Power efficient data management for dynamic applications

P. Marchal[1] J.I. Gomez[2] D. Atienza[2] S.Mamagkakis[3] F.Catthoor[1]

[1] *IMEC and Katholieke Universiteit Leuven, Belgium*

[2] *Universidad Complutense de Madrid, DACYA, Spain*

[3] *Democritus University of Thrace, Greece*

Abstract

In recent years, the semiconductor industry has turned its focus towards heterogeneous multi-processor platforms. They are an economically viable solution for coping with the growing setup and manufacturing cost of silicon systems. Furthermore, their inherent flexibility also perfectly supports the emerging market of interactive, mobile data and content services. The platform's performance and energy depend largely on how well the data-dominated services are mapped on the memory subsystem. A crucial aspect thereby is how efficient data is transferred between the different memory layers. Several compilation techniques have been developed to optimally use the available bandwidth. Unfortunately, they do not take the interaction between multiple threads running on the different processors into account, only locally optimize the bandwidth nor deal with the dynamic behavior of these applications. The contributions of this chapter are to outline the main limitations of current techniques and to introduce an approach for dealing with the dynamic multi-threaded of our application domain.

1 The design challenges of media-rich services

Business analysts forecast a 250 billion dollar market for media-rich, mobile wireless terminals [53]. These systems require an enormous computational performance (40GOPS[1]). Even though current PCs offer this performance requirement, they consume too much power (10-100W). Mobile devices should consume at least two or three orders of magnitude less power [30]. Furthermore, they should be cheap to successfully penetrate the consumer market. Consequently and in spite of the design issues, the engineering and manufacturing costs need to be reduced. Industry strongly believes that platforms are a potential way to meet the above challenges.

1.1 The era of platform-based design

A platform is a fixed micro-architecture together with a programming environment that minimizes mask-making costs and is flexible enough to work for a set of applications [4]. The production volumes can then remain high over an extended chip lifetime.

Given the strong energy constraints, we must choose the flavor of these platforms. Since power is cubic to the processing frequency, parallelism is an effective to reduce power and energy consumption. Then, multiple simple processors are preferred to one complex speculative and out-of-order processor. In the right application domain, we can get better performance and spend less energy. Besides parallelism, heterogeneity is an alternative way to decrease the energy cost. For instance, the TI OMAP platform combines a RISC processor with a Digital Signal Processor (DSP). The RISC is more energy-efficient for the input/output processing and simple control-dominated

[1]Giga Operations Per Second

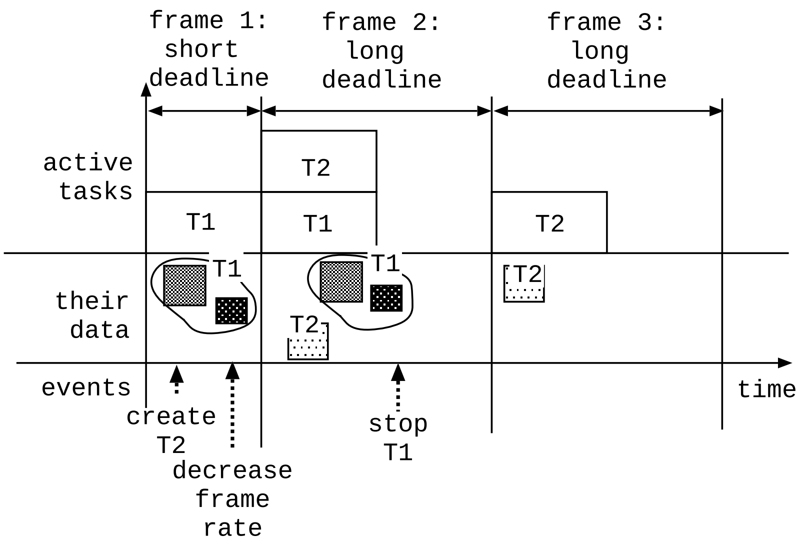


Figure 1: Characteristics of our application domain

applications. The DSP, on the other hand, provides the computational performance for audio and video processing, while keeping the energy cost bounded.

Taking a look to the market, it is clear that heterogeneous multi-processor platforms are conquering the world of low power embedded systems: ST Nomadik [40], Philips Nexperia [42], TI OMAP [22].

1.2 The desire for media-rich services

Platforms perfectly support the next wave of media rich, wireless applications, bound to flood the multi-billion dollar consumer market. Typical applications are media-players such the MPEG4 IM1 player.

