Optimal Unified Data Allocation and Task Scheduling for Real-Time Multi-Tasking Systems

Rony Ghattas, Gregory Parsons and Alexander G. Dean
Center for Embedded Systems Research
Department of Electrical and Computer Engineering
NC State University
Raleigh, NC 27695
rony_ghattas@ncsu.edu, gsparson@nc.rr.com, alex_dean@ncsu.edu

Abstract

Many real-time (RT) embedded systems can benefit from a memory hierarchy to bridge the processor/memory speed gap. These RT embedded systems usually utilize a cacheless architecture to avoid the time variability which complicates the timing analysis essential for RT systems. In the absence of a cache the burden of allocating the data objects to the memory hierarchy is on the programmer/compiler. There has been much research into allocating data objects into the memory hierarchy for efficient execution. However, existing methods have limited scope and ignore some aspects of RT multitasking embedded systems.

In this paper we propose a synergistic, optimal approach to allocating data objects and scheduling real-time tasks for embedded systems. We allocate data using integer linear programming (ILP) to minimize each task’s worst-case execution time (WCET), then perform preemption threshold scheduling (PTS) on the tasks to reduce stack memory requirements while still meeting hard RT deadlines. 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. The increased slack time can be used by PTS to reduce preemptions further, until a fixed point is reached. We evaluate the technique with several levels of data object granularity using both synthetic workloads and a realtime benchmark and find it to be highly effective.

1. Introduction

The processor-memory speed gap is a well-known bottleneck for server, desktop and high-performance computer systems, and has driven memory system design for decades [18]. These systems rely on hardware-managed caches and other hardware accelerating mechanisms to try to hide the gap. Embedded systems have different design constraints, making caches (and usually some other hardware accelerating mechanisms as well) 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 requirements will always be met. The dynamic nature of caches makes worst-case performance prediction extremely difficult resulting in WCET estimates that are usually too pessimistic for practical use.

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. Scratchpad memory may be used to provide guaranteed access times. In fact, a memory hierarchy may be present even in low-performance embedded systems. This can arise if a microcontroller lacks sufficient on-chip RAM and hence needs external memory, which will be slower for reasons such as external bus speed, multiplexed address/data bus, and narrow bus width. 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 and/or energy). The most frequently used objects should be allocated to the fastest or lowest energy memories. Without a hardware cache controller to handle this allocation dynamically, this is now the responsibility of the programmer and/or compiler.

There has been a significant amount of research invested in developing automatic and efficient data allocation techniques for cacheless systems[2–5,7–10,15–17,20–24]. However, most of these techniques are not
aimed at multi-tasking real-time applications. As we will show in this work, there is a strong interaction between the data allocation and the real-time scheduling problems in a multi-tasking real-time environment. Even if these two problems can be treated separately, doing so can waste many potential optimizations that can only be considered when these two design problems are solved simultaneously. In fact, in a multi-tasking real-time setting, the data allocation and real-time scheduling problems are iterative by nature. Solving one of them will affect the degree of optimality attainable by the other.

Preemption threshold scheduling (PTS) explores the middle ground between fully-preemptive and fully-non-preemptive scheduling approaches. PTS only allows preemptions which are necessary to maintain system schedulability. PTS statically allocates to each task a nominal priority (to preempt other tasks) and a preemption threshold (its effective priority while executing). 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[25] to find a feasible preemption threshold assignment if it exists. This algorithm was shown later[6] to be memory optimal in the sense that it finds the preemption threshold assignment resulting in the smallest total system stack space usage that maintains the system schedulability.

