Dynamic Shared SPM Reuse for Real-Time Multicore Embedded Systems

MORTEZA MOHAJJEL KAFSHDOOZ and ALIREZA EJLALI, Sharif University of Technology

Allocating the scratchpad memory (SPM) space to tasks is a challenging problem in real-time multicore embedded systems that use shared SPM. Proper SPM space allocation is important, as it considerably influences the application worst-case execution time (WCET), which is of great importance in real-time applications. To address this problem, in this article we present a dynamic SPM reuse scheme, where SPM space can be reused by other tasks during runtime without requiring any static SPM partitioning. Although the proposed scheme is applied dynamically at runtime, the required decision making is fairly complex and hence cannot be performed at runtime. We have developed techniques to perform the decision making offline at design time in the form of optimization problems combined with task scheduling/mapping. The proposed work is unlike previous works that either exploit static schemes for SPM space allocation or perform task scheduling/mapping and SPM space allocation incoherently. The experimental results show that our dynamic SPM reuse scheme can reduce WCET by up to 55% as compared to recent previous works on SPM allocation in real-time multicore embedded systems.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: Real-Time and Embedded Systems; D.4.2 [**Operating Systems**]: Storage Management

General Terms: Design, Performance

Additional Key Words and Phrases: Scratchpad memory, multicore processors, shared memory, scheduling, embedded real-time systems

ACM Reference Format:
Morteza Mohajjel Kafshdooz and Alireza Ejlali. 2015. Dynamic shared SPM reuse for real-time multicore embedded systems. ACM Trans. Architec. Code Optim. 12, 2, Article 12 (May 2015), 25 pages.
DOI: http://dx.doi.org/10.1145/2738051

1. INTRODUCTION

Multicore processors are increasingly used in the design of real-time embedded systems [Salamy and Ramanujam 2012; Guo et al. 2013]. Whereas some multicore processors use cache memory to reduce average execution time, such as in real-time applications in which the worst-case execution time (WCET) [Kopetz 2011] usually is more important, many real-time embedded systems use scratchpad memory (SPM) [Banakar et al. 2002] instead of cache [Guo et al. 2013]. Unlike cache, SPM is a software-controlled memory, and its content is visible to software [Suhendra et al. 2010]. Therefore, in contrast to cache, a software programmer (or compiler) can fully control the content of SPM to achieve the predictability required for analyzing and reducing WCET [Suhendra et al. 2005]. However, the cost of this predictability is a burden on the software to manage SPM. SPM management (allocating SPM space to system tasks) can be especially more

---

This article is not an extension of a conference paper.
Authors' addresses: M. M. Kafshdooz and A. Ejlali, Department of Computer Engineering, Sharif University of Technology, Azadi Ave., Tehran 1458889694, Iran; emails: mohajjel@ce.sharif.edu, ejlali@sharif.edu.

DOI: http://dx.doi.org/10.1145/2738051

complex when SPM is shared among the cores. Indeed, as resource sharing is one of the remarkable features of multicore processors [Zhuravlev et al. 2012], all or part of on-chip memory (cache or SPM) is usually shared among the cores.

Previous research works on shared SPM allocation in real-time multicore embedded systems either (1) provide static SPM space allocation techniques where the SPM space is statically partitioned among the tasks or cores, which means that SPM space allocated to each task or core remains unchanged during runtime (e.g., Salamy and Ramanujam [2012], Suhendra et al. [2006], and Chang et al. [2012]) or (2) perform task scheduling/mapping and SPM space allocation incoherently (e.g., Kandemir et al. [2004]; Suhendra et al. [2010]). Static allocation techniques can lead to waste of SPM space, thereby increasing WCET. When a part of SPM space is statically allocated to a task, it cannot be used by other tasks even when the task does not require its allocated SPM (e.g., like the works in Chang et al. [2012, 2014]). In addition, when SPM is statically partitioned among the cores, each task can only access to its host core SPM even if other cores do not utilize their SPM (e.g., like the work in Salamy and Ramanujam [2012]). With respect to the coherence between SPM space allocation and task scheduling/mapping, it should be noted that for efficient SPM management, we need to know which tasks are executed on different cores in parallel so that we can allocate proper SPM space to the tasks. This implies that for proper SPM space allocation, we need to have a task scheduling/mapping available—that is, we need to know when and where (on each core) each task must be executed. On the other hand, the information of task execution times, which is essential for scheduling/mapping, becomes available after SPM space allocation. These issues imply that it is more efficient to conduct SPM management and task scheduling/mapping coherently.

In this article, we propose a dynamic SPM reuse scheme for real-time embedded multicore systems, where the SPM is not statically partitioned among the cores, and whenever a task finishes, its allocated SPM space is reused by other SPM requiring tasks. In addition, in our proposed scheme, SPM allocation is combined with task scheduling/mapping to achieve higher efficiency of the system.

Although our proposed scheme is dynamic and applied at runtime,the required decision making is quite complex and consequently cannot be performed at runtime. Thus, we have developed techniques to perform the decision making offline at design time in the form of an optimization problem.

To address the optimization problem, we have developed two techniques: (1) an integer linear programming (ILP)-based technique, which finds optimal solutions for fairly small systems (small number of tasks and cores), and (2) a heuristic technique, which finds near-optimal solutions in reasonable time for large systems (large number of tasks and cores). To evaluate our techniques, we conducted several experiments using various benchmarks from the Mälardalen [Gustafsson et al. 2010] and MiBench [Guthaus et al. 2001] benchmark suites with different system configurations. The results show that our proposed techniques can reduce total WCET by up to 54% as compared to the recent previous techniques, which statically partition SPM among the cores, and by up to 55% as compared to static SPM allocation (SSA) to the tasks. Moreover, the results show that our proposed heuristic method is almost as effective as the optimal method, and in the worst case its WCETs are only 8% higher than the optimal WCET.

The rest of this article is organized as follows. In Section 2, we review related works. In Section 3, we describe our assumed application and architectural model. In Section 4, we present the proposed scheme. We describe our proposed techniques to solve the optimization problem in Sections 5 and 6. We explain experimental setup and obtained results in Section 7, and we conclude the article in Section 8.

2. RELATED WORKS

SPM management has been considered as a hot topic in real-time embedded systems. Although some research works have addressed the use of SPM in single-core real-time systems, relatively fewer works have considered the use of SPM in multicore real-time systems. Considering that in this article we aim to focus on real-time multicore embedded systems, in this section we review related previous works on SPM management for real-time multicore embedded systems (for more information about the use of SPM in single-core systems, please refer to Suhendra et al. [2010]).

On-chip memory (cache or SPM) organization and partitioning for data-intensive applications is studied in Kandemir et al. [2004] and Ozturk et al. [2005, 2006a]. Kandemir et al. [2004] proposed an approach to dynamically change the organization of shared on-chip memory to adapt to runtime variations in memory requirements and data sharing patterns. Ozturk et al. [2005] partitioned on-chip memory into several private and shared blocks and assigned these blocks to processor cores by capturing the private and the shared accesses of the cores to data. Moreover, they proposed an integrated data allocation and on-chip memory partitioning based on ILP in Ozturk et al. [2006a]. Although these works considered multicore processors with shared SPM, they differ from our work in at least in two main aspects. First, whereas we focus on on-chip memory (SPM) content management, they considered on-chip memory design and configuration. Second, in contrast to our scheme, they assume that task (program region) scheduling and mapping is known a priori.

SPM management for real-time multicore embedded systems with a priori known scheduling or mapping of tasks is studied in some other previous works. Suhendra et al. [2010] proposed an iterative SPM allocation algorithm to reduce the worst-case response time (WCRT) of concurrent applications. In their approach, SPM space is partitioned among concurrent tasks, and these tasks cannot exploit a common space of SPM. However, nonconcurrent tasks can exploit the common spaces of SPM. Although they considered a multicore system, in contrast to our scheme, they assume that SPM space is not shared among the cores and task mapping is known priori. With respect to the waiting time for accessing shared bus, Chattopadhyay and Roychoudhury [2011] proposed a bus delay aware iterative SPM allocation scheme to reduce the WCRT of an application running on a multicore processor with a virtually shared SPM (VS-SPM) [Kandemir and Choudhary 2002]. However, like Suhendra et al. [2010], they assume that task mapping to cores is known a priori. Guo et al. [2011, 2013] proposed dynamic programming approaches to place data in VS-SPM to reduce the total time of memory accesses and energy consumption. However, they assume that applications are partitioned a priori into parallel regions and for each region the mapping of threads to cores are known. Kandemir and Choudhary [2002] proposed an algorithm that targets the reduction of off-chip memory accesses caused by interprocessor communication for regular array-intensive applications. As in all of the previously mentioned works, they also assumed that the mapping of computations to cores has been performed a priori. Shared SPM management for concurrent array/loop-intensive applications is studied by Ozturk et al. [2006b]. In their approach, shared SPM space is partitioned among concurrent applications based on their SPM requirements. However, in contrast to our scheme, they determine the amount of SPM space for each application at runtime, which imposes timing overhead to the system. Moreover, in their scheme, application scheduling is performed independently from SPM management.

Some other works have considered the coherency of task scheduling/mapping and SPM management. For preemptive real-time systems, Wan et al. [2013] proposed a method based on a novel data structure (preemption graph [Wan et al. 2013]). In Ghattas et al. [2007] and Kang and Dean [2010], considering real-time constraints

to be an iterative method is proposed to reduce stack memory requirements of tasks. Although these works considered the coherency of SPM allocation and task scheduling, in contrast to our proposed work, they have not considered multicore systems.

For implicit deadline sporadic tasks, Chang et al. [2012, 2013, 2014] proposed approximation algorithms to perform task mapping and SPM allocation coherently. Their objective in Chang et al. [2013, 2014] is the reduction of the number of used islands in a island-based multicore processor, and the objective in Chang et al. [2012] is the reduction of processor core utilization in a multicore processor with a shared SPM. However, in contrast to our scheme, they assume that each task gets a private SPM space and that other tasks cannot use this space even when the task is not running.

Suhendra et al. [2006] proposed an optimal algorithm based on ILP to reduce the WCRT or the initial interval of an application running on a multicore processor with a VS-SPM. However, in contrast to our scheme, they do not reuse SPM space of completed tasks for other tasks. Zhang et al. [2010] proposed two heuristic algorithms to jointly schedule/map tasks and to partition the variables of an application running on a VS-SPM-based multicore processor. Gu et al. [2013, 2014] considered joint variable partitioning and task scheduling for a multibank [Gu et al. 2013] and multiport [Gu et al. 2014] VS-SPM-based multicore processor and proposed ILP and heuristic methods. Like [Suhendra et al. 2006], the last two mentioned works did not consider SPM space reuse to further reduce execution time.