We summarize the most important characteristics of the application domain in Fig. 1:

- *multi-threaded*: The systems contain multiple tasks which can execute in parallel. The tasks can either be independent or dependent. In figure 1, the system contains two parallel tasks (T1 and T2).
- *a closed system*: The entire set of possible tasks it is known at design time (i.e. we know the source code of every task to be executed in the system). However, the start time of each task an the exact instances of task being executed at a precise instant, it is only known at run time. User interaction and data dependent conditions introduce non-deterministic behavior in the system, making it impossible to accurately predict which tasks will be executed in parallel. We assume that no tasks can be downloaded on the system (such as e.g., Java applets or other software agents). For our example, this entails that no other types of tasks but T1 and T2 can occur at runtime. Conceptually, it is feasible to extend our framework and methodology for open systems, but we leave this for future work.
- *time-constraints*: Tasks within multi-media applications are usually bound to time-constraints. The most common deadline is the frame-rate (see above). To have a fluid video display the tasks of a thread-frame have to finish within its deadline. We indicate the deadline imposed by the frame-rate on our application in figure 1. In the first frame, we use a high frame-rate, i.e. a tight deadline for T1. Thereafter, an user event relaxes the frame-rate. In the remainder of this text, we mainly focus on the frame-rate, despite other deadlines will in practice also occur.

- *tasks are control/data flow graphs*: Each task is a control/data flow graph. Hence, parts of a task may be conditionally executed. As a result, which data and how frequently it is accessed may significantly vary at runtime. We take as a premise that at the start of each task we know how much memory it needs. The memory space can be used for the static data or as a heap for runtime allocated data. For instance, we assume that T1 requires two data structures whereas T2 only needs one.
- *data-dominated*:. The tasks are data-dominated. As a result, the energy of the data memory architecture dominates the system cost. On multi-media systems, this assumption is particularly true after the cost of the instruction memory hierarchy is optimized (e.g., with [1][32]). The data memory cost is then usually the remaining energy bottleneck. Consequently, optimizing the data memory is the top priority, even if it afterward slightly increases the processing energy consumption.

In the next subsection, we discuss the main challenges to integrate these applications on an embedded platform.

1.3 Memories rule power and performance

The memory system is an important contributor to the performance and power consumption of embedded software, particularly for multimedia applications [13][65]. The most well known technique for improving the performance of the memory subsystem is introducing a layered memory architecture. Large memories used to store multi-media data have long access times. Therefore, they are too slow to supply data at a sufficient rate to the processing elements. As a result, the processing elements stall, thereby wasting time and energy. To improve the performance and reduce the energy cost, designers create a layered memory hierarchy. Each layer contains smaller memories to buffer the data that is frequently accessed by the processor.

In this work, we focus on how to exploit a layered memory architecture. Particularly, we optimize the available bandwidth to the multiple memories/banks of each layer. This problem consists of detecting a data assignment and instruction schedule that satisfy all time constraints while minimizing the energy consumption. Despite many techniques already exist for this problem (section 3), they improve the bandwidth within a basic block and assume that the memories are accessed by a *single thread*. Moreover, these approaches require that access pattern of the application can be analyzed at design-time. Unfortunately, in our application domain multiple threads often share memory resources. Furthermore, the user determines which threads are running. As a consequence, we can only characterize the access pattern at runtime. We will show in sections 4.1.1, 4.2.1 and 5.1 that existing techniques breakdown under these circumstances, resulting in energy and performance loss.

We will overview the techniques which we have developed to overcome the above limitations. We have investigated on the one hand design-time techniques for globally optimizing the memory bandwidth, even across the tasks' boundaries (sections 4.1.2 and 4.1.3 for the shared layer and section 4.2.2 for the local layer). On the other hand, we have developed a combined design-time approach for dealing with the dynamic behavior (section 5). It makes run-time decisions based on an extensive design time analysis phase. Finally, we present how these run-time decisions can be energy-efficiently implemented at run-time (section 6). Before introducing our approach, we explain the memory architecture targeted throughout this text (section 2) and the related work more in detail (section 3).

2 Target architecture

During our research, we concentrate on a generic target architecture (figure 2). Different processing tiles contain multiple processing elements that share a local memory layer. The processing elements within the same tile are closely synchronized. A processing tile could be for instance a VLIW or a simple RISC processor (like on a TI OMAP). The local memory layer on a processing tile may comprise of multiple scratch pad memories/banks. Again

