. We show that this will not affect the schedulability of (1) any higher priority task, (2) any lower priority task.

1. Let T_h be any task in $\mathcal{HL}(T_i)$. We show that task T_h remains schedulable if T_i executes for $c_i \leq C_i$ time units. To this end, from equation 3.3 it should be clear that the blocking experienced by the higher priority task (i.e. B_h^{total}) will never increase if any lower priority task executes for less than its WCET budgets. We hence have two cases:

   (a) The length of the busy period of equation (3.4d), and in turn the WCRT of equation

(3.4a), will never increase and therefore T_h remains schedulable.

(b) Similarly, the blocking experienced by task T_h due to task T_i with a lower priority (i.e. B_h^{total}) in a dynamic-priority scheme will never increase if T_i consumes less than its WCET budget. Hence, B_h^{total} will remain less than or equal to the maximum blocking T_h can tolerate and the its schedulability will not be affected.

2. Let T_l be any task in $\mathcal{LL}(T_i)$ in either a static- or dynamic-priority scheme. We show that task T_l remains schedulable if T_i executes for $c_i \leq C_i$ time units.

   (a) It should be clear that the interference term in equation (3.4d) for task T_l will always be smaller if T_i finished earlier than its WCET. Hence, the schedulability of T_l is not altered.

   (b) On the other hand, let $T_i \in \mathcal{HL}(T_l)$ be any task with higher preemption level than that of of T_l. If T_i executes for $c_i \leq C_i$ than the following holds:

$$1 - \frac{c_i}{\min(D_i, P_i)} \geq 1 - \frac{C_i}{\min(D_i, P_i)}$$

   which implies that B_l^{max} will only increase if any task $T_i \in \mathcal{HL}(T_l)$ executes for $c_i \leq C_i$. On the other hand, the blocking experienced by T_l will not be affected by T_l executing for less than its WCET budget and hence will remain smaller than B_l^{max} implying that T_l remains schedulable.

This completes our proof. □

3.4.2 Improving System Responsiveness

A main advantage of using a fully-preemptive scheduling policy is to improve the system responsiveness to internal or external stimuli. However, limiting preemptions in general usually deteriorates system responsiveness, and PTS is no exception. Hence, if a real-time system developer is to choose between two PTS schedules, the one that leads to better system responsiveness is obviously preferred. In this section we show that it is possible to find an optimal

PT Assignment	Stack	AWCRT
$\Gamma^I = (1,2,3,4,5)$	160	26.6
$\Gamma = (1,2,3,5,5)$	150	28.4
$\Gamma = (1,2,4,5,5)$	150	29.6
$\Gamma = (1,2,5,5,5)$	140	29.6
$\Gamma = (1,3,5,5,5)$	110	31.2
$\Gamma = (1,4,5,5,5)$	110	31.6
$\Gamma = (1,5,5,5,5)$	110	31.6
$\Gamma = (2,5,5,5,5)$	110	38.2
$\Gamma^{opt} = (3,5,5,5,5)$	90	41.4
$\Gamma = (4,5,5,5,5)$	90	43.4
$\Gamma^{max} = (5,5,5,5,5)$	90	45.2

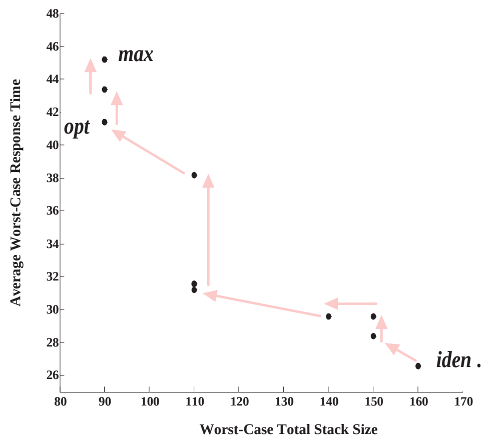


Figure 3.3: Solution path of the MPTAA

preemption mapping, which we shall denote by Γ^{opt}, that minimizes the total stack requirements as well as improves the system responsiveness.

Before we continue we need to define some way to be able to measure the system responsiveness. A well suited metric for measuring system responsiveness in a fixed-priority scheme is the *average worst case response time* (AWCRT) computed by averaging the WCRT over all tasks in a system. We will denote the AWCRT for the system $(\mathcal{T}, \Lambda, \Gamma)$ by $AWCRT(\mathcal{T}, \Lambda, \Gamma)$[6]. The system's AWCRT is a good measure of its responsiveness, as well as its robustness. The larger the AWCRT, the less slack time available for processing or power saving, and therefore, a smaller value is very desirable.

We show in this section that there might exist a preemption threshold assignment different from Γ^{max} that can result in the *same* optimal stack space requirements while improving the system's responsiveness as measured by the AWCRT. That is, we can find a feasible preemption threshold assignment, that we shall denote by Γ^{opt}, such that the following two properties hold.

$$\mathcal{S}(\mathcal{T}, \Pi^{stat}, \Gamma^{opt}) = \mathcal{S}(\mathcal{T}, \Pi^{stat}, \Gamma^{max}) \tag{3.7a}$$

[6]For simplicity, we focus in this section on fixed-priority systems. However, it should be emphasized that this treatment is equally applicable to dynamic-priority systems by defining some other metric to measure the system responsiveness.

$$AWCRT(\mathcal{T}, \Pi^{stat}, \Gamma^{opt}) \leq \mathcal{W}(\mathcal{T}, \Pi^{stat}, \Gamma^{max}) \tag{3.7b}$$

As an example, a randomly generated system of 5 tasks was used to explore the search path of the MPTAA. In this example we follow the MPTAA algorithm until it reaches the maximal assignment Γ^{max}. The search path for the MPTAA is shown in figure 3.3 with all the assignments visited starting from the identity assignment. As can be seen, the MPTAA finds the maximal threshold assignment that minimizes the system stack requirements from 160 to 90 units while maintaining schedulability. However, it is interesting to note the grouping of the data points in Figure 3.3; there are multiple vertically-aligned clusters of points with the same worst-case total stack size but differing AWCRT. The MPTAA reaches each cluster's lowest point (i.e. best AWCRT) first, but then proceeds upward, yielding worse AWCRT without improving the total stack size. Although the MPTAA minimizes the system's stack usage at 90 units, there are *two* other preemption threshold assignments that result in the *same* memory-optimal total stack requirement but with *better* system responsiveness. The best is Γ^{opt}, which has an AWCRT of 41.4 time units, versus 45.2 for Γ^{max}.

We also need to emphasize that the list of preemption threshold assignments given in figure 3.3 can also be used by system developers who prefer to have higher system responsiveness on the expense of some additional stack space to be allocated. To this end, the designer simply chooses from the list the desired level of responsiveness, as well as the number of preemptions they desire and the preemption threshold assignment corresponding to these parameters is readily available from the assignments list.

To compute Γ^{opt}, we simply traverse the problem graph backward starting with Γ^{max} and exit as soon as the total stack size starts increasing as shown in Figure 3.3. That is, after each iteration of algorithm 2, the preemption threshold assignment Γ is saved in a list. After Γ^{max} has been found, this list is traversed backwards to determine if the same optimal stack size can be obtained with better system responsiveness.

3.5 Case Studies and Simulations

Section 3.3 showed that PTS will always render the smallest stack size that will maintain the schedulability of the workload. We now evaluate the impact of PTS on stack space and response time in two ways. First, we use PTS to schedule a real workload developed for controlling an Unmanned Aviation Vehicle (UAV). Second, we use randomly-generated workloads to examine broad trends across a range of design points.

Paparazzi Benchmark

The ”Paparazzi” project of Brisset and Drouin [57] targets a cheap fixed-wing autonomous UAV executing a predefined mission. Nemer et al. [58] used the Paparazzi project to develop a real-time benchmark called ”PapaBench”. PapaBench is composed of two workloads with tasks for controlling the servo system, and handling navigation and stabilization. In this section we used PTS to schedule and optimize the the servo controller tasks from PapaBench listed in Table 5.2. The task set requires 120 bytes for global data and 108 bytes for stack data (detailed in the table). Hence a fully-preemptive scheduling approach would require a full 108 bytes to support all task stacks. Using PTS to limit preemptions reduces this total stack space requirement from 108 bytes to 34 bytes while maintaining schedulability. This holds for all utilization levels examined (37% to 97%). The total RAM required is reduced by 37%, a significant amount. Indeed, PTS with the MPTAA provides us with a simple systematic method that can be applied directly to any system independent of the priority assignment policy used. This method provides the real-time system designer with a tool to investigate the minimal memory requirements that maintain the schedulability of system, without which PTS would be an error-prone process.

Generic Real-Time Workloads

We next investigate workload characteristics that affect the stack size optimality level achievable through PTS. For example, the optimal stack utilization for some workloads through the use of PTS can be as small as 20% of the stack space utilization required by the fully-preemptive version of the system while others need 80%. In this section we simulate and analyze randomly

Table 3.2: The Fly-By-Wire Benchmark Task set

ID	**Name**	**Frequency**	**WCEC**	**Stack**
T1	receive_radio_task	40Hz	14,820	34B
T2	check_failsafe_task	20Hz	12,477	6B
T3	check_autopilot_values_task	20Hz	5,680	26B
T4	send_data_to_autopilot_task	40Hz	5,640	26B
T5	servo_transmit_task	20Hz	2394	10B
I1	servo_interrupt	_	80	2B
I2	spi_interrupt	_	193	2B
I3	radio_interrupt	_	76	2B

generated systems of tasks to better understand PTS.

To cover a wide range of design points, 40,000 systems with 10 tasks each were randomly generated. These were created so 1000 have a utilization of 50%, 1000 have 51% utilization, and so on up to 90%. Task periods have a normal distribution with a mean, $\bar{P}$, of 100 time units and a standard deviation, σ_P, of 25%, 50%, and 75%, respectively. Moreover, task deadlines were set equal to their respective periods (for simplicity, though not necessary). Tasks WCETs were set to incur the required overall system utilization. Task maximum stack space utilizations were chosen from a uniform distribution between 20 and 120 units. All 40,000 systems generated were schedulable with a fully-preemptive policy.

We first investigate the effect of the system utilization on the optimal stack utilization required with PTS. Using the MPTAA, the optimal stack space required by each system was computed and normalized to the stack space required by the fully-preemptive version of the system. The average normalized stack utilizations were then plotted as a function of the overall system utilization and the standard deviation in the task periods. The results are shown in figures 3.4 and 3.5 for the fixed-priority and the dynamic-priority schemes, respectively.

First, consider the fixed-priority scheme in figure 3.4. At low utilizations the system's stack space requirements might be less than 30% of those of a fully-preemptive system. However, at higher utilization levels, the variation in the tasks' periods increasingly affects the savings attainable. With $\sigma_P = 25\%$ the savings are only slightly dependent on the utilization level. On the other hand, as the standard deviation of the periods increases, the savings attainable

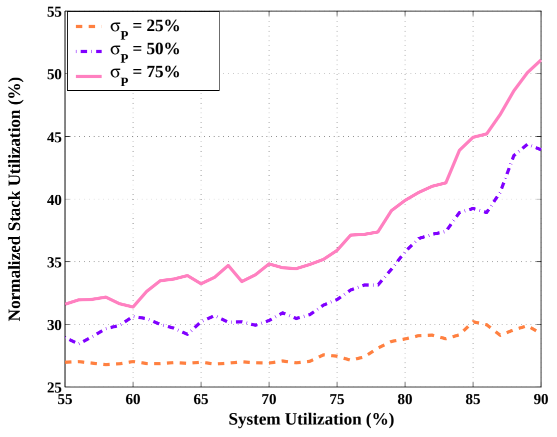


Figure 3.4: Fixed-priority PTS stack requirements.

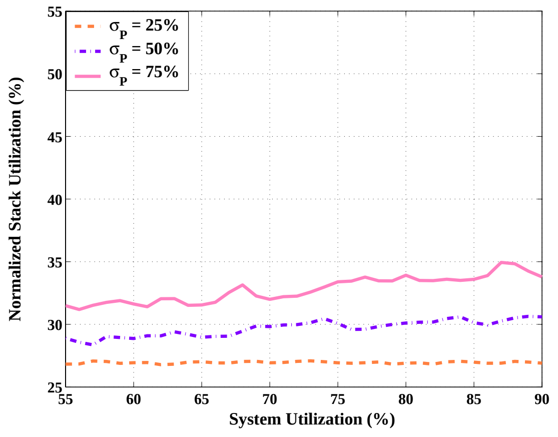


Figure 3.5: Dynamic-priority PTS stack requirements.

decrease significantly at higher utilizations. This can be attributed to the fact that for systems with large variations in their periods it is much harder to maintain the system's schedulability while minimizing preemptions. For example, if a system contains two tasks that have a large difference in their frequencies, it is difficult, if not impossible, to limit the higher frequency task from preempting the slower one while maintaining system schedulability. This obviously becomes much more apparent at high system utilizations where there is much less slack time.

Second, consider the dynamic-priority scheme in figure 3.5. The normalized stack space utilizations in this case are much more uniform across all utilization levels. This is because the dynamic (EDF) scheme is much more *adaptive* to the workload characteristics and has a higher *schedulable utilization*[7] than a fixed-priority scheme such as RM. This more-efficient scheduling allows more preemption limiting to occur before schedulability is lost.

Another interesting property is the distribution of the 40,000 systems among the different normalized stack space utilizations levels. To this end, a histogram was constructed showing the percentage of systems versus the normalized (optimal) stack space utilization achievable by each system. Figure 3.6 and figure 3.7 show this distribution for the overall system utilization levels of 60%, 70%, 80%, and 90%, respectively. Again, as can be seen, the workloads scheduled

[7]The *schedulable utilization* of a scheduling algorithm is defined by Liu [13] as the utilization level that guarantees that any system with this utilization can feasibly be scheduled with this algorithm. It is known that EDF has a schedulable utilization of unity as compared to RM with schedulable utilization of $N(2^{1/N} - 1)$ for N tasks.

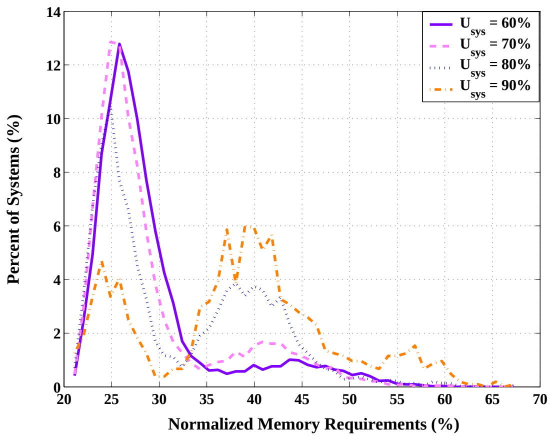


Figure 3.6: Dramatic reductions in stack space with fixed-priority PTS.

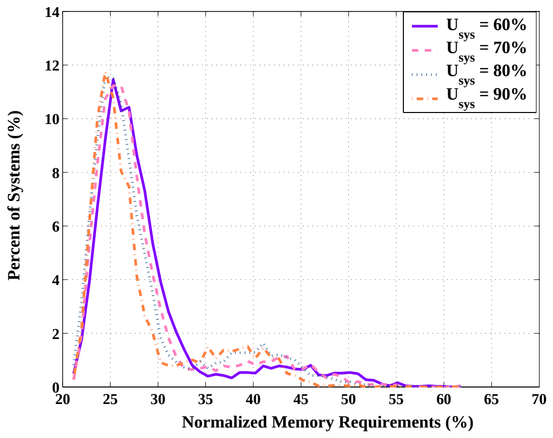


Figure 3.7: Dramatic reductions in stack space with dynamic-priority PTS.

with the dynamic-priority schemes depend less on the system utilization level than those in a fixed-priority scheme.

System Responsiveness with PTS

Limiting system preemptions has some undesirable side effects, including increasing a task's WCRT. The increase is not constant and depends on workload characteristics. In Section 3.4.2, we discussed reducing the effect of preemption limiting on the WCRT through backtracking.

To investigate this important issue, the workloads generated in the previous section were arranged in order of relative improvement in stack memory requirements. The AWCRT for the workloads was then computed and normalized to the optimal AWCRT of the fully-preemptive version of the system. For example, if the AWCRT of a system with PTS doubles as compared to its fully-preemptive AWCRT, then its normalized AWCRT is 200%. This data was then plotted as shown in figure 3.8 for the MPTAA with and without backtracking.

Figure 3.8 shows two relations. First, the normalized AWCRT is inversely related to the optimal normalized stack utilization; as the system's stack space requirements increase, its normalized AWCRT decreases. This is expected, since systems utilizing more stack space allows more preemptions, resulting in better system responsiveness. Second, backtracking results in greater improvement at higher stack utilizations. When a high level of preemptions is needed

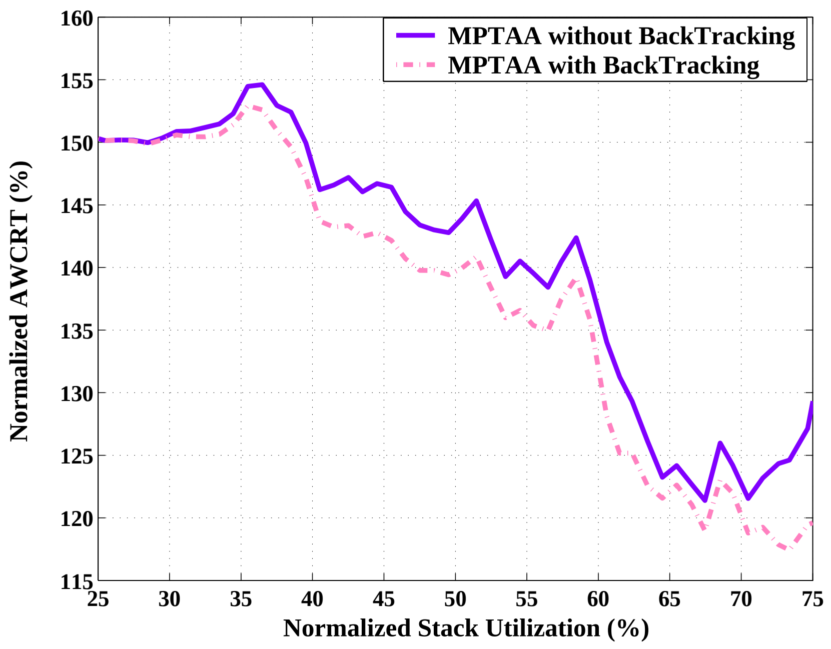


Figure 3.8: Responsiveness improvements of the MPTAA through backtracking.

for the system, the MPTAA over-constrains some of the tasks unnecessarily even though the optimal stack utilization has been reached and further limiting of preemptability will not succeed in minimizing the stack further.

Finally, we examine the improvement from backtracking the MPTAA. The histogram of Figure 3.5[8] shows that although most of the systems had between 0.5% to 1% improvement only, some systems had a dramatic AWCRT improvement – up to 50%. These generally modest system-level AWCRT improvements mask the task-level benefits. To see this, we examine a system of 20 tasks which saw only a 0.98% improvement in AWCRT on the system through backtracking (98% of systems see an improvement under 1%). The WCRTs of the individual tasks are plotted in figure 3.5. Remarkably, some task WCRTs improved by more than 50%. Hence, on the task level, backtracking the MPTAA can indeed be quite beneficial.