Salamy and Ramanujam [2012] proposed a heuristic algorithm to jointly partition shared SPM between processors and task scheduling/mapping in MPSoCs to reduce the total WCET of an application. Liu et al. [2011] considered cache instead of SPM. However, their approach is applicable to multicore processors with shared SPM. They proposed some heuristic to jointly assign tasks and cache partitioning with cache locking to reduce the total WCET of an application. However, in these works, shared on-chip memory (SPM) space is statically partitioned among cores, and hence each task at most can access to the space allocated to its host core even if other cores have unused spaces.

3. SYSTEM MODEL

The architectural model that we consider in this article is a homogeneous multicore processor with three levels of memory hierarchy:

(1) Local private SPMs that are tightly coupled to cores and each core can only access to its own local SPM
(2) A global shared SPM that can be accessed by each core via a separate port
(3) A main memory that can be accessed by all of the cores via a shared bus.

It should be noted that our proposed scheme in this article is also applicable to a one-level shared SPM. However, for many applications where high performance is required, a two-level SPM is preferable to a one-level SPM [Kandemir and Choudhary 2002; Liu and Zhang 2012]. Therefore, in this work, we consider a two-level SPM. It is noteworthy that a two-level (multilevel) SPM is used in some commercial processors, such as TMS320C6472 [Texas Instruments 2011]. Figure 1 depicts the considered architecture for a quad-core processor.

To avoid consistency problems in SPMs, we assume that at most one copy of each memory object (by *memory object*, we mean a program code or a data structure stored in system memory) can exist in SPMs. Therefore, as we do not have multiple copies of memory objects in SPMs, we do not have any consistency problem in the considered architecture. To manage the content of SPMs by the programmer or compiler, we assume that each SPM (shared or private) has its own separate nonoverlapping address space and that these address spaces are known at design time. Therefore, the compiler or programmer can directly determine the memory unit (SPM unit or main memory)

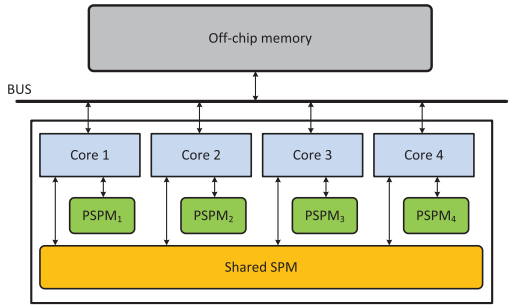

Fig. 1. Multicore processor with a two-level SPM.

where each memory object must be located. This can easily be performed by mapping each memory object to the address space of the desired memory unit. It should be noted that in our proposed scheme, the contents of private and shared SPMs are decided offline at design time, and during runtime no data interchange occurs between the private and shared SPMs. Like all the other offline techniques, this technique does not provide optimal performance, as offline techniques cannot provide best results if unexpected runtime variations occur. Nevertheless, we believe that for SPM space allocation, such an offline technique is preferable, as they do not impose time and energy overhead, which are very important for real-time embedded systems that have severe time and energy limitations.

In the considered architecture, it is assumed that each core incurs very low latency access to its own local SPM [Marwedel 2006] and slightly higher latency access to the global SPM. However, the access latency to the off-chip memory is high. It should be noted that it is hard to estimate the time overhead caused by bus conflicts among cores in access to off-chip memory. Therefore, in our analysis, we have assumed the worst-case scenario—that is, to guarantee a safe upper bound for execution time, we have assumed that each access to the off-chip memory incurs a constant (worst-case) latency.

The application model that we consider in this work is frame based, where the application consists of a set of N independent tasks, T_1, $T_2, \ldots, T_N$, with the same release time and deadline [Li and Wu 2014]. One example of such an application is systems that use timeline scheduling [Buttazzo 2011]. Timeline scheduling is used for hard real-time systems where the time axis is divided into equal time slots (frames) that are executed periodically and each time slot can contain several tasks based on the tasks periods [Buttazzo 2011]. For example, the control system described in Kim et al. [1996] uses a timeline scheduling scheme where up to seven tasks can be executed in each frame. Another example of real-time systems with a frame-based model is those real-time systems that use pipelined scheduling [Chatha and Vemuri 2002]. In these systems, during each initiation interval [Suhendra et al. 2006] of the pipeline, a set of independent stage tasks are executed (it is noteworthy that the concept of pipelining is based on this fact, considering that as various pipelining stages are operating on different and independent data units, tasks are independent from each other during each initiation stage). In each initiation interval, we have a set of stage tasks that must be executed so that the beginning and end of the initiation interval can be considered as the tasks' common release time and common deadline, respectively. Therefore, when we use pipelined scheduling, we indeed have a frame-based application where each initiation interval comprises a frame. Pipelined scheduling is used in many real-time systems, especially streaming applications [Suhendra et al. 2006]. In pipelined scheduling, the number of tasks in each frame depends on how the designer has partitioned

the application into pipelining stages. It is noteworthy that the number of tasks within each frame is not necessarily equal to the number of processing cores. For example, in Suhendra et al. [2006], a pipelined scheduling has been used for an image enhancement application where six tasks are executed during each frame while there are four processing cores. We have provided these examples (the use of timeline scheduling and pipeline scheduling in real-world applications) to illustrate the prominence of the frame-based model, and we believe that this is why many other researchers have also considered similar frame-based application models (with common release time and deadline for multiple tasks) in their research works (e.g., Gerards and Kuper [2013], Li and Wu [2014], Rusu et al. [2003], and Devadas and Aydin [2012]).

In the considered application model, each task is executed nonpreemptively on one of the processor cores. Each task can be allocated in some space of the SPM (global or local) to speed up its execution. The achieved speedup for each task, and hence its WCET, depends on the amount of SPM space allocated to it and the way in which it uses this space. Therefore, SPM management can be divided into two phases: (1) intratask SPM management, which determines how each task uses its allocated SPM space, and (2) intertask SPM management, which determines the amount of allocated SPM space for each task. It should be noted that considering our focus in this article is on intertask SPM management, we assume that for different sizes of allocated SPM space, intratask SPM management and WCET analysis are performed a priori.

One important point is that at first glance, it seems that since intratask SPM space allocation is performed before the intertask one, intratask SPM space allocation may have this negative influence that can freely consume a large amount of SPM, thereby leaving less SPM space for the intertask one. However, such an unbalanced scenario does not happen, because although the intratask SPM allocation is performed a priori, during the intertask SPM allocation we do not use a fixed (and unchangeable) result from intratask SPM allocation. Indeed, at first in intratask SPM allocation, we produce a set of results (options) containing numerous intratask SPM allocations for various sizes of allocated SPM space, then during the intertask SPM space allocation, we can choose proper intratask SPM space allocation from the available set.

It should be noted that wherea in some literature the terms *static* and *offline* (and also the terms *dynamic* and *online*) are used interchangeably, in this work these terms have quite different meanings. Throughout this article, the term *static* refers to "static SPM allocation," which means that the SPM space allocated to tasks and cores remains unchanged during runtime. In addition, the term *dynamic* refers to "dynamic SPM allocation," which means that the SPM space allocated to tasks and cores can vary during runtime. On the other hand, the term *offline* refers to "offline decision making," which means that the decisions (with respect to task scheduling or SPM allocation) are made offline at design time. Additionally, the term *online* refers to "online decision making," which means that such decisions are made online at runtime. Therefore, for example, it is possible to have offline but dynamic SPM allocation, which means that although the decisions about the SPM allocation are made offline at design time, they are applied at runtime to dynamically vary the SPM space allocated to tasks and cores [Udayakumaran et al. 2006; Zhuge et al. 2012; Falk and Kleinsorge 2009].

4. PROPOSED SCHEME

In this section, we explain our proposed dynamic SPM space reuse scheme, which is used for real-time multicore embedded systems to reduce WCET. Indeed, the proposed scheme can be defined by the following identifying features:

(1) Whenever a task starts running on the system, an amount of space from the shared SPM is exclusively allocated to it. Once the task finishes, its SPM space is released

so that it can be allocated to the other task. In Section 4.1, by means of a motivating example, we explain how such a strategy is superior to strategies used in previous related works.

(2) To determine the amount of SPM space that is allocated to each task, we use offline optimization techniques. We adopt offline techniques, as they allow having computationally expensive optimizations without incurring runtime overhead. It is noteworthy that online techniques also have their advantages—for example, they allow consideratio of runtime data, as there are data that are not available at design time. Nevertheless, online techniques cannot perform complex computation, as it incurs very high overhead. In our proposed scheme, as we want to exploit sophisticated optimizations, we have adopted offline techniques.

(3) In our proposed scheme, SPM space allocation and task scheduling/mapping are not separated and are performed jointly to achieve the required coherency between SPM space allocation and task scheduling/mapping. Indeed, our approach is that both the SPM space allocation problem and task scheduling/mapping problem are formulated into one same optimization problem (discussed in Item (2)). Therefore, once we solve the optimization problem, not only do we determine how SPM space must be allocated to the tasks but also we achieve a task scheduling/mapping. It should be noted that based on the discussions provided by previous works (e.g., Liu et al. [2011] and Salamy and Ramanujam [2012]), it can be concluded that this optimization problem is NP-complete and hence cannot be solved in polynomial time, at least with existing algorithms.

4.1. Motivating Example

In this section, we show by a simple motivating example that in our proposed scheme, more efficient solutions than in similar previous schemes can be achieved. To this end, we consider the following schemes:

—*Static SPM allocation (SSA) scheme*: This is based on the schemes presented in Suhendra et al. [2006] and Chang et al. [2012]. In this scheme, SPM space is statically partitioned among tasks.

—*Static SPM partitioning among cores (SSP) scheme*: This based on the schemes presented in Salamy and Ramanujam [2012] and Liu et al. [2011]. In this scheme, SPM space is statically partitioned among cores instead of tasks, and allocated SPM space to a task on a core can be reallocated only to other tasks on the same core.

—*Proposed dynamic SPM reuse scheme*: This is our proposed scheme in this article. In our scheme, SPM space allocated to a task can freely be reallocated to next tasks even on other cores.