In this study 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. Though finding the optimal allocation using ILP has been shown to be NP-hard, it was also shown by many researchers that many algorithms based on the simplex method converge most of the time to an optimal solution in polynomial time [19]. Once an optimal allocation has been found, the real-time properties of the system based on this allocation are computed. These properties are then used to schedule the system using preemption threshold scheduling (PTS). PTS minimizes the number of preemptions between tasks without violating any of their real-time constraints[6, 25]. 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 PTS to further limit the system preemption. This may result in even more tasks that use the smaller memory. This iterative data allocation/real-time scheduling approach is depicted in Figure 1.1. We examine how this iterative solution can be used at several allocation granularities depending on the amount of overheads (run-time and code overheads) 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 our unified data allocation/real-time scheduling scheme will always result in an enhanced system performance and better memory utilization. Though a similar strategy could be used to allocate program code to memory as well, we leave this extension to future work.

This paper is divided into seven sections. In Section II we present an overview of currently available automatic techniques for data allocation in cacheless systems. The motivation behind our design strategy is presented in Section III. Section IV presents a formal description of the problem as well as some of the tools used to solve it. We formulate the data allocation/real-time scheduling problem in Section V. To assess and analyze our proposed design strategy, we perform simulations and experiments in Section VI and present their results. Finally, we present the main conclusions of this work in Section VII.
2. Related Work

Many automatic allocation methods have been developed [2–5, 7–10, 15–17, 20–24]. To the best of our knowledge, only two studies [16,17] considered the data allocation problem in a multi-tasking (but non-real-time) environment, while only one [22] considered the problem in a real-time (but 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. It is much better suited for static timing analysis, and hence preferred for real-time applications. Many static allocation methods exist. The earliest two static methods presented by Sjodin et al. [20] and by Panda et al. [17] only addressed the allocation of global variables. A later study by Sjodin et al. [21] 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 [17], on the other hand, targeted an architecture that, besides having multiple software-managed memories, uses a hardware-managed cache. The main goal is finding a memory allocation that minimizes cache misses by placing conflict-miss-triggering data objects in the scratch-pad memory. Methods proposed by Avissar et al. [2] and Cao et al. [4] use an ILP formulation (similar to ours) to statically allocate the application’s data objects to memory, while a later paper [3] reformulated the problem when some of the memory unit sizes are unknown at compile-time [15].

Many dynamic data allocation methods exist 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. The instructions added to move data between memories at run-time can also dramatically increase the code size. 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 cache tags. This method, however, incurs 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 [5].

The work of Suhendra et al. [22] is the only one known that addressed the data allocation problem in a real-time environment. Suhendra et al. [22] used a static allocation strategy to allocate data objects to scratch-pad memories with the goal of minimizing the application’s WCET. To this end, they formulate an ILP optimization problem to allocate the data object to memory such that the application’s WCETs are minimized. However, all data objects are assumed to be alive all the time (i.e. stack variables are modeled as global variables). This limits the benefits attainable through their work considerably. Our proposed method, on the other hand, is aware of disjoint stack variables and tasks lifetimes. This allows 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. Furthermore, Suhendra et al. did not consider the effect of having more than one task in their framework as many real-time applications do.

The studies that considered the data allocation problem of a multi-tasking workload [16,17] assume that all tasks can be alive at the same time. Though this is true in a fully-preemptive system – techniques for limiting preemptions between tasks while maintaining the system schedulability in a real-time environment have been proposed (e.g. preemption threshold scheduling [6,25]). 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 [5]. Many real-time design standards prohibit the use of dynamic memory allocation for real-time applications as they are not considered real-time safe. For example, the OSEK/VDX standard 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. One reason is that dynamic memory allocation functions like malloc and new have execution time that can vary with each call. Another reason is to prevent run-time allocation errors. Although our proposed method does not explicitly support dynamic memory allocation, it could...

---

1The term fully-preemptive is used to denote workloads where preemptions are not limited to occur between particular task groups and not others as with the use of PTS[6].

2We say two tasks are mutually non-preemptive if neither can preempt the other. Two tasks can be made mutually non-preemptive in the framework of preemption threshold scheduling by setting the preemption threshold of each task equal to the priority of the other [6].

3The OSEK/VDX standard includes specifications for embedded operating systems, communication subsystems, and embedded network management systems.
be supported by simply assuming that all heap data will be allocated to the slowest memory unit available. This will not affect our allocation strategy.