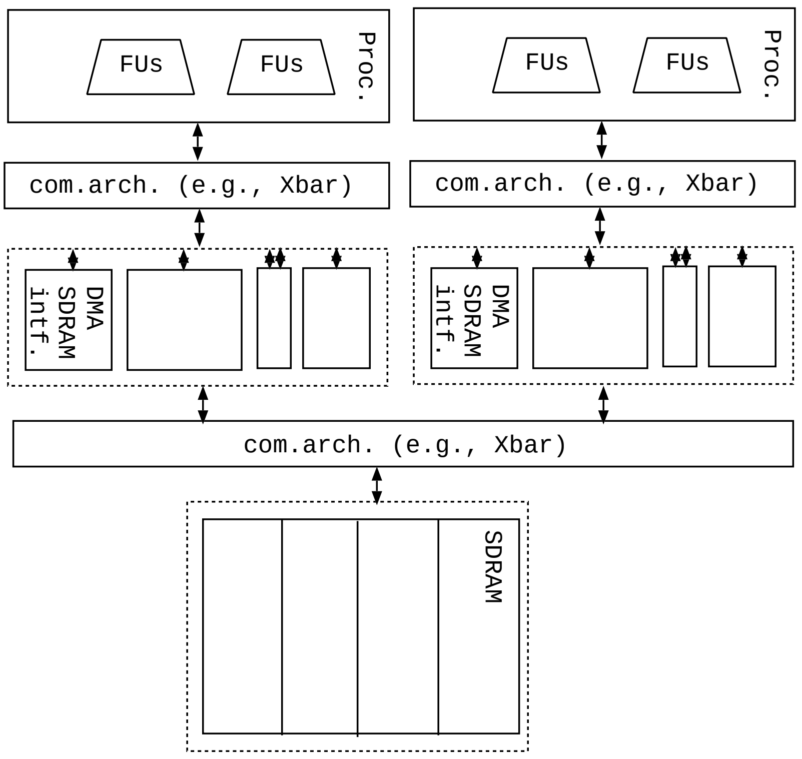


Figure 2: Target architecture for bandwidth optimization

this closely resembles ST LX [16] or TI C6X [22] DSPs where up to eight memories are included in the local layer. We do not directly exploit cache memories, but focus on scratch pad memories. These software controlled memories do not require complex tag-decoding logic [5]. Therefore, they have a lower energy cost per access compared to caches, and also reduce the indeterminacy of the system. To further reduce the global energy cost, we assume that they are heterogeneous: they can have different sizes, different number of ports and access time.

Furthermore, the processing tiles share an off chip SDRAM (like on the TI OMAP or Philips Nexperia). We include the SDRAM in our overall target architecture, because it may consume up to 30% of the system energy cost of a commercial hand-held device [37].[2]

We integrate a cross-bar as communication architecture both between the processing elements and the local layer as between the local layers and the shared SDRAM. Although a cross-bar is not the most energy-efficient architecture, its energy cost is currently limited to only 10% of the global data transfer cost.[3] With this configuration, the communication architecture does not have any impact on the achievable performance, but the ports of the different memory layers (local memories and SDRAM) become potential bandwidth bottlenecks.

3 Surveying memory bandwidth optimization

Memory bandwidth optimization is a widely researched topic. Most of the related work is focused on improving the bandwidth of a single memory layer. This layer can either consist of multiple SRAMs or a large SDRAM memory (figure 2). We discern two methods which are commonly applied/combined to optimize the memory bandwidth: data layout transformations and instruction scheduling techniques. Data layout transformations may comprise

[2]This percentage is for a complete system including speakers, LCD, etc.

[3]In principle, a more scalable communication architecture could be programmed or synthesized (such as e.g., [51]) However, research on advanced communication architectures fall outside the scope of this text.

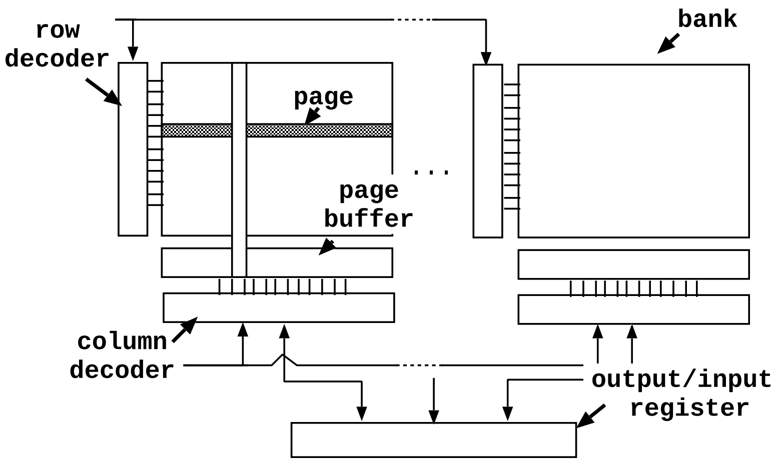


Figure 3: Multi-banked SDRAM architecture

several optimizations: from deciding the optimal address in memory (and thus, the optimal memory module) for each application variable to the array elements relative order in memory. Variables lifetime can also be exploited to fully exploit the available memory space. Instruction scheduling tries to find a memory access ordering that optimizes an specific cost. The goal is usually to increase the system's performance. Only a few methods exchange the performance gains for energy savings. In the next subsections, we outline them for SDRAMs and for the local memory layer.