[8]To quantify the improvement of backtracking, the absolute difference between the AWCRT and the AWCRT with backtracking was normalized to the optimal AWCRT and used to measure the improvement rendered through backtracking

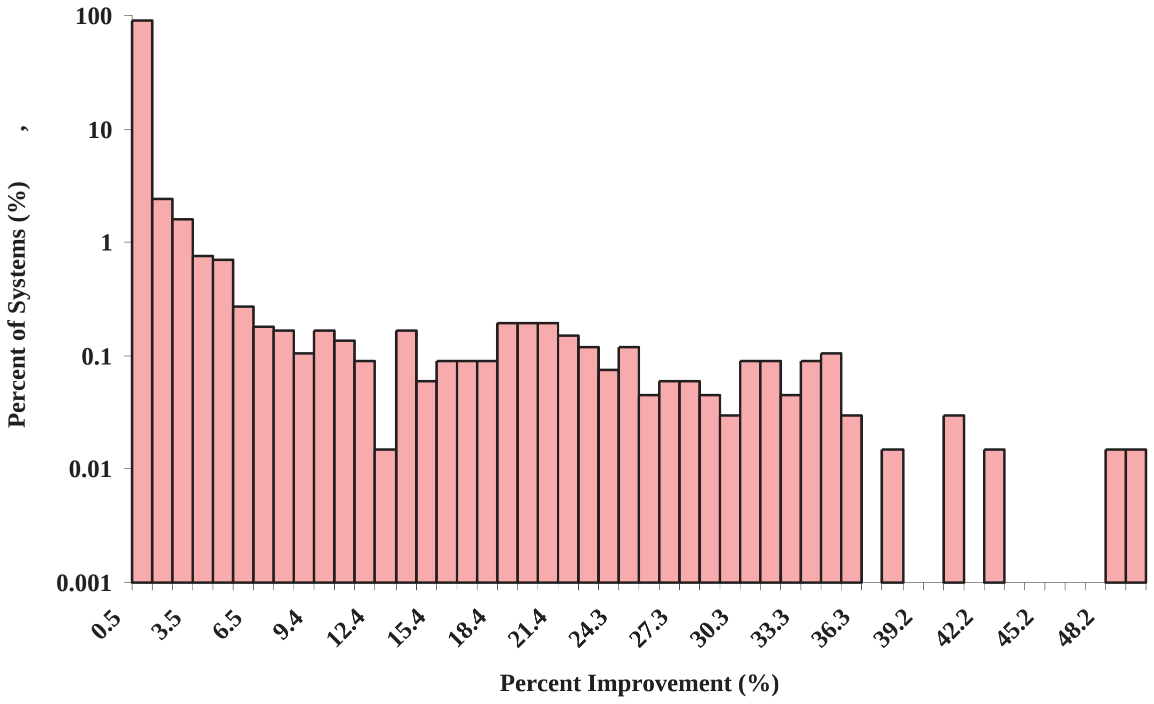


Figure 3.9: System-level distribution of AWCRT improvements from backtracking

3.6 Chapter Summary

In this chapter we developed a framework for using PTS with both dynamic and fixed priority systems. For dynamic-priority systems, the preemption level mapping was used to determine the preemption relations of the system *a priori* and enable us to statically analyze the dynamic-priority systems in the manner fixed-priority systems are analyzed. The developed framework is equally applicable with minor changes to both priority-driven schemes enabling providing the system developer with a fast ”what if” analysis of the system under consideration before committing to more involved simulations and analysis frameworks.

Using the unified PTS framework, we analytically proved that PTS along with the MPTAA is stack space optimal in the sense that no other preemption limiting technique can result in smaller memory requirements than PTS. We have also shown that PTS does not suffer from does not suffer from common undesired preemption limiting side effects like loss of robustness. To this end, it was shown that PTS maintains system schedulability in the event there

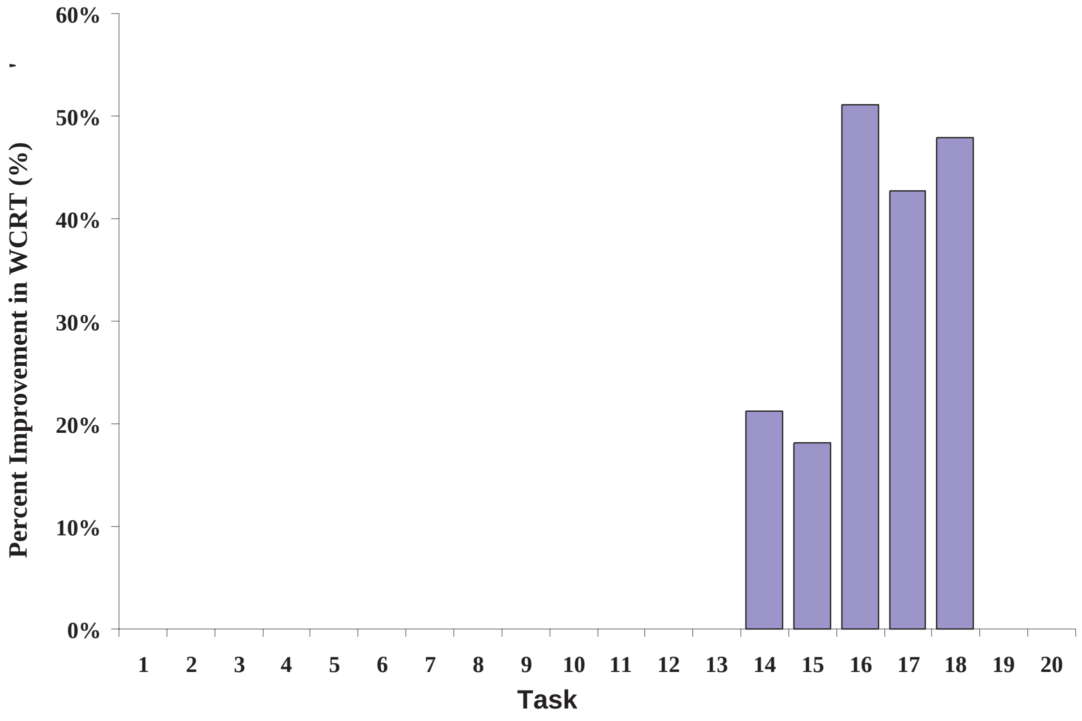


Figure 3.10: Task-level distribution of AWCRT improvements

are uncertainties in the workload's WCETs. Moreover, we showed that another undesired side effect of preemption limiting, namely deteriorated system responsiveness, can be improved for PTS through backtracking the MPTAA. Improvements of up to 50% for some tasks WCET were obtained using this backtracking scheme.

To aid system developers in deciding the most appropriate use of PTS in their own applications, we presented several simulations targeted at analyzing some of the properties of PTS. To this end, we showed that PTS can result in stack space savings of up to 78% for both fixed-priority and dynamic-priority systems. We also showed that in 40,000 systems simulated, not one system under PTS required more than 80% of the total stack space required with full preemption.

Chapter 4

Increasing Energy Efficiency in the Embedded Real-Time Domain

The proliferation of portable and battery-operated computing platforms continues to push research into techniques to save power and energy. One of the defining characteristics of these devices is their requirement to operate with limited energy budgets. Power and energy minimization techniques have been proposed on almost all levels of a system, including the switch level, the ISA, the operating system, and the compiler [27, 59, 25, 60].

One effective and extensively-studied method is DVS or dynamic voltage scaling, which has been demonstrated to be one of the most effective methods for minimizing the power dissipation at run time, and in turn, increase the energy efficiency of the system. This is because the power dissipated by CMOS-based integrated circuits is a quadratic function of the operating voltage as was shown in Section 2.2.1. Hence, scaling down the operational voltage linearly effectively scales down the power dissipation quadratically. Some processors have been equipped internally with dynamic voltage scaling support (e.g. Transmeta's Crusoe processor, Intel Pentium with SpeedStep [30]).

The bulk of the research available on DVS, and on power and energy management in general, addresses only high-end computing platforms (32-bit and 64-bit processors). Certain assumptions are made that might not be applicable to low-end, resource constrained systems (e.g.

Table 4.1: Sample of popular 8-bit microcontrollers

Microcontroller	**Program Memory**	**Data Memory**	**IOs**	**Speed**	**Timers**	**PWR-DWN Modes**
ATmega128	128KB FLASH	4KB SRAM	53	16MHz	5	6
C8051F120	128KB FLASH	8KB+ 256B RAM	80	100MHz	1	N / A
PIC18LF8720	128KB FLASH	3840B SRAM	68	25MHz	5	1
MC68HC705C8A	Up to 7744B PROM	Up to 304B RAM	31	4MHz	3	N / A
ATmega8	8KB FLASH	1KB SRAM	23	16MHz	3	5
PIC16LF877	14KB FLASH	368B SRAM	33	20MHz	2	1
MC68L11D3	4KB EPROM	192B RAM	26	2MHz	1	N / A
AT89S8253	12KB EPROM	256B RAM	32	24MHz	N / A	2
SX20AC	2KB FLASH	136B SRAM	12	75MHz	1	N / A
ATtiny26	2KB FLASH	128B SRAM	16	16MHz	2	4
PIC16LF84A	2KB FLASH	68B SRAM	13	20MHz	2	1

the cost of an adjustable power supply is negligible; an RTOS is present, etc). To the best of our knowledge, no studies have investigated power management for short-bit-width computing platforms (e.g. 8-bit processors). However, short-bit-width processors continue to dominate worldwide microcontroller sales volumes [32, 61, 62]. Many digital consumer applications use these devices because of their low price and the availability of extensive tools and documentation. Many applications have no need for more powerful processors. Examples of portable or battery operated applications range from automotive keyless entry systems, automotive theft alarms, universal remote controls, portable compact discs and digital audio players, to digital thermometers and blood pressure monitors. Moreover, market research firms generally predict a stable, if unexciting, future for short-bit-width systems. In 2006, it is expected that 8-bit units will continue to lead all microcontrollers in revenue and unit shipments [61].

This section investigates the applicability of advanced energy saving methods such as DVS and DFS in combination with built-in low-power modes for short-bit-width *commodity commercial off-the-shelf* (COTS) processors, where some of the assumptions made in previous studies might not hold. We also try to weigh the advantages of using those techniques for low-cost embedded systems based on COTS processors versus their relatively high cost and complexity of implementation. To conduct our simulations and analysis, we use eleven 8-bit microcontrollers from the most popular architectures [61]. Complete and general mathematical models for the power dissipation characteristics of those microcontrollers are developed and listed in this study.

4.1 Processors Energy Dissipation Models

A sample of low-end commercial microcontrollers was chosen to investigate the applicability of various power and energy management techniques like DFS and DVS to those resource constrained embedded platforms. In this section we develop empirical models capturing the energy and power dissipation characteristics of some popular 8-bit microcontrollers. The models developed and their statistical error bounds will be presented. In order to make this analysis tractable, we do not include I/O power. Digital I/O and peripherals (serial communication, analog interfacing, timers, etc.) that are used in an application-specific manner and need to be considered in a system-level analysis.

4.1.1 Microcontroller Sample

Eleven COTS microcontrollers were chosen to represent the most popular 8-bit architectures in the market [32, 61, 62]. This sample includes Intel's 8051 architecture, Microchip's PicMicro architecture, Motorola's 6805, 6811, and 6812 architectures, and Atmel's AVR architecture. Table 4.1 lists these microcontrollers along with some of their features.

4.1.2 Empirical Modeling of Processors Energy Consumption

As can be recalled from Section 2.2.1, three main relations completely characterize the energy dissipated in a CMOS-based circuit. These three relations are repeated here for convenience:

$$E_{dyn} = C_P \, f_{CLK} \, V_{CC}^2 \, \Delta t \tag{4.1a}$$

$$E_{stat} = I_{lkg} \, V_{CC} \, \Delta t \tag{4.1b}$$

$$\max[f_{CLK}] = \frac{K_p \, (V_{CC} - V_{th})^{1.8}}{V_{CC}} \tag{4.1c}$$

where the leakage current is given by the following [26]:

$$I_{lkg} = \mu C_{ox}(W/L)V_t^2 e^{\frac{V_{CC}-V_{th}}{n\,V_t}} \left(1 - e^{-\frac{V_{CC}}{V_t}}\right) \tag{4.1d}$$

where, f_{CLK} is the circuit's operational frequency, V_{CC} is the circuit's operational voltage, V_{th} is the circuit's threshold voltage, and I_{lkg} is the leakage current due to subthreshold currents. The problem with the above model is that it requires detailed knowledge of many low level parameters that are dependent on the fabrication technology as can be seen from the expression for the leakage current I_{lkg} given by equation (4.1d). Since at our level of abstraction this information is not available to us, and even if it was it will restrict our analysis to a chips fabricated using a particular technology, we will make a simplifying assumption.

As can be seen from (4.1d) there is dependence of the leakage current on the operational voltage V_{CC}. In combination with equation (4.1b), it should be clear that the static energy component has to depend quadratically on the operational voltage V_{CC}. To this end, we chose to model this relation using the following empirical equation (which is simply a second order approximation of the actual relation):

$$E_{stat} = S_P \, V_{CC}^2 \, \Delta t \tag{4.2}$$

Where S_P is a constant of proportionality that will only depend on the conductivity properties of the circuit with units of $\mho$ (or mhos).

To develop the need empirical models, we only need now to estimate the constants of proportionality C_P, S_P, and K_P such that the modeling error is minimized. The complete derivation of this model is presented in the appendix. The resulting empirical models developed and the nominal values of the estimated parameters are listed in Table 4.2.

4.1.3 Simulation Assumptions

In our simulations, we do not try to compare the various microcontrollers to find which is the most energy-efficient, as this depends on many other factors not considered here (e.g. ISA, compiler). We only try to compare the potential benefits of using the various power-saving methods for a given processor. To this end, we generate execution profiles with a specific utilization level and a specific granularity level (number of jobs, or releases (instances) of tasks) and compose our workloads from those execution profiles. Once the workloads are generated, they are passed to our simulator which uses the models developed earlier, and various other parameters (e.g. voltage transition rate for DVS, wake-up delay time for PDM) to calculate the normalized power and energy used with the various power management techniques. The minor ”humps and bumps” in the plots are noise resulting from the execution profiles’ discrete nature and limited length, and would be eliminated with longer profiles.

Table 4.2: Statistical power models of 8-bit microcontrollers

Model Parameters	**Nom.** C_P	**Min.** C_P	**Max.** C_P	**Nom.** S_P	**Min.** S_P	**Max.** S_P	**Nom.** K_P	**Min.** K_P	**Max.** K_P	**Nom. Power (mW)**
ATmega128	*0.3739*	*0.3596*	*0.3881*	*0.2322*	*0.0935*	*0.3708*	*5.1562*	*4.1239*	*6.1884*	$0.3739 f_{CLK} V_{CC}^2 + 0.2322 V_{CC}^2$
C8051F120	*0.1901*	*0.1846*	*0.1957*	*1.9783*	*1.6873*	*2.2692*	*51.613*	*27.119*	*76.106*	$0.1901 f_{CLK} V_{CC}^2 + 1.9783 V_{CC}^2$
PIC18LF8720	*0.1055*	*0.0988*	*0.1122*	*0.2221*	*0.1197*	*0.3245*	*8.2201*	*6.6372*	*9.803*	$0.1055 f_{CLK} V_{CC}^2 + 0.2221 V_{CC}^2$
MC68HC705C8A	*0.3212*	*0.279*	*0.3634*	*0.1054*	*0.0038*	*0.207*	*1.1804*	*1.067*	*1.2939*	$0.3212 f_{CLK} V_{CC}^2 + 0.1054 V_{CC}^2$
ATmega8	*0.2073*	*0.1908*	*0.2237*	*0.42*	*0.26*	*0.5801*	*5.2377*	*4.2031*	*6.2724*	$0.2073 f_{CLK} V_{CC}^2 + 0.42 V_{CC}^2$
PIC16LF877	*0.0445*	*0.0405*	*0.0484*	*0.1997*	*0.1506*	*0.2488*	*6.1159*	*5.3373*	*6.8946*	$0.0445 f_{CLK} V_{CC}^2 + 0.1997 V_{CC}^2$
MC68L11D3	*1.2866*	*1.1203*	*1.453*	*0.1401*	*0.0104*	*0.4032*	*0.5403*	*0.2026*	*0.878*	$1.2866 f_{CLK} V_{CC}^2 + 0.1401 V_{CC}^2$
AT89S8253	*0.0239*	*0.02*	*0.0279*	*0.498*	*0.461*	*0.522*	*6.436*	*5.1427*	*7.9665*	$0.0239 f_{CLK} V_{CC}^2 + 0.498 V_{CC}^2$
SX20AC	*0.2574*	*0.2416*	*0.2732*	*0.8702*	*0.2572*	*1.4833*	*23.034*	*16.942*	*29.125*	$0.2574 f_{CLK} V_{CC}^2 + 0.8702 V_{CC}^2$
ATtiny26	*0.1681*	*0.1588*	*0.1773*	*0.2093*	*0.1189*	*0.2997*	*5.3728*	*4.3587*	*6.3869*	$0.1681 f_{CLK} V_{CC}^2 + 0.20931 V_{CC}^2$
PIC16LF84A	*0.0386*	*0.0355*	*0.0418*	*0.0419*	*0.0044*	*0.0793*	*5.9998*	*5.4603*	*6.5393*	$0.0386 f_{CLK} V_{CC}^2 + 0.0419 V_{CC}^2$

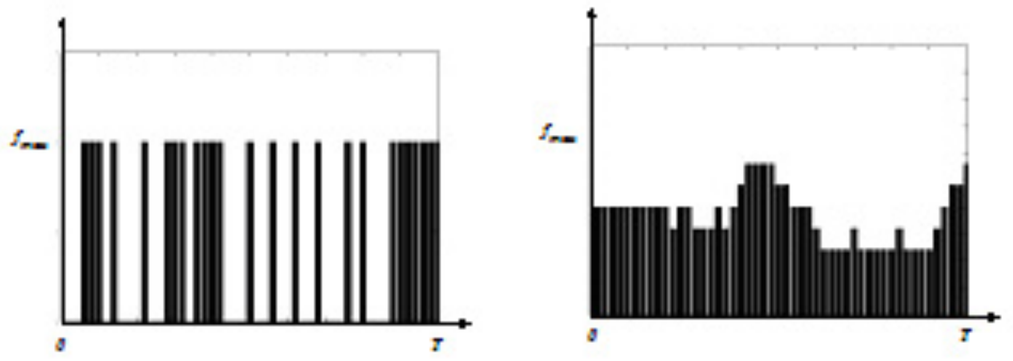


Figure 4.1: Execution profiles for scheduled versus non-scheduled workloads.

4.1.4 Benchmarks

Each workload is generated from an execution profile with a specific utilization and number of tasks. Since many applications might not support a power scheduler, we divide the workloads into scheduled and unscheduled as follows.

4.1.5 Non-Power-Scheduled Workloads

Each of the workloads used has N tasks that have a utilization of U over some fixed period of time t. Applications are released (invoked) randomly while maintaining the particular utilization. The execution profile of a sample workload with a 50% utilization (U = 50%) and 20 jobs (N = 20) is shown in figure 4.1.4. Note that in the absence of a real time scheduler, there is a substantial amount of idle time (where applications might be waiting on other applications, inputs, etc.) that is not utilized. Without applying any power management method, the processor will simply remain active during those idle slots consuming power while not performing any useful work.

4.1.6 Power-Scheduled Workloads

A scheduling process is crucial to any successful implementation of a power saving method. Moreover, scheduling for real-time systems is even more critical since these applications have real-time constraints that must be met for a correct and safe system operation. For this chapter we do not consider real-time scheduling. Instead, we simply implement a scheduler that uses the PAST prediction policy, and the Chan-style speed-setting policy. The scheduler calculates the utilization over some past window of time, U_{past}, and sets the current clock frequency to be $U_{past} \cdot f_{max}$. If the current utilization, $U_{current}$ turned out to be different from the past utilization, U_{past}, the difference, $U = |U_{current} - U_{past}|$ is added to the future utilization, $U_{future} = U_{current} + U$, and the clock is adjusted accordingly. A sample execution profile of the workload given in Section 4.1.5 after being scheduled is shown in figure 4.1.4.