3. Motivation

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’s performance by reducing the time spent on accessing data from slow memory units. This can also reduce energy by eliminating the need to fetch data from slower memory units which usually are further and require more energy per access[11]. A simple, yet naive, approach is to allocate the different data objects based on their access frequencies — allocate the more frequently accessed data objects to the faster memory units, while placing less frequently accessed data objects in slower memories. In a single-tasking non-real-time environment, this reduces to finding a strategy to allocate the application’s global variables, heap variables if a heap is used, and stack variables to the memory hierarchy.

A variable is alive from its definition to its last use [1]. Global variables are alive as long as the application is running. 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 on the slower, yet larger and cheaper, units of the memory hierarchy. For reasons discussed before, heap variables are not considered.

Stack variables (non-static automatic variables) differ from global variables because of their limited lifetimes. Like all of the other objects in the procedural stack frame4, are born (i.e. allocated) upon procedure entry and die (i.e. freed) upon exit and thus have limited lifetimes. Any two stack variables with disjoint lifetimes can share the same (fast) memory space. This enables substantial space and performance improvements over global variables since multiple procedural stack frames can share the fast memory units.

A multi-tasking real-time environment has additional characteristics and restrictions. First, real-time tasks have timing constraints that must be met. The particular data allocation strategy used affects the real-time properties of the tasks, and in turn, the scheduling characteristics of the system. The scheduling policy affects the system’s preemption relations, which should be exploited to optimize the data allocation method used. Hence, in a real-time environment, the data allocation phase and the task scheduling phase should be tightly coupled to avoid losing optimization opportunities. Second, real-time systems (especially safety-critical ones) need guarantees on their worst-case execution rather than their average-case. Most proposed data allocation techniques allocate data based upon average-case profiles. An optimal allocation for the average-case may not necessarily be the optimal allocation for the worst-case. These differences (between single-tasking systems in a non-real-time environment, and multi-tasking systems in a real-time one) call for a unified data allocation/real-time scheduling technique to take advantage of synergy.

4. 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 $T_k^w$ 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 $U_k = (T_k^r, T_k^w, L_k, W_k)$ 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 condition. The memory space where the global variable $g_i \in \mathcal{G}$ will reside at run-time, which we will denote by $x_i^G$, has to be allocated for the entire operational time of the system (global variables are always alive). The memory space associated with each real-time task $T_m \in \mathcal{T}$ where the task saves its procedural stack frames is called the task’s stack, which we denote by $x_m^T$, need not have this property. With these definitions, each global variable and each real-time task is treated 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 accessed much more frequently than others. Similarly, particular nested procedures in a real-time task might be called much more frequently than others.

---

4Stack frames, also called activation records, may contain automatic variables, parameters, return value space, return address, dynamic link, and space for temporary local storage.

5In 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
Figure 4.2. Data objects can be either allocated to a single memory unit or distributed between multiple memory units. This abstract data object can be a global variable, a procedural stack frame, or a task stack.

To remove this obstacle to optimization, we must consider finer levels of objects granularity. For example, a global variable can either be allocated to a single memory unit, or distributed between multiple memory units (Figure 2(a)). A procedural stack frame can either be allocated as a single entity, or distributed between multiple memory units (Figure 2(b)). A task stack can either be allocated as a single entity, or distributed between multiple memory units (Figure 2(c)), and so on. We emphasize that finer granularity levels may incur more run-time and code size overheads. For example, if a procedure is allocated to two different memory units, two stack pointers must 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 (i.e. nested procedures are not allocated to the same memory unit) between multiple memory units, the same argument holds, and so on. Nevertheless, it will be shown later that our method does not incur significant overheads because it is usually possible to obtain sufficient performance improvement at intermediate granularity levels when data allocation is combined with preemption limiting (i.e. in real-time scheduling).

