Operating system integrated energy aware scratchpad allocation strategies for multiprocess applications*

Robert Pyka[1], Christoph Faßbach[1], Manish Verma[2], Heiko Falk[1], Peter Marwedel[1]
[1]Department of Computer Science XII
University of Dortmund, Germany
{first.lastname}@uni-dortmund.de

[2]European Technology Center
Altera Europe, United Kingdom
mverma@altera.com

Abstract

Various scratchpad allocation strategies have been developed in the past. Most of them target the reduction of energy consumption. These approaches share the necessity of having direct access to the scratchpad memory. In earlier embedded systems this was always true, but with the increasing complexity of tasks systems have to perform, an additional operating system layer between the hardware and the application is becoming mandatory. This paper presents an approach to integrate a scratchpad memory manager into the operating system. The goal is to minimize energy consumption. In contrast to previous work, compile time knowledge about the application's behavior is taken into account. A set of fast heuristic allocation methods is proposed in this paper. An in-depth study and comparison of achieved energy savings and cycle reductions was performed. The results show that even in the highly dynamic environment of an operating system equipped embedded system, up to 83% energy consumption reduction can be achieved.

1 Introduction

Studies show that in an embedded system, significant amount of energy is being dissipated in the memory subsystem. Furthermore, not only the energy dissipation but also the access times are becoming a performance bottleneck in current embedded systems [19, 12]. Over the last decades, the improvements in processing speed and the memory access times have diverged increasingly. For different memory technologies, there is usually a trade-off between size, speed and energy consumption. Therefore, it is beneficial to incorporate various memories into a system, so they can fulfill all requirements on the capacity, and provide sufficient performance for the application's hotspots. Beside Caches, which have the advantage of being almost transparent to the software, but have a hard to predict behavior and consume more energy than equivalent plain memories. Therefore, small fast on-chip memories aka. *scratchpads* have been introduced into embedded systems. Since they are plain memories, it is up to the compiler to exploit them in an efficient way.

Exploiting scratchpads can be performed in different scenarios. For example, a fixed set of processes which are permanently ready for execution, or an unrestricted access to the entire scratchpad are usual assumptions. The development in embedded systems shifts towards complex multi process systems incorporating an operating system layer in their software. That is the target scenario in this work. The scratchpad allocation strategies presented here provide efficient scratchpad utilization in a highly dynamic environment where the set of active processes may change at runtime. The goal is to maximize the amount of accesses to the scratchpad memory over the entire runtime of the system, implying reduced energy consumption and reduced runtime of processes. To achieve this goal, a combined approach is proposed. At compile-time, the set of memory objects is determined, consisting of statically or dynamically allocated data and functions. For each memory object, the access pattern of each process is determined and translated to a profit value which is attached to that object. At runtime, a scratchpad memory manager which is incorporated into the operating system computes the set of objects to be placed onto

*This work has been partially supported by the European ARTIST2 Network of Excellence.

the scratchpad. At each context switch, this set of memory objects may change in such a way that high scratchpad utilization for the set of currently active processes is achieved. This work proposes several heuristic methods which are capable of performing the allocation decisions in real time. Furthermore, an ILP based optimal method for the runtime scratchpad allocation for evaluation purposes will be presented. The results show that despite the tight time constraints for the heuristic methods, the best heuristic achieves energy savings close to values of the ILP based method.

The scratchpad manager presented here has been implemented for the RTEMS operating system [11]. Tests and benchmarks have been performed on the MPARM SoC simulator [9]. The simulated hardware is highly configurable, so it was possible to perform experiments for various scratchpad sizes, while obtaining comparable energy values.

The structure of this paper is as follows: It starts with an overview of related work. In Section 3 compile–time transformations are presented. Hereafter, seven heuristic runtime methods are presented, divided into two groups differing in the way memory objects are treated. In Section 5, the ILP based approach is presented. Section 6 contains the results. Presented values show that peak energy consumption reductions of up to 83% compared to the non–optimized application are achieved. Best heuristic methods are close to the optimal solution. On average, the best heuristic is less then 6% off the ILP based solution. The final section is the conclusion.

2 Related Work

Classical dynamic memory allocation techniques and various ways of optimizing accesses have been summarized by Wilson et al. [18]. The significant amount of research devoted to this aspect of operating systems shows the demand on efficient allocation strategies. Nevertheless, Garey et al. [4] show that this problem is NP-hard, therefore it is a reasonable approach to concentrate on fast and efficient heuristics.

Additional research has been done concerning specialized allocation strategies. Three of them should be mentioned here; one is the work done by Zhao et al. [20], which utilizes pools of similar memory objects and prefetching to improve the cache utilization. The second one presented by Mamagkakis et al. [10] offers an automatically generated set of Pareto-optimal configurations to adjust the memory manager to a particular application. To achieve significant energy reductions of up to 82%, their approach also considers the utilization of scratchpad memories. Finally, work done by Lebeck et al. [8] tries to perform the allocation of memory pages in such a way that low-power and standby modes of memories can be utilized efficiently.