4.1.7 Simulation Methodology

The simulation will depend on the presence or absence of a power scheduler. In the absence of such a scheduler, only PDM can be used by embedding some power-down function in the application code. In the simulation, we assume such function has been implemented in every application composing the particular workload and calculate the power dissipation based on that.

In the presence of a power scheduler, PDM, DFS, or DVS can be used. We assume that when DFS or DVS are used, they are used in combination with PDM when available (e.g. DVS-PDM, DFS-PDM).

4.2 Results and Observations

4.2.1 Workload-Independent Power Dissipation Characteristics

Certain power dissipation characteristics are independent of the workload being executed and depend only on the microcontroller or the power management technique under consideration. These characteristics can help us decide and understand why a particular power management method is more suitable for a particular microcontroller than the other. For example, consider a microcontroller with a relatively large static power dissipation component and a ”power-down” mode which in reality merely stops the clock rather than removing power. For this MCU, DVS is preferred as it is the only method that directly reduces the leakage component

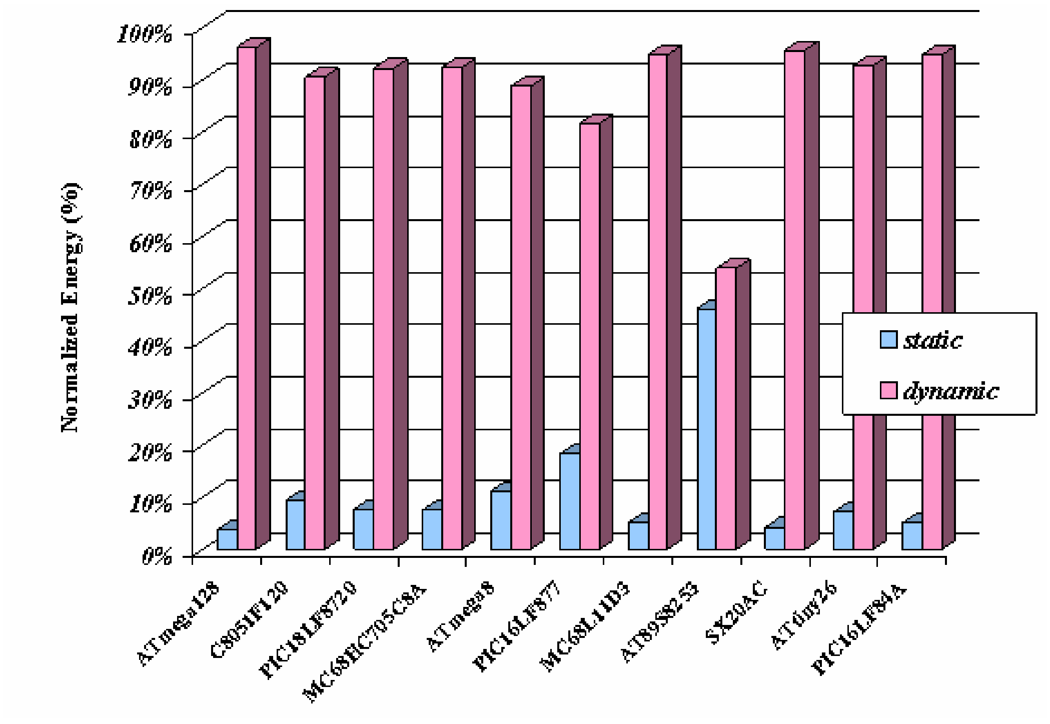


Figure 4.2: Normalized static vs. dynamic energy for popular microcontrollers

of power. As another example, a power management method with small transition times is expected to perform better with a large number of jobs (i.e. high granularity level).

4.2.2 Dynamic and Static Power Components

Figure 4.2.1 shows the normalized power components dissipated when the microcontrollers are running at their maximum operational voltage and frequency. Note how the AT89S8253 has a static component that is almost as large as its dynamic component. DVS should render the best savings for this microcontroller as it minimizes the large static component. The other devices should see less benefit from DVS.

4.2.3 Minimum Power Dissipation

Figure 3 shows the power dissipated when the particular microcontroller is always idle and a single power management technique (i.e. either PDM, DFS, or DVS) is used all the time. This provides a bound on the amount of energy that can be saved with the various methods.

For DFS, we assume that the frequency dividing circuit contains an 8-bit counter which divides the operating frequency by $2^8 = 256$, so the microcontroller runs at $f_{min} = f_{max}/256$. For DVS, the energy dissipation shown in figure 4.2.3 corresponds to using DVS to run the particular microcontroller at V_{min} (i.e. the minimal possible operational voltage for each microcontroller).

Microcontroller behavior falls into one of three categories: The first is microcontrollers with a usable built-in PDM, which always use the least power. This is lower by a factor of at least 1000 when compared to the energy dissipated using the two other power-saving methods. In the second category, DFS leads to the lowest power dissipation (when PDM is not available) and the second lowest power dissipation (when PDM is available). In the third category, DVS leads to lower power dissipation than DFS. This category includes only two microcontrollers (AT89S8253 and PIC16LF877). These two microcontrollers are those with the highest static energy / dynamic power ratio (which can be seen from Figure 4.2.1) and consequently, DVS leads to better results than DFS because it minimizes this large static power component while DFS does not.

4.2.4 Switching Power Supply

As discussed earlier, the voltage transition rate is a power supply design parameter that depends on various issues. In general, the voltage transition rate will be proportional to the power converter's design and implementation costs. This tradeoff must be evaluated by the system designer to decide whether a power converter with the required voltage transition rate will be worth its cost.

The transition time for DVS and DFS can be considered microcontroller independent since it will only depend on the converter's transition rate and frequency divider circuit for DVS and DFS respectively. On the other hand, the transition time for PDM depends on the wake-up delay of the particular microcontroller (the PDM transition delay in the figure has been simulated using the ATmega128 model as a representative microcontroller).

Note how the accumulating transition time for DVS has a higher rate of increase (this is

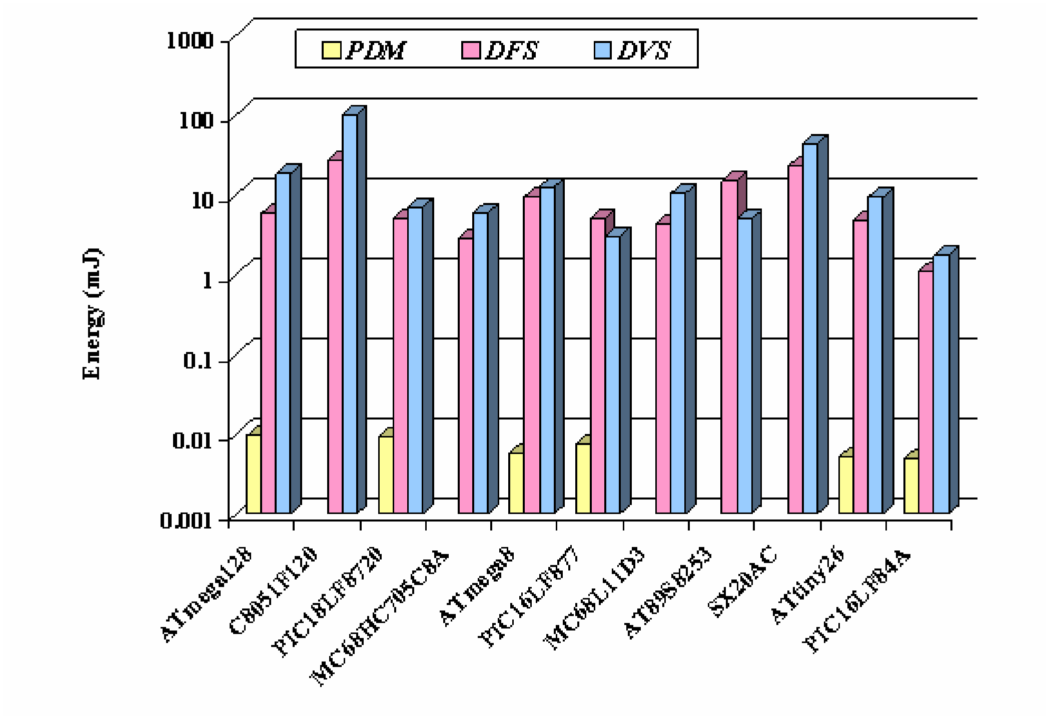


Figure 4.3: Physical limits on energy savings

a log-log plot) than both PDM and DFS. This is a drawback of DVS since the transition time ($|V_1 - V_2|/dV_{CC}/dt$) will almost always be much larger than the transition time for DFS (which will have a worst case of 2^n cycles when using an *n*-bit counter for frequency division), or the transition time for PDM (which usually will fall between a several cycles to a few thousand cycles). Hence, DVS is usually much more sensitive to the granularity level than DFS or PDM, which makes it less suitable for high-granularity workloads.

4.2.5 Voltage Transition Rates

The voltage transition rate will affect the sensitivity of DVS to the granularity level of the workload. Figure 4.2.5 shows the normalized energy dissipated by a microcontroller using DVS with three converters, with voltage transition rates of 0.01, 0.05 and 0.1 mV/ s. The smaller the transition rate is, the more sensitive the microcontroller to the number of transitions, and in turn, the number of jobs. Beyond a certain point, transition energy increases rapidly. Note that in the remainder of this study we use the transition rate of 1.95 mV/ s.

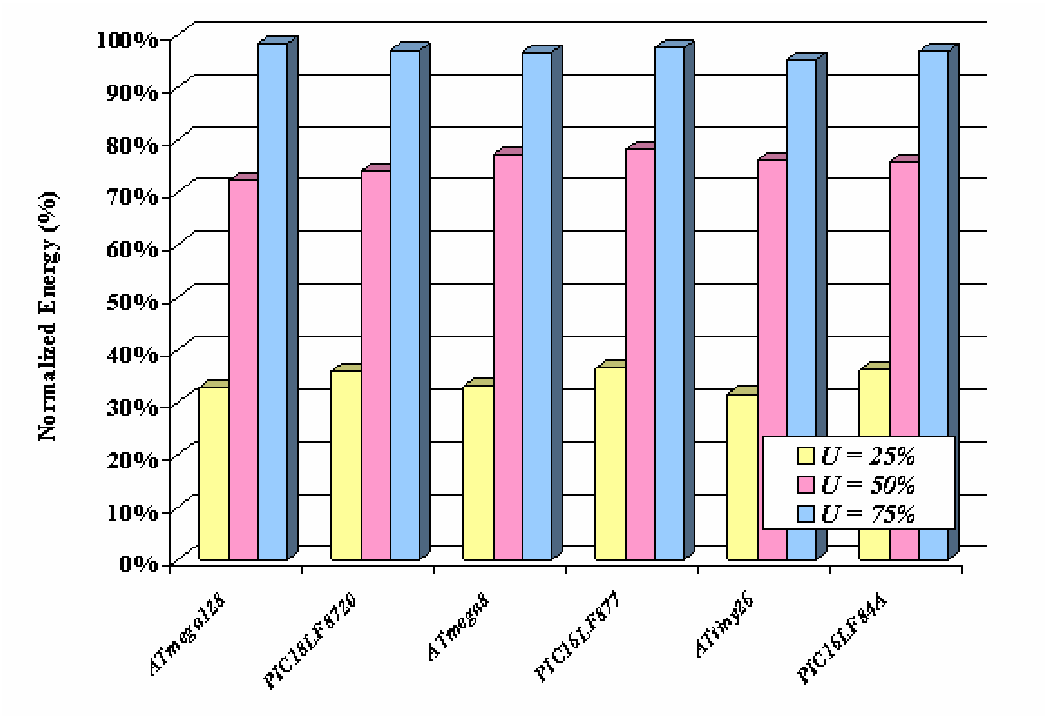


Figure 4.4: Energy usage without a power scheduler

4.2.6 Energy Use without a Power Scheduler

In the absence of a power scheduler, the system designer has no choice but the use of a built-in power down mode (if supported by the particular microcontroller). In this case, the programmer inserts power-down code that activates the microcontroller's low power state until some waking event occurs (e.g. timer overflow interrupt, external event). Figure 4.2.5 shows the energy dissipated per microcontroller as a function of the utilization (for the microcontrollers that support PDM mode), normalized to no power management. As the utilization increases, the energy used approaches that of a processor without any power management technique. This should be expected since power down modes can only save power while the processor is idle. As the utilization increases, the idle time decreases, and so does PDM's opportunity of reducing the system's energy consumption.

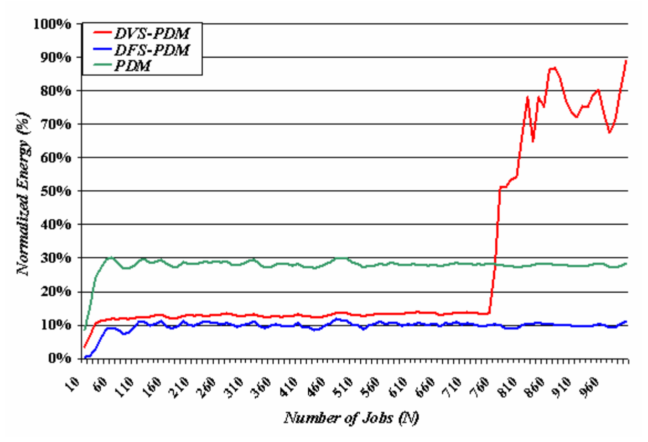
4.2.7 Energy Use with a Power Scheduler

Adding a power scheduler enables power and energy savings while the processor is active, opening the door to the advantages of DVS. When analyzing the power dissipation with a power scheduler, the microcontrollers fall in one of three categories. The first and largest category includes nine microcontrollers: the ATmega128, the PIC18LF8720, the MC68HC705C8A, the ATmega8, the PIC16LF877, the MC68L11D3, the SX20AC, the ATtiny26, and the PIC16LF84A. These microcontrollers have similar characteristics. The normalized energy used by the ATmega128 (a representative of this category) is given in Figures 4.5(a), 4.5(b) and 4.5(c), for utilization levels of 25%, 50%, and 75% respectively.

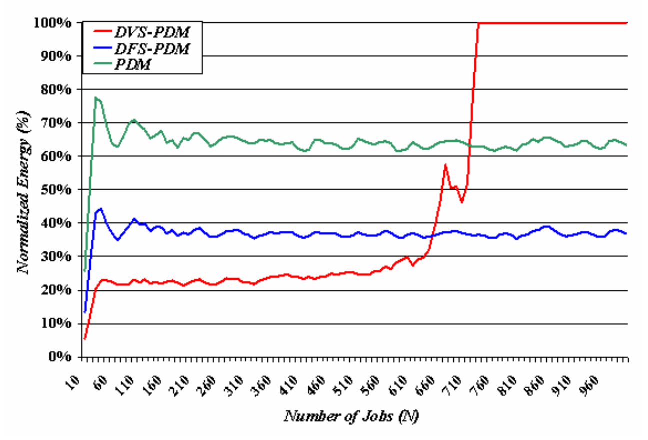
At low utilization (25%) and low granularity, DFS-PDM uses less power than DVS-PDM. DVS-PDM is much more sensitive to the number of jobs (granularity) than DFS-PDM or PDM alone. As granularity increases, the performance of DFS-PDM and DVS-PDM become similar. Beyond a certain number of jobs (the critical granularity level), the accumulated transition time is so large that the processor cannot reduce the voltage anymore and must run at full speed and power in order to maintain its throughput. As utilization increases to 50% and 75%, DVS-PDM becomes more efficient than DFS-PDM below the critical granularity level.

Two microcontrollers show different behavior. The energy used by the C8051F120 at utilization levels of 25%, 50%, and 75%, is shown in Figures 4.6(a), 4.6(b) and 4.6(c), respectively. Note that at low utilization, regardless of utilization DFS saves much more energy than DVS (90% by DFS as opposed to 70% by DVS). For medium utilization, both DFS and DVS provide energy saving of about 60% - 65%. Only for high utilization does DVS out-perform DFS with energy savings of about 60% for DVS and 50% for DFS. Even then, the difference is relatively small. The exceptional behavior of the C8051F120 is due to two characteristics. First, the narrow operational voltage range (2.7 to 3.6V) limits the potential benefits of DVS. Second, the minimum frequency for DVS is limited by this narrow voltage range, when compared with the potential clock division by 256 in DFS.

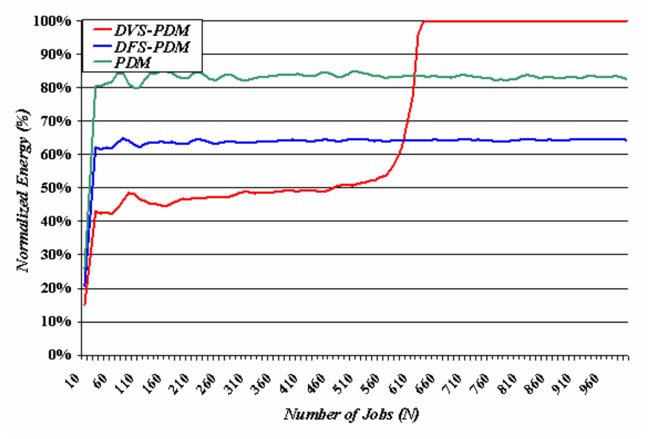
The AT89S8253 provides an exception on the other end of the spectrum. The normalized energy used by the microcontroller for utilization levels of 25%, 50%, and 75%, are shown in



(a) Normalized energy use for the ATmega128 with $U = 25\%$

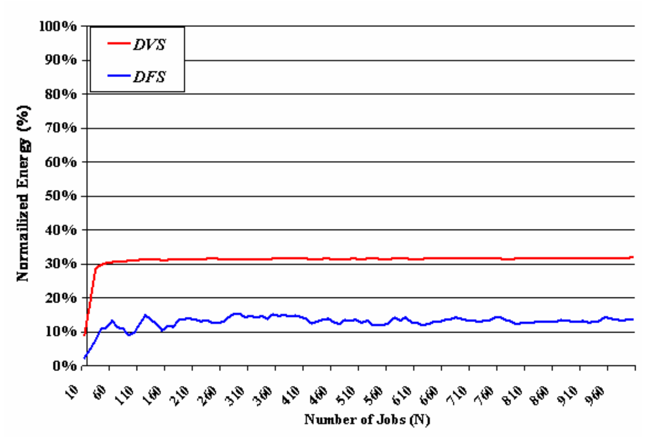


(b) Normalized energy use for the ATmega128 with $U = 50\%$

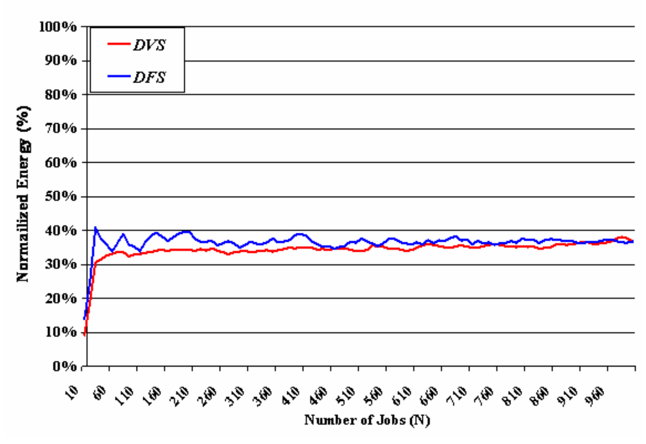


(c) Normalized energy use for the ATmega128 with $U= 75\%$

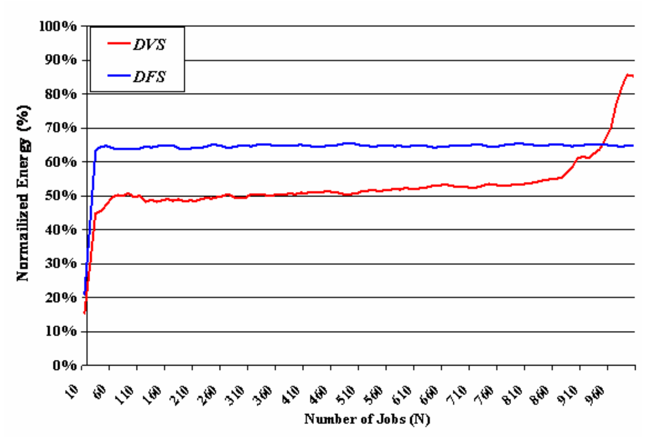
Figure 4.5: Normalized energy use for Atmega128



(a) Normalized energy use for the C8051F120 with *U = 25*%



(b) Normalized energy use for the C8051F120 with *U = 50*%



(c) Normalized energy use for the C8051F120 with *U = 75*%

Figure 4.6: Normalized energy use for C8051F120

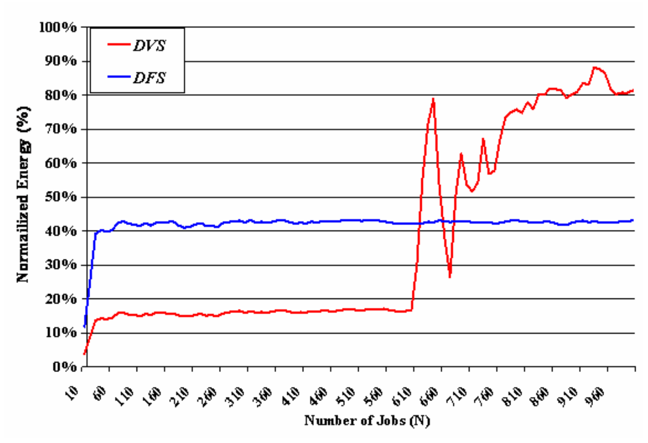
Figures 4.7(a), 4.7(b) and 4.7(c), respectively. Note how DVS outperforms DFS for all utilization levels. In fact, DVS provides from 20% to 30% more savings than DFS for all utilizations. This microcontroller has a large static power component, as explained earlier, and only DVS minimizes this leakage component. Hence, for this microcontroller DVS is a better energy management method.