To exploit these different granularity levels, we modify the above definitions. We assume that the memory space $x^G_i$ 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 $x^P_{ij}$ for $j \in [1, N_P(i)]$. We also assume that the memory space for each task stack $x^T_m$ 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 $x^{F}_{mn}$ for $n \in [1, N_F(m)]$. At the finest level of granularity, we assume that the memory space of each procedural stack frame $x^{F}_{mn}$ 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 $x^{V}_{mno}$ 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 appears in the generic memory map of Figure 4.3.

Procedural stack frames (and in turn procedural stack variables) are allocated on procedure entry and deallocated on its exit. Some 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 4.4. Two procedures have disjoint lifetimes within a task if there is no root-to-leaf-node path containing both procedures. In a similar sense, two tasks have disjoint lifetimes, and their stacks can be allocated to the same memory space, if they are mutually

16-bit word, a 32-bit word, etc.
non-preemptive. Hence, allocating task stack frames permanently by allocating dedicated space for each is not optimal. To perform a lifetime analysis at the task level, we use a preemption graph for a workload, consisting of a directed acyclic graph with the essential property that an edge exists from task A to task B if and only if task B can preempt task A. A preemption graph for a generic fully-preemptive workload with three tasks is shown in Figure 4.5; the scheduler is represented as $T_0$.

In this study we do not use profile data to determine the number of memory accesses to 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. This is handled implicitly, however, by the inner loop of Figure 1.1. 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. Finally, we let the number of memory words (i.e. size) of data object $x^O$ be denoted by $S(x^O)$.

Since real-time scheduling will be performed, we must specify the real-time properties of the task set. Tasks can be periodic or sporadic. If task $T_m$ 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 $\pi_n > \gamma_m$. In our previous work we showed that given a particular priority assignment (i.e. assigned using some scheduling algorithm like rate monotonic (RM) algorithm, deadline monotonic (DM) algorithm, etc.), a heuristic known as the maximal preemption threshold assignment algorithm (MPTAA) will always find the optimal preemption threshold assignment that results in the smallest stack size requirements. The MPTAA algorithm of Algorithm 1 is used in our framework to assign preemption thresholds and control the preemption relations of our real-time task set.

Algorithm 1 $\text{MPTAA}(T, \Pi)$

1: $\Gamma = \Gamma'$ /* 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_scheduled($T_i$);
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 $\Gamma$;

Finally, we assume that other compiler optimizations have been already performed on the application’s variables. In particular, we assume that the variable/argument promotion and register allocation has already been performed. Given the embedded application code, our goal is to determine the mapping of each

\(^6\)In our experiments we assume that loop bounds are known and there is no recursion. There are other static timing analysis techniques which could be used to address these constraints and tighten WCET bounds.
variable not bound to a processor register to the different memory units while maximizing the application’s overall memory access performance.

5. Problem Formulation

Given a real-time workload \( W = (T, G) \) 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. We define for each data object \( x^G \) a set of \( N_U \) decision variables \( I_n(x^G) \) for \( n \in [1, N_U] \), such that \( I_n(x^G) \) is non-zero if and only if \( x^G \) is allocated to memory unit \( U_n \) and is zero otherwise. That is, for each data object \( x^G \) we define the following set of decision variables for \( n \in [1, N_U] \):

\[
I_n(x^G) = \begin{cases} 
1 & \text{if object } x^G \text{ allocated to unit } U_n \\
0 & \text{otherwise}
\end{cases}
\]

(5.1)

Although there are various levels of granularity that can be considered, we choose here to formulate the problem only for the two boundary cases for brevity. We present the problem formulation for the coarsest and the finest granularity levels only, but the technique can be extended to apply to intermediate granularities.