We run SSA, SSP, and the proposed scheme on a system consisting of five tasks, two cores, 4KB shared (L2) SPM, and 128B private (L1) SPM per core. The WCET of the tasks per allocated L2 SPM size are depicted in Figure 2. The tasks are selected from the benchmarks explained in Section 7.1.

The obtained results for SSA, SSP, and the proposed scheme are depicted in Figures 3, 4, and 5, respectively. It should be noted that these results are optimal, as they have been obtained by an ILP approach (Section 5) with the objective of minimizing total WCET, which is our main goal in this article. As shown in these figures, the total WCETs of the example system using SSA, SSP, and the proposed scheme are 2.655, 1.888, and 1.738 million cycles, respectively. Therefore, in this example, the proposed scheme results in about 35% less WCET than SSA and about 8% less WCET than SSP. The reasons why the proposed scheme outperforms SSA and SSP are as follows.

In the SSA scheme, SPM space is wasted, since when a task is not running, its allocated SPM space cannot be reused by other tasks. Therefore, in systems with

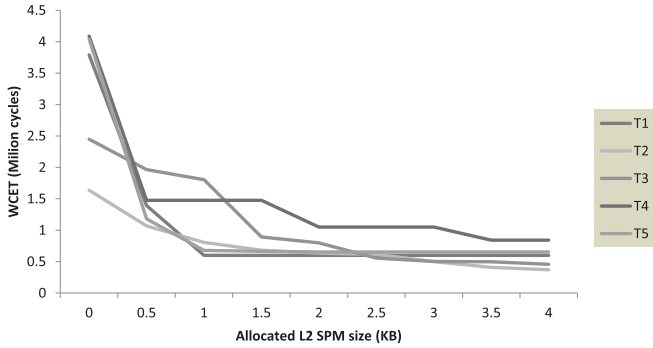

Fig. 2. WCET of the tasks related to the allocated L2 SPM size.

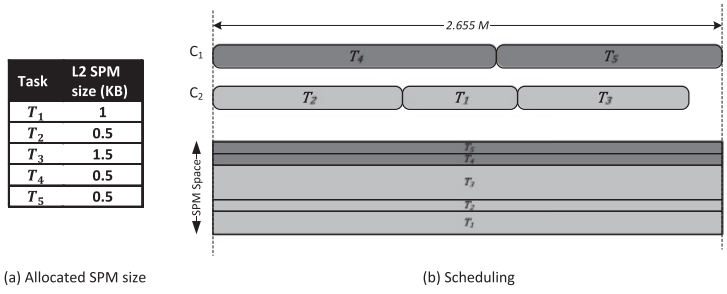

Fig. 3. The result of applying an SSA scheme for the example system.

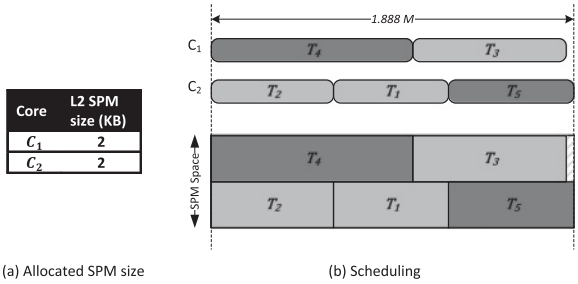

Fig. 4. The result of applying the SSP scheme for the example system.

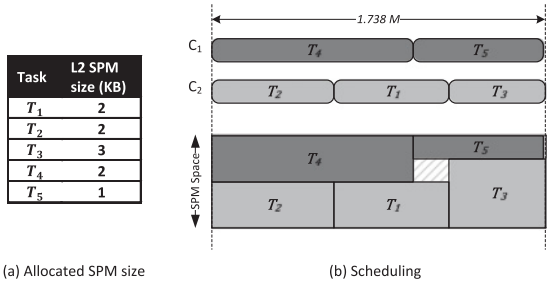

Fig. 5. The result of applying our proposed dynamic SPM reuse scheme for the example system.

numerous tasks or with a small SPM size related to application size, the use of an SSA scheme leads to allocating very small SPM portions to tasks, which can considerably increase the WCET. For the example system, as shown in Figure 3, when SSA is used, tasks T_2, T_4, T_5 are allocated 0.5KB of SPM, which is much less than the allocated SPM to these tasks when SSP or the proposed scheme is used.

In the SSP scheme, when a task finishes, its SPM space can be released for the next tasks, and therefore SPM space is used more efficiently as compared to when SSA is used. For the example system, as shown in Figure 4, when SSP is used, 2KB of SPM space is allocated to each task, which is more than the allocated SPM space when SSA is used, and hence the WCET of the system is reduced. However, the proposed scheme achieves even more WCET reduction, because it can use SPM space reuse even among the tasks that run on different cores. For the example system, as shown in Figure 5, when the proposed scheme is used, each of tasks T_1, T_2, and T_4 is allocated2 KB of SPM, which is equal to the allocated SPM space to these tasks when SSP is used. However, as in our scheme, SPM space allocated to a task can be reused by next tasks even on other cores, task T_3 can be allocated 50% more SPM space than its allocated SPM space in the SSP scheme, and hence the total WCET is further reduced. Note that in the proposed scheme, the allocated SPM space of task T_5 is less than its allocated SPM space in the SSP scheme, and hence its WCET is increased. However, as in the proposed scheme task T_3 is allocated more SPM space than in the SSP scheme and this task gets more benefit from increasing the amount of allocated SPM space for reducing its WCET than task T_5, in the proposed scheme the total WCET of the example system is reduced.

This simple example shows that our proposed scheme can lead to the further reduction of total WCET. In the next section, we formally define the optimization problem, which we solve to determine SPM allocation and task scheduling/mapping for implementing the proposed scheme.

5. OPTIMAL METHOD

We now present the ILP formulation, which we use to optimally solve the joint SPM allocation and task scheduling/mapping problem by an ILP solver such as CPLEX [ILOG Cplex 2007]. Before presenting the ILP formulation, recall that we assume an application consists of N independent tasks denoted as T_1, $T_2, \ldots, T_N$. Moreover, we assume the application is executed on a multicore processor, which contains M cores denoted as P_1, $P_2, \ldots, P_M$ and a shared SPM consisting of S blocks denoted as B_1, $B_2, \ldots, B_S$.

It should be noted that in the following formulation, we represent variables with a capital first letter and constants with all small letter words.

5.1. WCET of Tasks and Objective Function

The WCET of a task depends on the number of SPM blocks allocated to it. Therefore, we define the binary variable $Y_{i,k}$ as follows:

$$Y_{i,k} = \begin{cases} 1, & \text{if } k \text{ blocks of the SPM are allocated to the task } T_i \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

Since the number of SPM blocks allocated to a task dose not change through its runtime, for each task T_i the following equation must be satisfied:

$$\sum_{k=0}^{S} Y_{i,k} = 1. \tag{2}$$

Considering Equation (2), the WCET of the task T_i denoted as W_i can be obtained by Equation (3). The constant $w_{i,k}$ represents the WCET of the task T_i when the k blocks of the SPM are allocated to it. Recall from Section 3 that the WCET of tasks per allocated SPM size has been computed a priori.

$$W_i = \sum_{k=0}^{S} Y_{i,k} \times w_{i,k} \tag{3}$$

We represent the start time of the task T_i as $Start_i$ and its end time as End_i. Thus:

$$End_i = W_i + Start_i. \tag{4}$$

The objective function is to minimize the total WCET ($Total\_WCET$), which is the longest time, required to complete all of the tasks. Then,

$$\forall T_i : Total\_WCET \geq End_i. \tag{5}$$

5.2. Resource Usage Constraints

The tasks that use common resources (SPM blocks or processor cores) cannot be executed in parallel. To check that this constraint is satisfied, we need to specify

—the resources used by each task and
—the tasks that use at least one common resource.

To specify the resources that are used by each task, we define two sets of decision variables as follows:

$$X_{i,j} = \begin{cases} 1, & \text{if the task } T_i \text{ uses the core } P_j \\ 0, & \text{otherwise} \end{cases} \tag{6}$$

$$Q_{i,r} = \begin{cases} 1, & \text{if the task } T_i \text{ uses the SPM block } B_r \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

Each task can use (be mapped to) exactly one of the cores. This can be captured by

$$\sum_{j=1}^{M} X_{i,j} = 1. \tag{8}$$

As the binary variable $Y_{i,k}$ indicates if the task T_i uses exactly the k blocks of the SPM space, for each task T_i the following equation must be satisfied:

$$\sum_{r=1}^{S} Q_{i,r} = \sum_{k=0}^{S} k \times Y_{i,k}. \tag{9}$$

To specify the tasks that use common resources, we define the following sets of variables:

$$U_{i,i',j} = \begin{cases} 1, & \text{if the two tasks } T_i \text{ and } T_{i'} \text{ are mapped to the core } P_j \\ 0, & \text{otherwise} \end{cases} \tag{10}$$

$$U_{i,i'} = \begin{cases} 1, & \text{if the two tasks } T_i \text{ and } T_{i'} \text{ are mapped to the same core} \\ 0, & \text{otherwise} \end{cases} \tag{11}$$

$$V_{i,i',r} = \begin{cases} 1, & \text{if the two tasks } T_i \text{ and } T_{i'} \text{ use the SPM block } B_r \\ 0, & \text{otherwise.} \end{cases} \tag{12}$$

$$V_{i,i'} = \text{the number of common SPM blocks of the two tasks } T_i \text{ and } T_{i'} . \tag{13}$$

These variables can be obtained by the following equations:

$$U_{i,i'} = \sum_{j=1}^{M} U_{i,i',j} \tag{14}$$

$$V_{i,i'} = \sum_{r=1}^{S} V_{i,i',r} \tag{15}$$

$$U_{i,i',j} = X_{i,j} \times X_{i',j} \tag{16}$$

$$V_{i,i',r} = Q_{i,r} \times Q_{i',r}. \tag{17}$$

Note that as Equations (16) and (17) are not linear, they cannot be used in ILP formulation. Therefore, we use a linear equivalent of Equation (16), which is represented in Expressions (18) through (20), and a linear equivalent of Equation (17), which is represented in Expressions (21) through (23). To show the equivalence of these linear expressions with the original ones, consider for example the Equation (16) and its linear equivalent (Expressions (18) through (20)). Based on the values of $X_{i,j}$ and $X_{i',j}$, four cases are possible:

(1) ($X_{i,j} = 0$, $X_{i',j} = 0$): Expression (18) turns to $U_{i,i',j} \geq -1$, and Expressions (19) and (20) become $U_{i,i',j} \leq 0$, which finally results in $U_{i,i',j} = 0$.
(2) ($X_{i,j} = 0$, $X_{i',j} = 1$): Expression (18) turns to $U_{i,i',j} \geq 0$, Expression (19) becomes $U_{i,i',j} \leq 0$, and Expression (20) turns to $U_{i,i',j} \leq 1$, which finally results in $U_{i,i',j} = 0$.
(3) ($X_{i,j} = 1$, $X_{i',j} = 0$): Expression (18) turns to $U_{i,i',j} \geq 0$, Expression (19) becomes $U_{i,i',j} \leq 1$, and Expression (20) turns to $U_{i,i',j} \leq 0$, which finally results in $U_{i,i',j} = 0$.
(4) ($X_{i,j} = 1$, $X_{i',j} = 1$): Expression (18) turns to $U_{i,i',j} \geq 1$, Expression (19) becomes $U_{i,i',j} \leq 1$, and Expression (20) turns to $U_{i,i',j} \leq 0$, which finally results in $U_{i,i',j} = 1$.