4.3 Chapter Summary

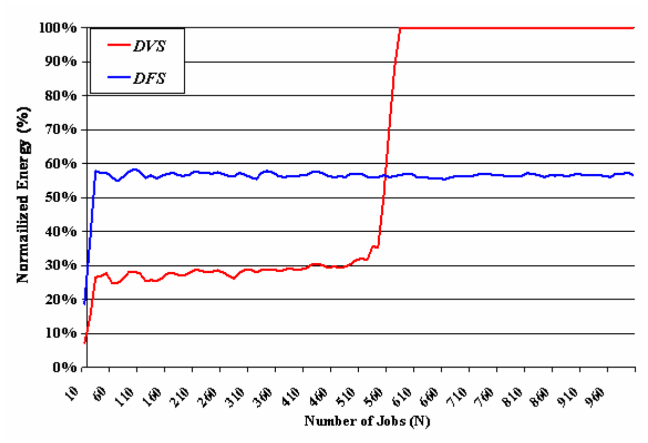
We find that DVS with PDM generally offers improvements over DFS with PDM for most short-bit-width microcontrollers with workloads of significant utilizations (50% and above) and moderate granularity. Systems with a large static power component benefit disproportionately from DVS-PDM. However, for workloads with low utilizations or high granularity, the overhead of voltage scaling outweighs the energy savings, making DFS-PDM an attractive, cost-effective alternative. Though often significant, the energy enhancements provided by DVS-PDM may not justify the additional cost of a switching power supply. The designer must carefully weigh this issue. However, some microcontrollers do not fit this rule, so the designer should measure the device's power characteristics to determine the best energy-saving approach.

DVS benefits from a wide voltage range to leverage its quadratic energy savings, but for all MCUs studied this range was relatively small (at most 2). This limitation also affects 32-bit microprocessors such as the IBM PowerPC405LP, TransMeta Crusoe TM5800 and Intel XScale 80200 [32]. DVS also requires a fast variable power supply (i.e. large dVcc/dt), but this may increase the circuit cost beyond the cost constraints, rendering it infeasible. Without fast transitions, the workload's job granularity becomes a bottleneck to saving energy. DFS can scale down the clock frequency by a factor of 256 with a simple 8-bit counter leading to at most a 256x power reduction with a much less expensive circuit.

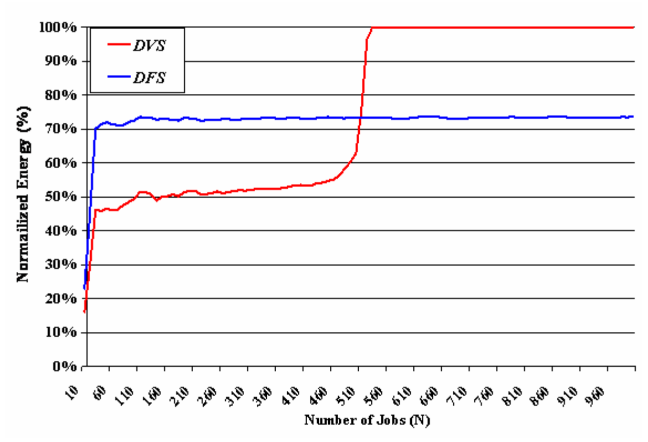
Finally, for most MCUs, the dynamic energy component is still much higher than the static energy component, making DFS still a more attractive solution especially in our cost-constrained design space. This may change as MCUs migrate to newer fabrication processes.



(a) Normalized energy use for the AT89S8253 with *U* = *25*%



(b) Normalized energy use for the AT89S8253 with *U* = *50*%



(c) Normalized energy use for the AT89S8253 with *U* = *75*%

Figure 4.7: Normalized energy use for AT89S8253

Chapter 5

Data Allocation with Real-Time Scheduling (DARTS)

As was explained in the introductory chapter, the efficient utilization of memory plays a very important role in the embedded design process, and often significantly impacts the embedded system's performance, energy dissipation, and overall cost of implementation. In order to alleviate the processor-memory gap, many embedded systems exploit a memory hierarchy of two or more on-chip and off-chip memory units. The advances in fabrication technology have made it possible to combine multiple heterogeneous memory units on the same chip. For example, it is now possible to combine a DRAM module with ordinary static logic on one chip [59]. It is therefore not uncommon to find a memory hierarchy composed of multiple heterogeneous[1] memory units with different sizes, access latencies, as well as bit-widths. The memory units found today in many embedded system can include on-chip SRAM, off-chip SRAM, on-chip DRAM, off-chip DRAM, and even EPROM or EEPROM that is writable by software.

For most general purpose computing platforms, the on-chip memory typically been implemented as a fast hardware-controlled cache that interfaces with one or more slower off-chip memory units (typically DRAM). This is particularly true for mid- to high-end general purpose

[1]We will say two memory units are *heterogeneous* if they use different fabrication technology, and / or they have different sizes, access latencies, and / or bit-widths.

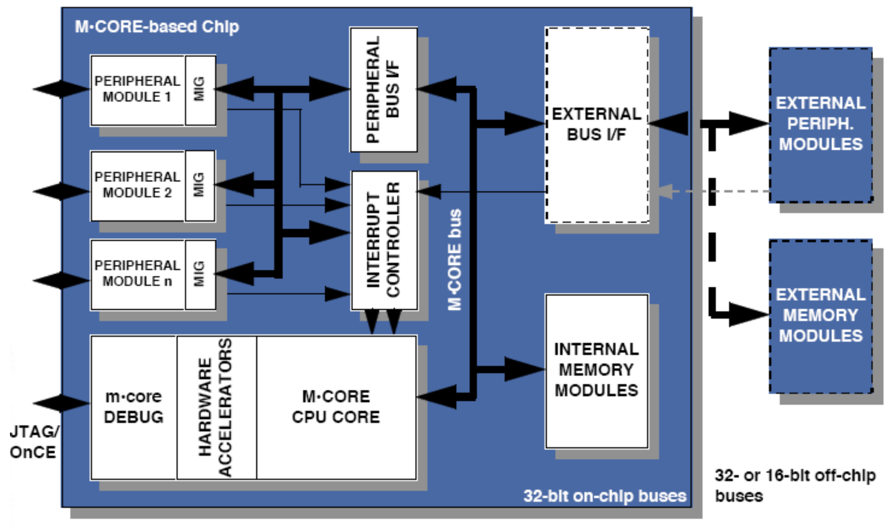


Figure 5.1: Block diagram of typical embedded processor configuration

processors [3]. The use of a cache in such system can greatly enhance the overall system performance by reducing the average case memory latency. Embedded systems, however, have different design constraints, making caches less appropriate or even unacceptable when trying to bridge the processor-memory speed gap. Many embedded systems are real-time and require *a priori* determination of worst-case execution times (WCETs) to guarantee at design time that timing constraints will always be met. The dynamic nature of caches make worst-case performance prediction extremely difficult resulting in WCET estimates that are usually too pessimistic for practical use. Another problem with caches is their energy consumption and memory footprint [12].

Embedded system designers may still decide to use a memory hierarchy to bridge the speed gap, choosing instead to manage it in a more predictable, and hence easily analyzed (and more tightly bounded) way. Scratch-pad memory (usually a small and fast on-chip SRAM memory) may be used to provide guaranteed access times. Both fast and slow embedded systems may have a memory hierarchy although for different reasons. A fundamental challenge is to allocate data objects into the hierarchy to provide optimal performance (e.g. run-time or energy). The most frequently used objects should be allocated to the fastest or lowest energy memories.

There are many examples of systems that opt to use a *cacheless* memory hierarchy. Examples include many low-end embedded processors, as well as many DSPs (Digital Signal Processors), including most 8-bit microcontrollers (e.g. Atmels' AVR series, PicMicro PIC series, Texas Instruments' TMS370Cx7x series), as well as many 16-bit and 32-bit embedded platforms (e.g. Motorola's 68HC12 and MCORE series, Intel's IXP network processor series, Analog Devices ADSP-21160M series, Atmel's ARM7TDMI series).

In the absence of a cache (for the reasons explained above and in chapter I) a new problem that was hidden by hardware control is now the responsibility of the programmer and / or compiler, namely, the mapping of the application's data objects to the memory levels. This mapping has to maintain good data locality in the sense that frequent data accesses should be from the fast memory levels rather than the slow ones. Until recently, this allocation had to be performed by the programmer using assembly directives, and many performance-critical kernels had to be completely written in assembly. Lately, there has been a significant amount of research invested in developing automatic and efficient data allocation techniques for cacheless systems [63, 64, 65, 66, 67, 29, 68, 69, 70, 71, 72, 1, 73, 74, 75, 76]. However, most of these techniques are not aimed at multitasking real-time applications. As we will show in this chapter, there is a strong interaction between the data allocation and the real-time scheduling problems in a multitasking real-time environment. In fact, even if these two design problems can be treated separately, doing so ignores many optimizations possible when considering them together. To this end, we will show that in a multitasking real-time setting, the data allocation and real-time scheduling problems are interdependent by nature. Solving one of them will affect the degree of optimality attainable by the other.

In this chapter we develop a unified, iterative data allocation and real-time scheduling approach to real-time systems design. We start by using *integer linear programming* (ILP) to find the data allocation map that optimizes the system performance. This allocation minimizes the number of cycles spent by each task accessing memory along its worst-case path minimizing the *worst-case execution times* (WCET)[2]. The resulting real-time properties are then used

[2]WCET is needed for real-time scheduling of the system.

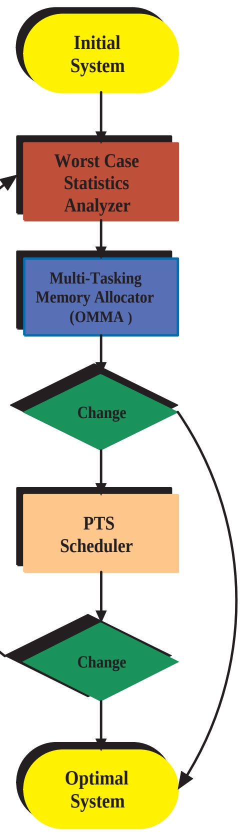

Initial System
Worst Case Statistics Analyzer
Multi-Tasking Memory Allocator (OMMA)
Change
PTS Scheduler
Change
Optimal System

Figure 5.2: The DARTS data allocation / real-time scheduling framework

to schedule the system using PTS which minimizes the preemptions between tasks without violating any of their real-time constraints. By limiting the preemptions, some tasks will have disjoint lifetimes, enabling them to share the same fast memory space, rather than forcing some to use slower memory. The data allocation phase is repeated, resulting in even more reductions in tasks' WCETs. This in turn introduces extra slack time that can be used by the PTS to further limit the system preemptions. This may result in even more tasks that use the smaller memory. This iterative data allocation / real-time scheduling approach is depicted in Figure 5.2. We examine how this iterative solution can be used at several allocation granularities depending on the amount of overheads (additional run-time cycles and additional overhead instructions) the system developer is willing to tolerate. To quantify the benefits of using our design approach, many simulations were performed on synthetic as well as real benchmarks. The results indicate that the using the DARTS framework will always result in an enhanced system performance, increased energy efficiency, and better memory utilization.

5.1 Related Work

The effectiveness of caches is primarily based on their ability to maintain, at each time during program execution, the subset of data that is frequently used at that time in the cache. In other words, caches are able to dynamically track the changing locality of references in a program. This is achieved automatically by hardware control and is completely invisible to the programmer/compiler. For cacheless systems, specific mechanisms are required to ensure that the mapping of data to the memory hierarchy address space takes full advantage of the locality of references in the program being executed. In the simplest case, the programmer lays out the allocation of data objects to the memory hierarchy address space manually using assembly directives based on the structure of the particular program. However, this often proves extremely tedious and prohibits any global compiler optimizations. Several automatic allocation methods have been developed to address this issue [63, 64, 65, 66, 67, 29, 68, 69, 70, 71, 72, 1, 73, 74, 75, 76]. Though these automatic data allocation methods have different flavors, they all try to optimally allocate the application's data objects (i.e. global variables, stack variables, and heap variables) to memory such that the system performance is enhanced and/or its energy utilization is improved.

Many automatic allocation methods have been developed and presented in the literature [63, 64, 65, 66, 67, 29, 68, 69, 70, 71, 72, 1, 73, 74, 75, 76]. To the best of our knowledge, however, only two studies [1, 63] considered the data allocation problem in a multi-tasking (non real-time) environment, while only one study [74] considered the problem in a real-time (only for a single-task) environment. None of the available studies considered the allocation problem in a real-time multi-tasking environment or the interaction between the data allocation and the real-time scheduling problems.

The available methods can be broadly categorized as being *static* or *dynamic*. Static methods are those whose allocation map does not change at run-time. Our data allocation strategy belongs to this category which is much more suitable for static timing analysis, and hence preferred for real-time applications. Many static allocation methods were proposed in the literature. The earliest two *static* methods presented by Sjodin et al. [71] and by Panda et al. [1] only addressed the allocation of global variables. The later study by Sjodin et al. [72] addressed the allocation of both global as well as stack variables with the goal of optimizing the use of pointers in addition to optimally allocating data to memory. Another study by Panda [1], on the other hand, targeted an architecture that, in addition to having multiple software-managed memories, uses a hardware-managed cache. In the presence of a cache, the main goal behind the data allocation method changes significantly from those addressing cacheless systems like ours. In the presence of a cache, the allocation goal becomes finding a memory map that would result in the minimal cache misses (especially conflict misses) by trying to allocate the data that can cause a conflict miss in the cache to the scratch-pad memory. The two studies by Avissar et al. [68, 73], and the study by Cao et al. [29] used a very similar ILP formulation like ours to statically allocate the application's data objects to memory, while a later paper by one of the authors of [73] reformulated the problem when some of the memory unit sizes are unknown at compile-time [64].

Many dynamic data allocation methods were presented in the literature as well. Dynamic allocation methods are those whose data allocation map can change at run-time. Those methods usually incur significant run-time overheads due to the reallocation of data at run-time. Moreover, the instructions added to move data between memories at run-time can dramatically increase code size as well. One dynamic data allocation technique is *software-managed caching*. This method emulates the working of a hardware cache in software; inserting instructions before loads and stores to check a software maintained cached tags. This method, however, incur large run-time overheads, code size, and data memory space for tags, and delivers poor real-time guarantees just like its hardware counterpart. Other dynamic allocation techniques which promise less run-time and code overheads have also been proposed in the literature [64, 66, 69, 70].

The work of Suhendra et al. [74] is the only one known that addressed the data allocation problem in a real-time environment. Suhendra et al. [74] used a static allocation strategy to allocate data objects to scratch-pad memories with the goal of minimizing the application's worst-case execution times (WCET). To this end, they formulated an ILP optimization problem, similar to ours, to allocate the data object to memory such that the application's WCETs are minimized. Nevertheless, this study assumed that all data objects are alive all the time (i.e. stack variables were modeled as global variables). This limits the benefits attainable through their work considerably since the lifetimes of stack variables (and of tasks as will be shown in this study) can be disjoint allowing more objects to be placed in the faster memory units, and in turn, resulting in significant improvements in the tasks WCETs as well as the overall performance of the application. Moreover, they did not consider the effect of having more than one task in their framework as many real-time applications do. On the other hand, the studies that considered the data allocation problem of a multi-tasking workload [1,63] assumed that all tasks can be alive at the same time. Though this is true in a FP system, techniques for limiting preemptions between tasks while maintaining the system schedulability in a real-time environment have been proposed in the literature (e.g. PTS presented previously). These techniques can be used to create groups of *mutually non-preemptive* tasks that can share the same memory space at run time as we will show in this study.

The allocation of heap data was considered in [76]. Nevertheless, in our proposed technique we do not consider the allocation of heap data for many reasons. First, dynamic memory allocation functions like `malloc()` and `new()` are not always available on every embedded platform. Second, many real-time design standards prohibit the use of these functions for real-time applications as they are not considered *real-time thread safe*. For example, the OSEK/VDX standard[3] specifies that all resources needed by the operating system have to be statically allocated during generation time and therefore no dynamic memory management is needed. The reason is that these functions can have a non-deterministic unpredictable behavior since their execution time can vary every time they are called [77]. By eliminating dynamic memory allocation, real-time applications are ensured that allocating any resource will never fail and will take a predictable amount of time. Still, if a heap has to be used with our proposed design

[3]The OSEK/VDX standard includes specifications for embedded operating systems, communication subsystems, and embedded network management systems [77].

strategy, it can be simply assumed that all heap data will be allocated to the slowest memory unit available without affecting the allocation strategy.

5.2 The Motivation behind DARTS

Given an application's workload, we would like to allocate its data objects to a memory hierarchy composed of a set of heterogeneous memory units which can have different access latencies, sizes, and/or bitwidths. The main goal of the allocation is to optimize the system performance and energy usage by reducing the time and energy spent on accessing data from slow memory units (which usually also have higher memory requirements per access [29]). A simple, yet naive, approach to this problem is to allocate the different data objects based on their access frequencies. That is, we can allocate the more frequently accessed data objects to the fast memory units (which also usually require less energy per access), while all less frequently accessed data objects are allocated to slower memories. In a single-tasking non real-time environment, the above problem simplifies to finding a strategy to allocate the application's global variables, heap variables if a heap is used, as well as stack variables, to the memory hierarchy.