Pure scratchpad allocation techniques can be divided into two major approaches; the first one allows the scratchpad memory to be distributed in the processor's address space. Approaches utilizing such architectures have been presented by Angiolini et al. [1, 2]. The advantage is that for a static distribution, the application does not have to be changed at all. Unfortunately, this kind of scratchpad memories is not very common in current systems, therefore, for our apporach we assume the scratchpad memory to be a plain memory bounded by a fixed address range.

A static allocation technique has been presented by Steinke et al. [14]. At compile time, the application is analyzed in terms of access profiles for each memory object. Data as well as code is taken into account. According to the profile results, a fixed partition of objects is computed. Energy savings of up to 78% compared to a system without scratchpad memory could be achieved.

Efficiently utilized scratchpad memories also outperform caches in terms of die size, energy consumption and runtime. Banakar et al. [3] compared a statically allocated scratchpad memory to a 2-way set-associative cache of the same size. The results show 34% smaller area requirements of the scratchpad memory and a lower energy consumption of up to 82% in the memory. Also, an improvement of the runtime of up to 16% could be achieved. Applications running on systems containing both scratchpads and caches do not have to be optimized only for one kind of memory. As shown in previous work [16], a scratchpad can be used to minimize the miss rate in the cache which also leads to increased energy savings of up to 29%.

According to work done by Wehmeyer et al. [17], the presence of partitioned scratchpads in a system can also be advantageous in terms of energy consumption. The work shows that additional energy savings of about 22% compared to a single scratchpad approach could be achieved. Finally, work also has been done to improve scratchpad utilization by subdividing memory objects into smaller parts [7].

Authors propose in [15] an approach which is able to optimize an application consisting of multiple concurrently running processes. Three different allocation strategies have been proposed. Beginning with a simple one, where the scratchpad is divided statically among applications, continuing with a dynamic one, which assigns the entire scratchpad to each application and ensures the content is copied properly on context switches. The third approach combines the previous two. Compared to the case where the application with the highest energy savings gets the entire scratchpad memory assigned and allocated statically, this approach was able to additionally save up to 37% of energy.

The usual assumption up till now was that the system does not provide an operating system layer between the application and hardware. In the approach presented in

this paper, contemporary state of the art embedded systems equipped with this kind of hardware abstraction layer are the target platform. Some work which is is closely related to the one presented in this paper, has already been done by Poletti et al. [13]. Poletti extended the memory manager present in the RTEMS operating system to be able to allocate on request some memory areas onto the scratchpad. Items once allocated to the scratchpad will stay there until deleted. The decision whether to allocate a memory area to the scratchpad is taken only based on the amount of free space available in the scratchpad memory. Therefore, the manager neither always ensures a good distribution of the scratchpad among the applications, nor achieves an adaptation to the current workload.

3 Compile–time Transformations

The operating system extension presented here requires some preparation steps to be performed at compile time. First of all, plain C code does not provide any notion of general memory objects. A memory object is a contiguous area of memory having a common set of properties. For example, global data arrays are the most common kind of memory objects. In our approach, both code and data may be grouped into memory objects. Therefore, the first step is to associate the global data, functions and dynamically allocated data present in the C code with a structure encapsulating all the memory object properties. The restriction to use function level granularity is mostly due to the source code scope of this approach. There are well known standard optimizations like function–exlining, which could be used to decrease the size of functions to add further opportunities for placing code objects on the scratchpad.

The main attributes of the object's properties structure are its size, the current address and its profit value. This value denotes how worthwhile it is to place a memory object on the scratchpad. The profit value is the number of accesses weighted with the difference in energy consumption per access between scratchpad and main memory. Objects which are frequently accessed are more valuable. Depending on the allocation strategy, an efficiency value is also precomputed for each memory object. This value takes into account the size of a memory object. Small, frequently accessed memory objects are most efficient.

$$\text{Efficiency} = \frac{\text{Profit value}}{\text{Object size}}$$

Gathering the profit values is done by profiling the application in a similar way as in previous single-application approaches. The application is compiled and simulated. The MPARM simulator records each memory access. Afterwards, the sequence of memory accesses together with

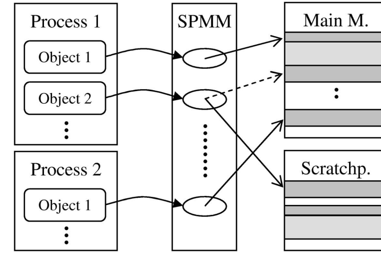

Figure 1: Dereferencing layer introduced into the application.

the address mapping file provided by the compiler is used to determine the number of accesses to each memory object.

The decision whether to put an object on the scratchpad and where it should be located is moved from the application to the runtime system. Therefore, the actual address of a memory object may change at runtime. This implies that whenever the application needs to access a memory object, it has to request its address from the runtime system. This is achieved by introducing an additional dereferencing layer in the access to memory objects. As shown in Figure 1, applications store only handles to the memory object.