Therefore, in all cases, the value of $U_{i,i',j}$ obtained from Expressions (18) through (20) is the same as the values obtained from Equation (16). A similar description can be provided for Equation (17).

$$U_{i,i',j} \geq X_{i,j} + X_{i',j} - 1 \tag{18}$$

$$U_{i,i',j} \leq X_{i,j} \tag{19}$$

$$U_{i,i',j} \leq X_{i',j} \tag{20}$$

$$V_{i,i',r} \geq Q_{i,r} + Q_{i',r} - 1 \tag{21}$$

$$V_{i,i',r} \leq Q_{i,r} \tag{22}$$

$$V_{i,i',r} \leq Q_{i',r} \tag{23}$$

To check that the two tasks T_i and $T_{i'}$ use at least one common resource, it is sufficient to check that the following expression is true:

$$U_{i,i'} + V_{i,i'} > 0. \tag{24}$$

Based on the preceding expression, we define a binary variable $L_{i,i'}$ to indicate whether or not the tasks T_i and $T_{i'}$ use at least one common resource. $L_{i,i'}$ can be formally defined as follows:

$$L_{i,i'} = \begin{cases} 1, & \text{if } U_{i,i'} + V_{i,i'} > 0 \\ 0, & \text{otherwise.} \end{cases} \tag{25}$$

As the preceding definition is also not linear, we use its linear equivalent, which is as follows:

$$L_{i,i'} \leq U_{i,i'} + V_{i,i'} \tag{26}$$

$$\infty \times L_{i,i'} \geq U_{i,i'} + V_{i,i'}. \tag{27}$$

We need to indicate the execution order of the two tasks T_i and $T_{i'}$, which use at least one common resource. Hence, we define the binary variable $Z_{i,i'}$ as follows:

$$Z_{i,i'} = \begin{cases} 1, & \text{if the task } T_i \text{ finishes before the start time of the task } T_{i'} \\ 0, & \text{otherwise.} \end{cases} \tag{28}$$

Based on the preceding definition, whenever $Z_{i,i'} = 1$, the inequality $End_i < Start_{i'}$ must be satisfied, and whenever $Z_{i,i'} = 0$, this inequality must be relaxed. Thus,

$$End_i < Start_{i'} + \infty \times (1 - Z_{i,i'}). \tag{29}$$

It should be noted that if the two tasks T_i and $T_{i'}$ use at least one common resource ($L_{i,i'} = 1$), then exactly one of the $Z_{i,i'}$ or $Z_{i',i}$ must be set to one. Otherwise, since there is no common resource between the two tasks T_i and $T_{i'}$ ($L_{i,i'} = 0$), both $Z_{i,i'}$ and $Z_{i',i}$ must be set to zero. This can be captured by

$$Z_{i,i'} + Z_{i',i} = L_{i,i'}. \tag{30}$$

6. HEURISTIC METHOD

In the previous section, we proposed an ILP-based technique to find an optimal solution for the joint SPM allocation and task scheduling/mapping problem. However, since this problem is NP-complete (Section 4.1), it is not possible (at least with existing algorithms) to find an optimal solution to this problem in polynomial time. Whereas this is of little importance for designing of systems consisting of a small number of tasks and cores, it can be a serious problem when designing systems with a large number of tasks and cores. Therefore, in addition to the optimal method, we proposed a heuristic method to find a near-optimal solution in a reasonable time for large systems.

Our proposed heuristic method is a genetic algorithm (GA) [Haupt and Haupt 2004]. GAs are metaheuristic search algorithms, which are successfully utilized in several optimization problems, such as task scheduling and mappings [Omara and Arafa 2010; Grajcar 1999; Kwok and Ahmad 1997]. In these nature-inspired algorithms, solutions (phenotypes) are encoded as strings or other data structures (genotypes). A GA starts with an initial pool (population) of genotypes, which is generated either randomly or by a heuristic method, and this population is evolved iteratively to find optimal or

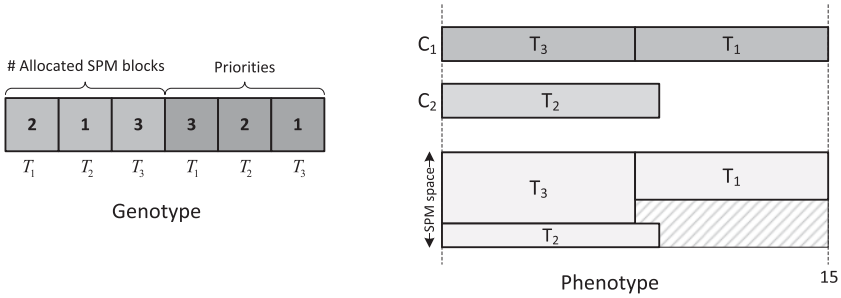

Fig. 6. A genotype and the generated phenotype by applying Algorithm 1.

near-optimal phenotypes. For this purpose, three operations, which are mimicked from nature, are performed: selection, crossover, and mutation. In the *selection* operation, some of the genotypes (usually the fittest) are chosen to be alive in the next generation and to produce new genotypes. In the *crossover* operation, two or more genotypes are taken from the selected genotypes to produce new genotypes, and finally in the *mutation* operation, one or more data of the selected genotypes are altered to maintain the diversity of the genotypes. A GA finishes when a certain stop condition is satisfied, such as when the maximum number of iterations has been exceeded. More information is provided on GAs in Haupt and Haupt [2004].

To design a GA that solves the joint SPM allocation and task scheduling/mapping problem, we should specify a way to encode each phenotype as a genotype. Since a phenotype for this problem is specified by two types of information (SPM allocation and task scheduling/mapping information), we use a two-part string to represent each genotype. In the first part of the string, the number of SPM blocks allocated to each task is specified; in the second part, similar to Ahmad and Dhodhi [1996], the priority of each task in scheduling/mapping is specified.

To generate a phenotype (SPM allocation and task scheduling/mapping) from a genotype (the number of SPM blocks allocated to each task and the priority of each task), we use a list-scheduling algorithm [Pinedo 2012], which is shown in Algorithm 1. The algorithm gets a string Str, which represents a genotype, the set T of tasks, and the system specifications (the number of processor cores and the number of SPM blocks) as input. Then, for each task t, its priority (Pri_t) and the number of SPM blocks allocated to it (SPM_t) are extracted from Str; after that, the WCET of the task t (W_t), which depends on SPM_t, is obtained (lines 1 through 5). Then, the tasks are sorted based on their priority and are stored in the sorted list ST (line 6). After that, the following works are performed iteratively. First, idle processor cores and unused SPM blocks in the current iteration are specified (lines 9 and 10). Then, each time a task is selected from the sorted list ST, if at least one idle core and the required SPM space of the task are available, then the task is scheduled/mapped and its required SPM blocks are allocated (lines 12 through 18). If all tasks have been examined or there is no idle core in the current value of $discrete\_time$, then $discrete\_time$ gets a new value for the next iteration (line 23). These steps are repeated until all tasks are scheduled. The algorithm finishes by returning the SPM allocation and tasks scheduling/mapping (phenotype), as well as the total WCET of the system (fitness function). Indeed, since our goal is to reduce the total WCET, we also use this algorithm as the fitness function and define the inverse of the total WCET as the fitness value. Figure 6 shows a genotype for a system consisting of three tasks and the phenotype that is generated by Algorithm 1 from the genotype.

ALGORITHM 1: Phenotype generation and fitness function

```
// A list-scheduling algorithm to generate phenotype (SPM allocation and task
   scheduling/mapping) from genotype (the priority and the number of SPM blocks
   allocated to each task)
Input:
—String Str // genotype
—Set T of tasks
—System specifications // # of processor cores and # of SPM blocks
Output: Phenotype and total WCET
1  foreach t ∈ T do
2  |  Pri_t ← Extract_priority(Str,t);
3  |  SPM_t ← Extract_SPM(Str,t) ;
4  |  W_t ← Get_WCET(SPM_t,t);
5  end
6  ST ← Sort(T , Pri);
7  discrete_time ← 0 ;
8  while Is_not_empty(ST) do
9  |  AP ← Find_available_procs(discrete_time);
10 |  ASPM ← Find_available_SPM(discrete_time);
11 |  foreach t ∈ ST do
12 |  |  if Is_not_empty(AP) then
13 |  |  |  if SPM_t ≤ ASPM then
14 |  |  |  |  Schedule_Map(t, W_t, discrete_time);
15 |  |  |  |  Allocate(t, SPM_t, W_t, discrete_time);
16 |  |  |  |  Update(AP, ASPM);
17 |  |  |  |  Delete(t, ST );
18 |  |  |  end
19 |  |  else
20 |  |  |  break;
21 |  |  end
22 |  end
23 |  discrete_time ← Update_time();
24 end
25 Total_WCET ← Compute_Total_WCET();
```