Global variables are always *alive*[4] in the sense that they have to be allocated for the entire operational time of the application. Allocating global variables to a memory hierarchy therefore simplifies to allocating the subset of all global variables that are frequently accessed to the fast memory units, while leaving all other global variables reside on the slower, yet larger and cheaper, units of the memory hierarchy. Heap variables, though usually left in the slower levels of the memory hierarchy, are beyond the scope of this paper, as explained earlier, since they are rarely used in safety-critical real-time systems. Stack variables, on the other hand, are different. Normally, the stack grows in units of stack frames (which we will refer to them as *procedural stack frames* hereafter to make a distinction between them and a higher level of abstraction, namely the *task stack* discussed later in this section) one per procedure, where a procedural stack frame is a block of contiguous memory locations containing all the local variables, parameters, and return variables of the procedure. Procedural stack frames, however,

[4]The lifetime of a variable, defined as the period between its definition and last use [78], is an important metric affecting register allocation.

are born (i.e. allocated) upon procedure entry and die (i.e. de-allocated) after its exit. Hence, stack variables have limited lifetimes and any two stack variables with the disjoint lifetimes can share the same memory space. This can enable substantial space and performance improvements over global variables since, for example, two procedural stack frames can share the fast memory unit, resulting in enhanced performance and better memory space utilization.

A multi-tasking real-time environment has different requirements. First, real-time tasks have real-time constraints that have to be met. The particular data allocation strategy used affects the real-time properties of those tasks, and in turn, the scheduling characteristics of the system. On the other hand, the scheduling policy used affects the preemption relations of the system. Those preemption relations can be exploited to optimize the particular data allocation method used. Hence, in a real-time environment, the data allocation phase and the scheduling phase are tightly coupled. These two phases should not be performed independently even if possible since many potential optimizations would be wasted. Second, allocating data objects to memory based on their access frequencies obtained through profiling helps improve the average case application performance, assuming in the first place that the profile data is representative of the average-case. Most data allocation techniques proposed in the literature fall in this category. Nevertheless, real-time systems, especially safety critical ones, need guarantees on their worst-case execution rather than their average-case. An optimal allocation for the average-case may not necessarily be the optimal allocation for the worst-case. As we will explain in this section, both of these differences between single-tasking systems in a non real-time environment, and multi-tasking systems in a real-time environment, seek some kind of a unified iterative data allocation / real-time scheduling technique.

As discussed earlier, currently all design strategies perform the data allocation phase independent of the real-time scheduling phase. Though sometimes possible, this design strategy cannot take advantage of many potential optimizations. To see why, suppose some real-time application developer decides to perform the data allocation first then independently perform the real-time scheduling. In this case, the system designer would use some data allocation method to allocate the application's data objects to the memory units. Later use some timing analysis technique to estimate the worst-case execution times of the application. Using this information, he would schedule the workload according to some scheduling policy. In this case, the system designer did not take advantage of many possible optimizations due to the preemption relations of the real-time application. As an example, suppose that two of the application's tasks are the most frequently executed. Suppose that these two tasks share some common resource (e.g. an input / output device) that prohibits them from ever executing simultaneously. Suppose further that there is enough space in the fast memory unit to hold the data objects of one of these tasks but not both. An intelligent scheduling policy will probably make these two tasks *mutually non-preemptive*. Being mutually non-preemptive, they can share the same memory space at run-time because they are guaranteed never to execute simultaneously. Nevertheless, the data allocation was performed independent of the scheduling and hence, the allocator did not have enough information about the preemption relation of these two task. If it did, it would have placed both of them in the fast memory unit resulting in an improved performance, energy utilization, as well as an overall smaller memory footprint.

If the system developer, on the other hand, decides to perform the real-time scheduling first, then to allocate the data objects to physical memory, other problems will arise as well. First of all, the application's execution properties used to construct the schedule may not be valid anymore after the data is re-allocated to multiple memory units with different latencies. Even if the schedule remains valid, it might be too conservative because the scheduling policy did not exploit possibility of moving the data objects between the different memory units to render more efficient schedules. In all cases, a significant potential for optimization has been wasted by treating the data allocation and the scheduling problems as two independent design phases. In this study, we show how this problem lends itself to a unified iterative solution as was shown in Figure 5.2. Using this unified approach, we can guarantee that both the data allocator and the real-time scheduling have all the information needed to to render optimal solutions while maintaining all the real-time guarantees required in a real-time environment.

Another reason for exploiting an iterative unified data allocation / real-time scheduling approach is concerned with the requirements imposed on a real-time application. All real-time applications, especially safety critical ones, require guarantees on their worst-case execution properties. Nevertheless, as was explained in section 5.1, most data allocation methods available use profiling data to optimize the average-case properties of the system and are very profile dependent. Assuming that the profiling data is representative of the actual execution of the system in the first place, a particular memory allocation method that is optimal for the average-case, may not necessarily be optimal in the worst-case. For a real-time system, we prefer to use the information available on the system's worst-case execution properties. To this end, we are interested in minimizing the memory access cycles along the worst-case path of execution. Nevertheless, as data is allocated to the memory hierarchy, a new execution path may become the worst-case path. This, in turn, will change many of the real-time properties of the system necessitating the process to be repeated. As a result, all non-iterative techniques proposed in the literature may not lead to optimal solutions when real-time systems are considered. Again, this presents another reason to exploit a unified iterative data allocation and scheduling approach.

Another reason for exploiting an iterative unified data allocation / real-time scheduling approach is concerned with the requirements imposed on a real-time application. Though the real-time system developers cannot usually make the lifetimes of two procedures disjoint without altering the functionality of the procedures, we will show in this study that this is not the case when it comes to the lifetimes of real-time tasks. Tasks have interfering lifetimes if and only if two tasks can preempt each other. Nevertheless, techniques to minimize the preemptions between tasks while not violating any of their real-time constraints have been presented in the literature. One method that we are particularly interested in is preemption threshold scheduling or PTS.

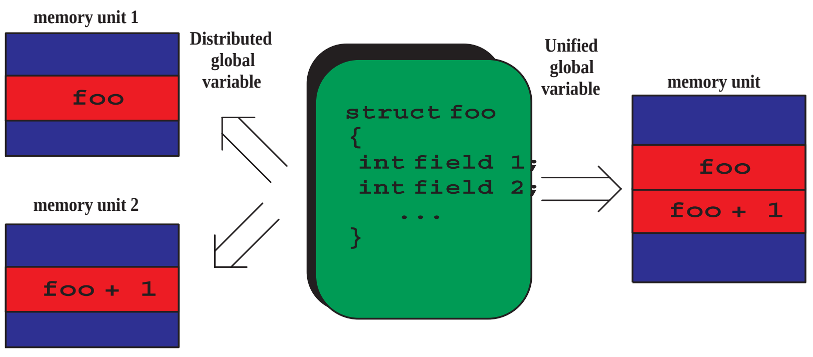
By limiting the preemptions, more tasks will have disjoint lifetimes enabling them to share the same memory space. That in turn means more data objects can fit in the faster units of memory resulting in enhanced run-time performance, and better energy utilization. But now that some tasks are allocated to the fast memory units their execution times is reduced, resulting in more slack time. This in turn enables even more task data to fit into the fast memory without violating the system's timing constraints. But this just means that even more tasks can fit into the fast memory now enhancing the system performance even more and leading to more slack time and so on.

5.3 Problem Description and Terminology

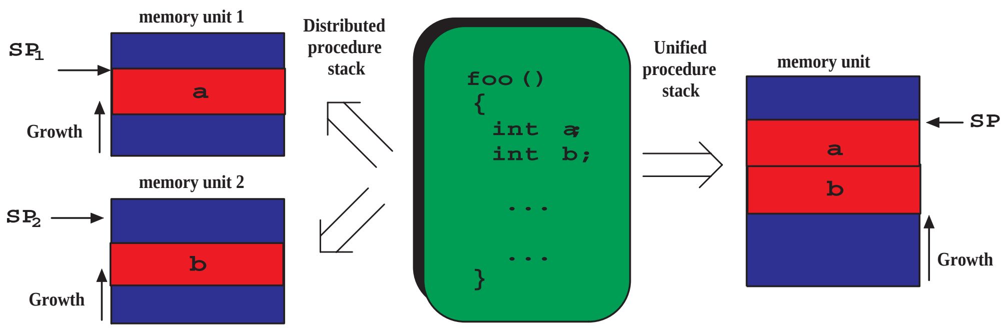
In this study we address the dual problem of data allocation and real-time scheduling combined. For our memory hierarchy, we assume that it is composed of N_U units. We define $t^r_k[t^w_k]$ as the time it takes to read[write] a single *word*[5] from[to] memory unit U_k for $k \in [1, N_U]$, respectively. We also define L_k as the number of lines in memory unit U_k and W_k as the number of words in each line. Hereafter, each memory unit in the system will be denoted by $U_k = \langle t^r_k, t^w_k, L_k, W_k \rangle$ for $k \in [1, N_U]$.

We are given a real-time workload, denoted by $\mathcal{W} = (\mathcal{G}, \mathcal{T})$, where $\mathcal{G}$ and $\mathcal{T}$ are two abstract entities representing a set of N_G global variables and a set of N_T real-time tasks, respectively. In the above definitions for the set of global variables $\mathcal{G}$ and the set of real-time tasks $\mathcal{T}$ we are not concerned with the actual form of these entities as long as they satisfy one basic conditions. The memory space where the global variable $g_i \in \mathcal{G}$ will reside at run-time, which we will denote by x^G_i, has to be allocated for the entire operational time of the system (this is just another way of saying that global variables are always alive). On the other hand, the memory space associated with each real-time task $T_m \in \mathcal{T}$ where the task saves its return addresses and/or uses for nested procedure calls and is called the *task's stack*, which we denote by x^T_m, need not have this property. A problem with these definitions that will show up later is the granularity level considered. The above definitions are very constraining since they force us to treat each global variable and each real-time task as a separate and indivisible entity that has to be allocated to a single memory unit. However, memory accesses of any data object are seldom uniform. For example, particular fields of a global structure construct, or particular locations of a global array, might be much more frequently accessed than others. Similarly, particular nested procedures in a real-time task might be much more frequently called than other.

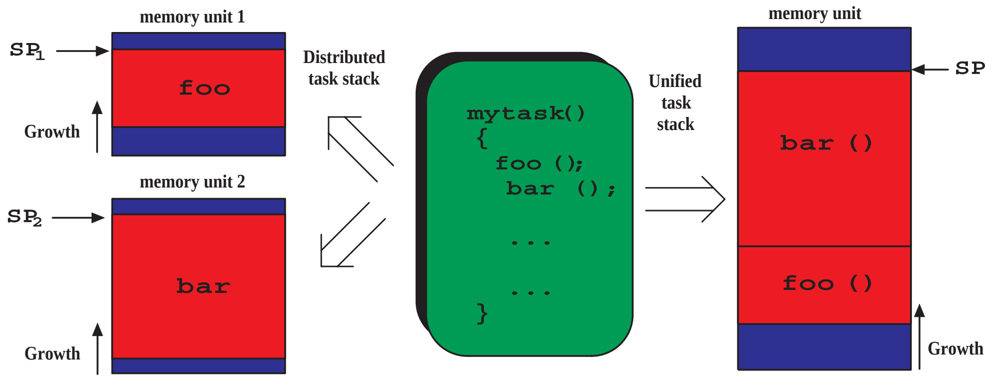
[5] In this study we use the term *word* to denote the smallest addressable memory unit and formulate all other quantities as multiples of a single word. This word might be an 8-bit word, a 16-bit word, a 32-bit word, etc.



(a) Global Variable



(b) Procedural Stack Frame



(c) Task Stack

Figure 5.3: Unified and distributed data objects

To alleviate the above problem with our definitions, we need to consider finer levels of object granularity. That is, we are not bound to alocate any single data object to a single memory unit. For example, a global variable can be either allocated to a single memory unit, or distributed between multiple memory units (figure 5.3(a)). A procedural stack frame can either be allocated as a single entity, or distributed between multiple memory units (figure 5.3(b)). A task stack can be allocated as a single entity, or distributed between multiple memory units (figure 5.3(c)), and so on. Nevertheless, we need to emphasize that the finer the granularity level adopted, usually the more run-time overhead needed. For example, if two procedures nested in the same real-time task are allocated to two different memory units, two stack pointers have to be used with all the additional instructions and run-time overheads to increment each stack pointer, decrement each stack pointer, etc. Similarly, if a task's stack is distributed between two memory units, the same argument holds, and so on. Still, as will be shown later in this study, sufficient performance improvements can usually be obtained by considering some intermediate granularity level rather than the finest one with the most overheads.

To exploit these different granularity levels, we need to modify the above definitions. To this end, we assume that the memory space x_i^G associated with each global variable g_i can be decomposed into a set of $N_P(i)$ *global partitions* and denote the memory space needed for each partition by x_{ij}^P for $j \in [1, N_P(i)]$. We also assume that the memory space for each task stack x_m^T associated with the real-time task T_m can be decomposed into a set of $N_F(m)$ *procedural stack frames* and denote the memory for each procedural stack frame by x_{mn}^F for $n \in [1, N_F(m)]$. At the finest level of granularity, we assume that the memory space of each procedural stack frame x_{mn}^F of each nested procedure can be further decomposed into a set of $N_V(m,n)$ *stack variables* and denote the memory space associated with each stack variable by x_{mno}^V for $o \in [1, N_V(m,n)]$. Clearly, with the above definitions we have two levels of granularity for the global variables, and three levels of granularity for the real-time tasks, for a total of six granularity levels for the overall real-time workload. The relation between these various data objects can be easily understood from the graphical representation of a generic memory map in Figure 5.4.

When trying to allocate the various data objects represented above at the different granular-

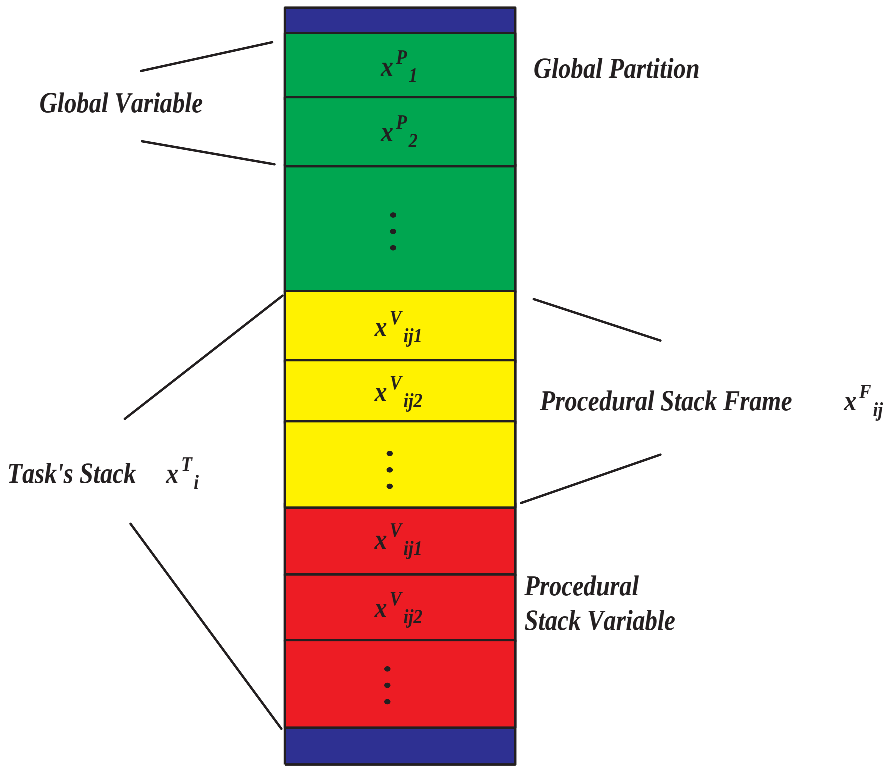


Figure 5.4: The different granularity levels for the data objects

ity levels, it is obviously possible to treat all of these objects as global variables that have to be allocated for all the time without violating any of the system's real-time constraints. Nevertheless, procedural stack frames, and in turn procedural stack variables, are automatic variables that are allocated on procedure entry and de-allocated on its exit. Hence, some of these procedural stack frames and stack variables might have disjoint life times enabling them to share the same memory space at run-time. To determine if two procedures have disjoint lifetimes, a call graph can be generated as shown in Figure 5.5. In a similar sense, two tasks have disjoint lifetimes, and their stacks can share the same memory space, if they are mutually non-preemptive. Hence, allocating task stack frames for all the time by dividing the memory space between them is not optimal. The reason is that performance as well as the memory footprint can be significantly improved by taking advantage of their disjoint lifetimes. To enable us of performing such a lifetime analysis on the task level, we need to define a call graph like structure. We define the *preemption graph* for a workload as a directed acyclic graph with the essential prop-

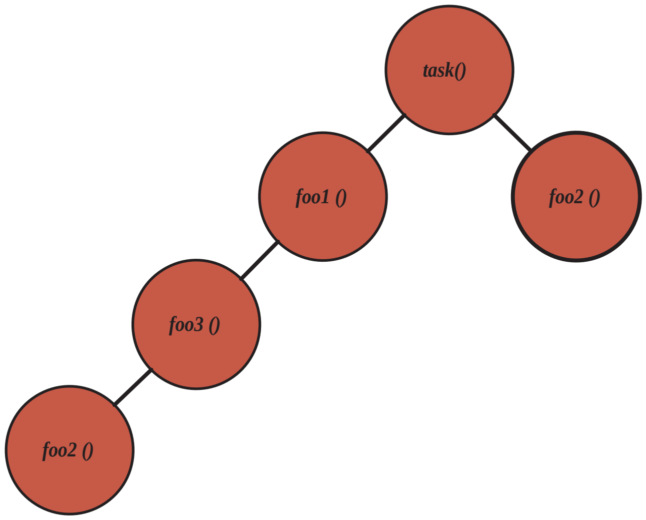


Figure 5.5: A call graph used to determine procedures lifetimes

erty that an edge exists from task A to task B if and only if task B can preempt task A. An example preemption graph for a generic *fully-preemptive* workload with three tasks is shown in Figure 5.6.

In this study we do not use profile data to determine the frequency of memory accesses to the different data objects. Instead, we use static timing analysis to determine the worst-case execution path the workload can take. We then try to minimize the number of memory access cycles on this worst-case path. As was explained earlier, the data allocation performed might change this worst-case path, but the iterative nature of our method implicitly accounts for that. To this end, we let $N^r_m(x^O)$ denote the number of reads (i.e. loads) of data object x^O (which can be a global memory space, global partition, task stack, etc.) on the worst-case execution path of our workload during any single invocation of task T_m. Similarly, we let $N^w_m(x^O)$ be the number of writes (i.e. stores) of data object x^O on the worst-case path that any invocation of task T_m can take. We would like to note here that the number of reads or writes of a single data object along the worst-case execution path of two tasks might be different. Nevertheless, this is also accounted for in our formulation as we show later. Finally, we let the number of memory words (i.e. size) of data object x^O be denoted by $S(x^O)$.