The runtime system is designed to be able to dynamically reorganize the scratchpad content. In general in the presence of a preemptive multitasking operating system, it is not always safe to move memory objects. At the C-Code level, accesses are represented by a single expression. As shown in Figure 2, this does not have to translate into a single assembly level instruction. Therefore, it is possible that a dispatcher-interrupt occurs while some instructions have been executed, but the final access didn't happen, yet. Since the SPMM is called, it could reorganize the scratchpad at that point in time, and move the memory object being accessed. This would lead to an invalid memory access the next time execution continues. To prevent this situation, the SPMM has to be informed which memory objects are in use. This is achieved by marking objects as locked before accessing them, and releasing the lock after the access has been completed. Basically, a lock is a flag in the properties of a memory object. The runtime system checks this flag to determine how the object has to be treated at each reorganization step. A set flag signals to the runtime system that such a memory object has to be located at exactly the same address, the next time the process execution continues. This may be achieved in different ways, depending on the allocation method. The straight forward method is not to move the object at all.

Besides the fact that locking is necessary to ensure correct code execution, it has the advantage of allowing to cache the dereference pointer to used memory objects, preventing significant overhead on each access.

Compile time transformations are fully automated. All

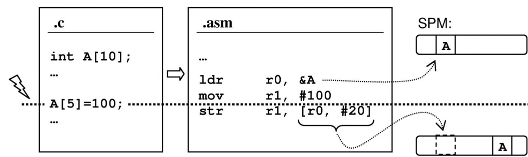

Figure 2: Invalid memory object access due to preemption.

source code modifications are performed on top of the ICD-C compiler framework [6]. The workflow can be divided into multiple steps:

- identify memory objects,
- generating code for locking accesses,
- add dereferencing layer,
- compile for profiling and execute,
- compute profit value and add it to the code.

4 Runtime Allocation Strategies

This work introduces a new component at the operating system level, which is capable of automatically managing the content of the scratchpad memory. From the developer's point of view, the approach should be absolutely transparent, therefore the compile-time transformation that need to be applied to the application are done automatically. Nevertheless, the interface between the scratchpad memory manager (SPMM) and the application is designed to be conveniently usable, so either manual utilization of the SPMM or more sophisticated compile–time analysis and transformations can be combined easily with this work.

The current implementation of the SPMM can handle statically allocated data objects and code objects corresponding to function which have to be introduced to the SPMM at startup, as well as data objects dynamically allocated at runtime. Each memory object provides a profit value to the SPMM indicating how worthwhile it is to be placed on the scratchpad. Several allocation strategies using these profit values have been implemented for the SPMM. The allocation strategies presented in Section 4.1 do not remove objects marked in use from the scratchpad memory. This ensures that the base address of these objects stays unchanged. Section 4.2 presents strategies which trade additional copy costs due to removing locked objects from the scratchpad for more flexibility while allocating new objects onto that memory.

Besides the allocation strategies, the most crucial part of the SPMM which decides about the actual energy savings is the way memory objects and free space on the scratchpad

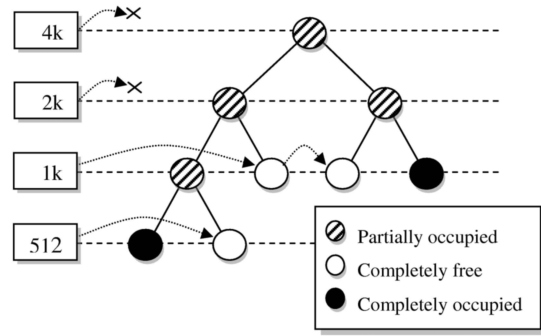

Figure 3: Tree based memory area management for 4k bytes of scratchpad memory.

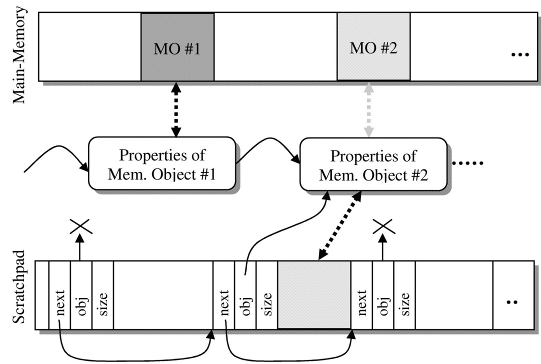

Figure 4: List based memory object management.

are managed. Three data structures have been investigated. As a first approach, a Skip-List based manager has been implemented. The basic idea is to manage lists of free areas which provide at least a particular amount of free space. A common approach is to manage lists for sizes of powers of two. Another approach would be a binary tree based buddy-system. The tree consists of nodes which represent a particular size. The root node is related to the entire scratchpad; if the entire scratchpad is free or completely occupied no other nodes are present. Otherwise, child nodes representing the upper and lower half of the area are inserted. Again, if one of these areas is partially occupied, the node has child nodes assigned. As shown in Figure 3, looking for free space is done by traversing nodes of a suitable minimal size. If no free area has been found, a larger area is split, if available. The third method uses a simple linked list of areas. Figure 4 shows an allocation example. The scratchpad areas are linked to their neighbors. Each area may be free or assigned to a memory object. If a scratchpad area is assigned to a memory object, the object pointer of that area is pointing to the properties structure of that object. A free area is denoted by an object pointer set to null.