Our proposed GA to solve the joint SPM allocation and task scheduling/mapping problem is shown in Algorithm 2. The algorithm gets the system specifications (the number of processor cores and the number of SPM blocks) as input and produces the best-found genotype (a genotype with maximum fitness among all visited genotypes). The algorithm starts by creating a random population of genotypes and computing their fitness values by Algorithm 1 (lines 1 and 2). Then, the following works are performed iteratively. First, the genotypes are ordered nonincreasingly based on their fitness value, and a genotype with maximum fitness (*current_best*) is selected (lines 6 and 7) and checked to see if it is better than the best genotype found so far (*best*) (line 8). If so, it becomes the new best-found genotype, and n becomes zero to show that a better genotype is found in the current iteration (lines 9 and 10). Otherwise, n is incremented by one to show the number of the consecutive iterations has been elapsed and the best-found genotype is not changed (line 12). After that, a proportion (α) of the best genotypes are selected to be alive in the next iteration and the others die out (line 14). Indeed, we use elitist selection, which means that the best genotypes in the current generation are retained in the next generation [Chambers 1995]. After the selection process, crossover operations are performed on the live genotypes to produce

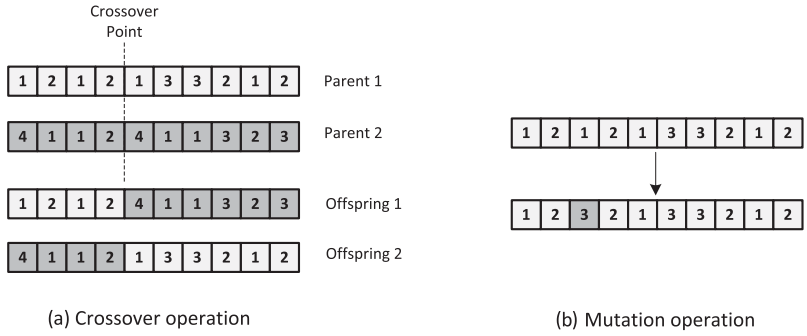

Fig. 7. Crossover and mutation operations.

ALGORITHM 2: Genetic algorithm to solve the joint SPM allocation and task scheduling/ mapping problem

```
   Input: System specifications // # of processor cores and # of SPM blocks
   Output: A genotype leading to minimum WCET among all visited genotypes
 1 Initialize(population);
 2 Compute_Fitness(population);
 3 iter ← 0;
 4 n ← 0;
 5 while iter < Max_iter and n < N do
 6 |   Sort(population);
 7 |   current_best ← Get_Best(population);
 8 |   if current_best.fitness > best.fitness then
 9 |   |   best ← current_best;
10 |   |   n ← 0;
11 |   else
12 |   |   n ← n + 1;
13 |   end
14 |   Selection(population, α);
15 |   Crossover(population);
16 |   Mutate(population,β);
17 |   Compute_Fitness(population);
18 |   iter ← iter + 1;
19 end
20 return best;
```

new genotypes for the next iteration instead of dead genotypes. Then, genotypes in the new population are occasionally mutated by probability of β to avoid rapid convergence of the algorithm in a local maximum. These works are repeated until the maximum number of iterations (Max_iter) is reached or in N consecutive iterations no improvement is achieved (line 5). The algorithm finishes by returning the best-found genotype.

Figure 7(a) shows an example of the crossover operation used in our GA. As shown in this figure, we use two-parent one-point crossover, which means that for each crossover operation two genotypes (parents) are selected, and one point in them is chosen randomly and all information on one side of that point in either genotypes is swapped. Consequently, two new genotypes (offspring) are produced, and these offspring are inserted to the next population. Figure 7(b) shows an example of the mutation operation used in our GA. In the mutation operation, one data of a genotype is selected

randomly and its value is changed to a new valid random value, which is less than or equal to the total available SPM size. It should be noted that as we use mutation and crossover, the sum of allocated SPM space to the tasks in a genotype can be greater than the available SPM size. However, this does not mean that more SPM space than the available SPM size is required. Indeed, whenever the sum of allocated SPM size to a set of tasks is greater than the available SPM size, this means that those tasks cannot be scheduled to be executed in parallel. Therefore, when the sum of allocated SPM space to the tasks in a genotype is greater than the available SPM size, during the scheduling phase (phenotype generation) the sum of allocated SPM space to the tasks that are executed in parallel is always less than the available SPM size. Hence, it is not required to check the sum of allocated SPM space in initial genotypes or in new genotypes that are produced by the crossover and the mutation operations.

7. EXPERIMENTAL EVALUATION

7.1. Experimental Setup

To evaluate our proposed methods, we used several benchmarks from the MiBench [Guthaus et al. 2001] and Mälardalen [Gustafsson et al. 2010] benchmark suites, which have been used for embedded and real-time systems studies by several previous works (e.g., [Ding et al. 2012; Plazar et al. 2012; Whitham et al. 2014]). Moreover, to increase the diversity of benchmarks, we not only conducted experiments with the original MiBench and Mälardalen benchmarks but also used additional benchmarks that are obtained by modifying the inputs of these benchmarks. Table I depicts the descriptions and the size (code + data) of the benchmarks.

To compile the benchmarks, we used the SimpleScalar GCC PISA cross-compiler [Burger and Austin 1997], and to evaluate the WCET of the benchmarks, we used the Chronos timing analyzer [Li et al. 2007]. For considering SPM in the experiments, we did not modify the cross-compiler. Rather, we modeled SPM in the Chronos timing analyzer. In Chronos, we specify which memory space is mapped to SPM and which one is mapped to main memory; based on this data, Chronos can conduct timing analysis considering the impact of SPM use. As described in Section 3, in the considered architecture we have a two-level SPM; hence, a two-level SPM model is included in the Chronos processor model to consider the effects of SPMs on WCET.

As mentioned in Section 3, in our proposed technique we have two levels of SPM space allocation: intertask SPM allocation and intratask SPM allocation (and for intertask SPM allocation, we do not propose any technique and exploit existing techniques). The intratask SPM allocation method that we considered in the experiments is an ILP method proposed in Suhendra et al. [2005]. In this method, for each memory object (code+data) a flag is defined to indicate that it must be placed in SPMs or not. Considering the limited size of SPMs, the values of these flags are selected to minimize WCET. It is noteworthy that in our proposed technique, we do not directly involve WCET analysis in intertask SPM space allocation; rather, WCET analysis is performed in the intratask allocation, and the intertask allocation uses the results obtained from the intratask allocation.

For a comprehensive evaluation of the proposed methods, we repeated the experiments for 80 sets of tasks, which were randomly selected from the Mälardalen benchmark suite. The task sets consisted of different number of tasks (5, 10, 15, and 20) (20 sets for each number of tasks). We also repeated the experiments for 10-task sets, which were randomly selected from the Mibench benchmark suite. Moreover, the experiments were performed for dual- and quad-core processors with different SPM sizes.

With respect to the SPM size, it should be noted that the designers of embedded systems usually avoid using large SPMs, as large SPMs can increase the cost, energy

Table I. Benchmarks

Name	Size (Bytes)	Description	Source
basicmath (small)	4348	Simple mathematical calculations (small input)	MiBench
basicmath (large)	5144	Simple mathematical calculations (large input)	
bitcount (small)	2708	Counting the number of bits in an array of integers (small input)	
bitcount (large)	2716	Counting the number of bits in an array of integers (large input)	
dijkstra	5010	Finding the shortest path between nodes using Dijkstra algorithm	
stringsearch (small)	3116	String search (small input)	
stringsearch (large)	13356	String search (large input)	
patricia (small)	791300	Searching, inserting and removing nodes in a Patricia trie (small input)	
patricia (large)	3150644	Searching, inserting and removing nodes in a Patricia trie (large input)	
crc32	20876	32 BIT ANSI X3.66 CRC checksum	
adpcm	8364	Adaptive pulse code modulation algorithm	Mälardalen
bs	1052	Binary search for the array of 15 integer elements	
crc	1372	Cyclic redundancy check computation on 40 bytes of data	
fdct	2424	Fast Discrete Cosine Transform of an 8x8 block	
insertsort	376	Insertion sort on an array of size 10	
lms	2000	LMS adaptive signal enhancement	
matmult	5164	Matrix multiplication of two 20x20 matrices	
ndes	5741	Complex embedded code	
statemate	10156	Automatically generated code	
bs1000	8252	Binary search for the array of 1000 integer elements	Modified from Mälardalen
crc2000	3164	Cyclic redundancy check computation on 2000 bytes of data	
fdct100	15156	Fast Discrete Cosine Transform of 100 8x8 blocks	
fft128	6688	128 point Fast Fourier Transform	
fft512	12832	512 point Fast Fourier Transform	
fir50	1292	Finite impulse response filter over a 50 items long sample	
fir500	4860	Finite impulse response filter over a 500 items long sample	
insertsort100	744	Insertion sort on an array of size 100	
ludcmp10	4008	LU decomposition algorithm for 10x10 matrices	
ludcmp5	3496	LU decomposition algorithm for 5x5 matrices	
matmult10	1564	Matrix multiplication of two 10x10 matrices	
matmult5	684	Matrix multiplication of two 5x5 matrices	

consumption, and SPM delay [Marwedel 2006]. On the other hand, if excessively small SPMs are used, one will not have enough space to keep highly accessed data, which has a considerable negative influence on the system performance (and WCET). Indeed, the suitable size of SPM varies based on the system application.

Analyzing the impact of SPM size variation on the performance of embedded systems is indeed an important research area, and some works (e.g., Nguyen et al. [2005] and Che and Chatha [2011]) have considered this issue. However, in this article, we do not intend to analyze the impact of SPM size variation; rather, we want to provide a technique for task scheduling/mapping and SPM space allocation to reduce WCET. Nevertheless, in the experiments, we considered a range of SPM size.

We assumed 64 and 256 Bytes for local SPM size and 1KB, 2KB, 4KB, 8KB, 16KB, 32KB, and 64KB for global SPM size. It should be noted that these values have been used as case studies, and evidently our proposed technique can be used for other sizes, considering that during the development of the techniques we never have any assumption with respect to the SPM size. Furthermore, it should be noted that as the size of the benchmarks varies from 0.3KB to 3MB, the range of SPM size that we assumed in the experiments (from 1KB to 64KB) is quite reasonable, because if we considered

bigger values in the experiments, it would not have any influence on the experiment results, as the execution time of the tasks does not change considerably by allocating bigger SPM space. Moreover, some commercial microprocessors (microcontrollers) have similar SPM (internal SRAM) size (e.g., XMOSs XS1-L4A-64-TQ48 multicore microcontroller [XMOS 2013] with 64KB internal SRAM and Atmels AT32UC3L016 microcontroller [Atmel 2012] with 8KB internal SRAM). Finally, we assume that the off-chip memory is large enough to hold all memory objects of the tasks and assume a latency of 1, 4, and 100 cycles for a single access to local SPM, global SPM, and off-chip memory, respectively. These assumptions are in line with the realistic data provided in Suhendra et al. [2010] and Salamy and Ramanujam [2012].