5.1. Allocation Granularity = coarsest

At this level of granularity, each global variable \( x_i^G \) and the stack of each task \( x_m^T \), is to be allocated to a contiguous memory space without being distributed between multiple memory units. No additional runtime or code overhead is incurred. We represent the memory access cycles of each task \( T_m \) for \( m \in [1, N_T] \) along its worst-case execution path as:

\[
\sum_{n=1}^{N_U} \sum_{k=1}^{N_G} I_n(x^G) \left[ T_m^T N^T(x^G_j) + T_m^w N^w(x^G_j) \right] S(x^G_j)
\]

\[
+ \sum_{n=1}^{N_U} I_n(x^T_m) \left[ T_m^T N^T(x^T_m) + T_m^w N^w(x^T_m) \right] S(x^T_m)
\]

(5.2)

The above expression represents the memory access cycle count of each task along its worst-case execution path. This path might change after the data allocation phase, but this is implicitly accounted for by the iterative nature of our method. 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 \( H \) (the least common multiple of the tasks’ periods or interarrival times). 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:

\[
\sum_{m=1}^{N_T} k_m \times MAC(T_m)
\]

(5.3)

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:

\[
\sum_{k=1}^{N_G} I_k(x^G_i) = 1 \quad (\forall i \in [1, N_G])
\]

(5.4a)

\[
\sum_{k=1}^{N_U} I_k(x^T_m) = 1 \quad (\forall m \in [1, N_T])
\]

(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 task stacks. Nevertheless, task stacks need not be allocated permanently 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 cannot share the same memory space since a preemption can occur. Let \( L \) be the set of all leaf node tasks in the system’s preemption graph. Let \( P_q(T_l) \) be the \( q^{th} \) unique path from the root node to the leaf node \( T_l \in L \), and let there be \( NP(T_l) \) such paths. Then the following constraint can be derived:

\[
\sum_{i=1}^{NP(T_l)} I_n(x^G_i) S(x^G_i) + \sum_{T_m \in P_q(T_l)} I_n(x^T_m) S(x^T_m) \leq L_k W_k
\]

(5.4c)

\((\forall k \in [1, N_U], \forall T_l \in L, \forall q \in [1, NP(T_l)])\)

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 stacks. Nevertheless, at this level of granularity it is hard to obtain significant performance improvement.
5.2. Allocation Granularity = finest

The finest granularity level incurs the most run-time and code overheads. Here, the memory access cycles count of any task during any of its invocations on its worst-case path can be represented as:

\[
N_{L} \sum_{k=1}^{N_{L}} \sum_{i=1}^{N_{P}(i)} I_k(x^p_{ij}) [T_k^p N_m(x^p_{ij}) + T_k^w N_m(x^p_{ij})] S(x^p_{ij})
\]

As with the previous level, there are three constraints that need be added. First, we force the ILP solver not to allocate any data object to any memory unit twice by using the following two constraints:

\[
N_{L} \sum_{i=1}^{N_{P}(i)} I_k(x^p_{ij}) = 1 \quad (\forall j \in [1, N_P(i)], \forall i \in [1, N_G]) \quad (5.6a)
\]

\[
\sum_{k=1}^{N_{L}} I_k(x^V_{mno}) = 1 \quad (\forall o \in [1, N_V(m, n)], \forall n \in [1, N_P(m)]
\]

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

\[
N_{L} \sum_{i=1}^{N_{P}(i)} I_k(x^p_{ij}) S(x^p_{ij}) + \sum_{T_m \in P_q(T_i)} \sum_{x^p_{ij} \in F_{ij}} \sum_{k=1}^{N_{V}(T_i)} N_{V}(T_i, \forall q \in [1, N_P(T_i)]
\]

\[
(\forall k \in [1, N_V], \forall T_i \in L, \forall q \in [1, N_P(T_i)]
\]

5.3. 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. The MPTAA shown earlier in Algorithm 1 will be used to find the optimal preemption threshold assignment resulting in the minimal total stack space usage as was explained earlier.

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 1.1 is repeated until a fixed point is reached, resulting in the final data allocation and real-time schedule.