While looking for a free space, the list is traversed. De-

pending on the allocation method, this may stop at the first sufficiently large area. Then, this area is split into one representing the memory object being placed there and a second one for the remaining free space. If memory objects are removed, free areas may be merged again. Mostly due to the limited size of the scratchpad, the list based method performed best.

4.1 Locking Allocation Strategies

Common to all allocation strategies is the way memory objects are selected to be candidates for being moved onto the scratchpad. Objects are placed onto the scratchpad in a greedy way, similar to the greedy solution of the knapsack problem. Each object is assigned an efficiency value. It is precomputed as the quotient of its profit value and the size. All objects currently in main memory are iterated in a decreasing order of their efficiency values. Each object which is worth being copied onto the scratchpad, because the copy costs do not exceed the profit value, is a candidate for being put on the scratchpad. If any memory objects are ready to be moved to the scratchpad, the final decision to do so and the actual placement is done according to one of the allocation strategies presented next.

Static Allocation assumes each memory object which has been placed once on the scratchpad to be locked. Therefore, objects never get removed from there until they are freed by the owning process. The placement is done in a first fit manner with a circulating (roving) pointer for the start area. There is a fixed minimal size of M bytes for each scratchpad area, allowing for a trade-off between runtime and scratchpad fragmentation. Basically, this prevents splitting the scratchpad into a large number of small areas. For example, an application requesting the SPMM to put many single–word global variables onto the scratchpad, could cause this situation. Due to the fact that usually, all global static data items and functions are introduced to the SPMM right before the application process starts, the first call to this allocation strategy will place them onto the scratchpad, leaving only a limited amount of free space for objects dynamically allocated at runtime. Also, processes starting later may or may not get their objects allocated to the scratchpad depending on the current utilization.

First Fit Allocation and the *Best Fit Allocation* are similar to the previous static strategy. In contrast to the static allocation, objects may be replaced on the scratchpad. Furthermore, these strategies have much higher runtime requirements, since for each candidate a new search at most over the entire list of scratchpad areas is required. With respect to the size, either a First-fit strategy or a Best-fit method is used to find a suitable location on the scratchpad. Only completely empty areas of suitable size or areas used by an unlocked object of another process are considered as possible locations to put the new object. The Best Fit Allocation prefers free and small areas. To avoid futile copy operations for objects of the same process with a lower efficiency, these objects are never considered being a replacement candidate for the current object. Since these objects already got allocated onto the scratchpad, it is very likely to have to reallocate them on a different location in later steps. An example of this situation is shown in Figure 5. Each line represents the scratchpad allocation after each allocation step. For example, memory object 3 causes object 2 to be removed from the scratchpad, since its profit is smaller. During the next call of the SPMM, the same situation repeats; in that case memory object 1 is removed and memory object 2 is moved back to the scratchpad. It continues until all objects are on the scratchpad. In the end, there were many futile data movements which could have been avoided.

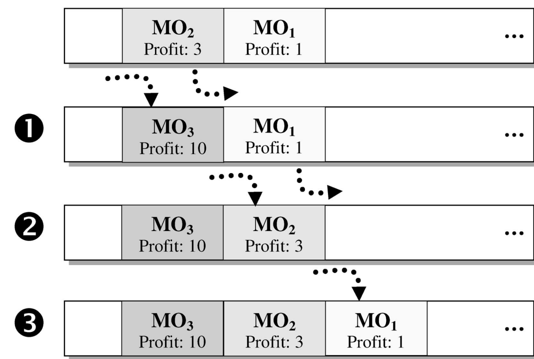

Figure 5: Futile reordering of objects of the same process.

Single Pass Allocation and *Triple Pass Allocation* strategies try to find a good allocation, while keeping the total runtime at the level of the static strategy. Similar to the first-fit and best-fit strategies, these strategies can remove unlocked memory objects of other processes from the scratchpad, but the idea here is to traverse the scratchpad only once for all candidate memory objects. In the single pass method, free areas and areas occupied by objects of other processes are treated equally, while in the triple pass method only free areas are considered first, then scratchpad areas occupied by objects of other processes and finally all unlocked objects are considered for replacement. For efficiency reasons, the passes are performed only once. An algorithmic description of the single pass allocation is presented in Algorithm 1.

4.2 Restoring Allocation Strategies

In contrast to the locking allocation strategies described in the previous section, a more relaxed handling of locked objects is performed here. In the previously described methods, these objects were never removed from the scratchpad. Now, these objects are also considered to be candidates for

Algorithm 1 *SinglePass*()

Require: MO_i a list of objects sorted by efficiency for process i.