With respect to the 100 clock cycle latency of off-chip memory access, it should be noted that if only one core accessed the off-chip memory at a time (i.e., assuming that other cores do not access the off-chip memory simultaneously), the latency would be much lower (e.g., about 20 cycles). However, as in this article we consider real-time systems, we need to consider a guaranteed upper bound for this latency, which is the worst-case latency that occurs when all cores access the off-chip memory simultaneously. For the worst case, we have considered the latency to be 100 cycles, and as mentioned previously, this assumption is in line with the data provided by previous works (e.g., Suhendra et al. [2010]). To compare to methods provided in previous related works, we implemented three other methods in addition to our proposed methods. The evaluated methods are as follows:

(1) Our optimal method for the dynamic SPM space reuse scheme (Section 5)
(2) Our heuristic method for the dynamic SPM space reuse scheme (Section 6)
(3) An optimal method based on the method presented in Suhendra et al. [2006] for the SSA scheme (Section 4.1),
(4) The heuristic method presented in Salamy and Ramanujam [2012] for the SSP scheme (Section 4.1), *HSSP1*
(5) The heuristic method presented in Liu et al. [2011] for the SSP scheme, *HSSP2*.

To find a proper value for each parameter of the proposed GA, we examined different values for each parameter by running several experiments. The obtained results are as follows:

—*Selection rate (α)*: The results showed that setting $\alpha = 0.5$ leads to better results than the other examined cases. Note that for very high selection rates (e.g., 0.9), very few offspring are generated and hence lower solution space is explored. On the other hand, for lower selection rates (e.g., 0.1), it is possible to miss many of the good solutions, which can be used to produce better offspring.
—*Mutation rate (β)*: The results of the experiments showed that setting $\beta = 0.1$ is better than the other examined cases. Note that for very high mutation rates (e.g., 0.9), the proposed algorithm becomes a random search algorithm, and for very low mutation rates (e.g., 0.001), the algorithm rapidly converges in a local maximum.
—*Population size*: Setting the population size to 200 makes a better trade-off between the execution time of the algorithm and the quality of solutions than in other cases. Note that for very low population sizes (e.g., 10), the diversity of genotypes in the population decreases and hence lower solution space is explored. On the other hand, for higher population sizes, the algorithm slows down.
—*Maximum allowable iterations (Max_iter) and maximum allowable consecutive iterations without any improvement in the result (N)*: Considering the results of the experiments, we set $N = 100$ and $Max\_iter = 1{,}000$. Note that for very low values for N (e.g., 5) and for Max_iter (e.g., 50), the quality of solutions considerably decreases. On the other hand, for very high values for N (e.g., 500) and for Max_iter

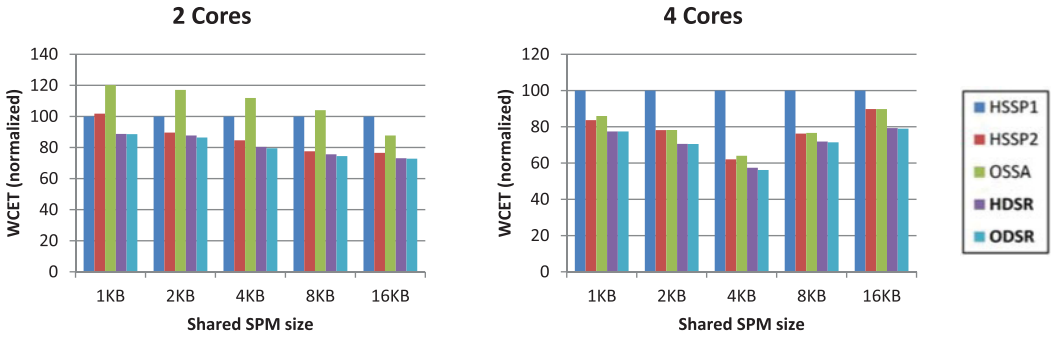

Fig. 8. The results of experiments for 5-task applications selected from the Mälardalen benchmark suite.

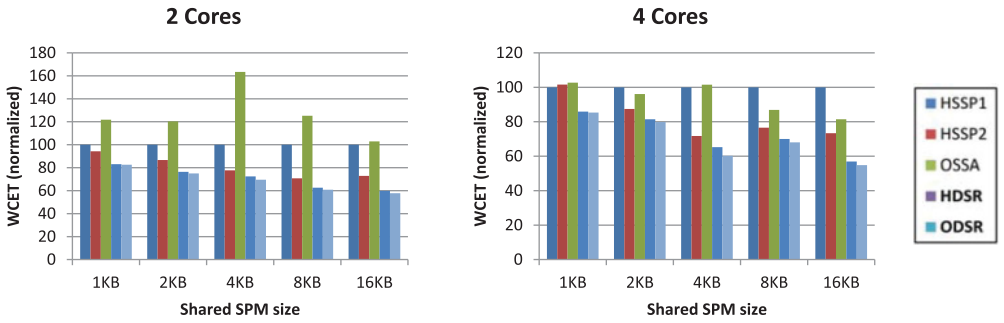

Fig. 9. The results of experiments for 10-task applications selected from the Mälardalen benchmark suite.

(e.g., 10,000), the execution time of the algorithm significantly increases without any noticeable improvement in solutions.

It should be noted that all of the experiments were run on a system with an Intel® Core™ i7-2630QM processor, and to optimally solve the joint SPM allocation and task scheduling/mapping problem (the proposed optimal method), we used the GAMS/Cplex solver [Brooke et al. 2003].

7.2. Results and Discussion

Figures 8 and 9 show the average normalized total WCETs of 5-task and 10-task sets that were selected from Mälardalen benchmark suite. The first column in these figures shows the results obtained by applying HSSP1, and the next four columns show the results obtained by applying HSSP2, OSSA, the proposed heuristic method (HDSR), and the proposed optimal method (ODSR), respectively. All of the results are normalized with respect to the results of HSSP1. Similar to Figures 8 and 9, the obtained results for 10-task sets that were selected from the MiBench benchmark suite are depicted later in Figure 12.

The results of HSSP1, HSSP2, and the proposed heuristic method (HDSR) for 15-task and 20-task sets (which were selected from the Mälardalen benchmarks) are shown in Figure 10 and Figure 11, respectively. For 15-task and 20-task sets, we do not report the results of OSSA and the proposed optimal method, as their execution did not finish within the time limit (3 hours).

In addition to comparing the WCET of the solutions, we also compare the execution time of the methods. Table II shows the execution time of each method for a system, which consists of two processor cores and 1KB of shared SPM. It should be noted that

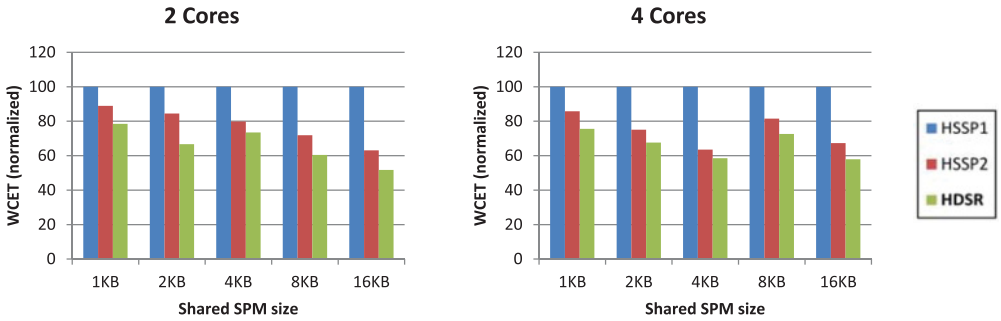

Fig. 10. The results of experiments for 15-task applications selected from the Mälardalen benchmark suite.

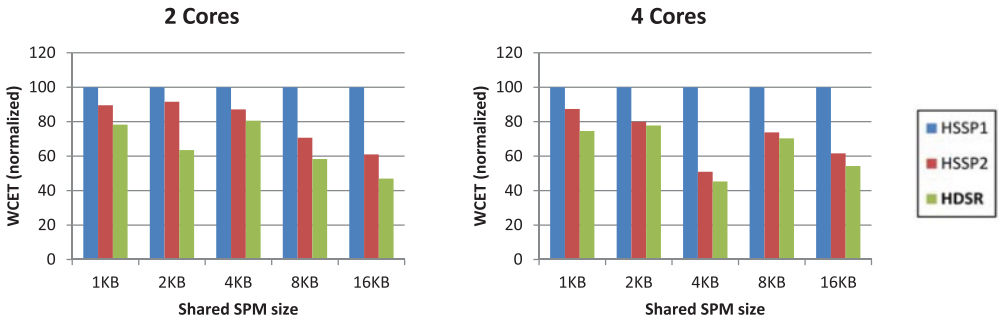

Fig. 11. The results of experiments for 20-task applications selected from the Mälardalen benchmark suite.

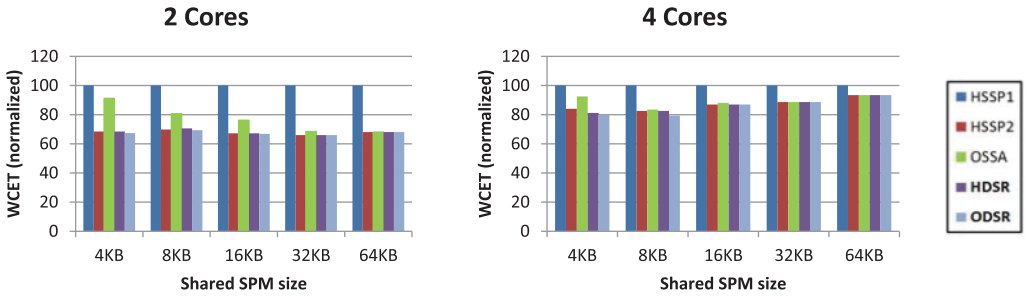

Fig. 12. The results of experiments for 10-task applications selected from the MiBench benchmark suite.