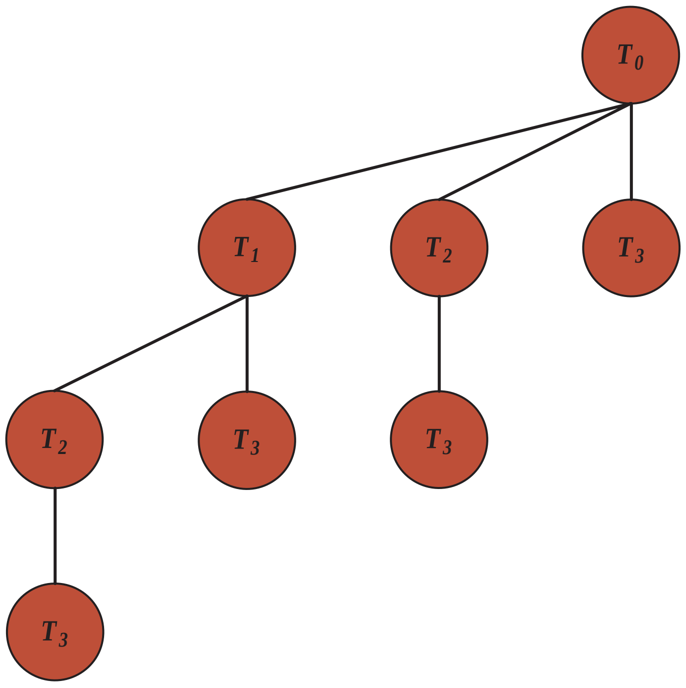


Figure 5.6: A preemption graph used to determine tasks lifetimes

Since real-time scheduling will also be performed, we will need to have the real-time properties of the task set. To this end, tasks can be *periodic* or *sporadic*. If T_m is periodic, the period p_m specifies a constant interval between arrival times of any two consecutive jobs (i.e. any two instances of the task), and if it is sporadic, p_m specifies a minimum interval between job arrivals. We say task T_m has a WCET of c_m time units if all instances of T_m can take no longer than c_m time units to execute. We also associate with each task T_m a unique priority $\pi_m \in \{1, 2, \ldots, N_T\}$ such that contention for resources is resolved in favor of the job with the highest priority that is ready to run. In addition to each task's priority, we associate with each task T_m a preemption threshold $\gamma_m \geq \pi_m$ which acts as the task's effective priority as it executes. That is, an executing task T_m cannot be preempted by a ready task T_n unless $\pi_n > \gamma_m$.

Finally, in this data allocation / real-time scheduling work, we assume that other compiler optimizations have been already performed on the application's variables. In particular, we assume that the register allocation has already been performed. Given the embedded application code, our goal is to determine the mapping of each variable not bound to a processor register to the different memory units while maximizing the application's overall memory access performance and minimizing its energy usage.

5.4 Problem Formulation

Given a real-time workload $\mathcal{W} = (\mathcal{T}, \mathcal{G})$ composed of N_T real-time tasks, N_G global variables, we would like to allocate the data objects of this workload to a memory hierarchy composed of N_U heterogeneous units. Before proceeding any further, however, we define for each data object x^O a set of N_U decision variables $I_n(x^O)$ for $n \in [1, N_U]$, such that $I_n(x^O)$ is non-zero if and only if x^O is allocated to memory unit U_n and is zero otherwise. That is, for each data object x^O we define the following set of decision variables for $n \in [1, N_U]$:

$$I_n(x^O) = \begin{cases} 1 & \text{if object } x^O \text{ allocated to unit } U_n \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

As was explained previously, there are various levels of granularity that can be considered. Nevertheless, we chose to only formulate the problem here for the two boundary cases due to space limitations. That is, we present below the problem formulation for the coarsest and the finest granularity levels only.

5.4.1 Allocation Granularity = Coarsest

At this level of granularity, the memory space for each global variables x_i^G, as well as the stack of each task x_m^T, is to be allocated to a contiguous memory space. In other words, data objects cannot be distributed between multiple memory units and no additional run-time or code overheads are needed. To this end, we can represent the memory access cycles of each task T_m for $m \in [1, N_T]$ along its worst-case execution path by the following expression:

$$\begin{aligned} &\sum_{n=1}^{N_U} \sum_{j=1}^{N_G} I_n(x_j^G) \left[T_n^r N^r(x_j^G) + T_n^w N^w(x_j^G) \right] S(x_j^G) \\ &+ \sum_{n=1}^{N_U} I_n(x_m^T) \left[T_n^r N^r(x_m^T) + T_n^w N^w(x_m^T) \right] S(x_m^T) \end{aligned} \tag{5.2}$$

The above expression represents the memory access cycles of each task along its worst-case execution path. Again, this worst-case execution path might change after the data allocation phase, however, this is implicitly accounted for by the iterative nature of our method. Moreover, since the application might be composed of multiple tasks, we need to create an objective function that accounts for all tasks in the application. To this end, we note that any task T_m can be invoked at max $k_m = H/p_m$ times during any *hyperperiod*[6] H [13]. Hence, an objective function that accounts for frequency of invocation of each task along with its worst-case path information is simply given by the following weighted sum:

$$\sum_{m=1}^{N_T} k_m \times MAC(T_m) \tag{5.3}$$

Where the *memory access cycles* (MAC) of each task is given by (5.2).

Minimizing the above objective function will result in the optimal data allocation of each task's objects along its worst-case execution path. Three constraints for the above problem are needed, however. First, any global variable memory space x_i^G and any task stack x_m^T needs only be allocated to a single memory unit. The following two constraints enforce this criteria:

$$\sum_{k=1}^{N_U} I_k(x_i^G) = 1 \ \ (\forall i \in [1, N_G]) \tag{5.4a}$$

$$\sum_{k=1}^{N_U} I_k(x_m^T) = 1 \ \ (\forall m \in [1, N_T]) \tag{5.4b}$$

We also need to guarantee that the total memory space available in the system's memory hierarchy is adequate for all global variables and the task stacks. Nevertheless, task stacks need not be allocated for all the time if two tasks are mutually non-preemptive. Hence, we can make use of the fact that some tasks do not preempt others to let them share the same memory space. This will minimize the overall memory requirements of the system, as well as enable us to locate more data objects to the faster memories, and hence, enhance the system's performance. As was explained earlier, the system's preemption graph can be used to analyze the preemption properties of our system. Any two tasks that lie on the same path from the root node to any of the leaf nodes in the system's preemption graph cannot share the same memory

[6] A workload's *hyperperiod* is defined as the least common multiple of the tasks' periods or inter-arrival times.

space since a preemption can occur. To this end, let $\mathcal{L}$ be the set of all leaf node tasks in the system's preemption graph. Moreover, let $P_q(T_l)$ be the q^{th} unique path from the root node to the leaf node $T_l \in \mathcal{L}$, and let there be $NP(T_l)$ such paths. Then the following constraint can be derived:

$$\begin{aligned}&\sum_{i=1}^{N_G} I_n(x_i^G)\, S(x_i^G) \;+ \sum_{\forall T_m \in P_q(T_l)} I_n(x_m^T)\, S(x_m^T) \leq L_k\, W_k \\ &(\forall k \in [1, N_U],\; \forall T_l \in \mathcal{L},\; \forall q \in [1, NP(T_l)])\end{aligned} \tag{5.4c}$$

Minimizing equation (5.2) subject to the constraints given by (5.4a), (5.4b), and (5.4c) will result in the optimal allocation of the global variables and the task stack frames. Nevertheless, at this level of granularity, it becomes very hard to obtain significant performance improvement. Memory requirements can be improved significantly, on the other hand, as will be shown in the simulations and experimentation section.

5.4.2 Allocation Granularity = Finest

This is the finest granularity level we address, and also the one requiring the most run-time and code overheads. For this granularity level, the memory access cycles of any task during any of its invocations on its worst-case path can be represented by the following expression:

$$\begin{aligned}&\sum_{k=1}^{N_U}\sum_{i=1}^{N_G}\sum_{j=1}^{N_P(i)} I_k(x_{ij}^P)\left[T_k^r N_m^r(x_{ij}^P) + T_k^w N_m^w(x_{ij}^P)\right] S(x_{ij}^P) \\ &+ \sum_{k=1}^{N_U}\sum_{n=1}^{N_F(m)}\sum_{o=1}^{N_V(m,n)} I_k(x_{mno}^V) T_k^r N_m^r(x_{mno}^V) S(x_{mno}^V) \\ &+ \sum_{k=1}^{N_U}\sum_{n=1}^{N_F(m)}\sum_{o=1}^{N_V(m,n)} I_k(x_{mno}^V) T_k^w N_m^w(x_{mno}^V) S(x_{mno}^V)\end{aligned} \tag{5.5}$$

Similar to the previous granularity level, there are four constraints that need be added. First, we still don't want any data object to be located to more than one memory unit. Hence, the first three constraints are given by the following:

$$\sum_{k=1}^{N_U} I_k(x_{ij}^P) = 1 \;\; (\forall j \in [1, N_P(i)], \forall i \in [1, N_G]) \tag{5.6a}$$

$$\sum_{k=1}^{N_U} I_k(x_{mno}^V) = 1 \;\; (\forall o \in [1, N_V(m,n)], \forall n \in [1, N_F(m)], \forall m \in [1, N_T]) \tag{5.6b}$$

The fourth constraint which guarantees that the memory available is adequate for the various data objects is given by the following expression:

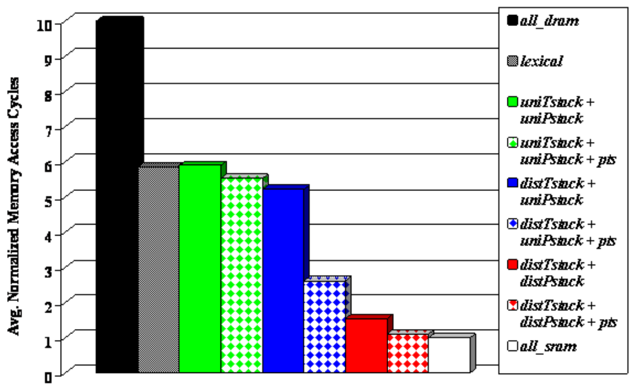
$$\sum_{i=1}^{N_G}\sum_{j=1}^{N_P(i)} I_k(x_{ij}^P)\,S(x_{ij}^P) \;+ \sum_{\forall T_m \in P_q(T_l)}\;\sum_{\forall x_{ij}^F \in P_{ir}(x_{il}^F)}\;\sum_{k=1}^{N_V(i,j)} I_n(x_{ijk}^V)\,S(x_{ijk}^V) \le L_n\,W_n \tag{5.6c}$$

$$\big(\forall k \in [1, N_U],\ \forall T_l \in \mathcal{L},\ \forall q \in [1, NP(T_l)],\ \forall x_{ml}^F \in \mathcal{L}_m,\ \forall r \in [1, NP_m(x_{ml}^F)]\big)$$

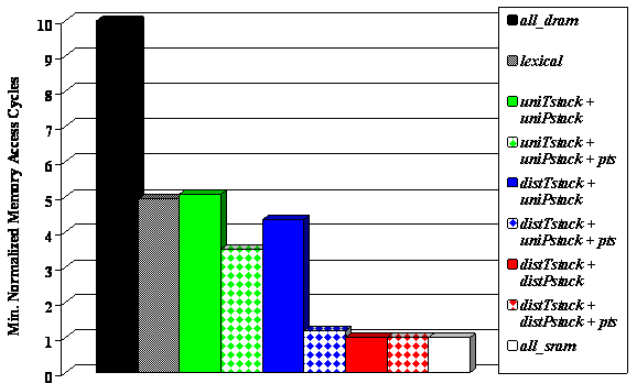
5.4.3 Preemption Limiting

As was shown in the previous sections, in a preemptive system the preemption relations between the tasks are part of the model (i.e. they affect the ILP constraints) and clearly will affect the allocation results. To exploit this property to our advantage, we try to prevent as many preemptions as possible without violating the system's real-time constraints. Preemption threshold scheduling, or PTS, limits preemptions to occur only when necessary to maintain system schedulability. Tasks that run non-preemptively with respect to each other can be mapped into the same run-time thread and share the same stack, reducing memory requirements and other preemption overheads. The preemption threshold assignment problem was considered in several studies. An algorithm known as the MPTAA (*maximal preemption threshold assignment algorithm* was developed by Wang et al. [16] to find a feasible preemption threshold assignment if it exists. This algorithm was shown later [79] to be memory optimal in the sense that it finds the preemption threshold assignment resulting in the minimal total stack space usage.

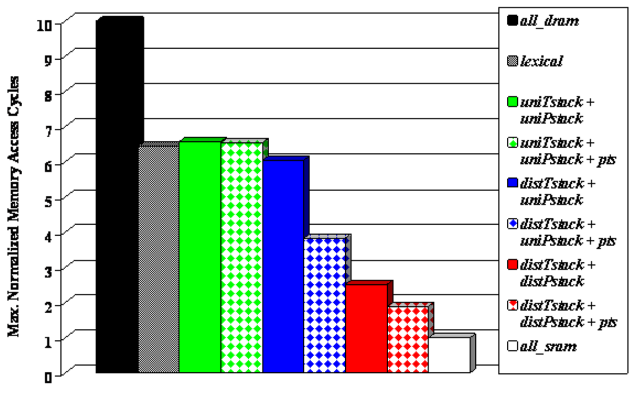
In our framework, once the data allocation objective functions have been minimized using an ILP solver, the MPTAA algorithm is then used to find the optimal preemption threshold assignment. Clearly this assignment will change the preemption relations of the system (as can be seen in the system's preemption graph), so the entire process shown in Figure 5.2 is repeated until a fixed point is reached, resulting in the final data allocation and real-time schedule.



(a) Average normalized memory access cycles



(b) Minimum normalized memory access cycles



(c) Maximum normalized memory access cycles

Figure 5.7: Normalized memory access cycles for synthetic workloads

Table 5.1: Normalized overhead cycles

Allocation Granularity	**Best-Case**	**Worst-Case**
uniTstack + uniPstack	0.12%	0.12%
distTstack + uniPstack	0.12%	0.62%
distTstack + distPstack	0.12%	3.33%

5.5 Simulation and Analysis

To assess and analyze our proposed unified data allocation / real-time scheduling method, several simulations were performed and their results analyzed. We present these simulations in the following section. We start by presenting our simulations for synthetic workloads, followed by our simulations for the Fly-By-Wire workload from the Autopilot and PapaBench benchmarks [58] and the GScope workload; a real-time oscilloscope emulator.

5.5.1 Synthetic Workloads

Matlab [38] was used to analyze our memory allocation method and compare it to existing methods using synthetic workloads. The memory hierarchy is composed of an on-chip SRAM with read / write latency of 1 cycle, an off-chip DRAM with read / write latency of 10 cycles, and an EEPROM with a 3 cycle read / latency and a 500 cycle write latency. The DRAM and EEPROM sizes are fixed at 8K words and 1K word, respectively. The SRAM size is varied as explained below.

One hundred synthetic workloads are used for our first two experiments, all with real-time constraints that must be met. The number of global variables in each workload is approximately 10 variables ranging from 1 word scalars up to 40 word arrays. Each of the tasks has between 1 and 10 nested procedures, with each procedure having between 1 and 16 stack variables (between 1 and 16 words in size). The load / store frequency ratio for global and stack variables is approximately 1:1; this only has an effect for memories with different read and write times.

The first simulation performed compares the memory execution cycles at different allocation granularities with different real-time scheduling schemes. The memory access cycles for

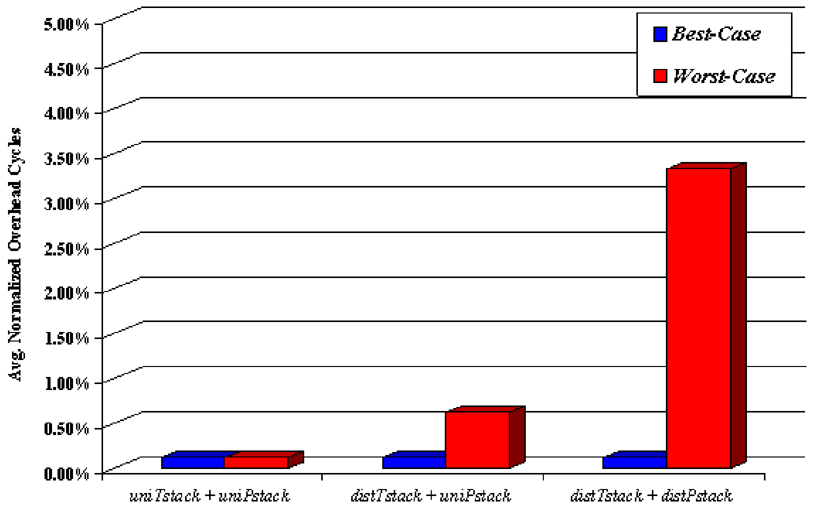
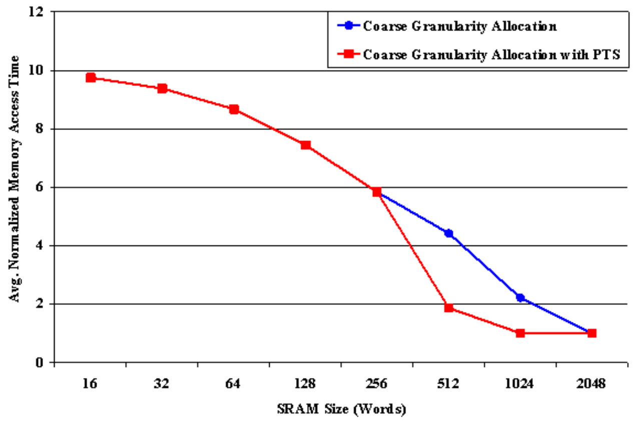


Figure 5.8: Normalized overhead cycles

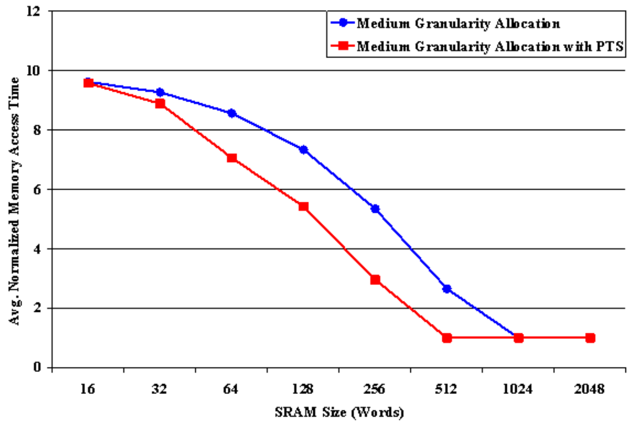
each experiment was normalized to the *all_sram* case (in which SRAM is large enough to hold *all* data). The average, minimum, and maximum, of these normalized memory access cycles for all workloads is presented in Figures 5.7(a), 5.7(b), and 5.7(c), respectively. The memory hierarchy for this experiment has on-chip 1-cycle SRAM and off-chip 10-cycle DRAM. The SRAM size s fixed at 256 words, which is about 20% of the total data memory required by each workload.

The memory hierarchy model used was chosen to be consistent with Motorola's M-Core chips [80]. The microcontrollers have a three level memory hierarchy; an EEPROM level with a 3 cycle read latency and a 500 cycles write latency, an external DRAM level with a constant 10 cycles read / write latency, and an internal SRAM level with a single cycle read / write latency. We emphasize here that this simple memory model is sufficient for evaluating the proposed scheme, however, more realistic models can be used without affecting the resulting framework. For example, different DRAM access modes like *fast page mode* can be incorporated into our model without affecting our formulation or most of the results presented in this section. Moreover, we perform a sensitivity analysis later on some of the parameters used in this simulation to investigate their effect on the proposed scheme.