1: $p = SPM_{start}$
2: **for all** s in MO_i **do**
3:   **while** $p < SPM_{end}$ && $(!IsSuitable(p,s) \parallel IsLockedObjectAt(p))$ **do**
4:     $p = NextArea()$
5:   **end while**
6:   **if** $p == SPM_{end}$ **then**
7:     **return** {End of SPM, stop allocation}
8:   **end if**
9:   **if** $IsOccupiedArea(p)$ && $!IsCodeObjectAt(p)$ **then**
10:     $MoveToMainMemory(GetMemoryObjectAt(p))$
11:   **end if**
12:   $MarkAreaFree(p)$ {Join free areas, if possible}
13:   $MarkAreaUsed(p,s)$ {Split area, if necessary}
14:   $MoveFromMainMemory(s)$
15:   $p = NextArea()$
16: **end for**

replacement. The downside is that this may introduce significant overhead, since all locked objects have to be restored at their original place when the process the objects belong to continues execution.

Dynamic Allocation directs the SPMM to use the entire scratchpad for each process. Objects of other processes are always removed first from the scratchpad. The advantage of this method is that there are no interferences between the processes, so from the point of view of predictable allocation and execution times, this method is superior to others presented here. Nevertheless, the copy cost will be quite high on each context switch. Intra-process allocation of objects to the scratchpad will be done in the same way the static allocation works. This is due to the fact that it is not beneficial to replace unlocked, less valuable objects of the same process. Objects of other processes won't be present on the scratchpad, anyway.

Chunk Allocation tackles two major disadvantages of the dynamic allocation; the excessive amount of additional buffers each processes needs for backup in the main memory and the high copy costs due to the complete write-back of the scratchpad on each context switch. Objects of each process are sorted by their efficiency value. Objects of highest values whose total size do not exceed the size of the scratchpad and whose copy overhead is less than the gains which could be achieved by placing them on the scratchpad are grouped together into a chunk. In the second step, this chunk is placed onto the scratchpad. First, a sufficiently large empty area is searched. If it is not available, the chunk is placed at the beginning of the scratchpad, forcing objects

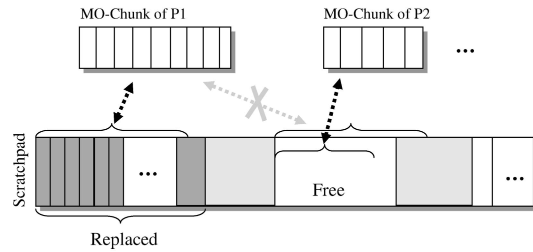

Figure 6: Placement strategy for the Chunk Allocation.

of other processes to be removed. Figure 6 depicts this situation. For the chunk P2, there is a sufficient large space on the scratchpad. For chunk P1, no suitable space has been found, therefore it will be placed at the beginning of the scratchpad memory. The dark grey areas depict the set of objects which have to be removed from the scratchpad, since they will be replaced by chunk P1. In the subsequent runs the chunk is placed at the same address it was placed first. This ensures that blocked objects keep their addresses.

In contrast to the dynamic method, the size of the backup storage in the main memory is reduced to the minimum required to store the objects placed on the scratchpad. This is not only beneficial in terms of memory utilization, but also in terms of copy overhead. Only the actually required amount of data is moved between scratchpad and main memory. Grouping of objects has the advantage of reduced overhead at further context switches. As long as no memory objects have been created or deleted, the set of objects belonging to a chunk does not change, therefore the overhead of finding and grouping these objects can be saved on subsequent context switches. Finally, the possibility of keeping chunks in the scratchpad across context switches may also save copy overhead.

5. Optimal Allocation Strategies

In addition to the runtime strategies presented in Section 4, an ILP based offline allocation strategy has been developed. It provides a baseline for the best possible scratchpad allocation to show the quality of the runtime approaches. To achieve this, the ILP based solution needs a global view over the entire runtime of the system. This helps to avoid locally optimal allocations which would prevent higher gains in future steps. Since there is a mutual dependency between the execution speed of each process and the scratchpad allocation, it is not possible to stick to fixed time slices, but context switches have to occur at exactly the same points of control the solution has been precomputed for. This is achieved by counting calls to SPMM functions and taking at runtime the precomputed actions assigned for a particular point of control.

The objective function to be maximized is defined as fol-

lows:

$$\sum_{t\in T}\left(\sum_{i\in I_t} x_{i,t}\cdot P_i\cdot T_{i,t} - \sum_{i\in I_t\cap I_{t-1}} \psi_{i,t}\cdot C_{MM\to SPM,SIZE_i} - \sum_{i\in I_t\cap I_{t-1}} \omega_{i,t}\cdot C_{SPM\to MM,SIZE_i}\cdot DATA_i - \sum_{i\in I_t\setminus I_{t-1}} x_{i,t}\cdot C_{MM\to SPM,SIZE_i}\cdot STAT_i\right) \to max$$

where $x_{i,t}$ is a binary decision variable denoting whether an object i is on the scratchpad at point of control t. For each point of control, there is a set of available objects I_t. $\omega_{i,t}$ and $\psi_{i,t}$ are binary decision variables stating whether to move an object at a particular point of control t. $\omega_{i,t}$ describes the movement from the main memory to the scratchpad. $\psi_{i,t}$ will be set to one if a memory object i has to be moved back to the main memory at a particular point of control. P_i denotes the constant profit of an object if it is placed on the scratchpad. $T_{i,t}$ is a binary constant set to one if the process to which object i belongs is being executed at point of control t.