5.4. Algorithm Convergence

Following the above steps our data allocation/real-time scheduling algorithm proceeds as shown in Figure 1.1. The convergence of this algorithm to an optimal solution is dependent on the convergence of the inner loop, followed by the outer one. As was explained earlier, the inner loop exists because changing the allocation might change the worst-case path, and hence, need to be recomputed. This inner loop will always converge since the number of possible locations in memory are finite as well as the number of data objects. In our experiments, the convergence of the inner loop was in fact very fast. The outer loop, on the other hand, is dependent on the MPTAA which will always converge to an
optimal preemption threshold assignment as was shown in [6]. Hence, our data allocation/real-time scheduling algorithm will indeed always converge to an optimal allocation and real-time schedule.

6. 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 with simulations for synthetic workloads, followed by those for the Fly-By-Wire workload from the Autopilot and PapaBench benchmarks[14].

6.1. Synthetic Workloads

Matlab[13] 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 and different real-time scheduling schemes. The memory access cycles for each experiment was normalized to the all_sram case (in which SRAM is large enough to hold all data). The average of these normalized memory access cycles for all workloads is presented in Figure 5.6. The SRAM size for this experiment is fixed at 256 words, which is about 20% of the total data memory required by each workload. The all_dram case shows the worst performance; all data objects are now allocated to the slow DRAM. The linear 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 linear 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 case. 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 tasks preemptions as can be seen in the distTstack + uniPstack + pts case. In fact, the

<table>
<thead>
<tr>
<th>Allocation Granularity</th>
<th>Best-Case</th>
<th>Worst-Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>uniTstack + uniPstack</td>
<td>0.12%</td>
<td>0.12%</td>
</tr>
<tr>
<td>distTstack + uniPstack</td>
<td>0.12%</td>
<td>0.62%</td>
</tr>
<tr>
<td>distTstack + distPstack</td>
<td>0.12%</td>
<td>3.33%</td>
</tr>
</tbody>
</table>
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 \( \text{dist}T\text{stack} + \text{dist}P\text{stack} \) 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 \( \text{dist}T\text{stack} + \text{dist}P\text{stack} + \text{pts} \) 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 allSRAM. In this case the average memory access cycles was only 11.2% more than the ideal allSRAM 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 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 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 tasks, this will lead to the highest number of overhead cycles. As can be seen from the Figure, 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 8(a), 8(b), and 8(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 8(b) and 8(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.

### 6.2. 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 2. The experimental platform used is depicted in Figure 6.7. In this framework, 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. OMMA (the Optimal Multitasking Memory Allocator) and the PTS scheduler are both implemented using Matlab\[13\]. Once the optimal allocation and schedule are found, the resulting run-time characteristics are calculated. As in the experiment performed in the previous sec-

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>receive_radio_task</td>
<td>40Hz</td>
</tr>
<tr>
<td>T2</td>
<td>send_data_to_autopilot_task</td>
<td>40Hz</td>
</tr>
<tr>
<td>T3</td>
<td>check_autopilot_values_task</td>
<td>20Hz</td>
</tr>
<tr>
<td>T4</td>
<td>servo_transmit_task</td>
<td>20Hz</td>
</tr>
<tr>
<td>T5</td>
<td>check_failsafe_task</td>
<td>20Hz</td>
</tr>
<tr>
<td>I1</td>
<td>servo_interrupt</td>
<td>-</td>
</tr>
<tr>
<td>I2</td>
<td>spi_interrupt</td>
<td>-</td>
</tr>
<tr>
<td>I3</td>
<td>radio_interrupt</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2. The Fly-By-Wire tasks.
Figure 6.8. Average normalized execution time of different allocation granularities with and without real-time preemption threshold scheduling. PTS benefits from finer granularities.

Figure 6.9. Normalized memory access cycles and execution cycles for the Fly-By-Wire benchmark with allocation at different granularity levels. On-chip SRAM latency is 1-cycle; off-chip DRAM is 10 cycles.

7. Conclusions

In this paper we present a novel, unified data allocation/real-time scheduling technique which improves run-time performance for real-time multitasking 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 to be most flexible. We find that our technique 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.

References