3.1 SDRAM bandwidth

SDRAMs are mostly used for storing large multimedia data. The access time and energy cost of an SDRAM heavily depend on how it is used. In general, an SDRAM consists of several banks (figure 3). Fetching or storing data in an SDRAM involves three memory operations. An activation operation decodes the row address, selects the appropriate bank an moves a page/row to the page buffer of the corresponding buffer. After a page is opened, a read/write operation moves data to/from the output pins of the SDRAM. Only one bank can use the pins at a time. When the next read/write accesses hit in the same page, the memory controller does not need to activate the page again (a *page hit*). The application can read these data elements at a lower access latency and lower energy cost. However, when another page is needed (a *page miss*), precharging the bank is needed first. Only thereafter the new page can be activated and the data can be read. Note that pages from different banks can be opened simultaneously. We can then interleave the access among banks in order to minimize the number of page-misses. The less page-misses occur, the better the performance and energy consumption of the SDRAM become. Most methods below focus on transforming the application such that page-misses are avoided.

3.1.1 Data layout transformations and data assignment techniques

For a fixed access schedule, the layout of the data in a memory bank defines how many page-misses occur (figure 4). To illustrate this, we map the scalars a,b,c,d,e,f in two different ways onto the pages of an SDRAM bank. If a memory operation accesses an open page, a page-hit occurs (H). If, on the other hand, the next operation reads/writes to another page, a page-miss happens (M). E.g., in the first layout, an access to c after one to a results in a page-hit, while an access to e after one to a causes a page-miss. Given the presented access sequence, four page-misses occur in the left layout. If we change the data layout, we can reduce the number of page-misses. E.g., when we move e to the first page and b to the second one, only two page-misses remain (figure 4-right). Furthermore, it reduces

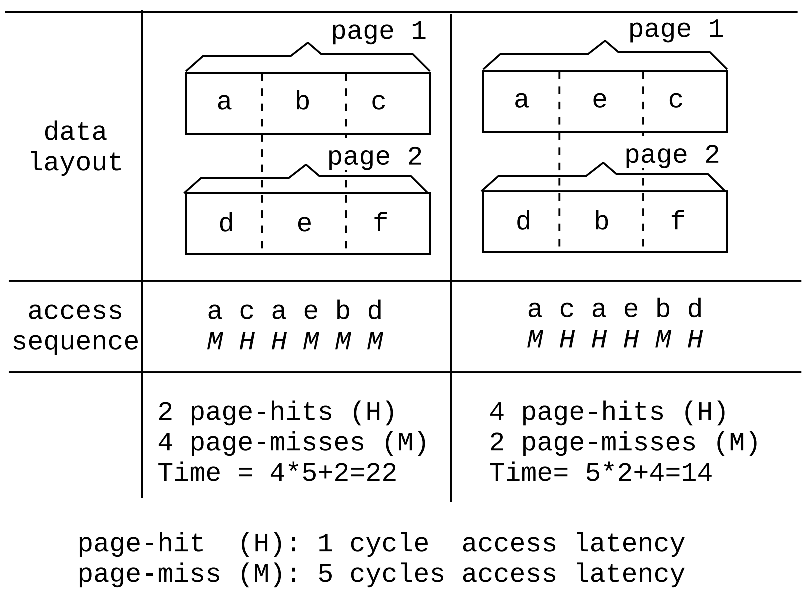


Figure 4: Different data layouts impact the number of page-misses

the execution time from 22 to 14 cycles. Since the data layout has such a large impact on the performance, several authors have proposed techniques to optimize it. [55] partitions arrays into tiles, each fitting into a single page. The tiles are derived such that the number of transitions between the tiles, and thus the number of page-misses, is minimized. [66] proposes to layout the scalar variables inside the program text, reducing the overall page-misses.