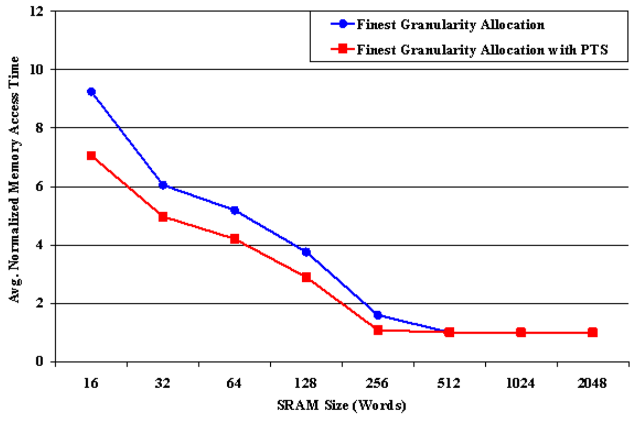
The *all_dram* case shows the worst performance; all data objects are now allocated to the



(a) Coarsest Allocation Granularity



(b) Medium Allocation Granularity



(c) Finest Allocation Granularity

Figure 5.9: Normalized memory access cycles as a function of SRAM size

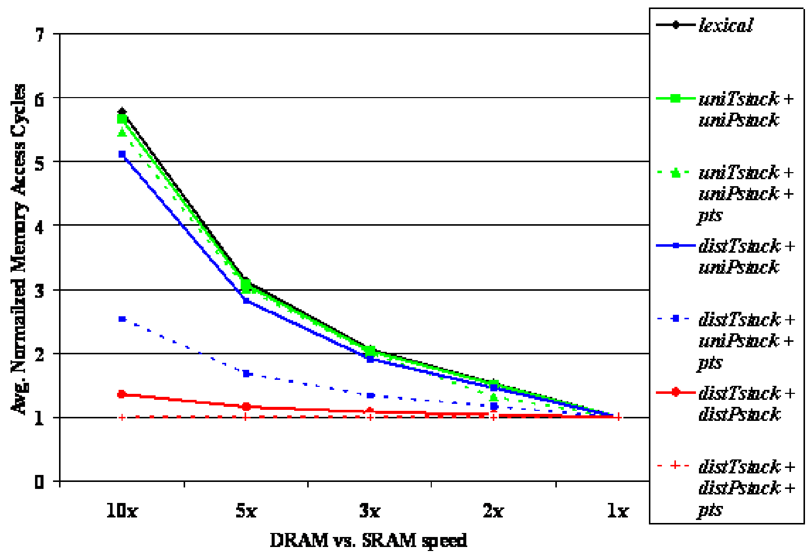


Figure 5.10: Avg. normalized memory access cycles as a function of DRAM speed

slow DRAM. The *lexical* case shows the average execution time when all data objects are allocated to memory sequentially in the order they are found in the program code or symbol table.

The intermediate cases are combinations of various options. First, all task stacks can be allocated to a single memory unit (*uniTstack*) or they can be distributed (*distTstack*). Second, all procedural stack frames for a given task can be allocated to a single memory (*uniPstack*) or distributed (*distPstack*). Third, the scheduler may ignore possible preemption limitations (no label) or it may use PTS (*pts*).

In the *lexical* case, no frequency information is used; some frequently accessed variables may still be allocated to DRAM. For *uniTstack* + *uniPstack*, the frequency information cannot be used because although almost every task has some variables that are more frequently accessed than others, we are forcing the allocator to group the tasks' data objects together, so this information is lost. There is some minor improvement here when we limit the preemptions using PTS (*uniTstack* + *uniPstack* + *pts*) because the allocator can now share parts of the fast memory between the tasks with the highest memory traffic. At a finer level of allocation granularity, we can see some improvements in the *distTstack* + *uniPstack*. Here the allocator has more freedom to allocate the procedural stack frames with the highest memory access from different tasks to the fast memory and not bound to group them with the rest of the data objects of the parent task. Still, in this case the fast memory can only be shared between procedural stack frames with disjoint lifetimes, as the preemption relation of the parent tasks are not accounted for yet. A significant improvement at this same allocation granularity can now be obtained when PTS is used to limit the preemptions between tasks as can be seen in the *distTstack* + *uniPstack* + *pts* case. In fact, the average execution time was reduced by 50% when PTS was used. This significant improvement is due to the fact that the most frequently called procedures from all mutually non-preemptive tasks are now grouped in the fast memory. The *distTstack* + *distPstack* case gives the allocator the most freedom in allocating the frequently accessed data objects to fast memory. At this level of granularity, stack variables belonging to more than one procedure, which might belong to more than one task, can be distributed between multiple memory units. This, however, is at the expense of significant run-time and code overheads as was explained previously. In the final *distTstack* + *distPstack* + *pts* case, we can allocate enough frequently-accessed data objects to fast memory to result in an average memory access cycles that is close to the ideal case of *all_sram*. In this case, the memory accesses are only 11.2% more than the ideal *all_sram* case with just 20% of the memory footprint.

The overheads imposed by using each of the three non-trivial granularity methods are shown in Table 5.1. These values are the average (over all workloads) normalized number of cycles spent adjusting and maintaining multiple stacks for the tasks and procedures. We assume that six cycles are needed to switch the storage context between two memory units[7]. At the coarsest granularity level, the overhead is 0.12% in both the best and worst cases. This is expected at this allocation granularity since we only need to switch between memory units at the tasks boundaries (i.e. on the entry or exit point of a task) which is only dependent on the invocation frequencies of the tasks. At the procedure granularity level, we have two cases to consider. In the best case, all procedures nested in a single task would be allocated to the same memory unit, and hence only task boundaries need to be considered where a switch between

[7]We assume a pointer must be swapped with another register or memory. Architectures with many pointer registers will have less overhead.

two memory units would occur. In the worst-case, we might have to switch between two memory units at the entry and exits of every procedure. Still, the overhead in this case is at most 0.62%. Finally, at the finest granularity level, we might have to switch between two memory units whenever we access a stack variable within a procedure. Depending on the number of variables per procedure, and the number of procedures per task, as well as the invocation frequencies of the different task, this will lead to the highest number of overhead cycles. As can be seen from the Figure 5.8, this can be a number as large as 3.33% of the total memory access cycles, which might be significant for some systems. Code motion could be used to group data accesses to the same unit, reducing memory switches. We leave this for future work.

Our second experiment explores the amount of fast memory (i.e. SRAM) needed to obtain certain normalized execution times. The results are shown in Figures 5.9(a), 5.9(b), and 5.9(c), for the different allocation granularities with and without the use of PTS. The amount of SRAM available was varied from 16 to 2048 words, and the average normalized memory access cycles were calculated. At the coarsest allocation granularity (i.e. uniTstack + uniPstack), PTS does not improve the average memory cycles until the memory size reaches a certain level. This is because *none* of the task stacks can fit into the fast (but small) memory to begin with, so limiting the preemptions cannot improve system performance. This is not the case for finer granularities, shown in Figures 5.9(b) and 5.9(c). Here PTS improves the system performance for almost all sizes of the available memory. Since the data objects are much smaller than in the coarse granularity case, many more are small enough to fit into the fast memory. Hence by limiting the preemptions we can fit many of those objects in the same fast memory space, resulting in significant performance improvements.

As a final experiment, the sensitivity of DARTS to the model parameters used at the different allocation granularities is investigated. To this end, the SRAM and DRAM sizes were fixed, and the DRAM speed was varied from 10 times the SRAM speed, to the same speed of the SRAM (i.e. until no more benefits will be obtained from allocating the data to the SRAM since the DRAM has the same speed). The average memory access cycles are shown as a function of the DRAM versus SRAM speed in Figure 5.10. As can be seen from the figure, limiting

Table 5.2: The real-time tasks composing the Fly-By-Wire benchmark

ID	Name	Frequency
T1	receive_radio_task	40Hz
T2	send_data_to_autopilot_task	40Hz
T3	check_autopilot_values_task	20Hz
T4	servo_transmit_task	20Hz
T5	check_failsafe_task	20Hz
I1	servo_interrupt	_
I2	spi_interrupt	_
I3	radio_interrupt	_

preemptions with DARTS still results in improved performance at all relative DRAM speeds. However, at finer allocation granularities, the improvements are much more significant and increase in magnitude with the decrease in the DRAM speed relative to the SRAM.

5.5.2 The Fly-By-Wire Workload

We next assess our proposed method by evaluating the Fly-By-Wire workload available in the Autopilot and PapaBench benchmark for the AVR architecture. This workload is composed of five periodic tasks and three sporadic interrupt handlers as shown in table 5.2. The experimental platform used was earlier shown in Figure 2.1 where the application's source files are compiled into assembly code using the AVR-GCC compiler. The resulting assembly code is then analyzed to determine the number of reads and writes along the worst-case path of execution of each task using the AVR-SAT tool. OMMA (the Optimal Multitasking Memory Allocator) and the PTSS toolboxes are then used to perform the optimal allocation and real-time scheduling phases.

The normalized memory access cycles for the Fly-By-Wire benchmark and its normalized total execution cycles are shown in figures 5.11 and 5.12, respectively. As can be seen, the improvement in the memory access cycles are evident. Nevertheless, the Fly-By-Wire workload spends less than 4% of its execution time accessing memory, and therefore the effect of the data allocation policy on the overall execution time is small as can be seen from Figure 5.12. Another interesting observation can be seen from Figure 5.11 where the memory access cycles at the procedure granularity with PTS results in better performance that at the finer granularity

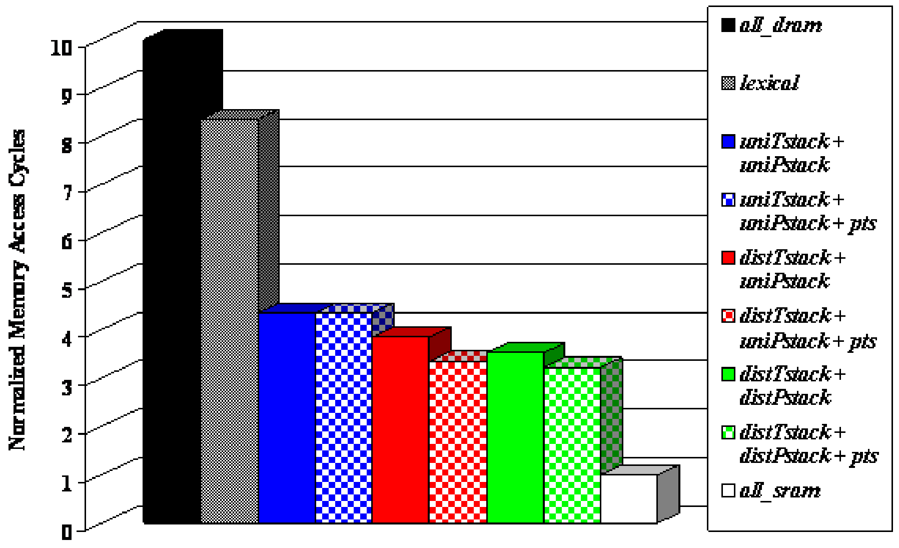


Figure 5.11: Normalized memory access cycles for Fly-By-Wire workload

level in the absence of PTS. This shows that we can sometimes avoid the additional run-time overheads of the finer granularity levels because similar improvements can be obtained with PTS.

It is also interesting to note how the energy dissipation spend on the memory traffic was minimized under the DARTS framework as can be seen in Figure 5.13. As expected, limiting the preemption through PTS minimized the preemption overheads including the energy per access. In other words, fewer preemptions now take place resulting in better energy utilization as was pointed to earlier.

5.5.3 The GScope Workload

Finally, we evaluate DARTS on the GScope workload. GScope is an oscilloscope emulator implemented on an Atmega128 microcontroller by Atmel. The workload is composed of three tasks, and as it turns out, is a little bit more memory intensive than the Fly-By-Wire workload.

As in the experiment performed in the previous section, the normalized memory access cycles for the GScope workload are shown in Figure 5.14. Again, the improvement in the memory

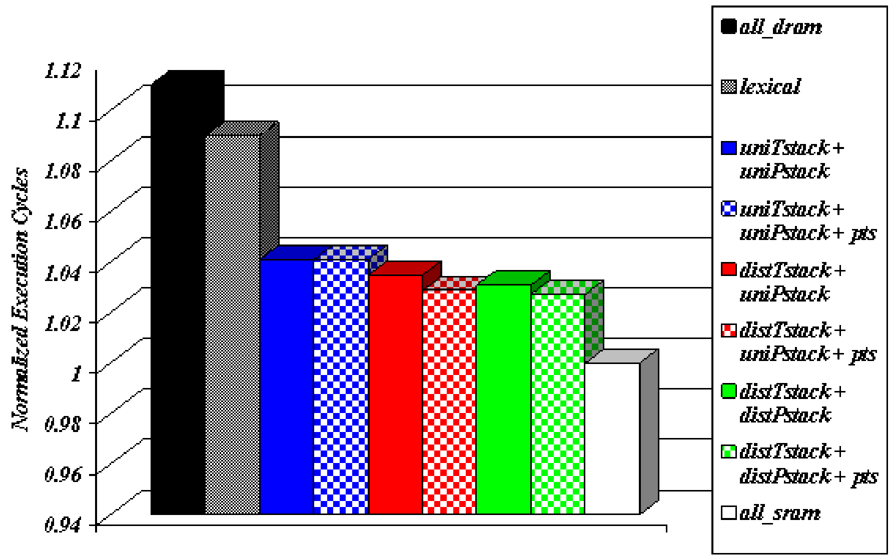


Figure 5.12: Normalized worst-case execution cycles for Fly-By-Wire workload

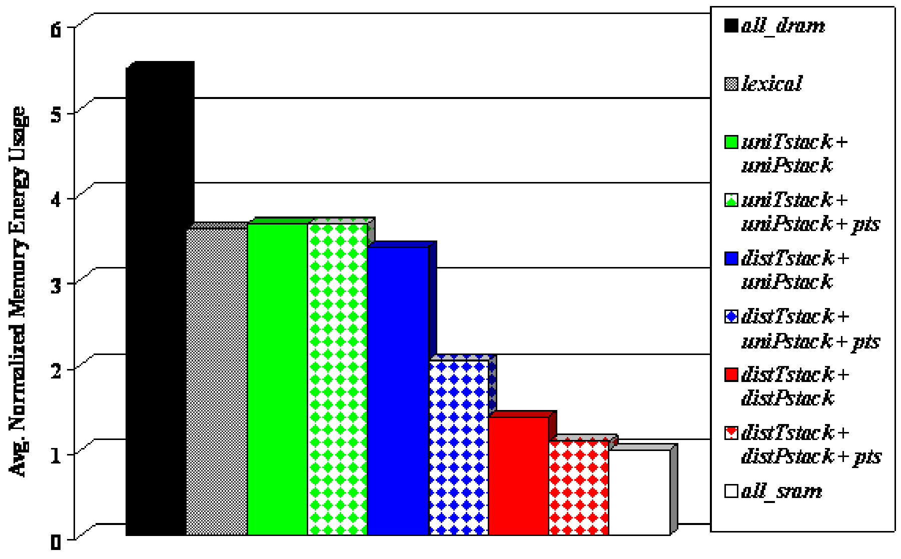


Figure 5.13: Normalized energy dissipation for the Fly-By-Wire workload

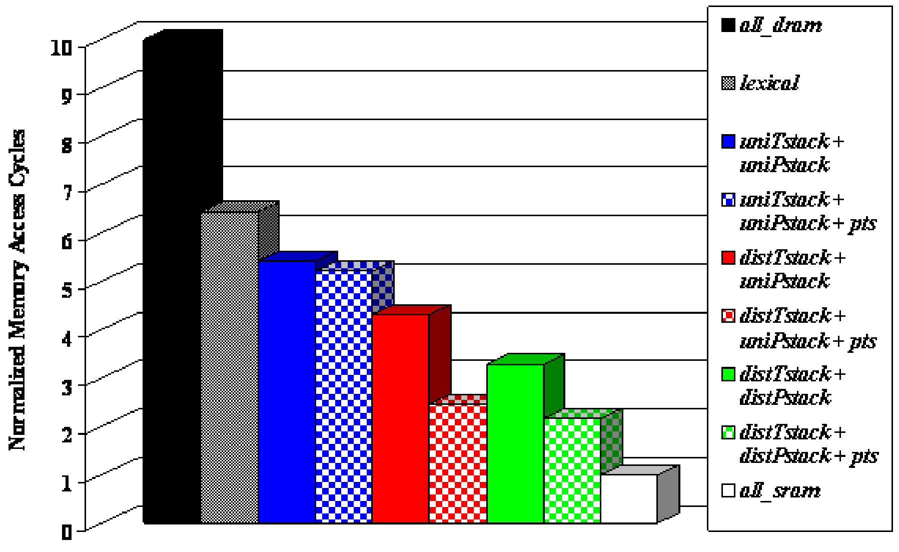


Figure 5.14: Normalized memory access cycles for GScope workload

access cycles is evident. In a similar manner to the Fly-By-Wire workload, we obtain significant improvements when we limit preemptions at the procedure granularity level; reducing the need for addressing the finer granularity level with the additional overheads associated with it.

5.6 Chapter Summary

In this chapter we present a novel, unified data allocation/real-time scheduling technique which improves run-time performance for real-time multi-tasking systems with memory hierarchies. Our method allocates data to memory to minimize worst-case execution time and improve performance while combining with preemption threshold scheduling to reduce stack memory requirements. Our technique was formulated at different allocation granularities for flexibility.

After evaluating the proposed technique, we find that it leads to significant performance improvements in general, and can reduce overall memory access cycles significantly. In fact, an improvement factor of 5 was obtained in some cases. Still, even in situations where memory access cycles are not improved dramatically, there can be a large improvement in memory energy use.

Chapter 6

Future Work

In this work we have shown how software-managed data allocation and real-time scheduling need not be treated as a separate design problems anymore with the help of the DARTS framework. Nevertheless, there are still many interesting problems that have not been addressed in this work and are left to future work. Some of these are (in random order):

- Investigate the possibility of using the slack time generated online due to early task completion to further enhance our data allocation and schedule. Clearly part of the allocation will have to be done online in that case.

- A study by Venkatesan et al.[81] showing how that different pages in DRAM have substantially different retention times. DARTS can be used in this case to perform the allocation not just based on the frequency of memory access, but also on the retention rate of the memory.

- A study by Bankar et al[12] claimed that caches will always consume more energy than other memories of the same capacity. Nevertheless, it might be the case that a smaller capacity cache would lead to the same performance needed by a large memory. Comparing the energy dissipation in this case is not a sufficient metric to make any conclusion. In our future work we propose to perform a comparison of the energy dissipated to reach a certain level of performance.

These and many other questions have not been answered completely by our work. We leave them, and many others, to be investigated in future studies.

Bibliography

[1] P. R. Panda, N. D. Dutt, and A. Nicolau, “On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, no. 3, pp. 682–704, 2000.

[2] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann, 2002.

[3] J. Handy, *The cache memory book*. San Diego, CA, USA: Academic Press Professional, Inc., 1993.

[4] F. Mueller, “Static cache simulation and its applications,” Ph.D. dissertation, Tallahassee, FL, USA, 1995.

[5] Y.-T. S. Li, S. Malik, and A. Wolfe, “Efficient microarchitecture modeling and path analysis for real-time software,” in *RTSS ’95: Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS ’95)*. Washington, DC, USA: IEEE Computer Society, 1995, p. 298.

[6] R. T. White, C. A. Healy, D. B. Whalley, F. Mueller, and M. G. Harmon, “Timing analysis for data caches and set-associative caches,” in *RTAS ’97: Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS ’97)*. Washington, DC, USA: IEEE Computer Society, 1997, p. 192.

[7] F. Mueller, “Generalizing timing predictions to setassociative caches,” in *9th Euromicro workshop on real-time systems*, Toledo, Spain, 1997.

[8] T.-Y. Huang, J. W.-S. Liu, and D. Hull, "A method for bounding the effect of dma i/o interference on program execution time," in *RTSS '96: Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*. IEEE Computer Society, 1996, p. 275.

[9] A. E.-H. Mahmoud, "Hard-real-time multithreading: A combined microarchitectural and scheduling approach," Ph.D. dissertation, Raleigh, NC, USA, 2006.

[10] K. Lahiri, S. Dey, D. Panigrahi, and A. Raghunathan, "Battery-driven system design: A new frontier in low power design," in *ASP-DAC '02: Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design*. Washington, DC, USA: IEEE Computer Society, 2002, p. 261.