Table II. Execution Time of HSSP1, HSSP2, OSSA, ODSR, and HDSR

	HSSP1	HSSP2	OSSA	ODSR	HDSR
5	<1ms	<1ms	0.1s	0.2s	0.2s
10	<1ms	<1ms	13s	134s	1.35s
15	<1ms	<1ms	<3h	<3h	2.35s
20	<1ms	<1ms	N/A	N/A	6.25s

as for other system configurations, similar results to the ones in Table II were obtained, but we do not report them.

The important results of the experiments and related discussions are as follows:

—As shown in Figures 8, 9, and 12, in all cases the solutions of the proposed optimal method (ODSR) are better than the solutions of the other methods. This result was expected, as the proposed optimal method finds optimal solutions in the proposed dynamic SPM space reuse scheme. Note that as we discussed in Section 4, in this scheme, the SPM space is used more efficiently than in the SSA and SSP schemes, and hence better solutions can be achieved when this scheme is used.
—The solutions of the proposed heuristic method (HDSR) in the worst case are only 8% and 3% worse than the solutions of the proposed optimal method for task sets selected from the Mälardalen and MiBench benchmarks, respectively. This shows the proposed heuristic method efficiency.
—In the best case, the solutions of the proposed heuristic method for the task sets selected from the Mälardalen benchmark suite are about 55%, 54%, and 30% better than the solutions of OSSA, HSSP1, and HSSP2, respectively. Similarly, for the task sets selected from the MiBench benchmark suite, the solutions of the proposed heuristic method are about 13%, 33%, and 3% better than the solutions of OSSA, HSSP1, and HSSP2, respectively. Moreover, in the worst case, the results obtained for the proposed heuristic method is about 6% better than the results of HSSP1 and is almost equal to the results of HSSP2 and OSSA. Therefore, the proposed heuristic method in almost all cases produces better results than HSSP1, HSSP2, and OSSA. These results were also expected, as the proposed heuristic method similar to the proposed optimal method uses the proposed dynamic SPM space reuse scheme.
—The solutions of HSSP2, in most cases, are better than the solutions of HSSP1. This is because in HSSP1, after finding an initial solution, only the size of the SPM space allocated to each core can be adjusted. However, in HSSP2, in addition to the size of the allocated SPM spaces, task scheduling/mapping can also be adjusted [Liu et al. 2011], and hence HSSP2 usually can find better solutions than HSSP1.
—As it can be seen from Table II, the execution time of the proposed heuristic method (HDSR) for 20-task applications (the largest examined applications in the experiments) on average is 6.25 seconds. Evidently, this execution time is quite suitable, as we use this method at design time. Indeed, the results generated by this GA-based method are exploited at runtime. Nevertheless, we do not propose to adopt a GA-based technique to be used at runtime. This is because, as mentioned in Section 4, online techniques cannot be computationally expensive, and GA-based methods basically require a lot of computation [Haupt and Haupt 2004]. For example, in our proposed scheme, although the 6.25-second overhead is quite acceptable at design time, it is not feasible to have such overheads at runtime, especially considering that this 6.25-second execution time is obtained by an Intel core i7 system, and the overhead on an embedded processor (which is usually much simpler) must be much higher.
—As shown in Table II, the execution time of the proposed optimal method (ODSR) is reasonable for the small number of tasks. Hence, as it finds the best solution, it can be recommended for fairly small systems. However, its execution time grows rapidly by increasing the number of tasks. Therefore, for the systems with large number of tasks, it is not applicable.
—The timing overheads of HSSP1 and HSSP2 in all cases are less than 1ms, which is much lower than the execution time of the proposed methods. However, the solutions that HSSP1 and HSSP2 find are usually worse than the solutions of the proposed methods.

8. CONCLUSIONS AND FUTURE WORKS

In this work, we studied shared SPM management for real-time multicore embedded systems and proposed a dynamic SPM space reuse scheme to manage SPM efficiently. Moreover, in our proposed scheme, we considered the coherency of SPM management and task scheduling/mapping. To apply our scheme, we proposed two methods: (1) an ILP based method, which finds optimal solutions for fairly small systems, and (2) a heuristic method based on GA, which finds near-optimal solutions in reasonable time for large systems. We evaluated the proposed methods by performing several experiments on different benchmarks and different system configurations. The important results of the experiments can be summarized as follows:

(1) As expected, in all cases the proposed optimal method outperforms the other methods provided in previous related works.
(2) The solutions of the proposed heuristic method in the worst case are only 8% worse than the solutions of the proposed optimal method.
(3) The proposed heuristic method has much lower execution time than the optimal method for large systems.
(4) The execution time of the proposed heuristic method is slightly higher than HSSP1 and HSSP2. However, its solutions are up to 54% and 30% better than the solutions of HSSP1 and HSSP2, respectively.

To extend the proposed scheme, the following future works are suggested:

(1) The proposed scheme is only applicable to a frame-based application model running on a nonpreemptive system. One possible future work is to extend the proposed scheme to other application and system models, such as periodic and sporadic. This requires developing new analytical models and new algorithms, and new experiments must be conducted to validate the models and algorithms.
(2) Many embedded real-time systems have limited energy resources. Therefore, considering energy consumption in the proposed scheme is an interesting extension.
(3) Considering tasks execution time variations can help to utilize available SPM space more efficiently. Therefore, one possible extension is to consider these variations in SPM allocation.
(4) Analyzing the impact of malicious code and reliability issues on SPM management in multicore system is another important issue that can be considered in future works. This requires developing reliability models and providing techniques to resist against malicious codes and tolerate faults. Furthermore, experimental analysis (e.g., fault injection experiments) must be conducted to analyze the system reliability against malicious codes and faults.

REFERENCES

Imtiaz Ahmad and Muhammad K. Dhodhi. 1996. Multiprocessor scheduling in a genetic paradigm. *Parallel Computing* 22, 3, 395–406. DOI:http://dx.doi.org/10.1016/0167-8191(95)00068-2

Atmel. 2012. 32-bit Atmel AVR Microcontroller. Retrieved March 27, 2015, from http://www.atmel.com/Images/doc32099.pdf.

Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. 2002. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES'02)*. ACM, New York, NY, 73–78. DOI:http://dx.doi.org/10.1145/774789.774805

Anthony Brooke, David Kendrick, Alexander Meeraus, and Ramesh Raman. 2003. *GAMS/CPLEX 9.0*. GAMS Development Corporation, Washington, DC.

Doug Burger and Todd M. Austin. 1997. The SimpleScalar tool set, version 2.0. *ACM SIGARCH Computer Architecture News* 25, 3, 13–25.

Giorgio C. Buttazzo. 2011. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Vol. 24. Springer.

Lance Chambers. 1995. *Practical Handbook of Genetic Algorithms*. CRC Press, Boca Raton, FL.

Che-Wei Chang, Jian-Jia Chen, Tei-Wei Kuo, and Heiko Falk. 2013. Real-time partitioned scheduling on multi-core systems with local and global memories. In *Proceedings of the 18th Asia and South Pacific Design Automation Conference (ASP-DAC'13)*. IEEE, Los Alamitos, CA, 467–472. DOI:http://dx.doi.org/10.1109/ASPDAC.2013.6509640

Che-Wei Chang, Jian-Jia Chen, Tei-Wei Kuo, and Heiko Falk. 2014. Real-time task scheduling on island-based multi-core platforms. *IEEE Transactions on Parallel and Distributed Systems* PP, 99, 1. DOI:http://dx.doi.org/10.1109/TPDS.2013.2297308

Che-Wei Chang, Jian-Jia Chen, Waqaas Munawar, Tei-Wei Kuo, and Heiko Falk. 2012. Partitioned scheduling for real-time tasks on multiprocessor embedded systems with programmable shared SRAMs. In *Proceedings of the 10th ACM International Conference on Embedded Software (EMSOFT'12)*. ACM, New York, NY, 153–162. DOI:http://dx.doi.org/10.1145/2380356.2380384

Karam S. Chatha and Ranga Vemuri. 2002. Hardware-software partitioning and pipelined scheduling of transformative applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 10, 3, 193–208. DOI:http://dx.doi.org/10.1109/TVLSI.2002.1043323

Sudipta Chattopadhyay and Abhik Roychoudhury. 2011. Static bus schedule aware scratchpad allocation in multiprocessors. *ACM SIGPLAN Notices* 46, 5, 11–20. DOI:http://dx.doi.org/10.1145/2016603.1967680

Weijia Che and Karam S. Chatha. 2011. Scheduling of stream programs onto SPM enhanced processors with code overlay. In *Proceedings of the 9th IEEE Symposium on Embedded Systems for Real-Time Multimedia (ESTIMedia'11)*. IEEE, Los Alamitos, CA, 9–18. DOI:http://dx.doi.org/10.1109/ESTIMedia.2011.6088530

ILOG Cplex. 2007. 11.0 Users Manual. IBM. Available at http://www.ilog.com/products/cplex/.

Vinay Devadas and Hakan Aydin. 2012. On the interplay of voltage/frequency scaling and device power management for frame-based real-time embedded applications. *IEEE Transactions on Computers* 61, 1, 31–44. DOI:http://dx.doi.org/10.1109/TC.2010.248

Huping Ding, Yun Liang, and Tulika Mitra. 2012. WCET-centric partial instruction cache locking. In *Proceedings of the 49th ACM/EDAC/IEEE Design Automation Conference (DAC'12)*. IEEE, Los Alamitos, CA, 412–420.