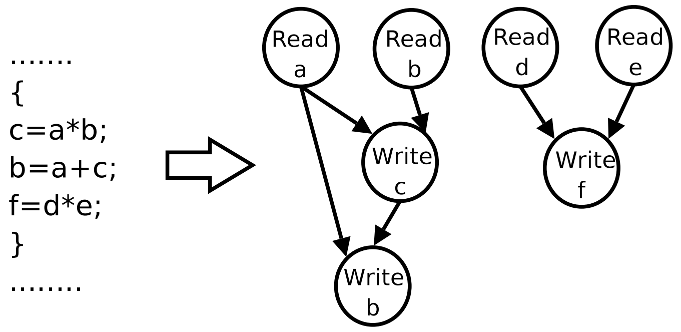
In contrast with the older DRAM architectures, most SDRAMs nowadays have more than one bank. E.g., the Rambus' SDRAMs have up to 32 banks. Multiple banks provide an alternative way to eliminate page-misses. For instance, [20] distribute data with a high temporal affinity over different banks such that page-misses are avoided. Their optimizations rely on the fact that the temporal affinity in a single-threaded application is analyzable at design time.

Thus, despite data assignment techniques exist for limiting the page-miss penalty, they are restricted to single-threaded, design-time analyzable tasks. As we will motivate in section 5.1, these techniques breakdown for dynamic multi-threaded applications.

3.1.2 Memory access reordering techniques

The access order also influences the number of page-misses. Consider the code of the basic block shown in Fig. 5-a. Its data flow graph is also shown. After data dependence analysis, several memory access schedules are feasible (just two of them are shown). However, as depicted in the figure, the choice impacts the number of page-misses (and thus, the performance and energy consumption). We assume the left data layout in Fig. 4. The data dependence analysis reveal that read accesses to a,b, d and e may performed in any order. To hide the multiplication latencies,we may opt to generate the top schedule of Fig. 5-b. It causes 5 page-misses out of 7 accesses; reordering the accesses as shown in the second option helps to reduce the number of page-misses to just 2. The potential performance improvement derived from the first schedule will very likely become a time penalty because of the extra page-misses. Moreover, the total energy consumption will be significantly larger.

As we will see in Section 4.1.1 for multi-threaded contexts, memory accesses from different concurrent tasks interfere with each other. As a consequence, task scheduling is a higher level way of changing the final access ordering.



```
.......
{
c=a*b;
b=a+c;
f=d*e;
}
........
```

(a)

possible access sequences	a b c b d e f M H H H M H H	5 page-hist (H) 2 page-misses (M) Time = 5*2+5 = 12
	a b d e c f b M H M H M M M	2 page-hist (H) 5 page-misses (M) Time = 5*5+5 = 27
	page-hit (H): 1 cycle access latency page-miss (M): 5 cycles access latency	

(b)

Figure 5: Access order impacts the number of page-misses

We may classify the existing work in this area in two main threads: hardware approaches (trying to reorder accesses through smart memory controllers) or by software approaches (that rely on the compiler to perform code transformations and instruction scheduling optimizations). Several authors ([56], [17]) propose hardware controllers to reorder the accesses. Typically, they buffer and classify memory access operations according to their type (precharge, row memory accesses and column memory accesses) and according to the accessed bank and the row. The hardware logic of the memory manager selects from this classified set which operation to execute first. Since we focus in low-power design, we strive to simplify the hardware to the bare minimum and put the complexity of our designs as much as possible in the design-time preparation phase (subsection 5.2). In this way, we avoid the extra hardware which increases the energy consumption of all memory accesses.

Several software approaches have been presented too. [47] exposes the special access modes of SDRAM memories to the compiler. As a result, their scheduler can hide the access latency to the SDRAMs. The work was started in the context of system synthesis, but later on extended to VLIW compilers [41]. Finally, [9] combines the scheduling technique of [47] with the memory energy model of [61] for reducing the static SDRAM energy.

The above existing techniques rely on the fact that the access pattern can be analyzed at design time for single-threaded applications. This is not the case in our application domain. Dynamism and data-dependent control flow in modern applications makes quasi unpredictable the final access pattern of a single thread. Things become more complex in the multi-thread context: memory accesses from different threads are interleaved. Currently, no techniques analyze the access pattern across threads. Moreover, the dynamic behavior of some multi-threaded applications further complicates the problem: the active task-set (set of tasks executing simultaneously) is only known at run-time. Therefore, it is impossible to predict the inter-tasks memory access interactions at design-time, since we cannot even know which tasks will be executed in parallel!