Points of control are the time instants at which the scratchpad allocation could be changed. This may happen on context switches and whenever dynamic memory is being allocated or deallocated. To determine the sequential order of control points, the application and the SPMM are compiled for profiling. In that mode, all memory objects are kept in main memory. Nevertheless, the calls to SPMM functions have to be preserved, since the solution of the ILP has to be applied at a particular call. Additionally, some constants are defined for the ILP formulation. C constants denote the copy cost for moving objects of particular size between the scratchpad and main memory. Finally, the $DATA_i$ and $STAT_i$ binary constants describe the type of a memory object. $DATA_i$ is set to one for any kind of data objects. Additionally, $STAT_i$ is set to one for statically allocated data objects. Basically, the target function to be maximized states that only profits of objects that are on the scratchpad while the process they belong to is running have to be counted. However, the gains have to be decreased by the overhead due to object movement.

The constraints have to ensure that

- locked memory objects are not moved between scratchpad and main memory,
- the sizes of objects on the scratchpad do not exceed the total scratchpad size,
- objects have to fit completely onto the scratchpad,
- each scratchpad location is occupied by at most one memory object at a time,
- objects must not be moved within the scratchpad,
- the decision variables x, ω and ψ correlate to each other.

In the case of restoring allocation strategies, constraints have to be slightly modified,

- it is not necessary to prevent locked memory objects from being moved,
- locked objects have to be placed at the same location on the scratchpad,
- if a process is running, all locked objects which were on the scratchpad the last time the process was executed, have to be placed there again.

6 Results

Five different sets of benchmarks have been used to analyze the quality of the proposed allocation strategies. The benchmarks target a wide range of applications.

- **AUTO** benchmark is derived from the MiBench [5] Testsuite. It consists of benchmarks from the automotive and industrial domain: BASICMATH, BITCOUNT, QSORT and SUSAN have been used.
- **TELECOM** benchmarks consist of typical encoding tasks, like CRC32, FFT, IFFT, ADPCM and GSM.
- **MEDIA-** benchmarks consist of AV processing applications: ADPCM, G723 and EDGE-DETECTION
- **MEDIA+** benchmarks incorporates in addition to the MEDIA- the MPEG2 decoder.
- **SORT** is a collection of sorting algorithms. Included are five algorithms: BUBBLESORT, HEAPSORT, INSERTIONSORT, QUICKSORT and SELECTIONSORT.

Table 1 summarizes the benchmark sizes in bytes and the number of active processes. Applications of each Benchmark are run in parallel. Some applications consist of subtasks (i.e. ADPCM-Encode and ADPCM-Decode) which are also assigned to separate processes. The applications are executed under the control of the RTEMS operating system. The binary executables have been generated using the ARM-GCC compiler. The SPMM uses the list based memory object management techniques introduced in Section 4.

	Code size	Input size	Processes
AUTO	13936	15100	6
TELECOM	27552	24401	7
MEDIA-	7628	4864	5
MEDIA+	23300	11660	6
SORT	7044	6796	8

Table 1: Benchmark sizes and process counts.

The simulation runs were performed on the MPARM SoC simulator. The energy consumption and runtime cycles have been obtained from models provided by MPARM [9]. All experiments have been performed for various scratchpad sizes of 256 Bytes up to 16 kBytes. In the cache based experiments, a 2-way and 4-way set associative cache of the same size as the scratchpad was used. To avoid interferences due to increased size of available fast memory, the cache based experiments were run without a scratchpad. In all other runs, no caches were present in the system.

The results consist of two main sets of comparisons. The first set provides comparisons to the original application and an operating system setup without a scratchpad manager. The second set determines the quality of the heuristic methods relative to the ILP based method. Results presented in this paper concentrate on the energy consumption since this is the main optimization goal. Nevertheless, measurements of the runtime have also been performed.

Figure 7 shows the average deviation of energy consumption compared to the ILP based solution. Average values over scratchpad sizes of 256 Bytes up to 4k Bytes have been computed for each benchmark. Additionally, overall average values are also included in the diagram. Especially the restoring strategies perform very well, achieving an average deviation of about 6%. The restoring strategies perform on average better than the blocking ones, because it is quite cheap to move data around in our target architecture. The benefit which can be achieved because of the additional free space on the scratchpad often countervails the additional copy costs. Additionally, a second effect contributes to this result; keeping blocked objects on the scratchpad means to split the free space into many smaller areas, and therefore the computational effort needed to find a good solution increases. The first-fit strategy has a very poor performance especially for SORT, because it allocates the memory objects in a really straight forward way. It has been implemented, because in previous work for dynamic storage allocation in main memories, this kind of heuristics could achieve very good results. This is not true in the context of a limited size, highly utilized scratchpad. The chunk allocation performs best, because it combines the advantage of the dynamic allocation with the possibility to keep some objects in the upper regions of the scratchpad across multiple context switches and additionally save copy overhead this way.