Heiko Falk and Jan C. Kleinsorge. 2009. Optimal static WCET-aware scratchpad allocation of program code. In *Proceedings of the 46th Annual Design Automation Conference (DAC'09)*. ACM, New York, NY, 732–737. DOI:http://dx.doi.org/10.1145/1629911.1630101

Marco E. T. Gerards and Jan Kuper. 2013. Optimal DPM and DVFS for frame-based real-time systems. *ACM Transactions on Architecture and Code Optimization* 9, 4, Article No. 41. DOI:http://dx.doi.org/10.1145/2400682.2400700

Rony Ghattas, Gregory Parsons, and Alexander G. Dean. 2007. Optimal unified data allocation and task scheduling for real-time multi-tasking systems. In *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*. IEEE, Los Alamitos, CA, 168–182. DOI:http://dx.doi.org/10.1109/RTAS.2007.23

Martin Grajcar. 1999. Genetic list scheduling algorithm for scheduling and allocation on a loosely coupled heterogeneous multiprocessor system. In *Proceedings of the 36th Design Automation Conference (DAC'99)*. IEEE, Los Alamitos, CA, 280–285. DOI:http://dx.doi.org/10.1109/DAC.1999.781326

Shouzhen Gu, Qingfeng Zhuge, Jingtong Hu, Juan Yi, and Edwin H.-M. Sha. 2013. Efficient task assignment and scheduling for MPSoC DSPS with VS-SPM considering concurrent accesses through data allocation. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'13)*. IEEE, Los Alamitos, CA, 2615–2619. DOI:http://dx.doi.org/10.1109/ICASSP.2013.6638129

Shouzhen Gu, Qingfeng Zhuge, Juan Yi, Jingtong Hu, and Edwin H.-M. Sha. 2014. Optimizing task and data assignment on multi-core systems with multi-port SPMs. *IEEE Transactions on Parallel and Distributed Systems* PP, 99, 1. DOI:http://dx.doi.org/10.1109/TPDS.2014.2356194

Yibo Guo, Qingfeng Zhuge, Jingtong Hu, Meikang Qiu, and Edwin H.-M. Sha. 2011. Optimal data allocation for scratch-pad memory on embedded multi-core systems. In *Proceedings of the International Conference on Parallel Processing (ICPP'11)*. IEEE, Los Alamitos, CA, 464–471. DOI:http://dx.doi.org/10.1109/ICPP.2011.79

Yibo Guo, Qingfeng Zhuge, Jingtong Hu, Juan Yi, Meikang Qiu, and Edwin H.-M. Sha. 2013. Data placement and duplication for embedded multicore systems with scratch pad memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 32, 6, 809–817. DOI:http://dx.doi.org/10.1109/TCAD.2013.2238990

Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The Mälardalen WCET benchmarks—past, present and future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*. 137–147.

Matthew R. Guthaus, Jeffrey S. Ringenberg, Daniel Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE International Workshop on Workload Characterization (WWC-4)*. IEEE, Los Alamitos, CA, 3–14. DOI:http://dx.doi.org/10.1109/WWC.2001.990739

Randy L. Haupt and Sue Ellen Haupt. 2004. *Practical Genetic Algorithms*. John Wiley and Sons, Hoboken, NJ.

Mahmut Kandemir and Alok Choudhary. 2002. Compiler-directed scratch pad memory hierarchy design and management. In *Proceedings of the 39th Annual Design Automation Conference (DAC'02)*. ACM, New York, NY, 628–633. DOI:http://dx.doi.org/10.1145/513918.514077

Mahmut Kandemir, Ozcan Ozturk, and Mustafa Karakoy. 2004. Dynamic on-chip memory management for chip multiprocessors. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'04)*. ACM, New York, NY, 14–23. DOI:http://dx.doi.org/10.1145/1023833.1023838

Mahmut Kandemir, Jagannathan Ramanujam, and Alok Choudhary. 2002. Exploiting shared scratch pad memory space in embedded multiprocessor systems. In *Proceedings of the 39th Annual Design Automation Conference (DAC'02)*. ACM, New York, NY, 219–224. DOI:http://dx.doi.org/10.1145/513918.513974

Sangyeol Kang and Alexander G. Dean. 2010. DARTS: Techniques and tools for predictably fast memory using integrated data allocation and real-time task scheduling. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10)*. IEEE, Los Alamitos, CA, 333–342. DOI:http://dx.doi.org/10.1109/RTAS.2010.36

Namyun Kim, Minsoo Ryu, Seongsoo Hong, Manas Saksena, Chong-Ho Choi, and Heonshik Shin. 1996. Visual assessment of a real-time system design: A case study on a CNC controller. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*. IEEE, Los Alamitos, CA, 300–310. DOI:http://dx.doi.org/10.1109/REAL.1996.563726

Hermann Kopetz. 2011. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, New York, NY.

Yu-Kwong Kwok and Ishfaq Ahmad. 1997. Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm. *Journal of Parallel and Distributed Computing* 47, 1, 58–77. DOI:http://dx.doi.org/10.1006/jpdc.1997.1395

Dawei Li and Jie Wu. 2014. Minimizing energy consumption for frame-based tasks on heterogeneous multiprocessor platforms. *IEEE Transactions on Parallel and Distributed Systems* PP, 99, 1. DOI:http://dx.doi.org/10.1109/TPDS.2014.2313338

Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudury. 2007. Chronos: A timing analyzer for embedded software. *Science of Computer Programming* 69, 1–3, 56–67. http://www.comp.nus.edu.sg/~rpembed/chronos.

Tiantian Liu, Yingchao Zhao, Minming Li, and Chun Jason Xue. 2011. Joint task assignment and cache partitioning with cache locking for {WCET} minimization on {MPSoC}. *Journal of Parallel and Distributed Computing* 71, 11, 1473–1483. DOI:http://dx.doi.org/10.1016/j.jpdc.2011.05.006

Yu Liu and Wei Zhang. 2012. Exploiting multi-level scratchpad memories for time-predictable multicore computing. In *Proceedings of the 30th IEEE International Conference on Computer Design (ICCD'12)*. IEEE, Los Alamitos, CA, 61–66. DOI:http://dx.doi.org/10.1109/ICCD.2012.6378618

Peter Marwedel. 2006. *Embedded System Design*. Vol. 1. Springer, New York, NY.

Nghi Nguyen, Angel Dominguez, and Rajeev Barua. 2005. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES'05)*. ACM, New York, NY, 115–125. DOI:http://dx.doi.org/10.1145/1086297.1086313

Fatma A. Omara and Mona M. Arafa. 2010. Genetic algorithms for task scheduling problem. *Journal of Parallel and Distributed Computing* 70, 1, 13–22. DOI:http://dx.doi.org/10.1016/j.jpdc.2009.09.009

Ozcan Ozturk, Guangyu Chen, Mahmut Kandemir, and Mustafa Karakoy. 2006a. An integer linear programming based approach to simultaneous memory space partitioning and data allocation for chip multiprocessors. In *Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*. IEEE, Los Alamitos, CA. DOI:http://dx.doi.org/10.1109/ISVLSI.2006.22

Ozcan Ozturk, Mahmut Kandemir, Guangyu Chen, Mary J. Irwin, and Mustafa Karakoy. 2005. Customized on-chip memories for embedded chip multiprocessors. In *Proceedings of the Asia*

and South Pacific Design Automation Conference (ASP-DAC'05). ACM, New York, NY, 743–748. DOI:http://dx.doi.org/10.1145/1120725.1121008

Ozcan Ozturk, Mahmut Kandemir, and Ibrahim Kolcu. 2006b. Shared scratch-pad memory space management. In *Proceedings of the 7th International Symposium on Quality Electronic Design (ISQED'06)*. IEEE, Los Alamitos, CA. DOI:http://dx.doi.org/10.1109/ISQED.2006.115

Michael Pinedo. 2012. *Scheduling: Theory, Algorithms, and Systems*. Springer, New York, NY.

Sascha Plazar, Jan C. Kleinsorge, Peter Marwedel, and Heiko Falk. 2012. WCET-aware static locking of instruction caches. In *Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO'12)*. ACM, New York, NY, 44–52. DOI:http://dx.doi.org/10.1145/2259016.2259023

Cosmin Rusu, Rami Melhem, and Daniel Mossé. 2003. Maximizing rewards for real-time applications with energy constraints. *ACM Transactions on Embedded Computing Systems* 2, 4, 537–559. DOI:http://dx.doi.org/10.1145/950162.950166

Hassan Salamy and Jagannathan Ramanujam. 2012. An effective solution to task scheduling and memory partitioning for multiprocessor system-on-chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 5, 717–725. DOI:http://dx.doi.org/10.1109/TCAD.2011.2181848

Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. 2005. WCET centric data allocation to scratchpad memory. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)*. IEEE, Los Alamitos, CA, 223–232. DOI:http://dx.doi.org/10.1109/RTSS.2005.45

Vivy Suhendra, Chandrashekar Raghavan, and Tulika Mitra. 2006. Integrated scratchpad memory optimization and task scheduling for MPSoC architectures. In *Proceedings of the 2006 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'06)*. ACM, New York, NY, 401–410. DOI:http://dx.doi.org/10.1145/1176760.1176809

Vivy Suhendra, Abhik Roychoudhury, and Tulika Mitra. 2010. Scratchpad allocation for concurrent embedded software. *ACM Transactions on Programming Languages and Systems* 32, 4, Article No. 13. DOI:http://dx.doi.org/10.1145/1734206.1734210

Texas Instruments. 2011. TMS320C6472 Fixed-Point Digital Signal Processor. Retrieved March 27, 2015, from http://www.ti.com/lit/ds/symlink/tms320c6472.pdf

Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. 2006. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Transactions on Embedded Computing Systems* 5, 2, 472–511. DOI:http://dx.doi.org/10.1145/1151074.1151085

Qing Wan, Hui Wu, and Jingling Xue. 2013. Scratchpad memory aware task scheduling with minimum number of preemptions on a single processor. In *Proceedings of the 18th Asia and South Pacific Design Automation Conference (ASP-DAC'13)*. IEEE, Los Alamitos, CA, 741–748. DOI:http://dx.doi.org/10.1109/ASPDAC.2013.6509689

Jack Whitham, Neil C. Audsley, and Robert I. Davis. 2014. Explicit reservation of cache memory in a predictable, preemptive multitasking real-time system. *ACM Transactions on Embedded Computing Systems* 13, 4s, Article No. 120. DOI:http://dx.doi.org/10.1145/2523070

XMOS. 2013. XS1-L4A-64-TQ48 Datasheet. Retrieved March 27, 2015, from https://www.xmos.com/download/public/XS1-L4A-64-TQ48-Datasheet(1.2).pdf.

Lei Zhang, Meikang Qiu, Wei-Che Tseng, and Edwin H.-M. Sha. 2010. Variable partitioning and scheduling for MPSoC with virtually shared scratch pad memory. *Journal of Signal Processing Systems* 58, 2, 247–265. DOI:http://dx.doi.org/10.1007/s11265-009-0362-3

Qingfeng Zhuge, Yibo Guo, Jingtong Hu, Wei-Che Tseng, Chun J. Xue, and Edwin H.-M. Sha. 2012. Minimizing access cost for multiple types of memory units in embedded systems through data allocation and scheduling. *IEEE Transactions on Signal Processing* 60, 6, 3253–3263. DOI:http://dx.doi.org/10.1109/TSP.2012.2189768

Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. 2012. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys* 45, 1, Article No. 4. DOI:http://dx.doi.org/10.1145/2379776.2379780

Received November 2014; revised January 2015; accepted February 2015