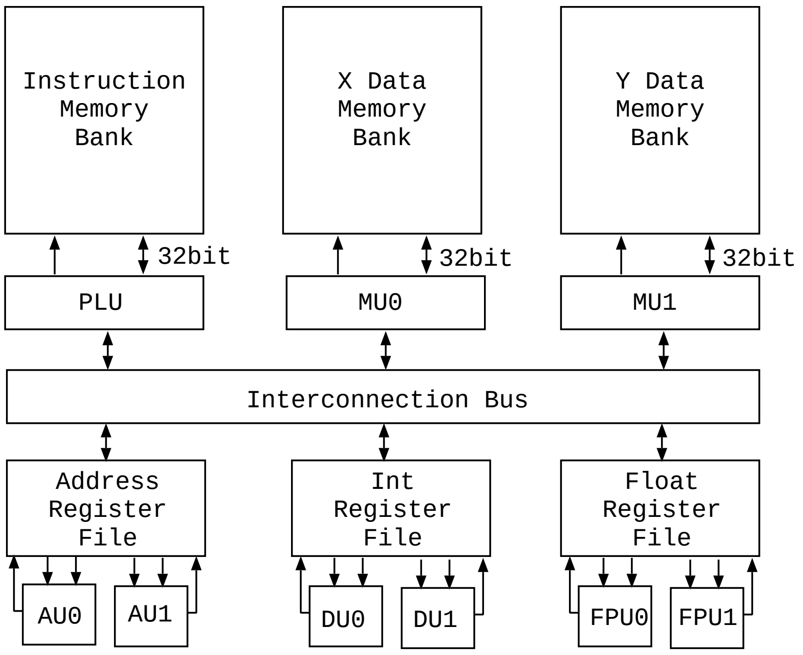


Figure 6: A VLIW architecture borrowed from [35]

3.2 Bandwidth to the local memory layer

Complementary to the SDRAM layer, memory bandwidth optimization has also been researched for the local memory layer. Their optimization objective is mostly reducing memory area/energy while guaranteeing performance. In this subsection, we discern again techniques which only change the data assignment and the ones which combine it with instruction scheduling.

3.2.1 Data layout based techniques

In the synthesis community, many techniques were developed for synthesizing a memory architecture with provides sufficient memory bandwidth, but is energy or area efficient too (e.g., [21] and [8]). They generate a memory architecture and decide on the data to memory assignment in a single step. As a consequence, this makes them not directly applicable for predefined memory architectures (such as on ASIPs or DSPs).

Modern DSPs usually have a local memory layer which consists of multiple SRAM memories. Let's consider the architecture in figure 6 as an example. It has two single ported memory banks (X,Y) which can be read in parallel. Most compilers would model this memory layer as a monolithical memory with multiple ports. Under this assumption, data layout has almost no influence in the potential performance. The compiler will schedule in parallel as many memory operations can as load/store units exist on the architecture. Since the compiler is not bank-aware, it will even schedule accesses to the same memory bank in parallel. Although this simplifies the instruction scheduling, special hardware at runtime needs to serialize the parallel accesses to the same memory resource. The DSP is then stalled and performance is lost. Several authors therefore expose the local memory architecture to the linker.

A conscious data layout may help to alleviate this problem. [35] maximizes the performance by carefully distributing the data across the different memories. In this way, it ensures that as many accesses as possible can be executed in parallel. Some very recent approaches ([36]) dynamically reallocate data in the local layer. At compile time they perform a life analysis of the task and insert instructionc to dynamically copy code segements and variables onto the scratchpad at runtime. They report energy reductions up to 34% compared to static allocation. However their technique cannot efficiently handle dynamic applications. A static analysis of a dynamic application cannot reaveal which data will be accesses at any point of the task. Furthermore, multi-tasked environments, where the local layer is shared between several tasks, are not considered at all.