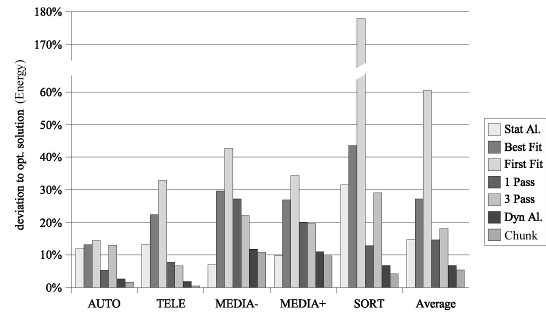

Figure 7: Deviation of energy consumption compared to ILP based solution.

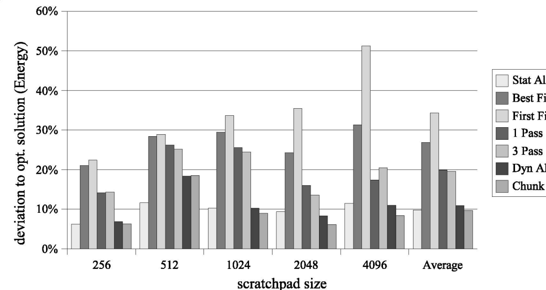

Figure 8: Deviation of energy consumption compared to ILP based solution for MEDIA+.

A comparison for scratchpad sizes larger than 4kBytes could not be performed for all benchmarks because of the excessive increase in the ILP solver's runtime.

Detailed measurements have been performed for all benchmarks. A similar behavior has been observed in all cases. Therefore, only the MEDIA+ benchmark is presented in detail here. Figure 8 shows the comparison of heuristic methods with the ILP based approach. Restoring strategies perform well for all scratchpad sizes, while the first and best fit methods always show the highest deviation. Especially in the case of the best fit method, the additional overhead due to the search for a best suited space does not pay off. The first fit method performs even worse. This method unnecessarily often removes objects of other processes from the scratchpad while free space may be available in other regions of the memory.

Figure 9 shows the relative energy consumption for the MEDIA+ benchmark. A system without a scratchpad memory manager running the MEDIA+ benchmark represents the 100% baseline. The highest energy savings were achieved by the static allocation for a 16k scratchpad. The energy consumption was reduced by 58%. Except for this particular setup, the chunk allocation achieved always the highest savings. Using that allocation strategy the energy consumption could be reduced by 49%. Besides the fact that with increasing scratchpad size the energy consumption goes down, also the highest energy reduction shifts

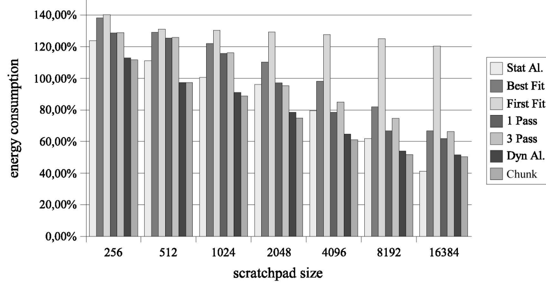

Figure 9: Energy consumption for MEDIA+ benchmark.

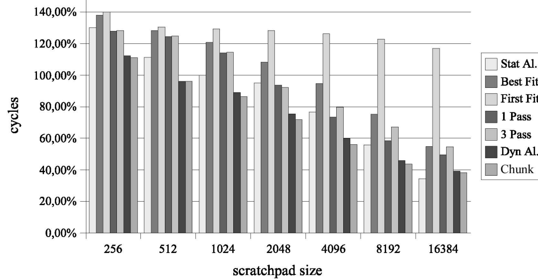

Figure 10: Runtime for MEDIA+ benchmark.

from dynamic methods for small memories to the very simple static method for sufficiently large memories. For other benchmarks, a saturating effect could be observed. Starting from a particular size, all allocation strategies except First Fit achieved similar cycle and energy savings.

The required amount of cycles to run MEDIA+ is shown in Figure 10. The overall observations to the savings in energy consumption also apply to the reductions in cycle counts. The highest runtime reduction also could be observed for the static allocation for a 16k scratchpad. The runtime was reduced by 65%. Using the chunk allocation strategy the runtime could be reduced by 61%. The strong relation between energy consumption and runtime requirements exists due to the properties of the target architecture. Program execution is faster when scratchpad is being utilized, so the total CPU energy consumption goes down. Additionally, accesses to scratchpad need less wait cycles as well as less energy.

The results for the SORT benchmark are shown in Figure 11. They include the energy savings that could be achieved by incorporating caches into the system. To keep the results comparable to other diagrams, the baseline of 100% represents the energy consumption of the original code run on a system without caches. According to this figure, the SPMM based approach is capable of outperforming a cache equipped system for the SORT benchmark. The optimized energy consumption was reduced by 83% compared to the original for a scratchpad size of 4k Bytes. The cache based system was able to achieve a peak energy reduction of 75% for a 2k 2–way cache. Compared to the cache based system, better performance of the SPMM has not been observed for all benchmarks. That is due to the fact that caches intercept access to all memory locations, while the SPMM based methods currently do not take into account the accesses performed in the operating system and those performed in standard libraries.