[11] J. M. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits*. New Jersey, USA: Prentice-Hall, 1996.

[12] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: A design alternative for cache on-chip memory in embedded systems," in *Proc. of the 10th International Workshop on Hardware/Software Codesign, CODES*, Estes Park, Colorado, 2002.

[13] J. W. S. W. Liu, *Real-Time Systems*. Upper Saddle River, NJ, USA: Prentice Hall, 2000.

[14] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, pp. 284–292, 1993. [Online]. Available: citeseer.ist.psu.edu/audsley93applying.html

[15] S. K. Baruah, L. E. Rosier, and R. R. Howell, "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor," *Real-Time Syst.*, vol. 2, no. 4, pp. 301–324, 1990.

[16] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *RTCSA '99: Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, Hong Kong, China, 1999, p. 328.

[17] M. Saksena and Y. Wang, "Scalable multi-tasking using preemption thresholds," in *The 6th IEEE Real-Time Technology and Application Symposium*, Cheju Island, South Korea, 2000.

[18] *ThreadX User Guide*, 2003. [Online]. Available: http://www.expresslogic.com

[19] J. Lehoczky, “Fixed priority scheduling of periodic task sets with arbitrary deadlines,” in *11th Real-Time Systems Symposiom (RTSS’ 90)*, Lake Buena Vista, FL, 1990.

[20] K. W. Tindell, A. Burns, and A. J. Wellings, “An extendible approach for analyzing fixed priority hard real-time tasks,” *Real-Time Syst.*, vol. 6, no. 2, pp. 133–151, 1994.

[21] L. Sha, R. Rajkumar, and J. P. Lehoczky, “Priority inheritance protocols: An approach to real-time synchronization,” *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, 1990.

[22] J. B. Goodenough and L. Sha, “The priority ceiling protocol: A method for minimizing the blocking of high priority ada tasks,” in *IRTAW ’88: Proceedings of the second international workshop on Real-time Ada issues.* New York, NY, USA: ACM Press, 1988, pp. 20–31.

[23] T. P. Baker, “Stack-based scheduling for real-time processes,” *Real-Time Syst.*, vol. 3, no. 1, pp. 67–99, 1991.

[24] J. Regehr, “Scheduling tasks with mixed preemption relations for robustness to timing faults,” in *RTSS ’02: Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS’02)*, Austin, TX, 2002, p. 315.

[25] T. D. Burd and R. W. Brodersen, “Design issues for dynamic voltage scaling,” in *ISLPED ’00: Proceedings of the 2000 international symposium on Low power electronics and design*. Rapallo, Italy: ACM Press, 2000.

[26] A. Ferre and J. Figueras, “On estimating leakage power consumption for submicron cmos digital circuits,” 1995. [Online]. Available: citeseer.ist.psu.edu/687140.html

[27] T. D. Burd and R. W. Brodersen, “Energy efficient cmos microprocessor design,” in *HICSS ’95: Proceedings of the 28th Hawaii International Conference on System Sciences.* Kauai, HI: IEEE Computer Society, 1995, p. 288.

[28] A.Qadi, S.Goddard, and S.Farritor, “A dynamic voltage scaling algorithm for sporadic tasks,” in *RTSS ’03: Proceedings of the 24th IEEE International Real-Time Systems Symposium*. Los Alamitos, CA, USA: IEEE Computer Society, 2003.

[29] Y. Cao, H. Tomiyama, T. Okuma, and H. Yasuura, "Data memory design considering effective bitwidth for low-energy embedded systems," in *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*. Kyoto, Japan: ACM Press, 2002, pp. 201–206.

[30] A. Miyoshi, C. Lefurgy, E. V. Hensbergen, R. Rajamony, and R. Rajkumar, "Critical power slope: Understanding the runtime effects of frequency scaling," in *ICS '02: Proceedings of the 16th international conference on Supercomputing*, New York, NY, USA, 2002, pp. 35–44.

[31] V. Gutnik and A. P. Chandrakasan, "Embedded power supply for low-power dsp," *IEEE Trans. Very Large Scale Integr. Syst.*, pp. 425–435, 1997.

[32] E. Nisley, "Rising tides," *Dr. Dobb's Journal, vol. 346*, 2003.

[33] G. Qu, "What is the limit of energy saving by dynamic voltage scaling?" in *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, San Jose, California, 2001.

[34] J. R. Lorch and A. J. Smith, "Improving dynamic voltage scaling algorithms with pace," in *SIGMETRICS '01: Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, Cambridge, Massachusetts, United States, 2001, pp. 50–61. [Online]. Available: citeseer.ist.psu.edu/lorch01improving.html

[35] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*. Banff, Alberta, Canada: ACM Press, 2001, pp. 89–102.

[36] G. Quan and X. Hu, "Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors," in *DAC '01: Proceedings of the 38th conference on Design automation*. Las Vegas, Nevada, United States: ACM Press, 2001.

[37] A. Sinha and A. P. Chandrakasan, "Energy efficient real-time scheduling," in *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*. San Jose, California: IEEE Press, 2001, pp. 458–463.

[38] *Matlab 6.5*, The MathWorks, 2003. [Online]. Available: http://www.themathworks.com

[39] *AVR-GCC*, 2005. [Online]. Available: www.avrfreaks.net/AVRGCC/

[40] *AiSee Features*, 2004. [Online]. Available: http://www.aisee.com/features.htm

[41] J. R. Levine, T. Mason, and D. Brown, *Lex & Yacc (A Nutshell Handbook)*. Boston, MA, USA: O'Reilly Media, 1992.

[42] R. Ghattas, *ssRTOS: Singe-Stack RTOS with Preemption Threshold Support*, 2006. [Online]. Available: http://www4.ncsu.edu/~rghatta/ssrtos/

[43] J. J. Labrosse, *Microc/OS-II*, 2nd ed. CMP Books, 2002.

[44] R. Barry, *Free Real-Time Operating System (FreeRTOS)*, 2006. [Online]. Available: http://www.freertos.org/

[45] L. Barello, *AvrX Real-Time Kernel*, 2006. [Online]. Available: http://barello.net/avrx/

[46] A. Coombs, "Designing an efficient rtos for a resource-constrained 8-bit microprocessor," Tech. Rep., 2001. [Online]. Available: www.omimo.be

[47] R. Davis, N. Merriam, and N. Trace, "How embedded applications using an (rtos) can stay within on-chip memory limits," in *In 12th Proceedings of Euromicro Conference on Real-Time Systems*, Maastricht, Netherlands, June 2000.

[48] W. Chen, Z. Wu, and X. Wang, "Minimizing memory utilization of task sets in smartosek," *aina*, vol. 02, pp. 552–558, 2005.

[49] *Nucleus OSEK*. [Online]. Available: http://www.mentor.com

[50] P. Gai, G. Lipari, and M. D. Natale, "Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip," in *RTSS '01: Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, London, UK, 2001, p. 73.

[51] S. Kim, S. Hong, and T.-H. Kim, "Perfecting preemption threshold scheduling for object-oriented real-time system design: from the perspective of real-time synchronization," in *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*. Berlin, Germany: ACM Press, 2002, pp. 223–232.

[52] D.-Z. He, F.-Y. Wang, and W. Li, “Dynamic preemption threshold scheduling for specific real-time control systems,” in *Proceedings. 2005 IEEE conference on Networking, Sensing and Control*. Beijing, China: IEEE Computer Society, 2005.

[53] W. Kim, J. Kim, and S. L. Min, “Preemption-aware dynamic voltage scaling in hard real-time systems,” in *Proceedings of the 2004 international symposium on Low power electronics and design (ISLPED’04)*. Newport Beach, California, USA: ACM Press, 2004, pp. 393–398.

[54] M. Saksena, P. Karvelas, and Y. Wang, “Automatic synthesis of multi-tasking implementations from real-timeobject-oriented models,” in *Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Newport, CA, USA, 2002.

[55] J. Chen, A. Harji, and P. Buhr, “Solution space for fixed-priority with preemption threshold,” in *RTAS ’05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*. San Francisco, CA: IEEE Computer Society, 2005, pp. 385–394.

[56] A. K. Mok, “Tracking real-time system requirements,” in *7th International Workshop on Real-Time Computing and Applications Symposium (RTCSA 2000)*. Cheju Island, South Korea: IEEE Computer Society, 2000.

[57] *The Paparazzi Project*. [Online]. Available: http://www.recherche.enac.fr/paparazzi/

[58] F. Nemer, H. Cass, P. Sainrat, J.-P. Bahsoun, and M. D. Michiel, “Papabench: a free real-time benchmark,” in *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, F. Mueller, Ed., Dresden, Germany, 2006.

[59] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing : A VLIW Approach to Architecture, Compilers and Tools*. Boston, NY, USA: Morgan Kaufmann, 2005.

[60] C. Chakrabarti and D. Gaitonde, “Instruction level power model of microcontrollers,” in *IEEE International Symposium on Circuits and Systems*, 1999. [Online]. Available: citeseer.ist.psu.edu/260043.html

[61] J. Shandle, “More for less: Stable future for 8-bit microcontrollers,” *TechOnLine*, 2004. [Online]. Available: http://www.techonline.com

[62] M. Le, “8-bit microcontrollers: Still going . . .” *EE Times*, 2004. [Online]. Available: http://www.eetimes.com

[63] O. Ozturk, M. Kandemir, and I. Kolcu, “Shared scratch-pad memory space management,” in *7th International Symposium on Quality of Electronic Design (ISQED 2006)*. San Jose, CA: IEEE Computer Society, 2006.

[64] N. Nguyen, A. Dominguez, and R. Barua, “Memory allocation for embedded systems with a compile-time-unknown scratch-pad size,” in *CASES ’05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, San Fransisco, CA, 2005, pp. 115–125.

[65] M. Verma, L. Wehmeyer, and P. Marwedel, “Dynamic overlay of scratchpad memory for energy minimization,” in *CODES+ISSS ’04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. Stockholm, Sweden: ACM Press, 2004, pp. 104–109.

[66] J. D. Hiser and J. W. Davidson, “Embarc: An efficient memory bank assignment algorithm for retargetable compilers,” in *LCTES ’04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, Washington, DC, USA, 2004, pp. 182–191.

[67] S. Udayakumaran and R. Barua, “Compiler-decided dynamic memory allocation for scratch-pad based embedded systems,” in *CASES ’03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, San Jose, California, USA, 2003, pp. 276–286.

[68] O. Avissar, R. Barua, and D. Stewart, “Heterogeneous memory management for embedded systems,” in *CASES ’01: Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, Atlanta, Georgia, USA, 2001, pp. 34–43.

[69] M. Kandemir, J. Ramanujam, J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, “Dynamic management of scratch-pad memory space,” in *DAC ’01: Proceedings of the 38th*

conference on Design automation. Las Vegas, Nevada, United States: ACM Press, 2001, pp. 690–695.

[70] M. Kandemir and A. Choudhary, "Compiler-directed scratch-pad memory hierarchy design and management," in *DAC '02: Proceedings of the 39th conference on Design automation*, New Orleans, Louisiana, USA, 2002, pp. 628–633.

[71] J. Sjodin, B. Froderberg, and T. Lindgren, "Allocation of global data objects in on-chip ram," in *In Proc. Workshop on Compiler and Architectural Support for Embedded Computer Systems*, 1998. [Online]. Available: citeseer.ist.psu.edu/sjodin98allocation.html

[72] J. Sjodin and C. von Platen, "Storage allocation for embedded processors," in *CASES '01: Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, Atlanta, Georgia, USA, 2001, pp. 15–23.

[73] O. Avissar, R. Barua, and D. Stewart, "An optimal memory allocation scheme for scratch-pad-based embedded systems," *Trans. on Embedded Computing Sys.*, vol. 1, no. 1, pp. 6–26, 2002.

[74] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "Wcet centric data allocation to scratchpad memory," in *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*. Miami, Florida: IEEE Computer Society, 2005.

[75] E. G. Hallnor and S. K. Reinhardt, "A fully associative software-managed cache design," in *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*. Vancouver, British Columbia, Canada: ACM Press, 2000, pp. 107–116.

[76] A. Dominguez, S. Udayakumaran, and R. Barua, "Heap data allocation to scratch-pad memory in embedded systems," *Journal of Embedded Computing(JEC)*, 2005.

[77] *The OSEK/VDX Standards*. [Online]. Available: http://www.osek-vdx.org/

[78] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[79] R. Ghattas and A. G. Dean, “Memory-optimal and robust preemption threshold scheduling with improved system responsiveness,” Tech. Rep., 2006. [Online]. Available: http://www.cesr.ncsu.edu/agdean/Papers/pts_tech_report.pdf.

[80] *M-Core Reference Manual*, 2005. [Online]. Available: www.freescale.com

[81] R. Venkatesan, S. Herr, and E. Rotenberg, “Retention-aware placement in dram (rapid): software methods for quasi-non-volatile dram,” in *Symposium on The Twelfth International High-Performance Computer Architecture, 2006*, 2006.

[82] E. K. P. Chong and S. H. *Zak, *An Introduction to Optimization*, New York, USA, 2002.

[83] E. Kreyszig, *Introductory Functional Analysis with Applications*, New York, USA, 1978.

[84] Montgomery and G. C. Runger, *Applied Statistics and Probability for Engineers*, New York, NY, 1999.

Appendices

Empirical Modeling of Proccesors Energy Consumption

For completence, we list below the three relations the govern the energy and power consump-toin of any CMOS based processor.

$$E_{dyn} = C_P \; f_{CLK} \; V_{CC}^2 \; \Delta t \tag{A-1a}$$

$$E_{stat} = S_P \; V_{CC}^2 \; \Delta t \tag{A-1b}$$

$$\max[f_{CLK}] = \frac{K_p \; (V_{CC} - V_{th})^{1.8}}{V_{CC}} \tag{A-1c}$$

To develop the needed imperial models, the *microcontroller automated power analyzer* or MAPA (see section 2.4.6), in addition to the manufacturers data sheets, were used to determine the current required for operation the circuit at different operating voltages. To start with, the constants of proportionality C_P and S_P are to be estimated in a way that would minimize the modeling error. Fortunately, finite-dimensional mathematical optimization along with statistical interference provide us with the needed framework to construct the needed *unbiased* estimators[82, 83, 84]. Once the first two constant's of proportionality have been estimated, we proceed to obtain an estimate of our third and last constant of proportionality, namely K_P, in a very similar manner.

Before we proceed, we need to recall that the total energy dissipated by a CMOS ciruit is given by the followiing expresoin:

$$E_{CMOS} = C_P \ f_{CLK} \ V_{CC}^2 \ \Delta t + S_P \ V_{CC}^2 \ \Delta t \tag{A-2}$$

While the total power dissipated by a CMOS circuit is simply obtained by diffirenciating the above expression to get:

$$P_{CMOS} = C_P \ f_{CLK} \ V_{CC}^2 + S_P \ V_{CC}^2 \tag{A-3}$$

Moreover, the following expression presents the supply current I_{CC} at the supply voltage V_{CC}:

$$I_{CC} = C_P \ f_{CLK} \ V_{CC} + S_P \ V_{CC} \tag{A-4}$$

Givent the above expression for the supply current I_{CC}, we can proceed to construct a linear optimization framework to estimate the reqiured parameters. To this end, note that equation (A-4) relates the total supply current of a CMOS based circuit to its clock frequency/supply voltage product *linearly* through the constant of proportionality, C_p, and to the supply voltage alone (again linearly) through the constant of proportionality, S_p. In other words, that equation can also be written as follows:

$$I_{CC} = \begin{bmatrix} f_{CLK} \times V_{CC} & V_{CC} \end{bmatrix} \begin{bmatrix} C_P \\ S_P \end{bmatrix} \tag{A-5}$$

Assuming that we have P data points for the supply current I_{CC} at different clock frequencies and operating voltages, then the following also holds for $k = [1, 2, \dots, P]$:

$$\langle I_{CC} \rangle_k = \begin{bmatrix} \langle f_{CLK} \times V_{CC} \rangle_k & \langle V_{CC} \rangle_k \end{bmatrix} \begin{bmatrix} C_P \\ S_P \end{bmatrix} \tag{A-6}$$

We now define the following matrices and vectors:

$$\mathbf{A}_N = \begin{bmatrix} \langle f_{CLK} \times V_{CC} \rangle_1 & \langle V_{CC} \rangle_1 \\ \langle f_{CLK} \times V_{CC} \rangle_2 & \langle V_{CC} \rangle_2 \\ \vdots & \vdots \\ \langle f_{CLK} \times V_{CC} \rangle_P & \langle V_{CC} \rangle_P \end{bmatrix} \quad \mathbf{y}_P = \begin{bmatrix} \langle V_{CC} \rangle_1 \\ \langle V_{CC} \rangle_2 \\ \vdots \\ \langle V_{CC} \rangle_P \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} C_P \\ S_P \end{bmatrix} \tag{A-7}$$

yhe symbols $\mathbf{A}_P$, $\mathbf{y}_P$, and $\mathbf{x}$, were used to be consistent with the technical literature []. It should be obvious that with the above definitions solving for our first two constants of proportionality becomes simply solving the linear set of equations. Moreover, since $rank(\mathbf{A}_P) = 2 < P$, this is just a classical example of what is referred to as an *over-determined* set of linear equations. A solution to the above problem can be found by finding some vector, $\hat{\mathbf{x}} = [\hat{C}_P \; \hat{S}_P]^T$ such that the difference, (i.e. error), between the actual values given by the vector $\mathbf{y}_P$, and the calculated values given by $\hat{\mathbf{y}}_P = \mathbf{A}_P\hat{\mathbf{x}}$ is minimal in some sense (as will be seen shortly, turns out to be in least sum-of-squares sense).

Fortunately, this ends up to be a simple, finite-dimensional, optimization problem over the complete inner-product space (more formally the P dimensional Hilbert space of real numbers $\mathbb{R}^P$). Therefore, the principle of orthogonality holds, and there does exist a unique vector, $\hat{\mathbf{x}} \in \mathbb{R}^2$, such that the following holds:

$$\|\mathbf{e}_P\|^2 = \left\|\mathbf{y}_P - \hat{\mathbf{y}}_P\right\|^2 = \|\mathbf{A}_P\mathbf{x} - \mathbf{A}_P\hat{\mathbf{x}}\|^2 \leq \left\|\mathbf{y}_P - \mathbf{A}_P\mathbf{z}\right\|^2 \quad \vee\, \mathbf{z} \in \mathbb{R}^2 \tag{A-8}$$

where $\|\cdot\|^2$ is simply the square of the Euclidian length defined over $\mathbb{R}^P$, and for any vector $\mathbf{v}^T = [v_1 \; v_2 \; \ldots \; v_P] \in \mathbb{R}^P$, is given by the following:

$$\|\mathbf{v}\|^2 = \sum_{i=1}^{P} (v_i)^2 \tag{A-9}$$

Clearly, the above norm represents nothing more than the sum of the squared error, and minimizing such an error is usually referred to as a *least-squares regression analysis problem*, and in the statistical jargon, it is referred to as a *multiple linear regression analysis*. The unique solution to the above problem has been given in the literature by the following []:

$$\hat{\mathbf{x}} = \left(\mathbf{A}_P^T \, \mathbf{A}_P\right)^{-1} \; \mathbf{A}_P^T \, \mathbf{y}_P \tag{A-10}$$

It was shown in the literature that the above estimate not only results in the lowest modeling error (in the sun-of-squares sense), but it also posses some useful statistical properties as will be explained in the next section when we establish the necessary statistical bounds on our estimated parameters. Finally, once C_P and S_P have been calculated, we shall use (A-1c) to calculate an estimate for our third and last constant of proportionality, namely, K_P, using a very similar procedure. Once the empirical models have been developed, statistical analysis has been used to bound the estimation errors. The complete models developed have been already presented in table 4.2.