3.2.2 Access order

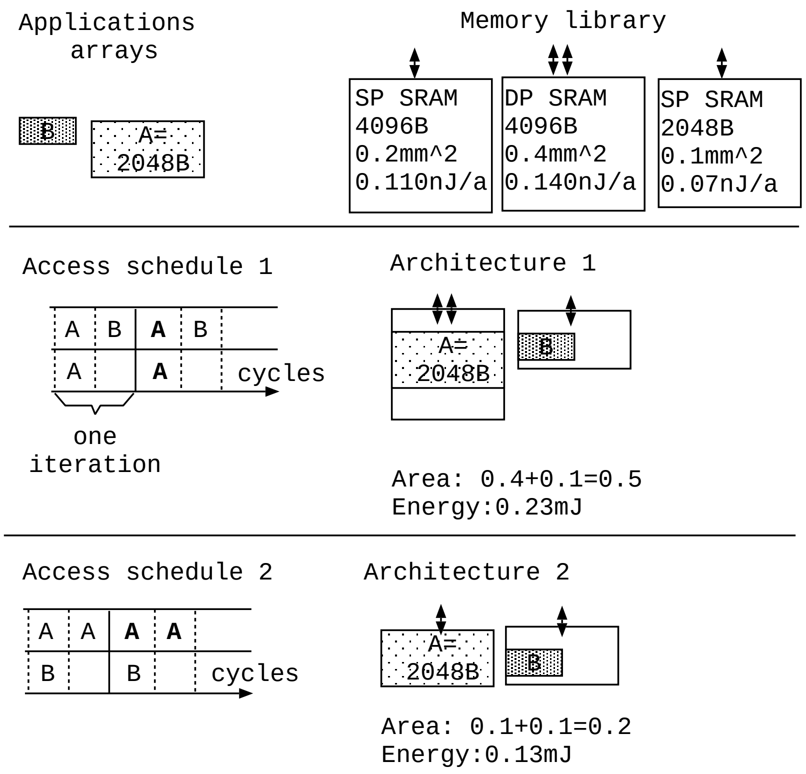


Figure 7: Reducing the memory cost with access ordering

The memory accesses order has also an important impact on the performance. and energy consumption. Indeed, changing the access ordering may allow to achieve the same performance with a more energy efficient memory system. Let's see how with an example (see figure 7). The application one small array (B) and a bigger one (A). The memory system may consist of several instances of any of the modules in the *Memory library*. Considering the access schedule 1, the architecture 1 is the most energy-efficient that enables the schedule. However, the dual port memory has an important impact on the energy consumption of this architecture. By rescheduling the memory accesses of the inner loop, we can eliminate the need for this memory (see architecture 2 in figure 7). It retains the same performance, but both data structures can now be mapped in a single port memory, thereby reducing the energy cost from 0.23mJ to 0.13mJ. From this example, it is clear that data layout and access scheduling are very effective in lowering the architecture cost. Because both techniques are so closely coupled, several authors propose to optimize memory layout and access ordering together.

An example is [57]. It optimizes the memory bandwidth in a separate step before compilation, thereby outputting a (partial) data assignment which constrains the final instruction scheduling. It guarantees that enough memory bandwidth exists to meet the deadline, while remaining as energy-efficient as possible. This technique optimizes the storage bandwidth within a basic block for memories with a uniform access time. of the application. An extension to this work ([14]) indicates how this technique, initially developed for a system synthesis, can also be used on existing processor architectures. But in both cases, they only reorder the memory accesses within the scope of a basic block.

More global optimization techniques can further improve the performance. In the past, several authors have proposed techniques to globally schedule instructions to parallelize code [28], but they do not consider how to optimize the memory bandwidth. [63] defines an operation schedule which reduces the number of memory ports. However, it does not take into account which data structures are accessed or how they are mapped onto the memory.

In summary, the main limitations of the above bandwidth optimization techniques for both the local and the shared memory layer are:

1. *single-threaded applications*: they optimize the memory bandwidth for a single task at a time. As a result, we cannot directly use them in our context, since we want to optimize the bandwidth across multiple tasks (e.g., on the shared SDRAM layer).

2. *static applications*: the above data layout/assignment techniques obtain information on the locality for the data at design time. The locality depends on which tasks are executing in parallel. Since in our application domain the actual schedule is only known at run time, we can no longer extract it at design time.

3. *no global optimization*: the existing techniques only reorder the memory accesses within the scope of a basic block. No optimizations across the boundaries of the basic blocks are systematically applied. As we will show for the local memory layer (section 4.2), this significantly reduces the potential performance gains and energy savings.

For our application domain and target architecture, several extensions are clearly needed for dealing with multiple threads and coping with the dynamic behavior. We discuss now techniques which optimize the memory hierarchy across the boundaries of a single task (subsection 3.3) and overview the techniques for managing dynamic behavior (section 3.4).

3.3 Memory optimization in multi-threaded applications

Different design communities have researched the influence of the communication architecture and memory subsystem on the performance of a multi-threaded application.

A large body of research exists in the high-performance computing domain on parallelizing applications while reducing the communication cost (e.g., the SUIF-project [19] and the Paradigm compiler [6]). However, they target an architecture which is very different from ours. E.g., they rely on complex hardware to guarantee data coherency and consistency, which may come at an important energy penalty. Furthermore, their techniques only work for statically analyzable code and cannot cope with runtime variations which are typically present in modern multi-media applications. These limitations render this prior-art not directly applicable to our context.

Also in the embedded system's context many authors have studied multi-threaded applications. [27] proposes a top-down hierarchical approach, compiling code on a heterogeneous multi-processor. The main disadvantage of this approach is that they use a synchronous data-flow model. It covers only a limited application domain and is not sufficient for our target domain.

Finally, since the middle of the last decennium, multi-processors systems have been widely researched in the system-level design community. Most techniques explore how to combine IP-blocks such that the system cost (albeit performance, energy or area) is reduced. In this context, the ordering and assignment of the tasks to the processing elements plays an important role in the system's performance (see [12] for an overview). However, in recent years, the energy consumption has become an important bottleneck too. When energy is considered at all in task scheduling, the focus has been on the processing cores. [52] presents a complete methodology to map multi-tasked and multimode applications onto heterogeneous multi-processor platforms. They focus on task and communication mapping, tasks scheduling and dynamic voltage scheduling. However their model does not specifically include the memory system.