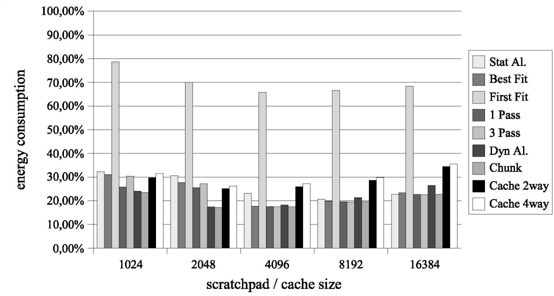

Figure 11: Comparison of SPMM to Caches for SORT benchmark.

7 Conclusion

This work presents a new approach to integrate scratchpad memory support into the operating system. In contrast to previous work, this approach is based on transformed input programs. Transformations can be done automatically, it does not affect the development process. Various allocation strategies have been studied. The results show that for current scratchpad sizes, it may be beneficial to risk higher copy costs instead of using runtime intensive allocation strategies. Also, usually the gain due to the additional free space exceeds the overhead of removing and restoring objects on the scratchpad. The main limitation of this approach is that currently, neither the RTEMS operating system nor the standard libraries participate in the competition for free space on the scratchpad. Therefore, extending this approach in that direction would probably lead to higher energy savings, and would definitely outperform caches for any kind of applications. From the point of view of each single process this approach resembles a hardware based cache in terms of predictability. But in contrast to caches, the software based implementation of SPMM allows for a better performing, highly configurable, jet dye area saving way of implementing on chip data and code caching.

References

[1] F. Angiolini, L. Benini, and A. Caprara. Polynomial-time algorithm for on-chip scratchpad memory partitioning. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 318–326. ACM Press, 2003.

[2] F. Angiolini, L. Benini, and A. Caprara. An efficient profile-based algorithm for scratchpad memory partitioning. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 24(11):1660–1676, 2005.

[3] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory : A Design Alternative for Cache On-chip memory in Embedded Systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign (CODES)*, 2002.

[4] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide To the Theory of NP-Completeness*. Freeman, 1979.

[5] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th Annual IEEE International Workshop on Workload Characterization (WWC)*, pages 3–14. IEEE, 2001.

[6] ICD - Informatik Centrum Dortmund e.V. ICD-C Compiler framework. `http://www.icd.de/es/icd-c/icd-c.html`, 2006.

[7] M. T. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. Dynamic Management of Scratch-Pad Memory Space. In *Proceedings of the 38th DAC conference*, pages 690–695. ACM Press, 2001.

[8] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power aware page allocation. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 105–116. ACM Press, 2000.

[9] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon. Analyzing On-Chip Communication in a MP-SoC Environment. In *Proceedings of the 7th DATE conference*. IEEE, 2004.

[10] S. Mamagkakis, D. Atienza, C. Poucet, F. Catthoor, D. Soudris, and J. M. Mendias. Automated exploration of Pareto-optimal configurations in parameterized dynamic memory allocation for embedded systems. In *Proceedings of the 9th DATE conference*, pages 874–875. EDAA, 2006.

[11] OAR Corporation. RTEMS Homepage. `http://www.rtems.com/`, 2003.

[12] P. R. Panda, N. Dutt, and A. Nicolau. *Memory Issues in Embedded Systems-On-Chip*. Kluwer Academic Publishers, 1999.

[13] F. Poletti, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. M. Mendias. An integrated hardware/software approach for run-time scratchpad management. In *Proceedings of the 41th DAC conference*, pages 238–243. ACM Press, 2004.

[14] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Proceedings of the 5th DATE conference*, pages 409–417. IEEE, 2002.

[15] M. Verma, K. Petzold, L. Wehmeyer, H. Falk, and P. Marwedel. Scratchpad sharing strategies for multiprocess embedded systems: A first approach. In *Proceedings of 3rd Workshop on Embedded System for Real-Time Multimedia (ESTIMedia)*. IEEE, 2005.

[16] M. Verma, L. Wehmeyer, and P. Marwedel. Cache-aware scratchpad allocation algorithm. In *Proceedings of the 7th DATE conference*, pages 1264–1269. IEEE, 2004.

[17] L. Wehmeyer, U. Helmig, and P. Marwedel. Compiler-optimized usage of partitioned memories. In *Proceedings of the 3rd workshop on Memory performance issues (WMPI)*, pages 114–120. ACM Press, 2004.

[18] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management (IWMM)*, 1995.

[19] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *IEEE Computer Architecture News*, 23(1), March 1995.

[20] Q. Zhao, R. Rabbah, and W.-F. Wong. Dynamic memory optimization using pool allocation and prefetching. *SIGARCH Computer Architecture News*, 33(5):27–32, 2005.