They incorporate communication costs between tasks, but this information is not enough to efficiently optimize the memory hierarchy.

Unfortunately, only limited research exists in reducing the energy cost of the memory system. [10] and [31] describe both a heuristic which does allocation, assignment, scheduling of multiple task-graphs. [24] and [58] compile a task-graph on a given heterogeneous architecture. They explicitly model the memory system, interconnect and processing elements. The algorithm answers the following question: is it better to distribute the data (at a higher communication cost) or to keep data local (at a higher local memory cost). The above approaches use a naive memory architecture model and hardly incorporate the real behavior of the interconnections and memories.

Recently, [33] discuss how many processors are required to execute code as energy-efficiently as possible. The task interaction is empirically accounted for (based on simulation), but this is not a scalable approach. [25] partitions the data space of a linked binary. Each part is then mapped onto a memory bank. It selects the partition which optimizes the energy cost compared to a dual port memory. The performance of each partition is not accurately estimated since the technique does not account for memory stalls.

We identify the following limitations to the above techniques:

1. they target an architecture which is either too different from ours or is not detailed enough. As a result, we cannot reuse them to optimize the interaction between parallel executing tasks.
2. their program model is too limited for our application domain in which dynamic behavior plays an important role too.

In the next section, we review the current techniques for coping with the dynamic behavior.

3.4 Runtime memory management

Dynamic applications are slowly becoming desirable in the context of embedded systems. The unpredictability generated by the dynamism entails the usage of run-time policies for effective optimizations. These policies must be implemented efficiently to minimize the resulting overhead (figure 8).

As indicated in the previous sections, for memory bandwidth optimization, the policy making consists of scheduling the tasks (or their instructions) and (re)distributing their data across the available memories (step 1). To efficiently implement these decisions, we need to manage the memory space at runtime (step 2). In this section, we overview both the runtime decision taking and implementation techniques.

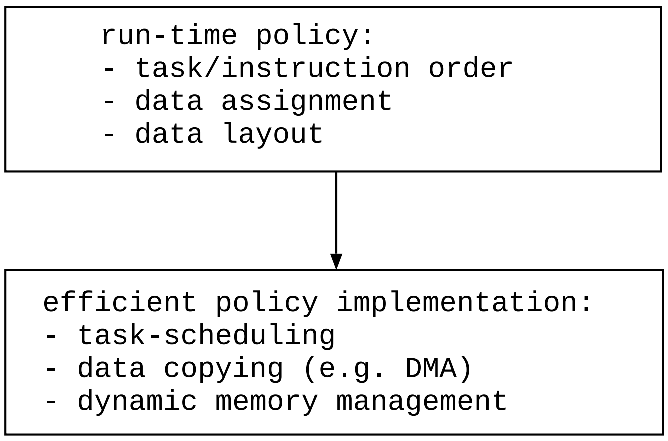


Figure 8: Runtime memory optimization decomposed in two problems: decision taking and implementation

3.4.1 Runtime policies

In the context of embedded systems, only few techniques decide where to store the data at runtime taking the memory architecture into account. For instance, [59][5] decide at design time for each call-site to malloc/new to which memory the data should be assigned. They base their decision on simple criteria: object co-location to avoid conflict misses, object size and access frequency. Nearly no work has been done on memory-aware task scheduling for dynamic multimedia applications. One of the only contributions in this area is [60]. There, an OS scheduler directs the power mode transitions of the SDRAM modules, but performs no access scheduling or bank assignment.

3.4.2 Enforcing runtime policies

Dynamic memory management is a well known problem. It has been widely researched in the context of general purpose computing. The main reason is that high-level programming languages intensively allocate data on the heap at runtime. E.g., every time in C++ a new object is created, the *new* function dynamically allocates memory space on the heap. Since applications allocate many differently sized data structures, the heap space easily becomes fragmented. This significantly reduces the available memory space and increases the allocation overhead. Several dynamic memory managers have been proposed for reducing fragmentation (see [11][15] for an overview). An important technique to eliminate fragmentation is adapting the dynamic memory manager to the allocation requests of the applications. [64] splits the available memory in pools. Every pool is then managed by a separate dynamic memory manager, which deals with a subset of the allocation requests. Usually, the subsets consists of the allocation requests with a similar size.

The authors of [54] present a deterministic hardware-based dynamic memory manager. The memory is hierarchically managed. Each processor has its own memory pool which is controlled by the RTOS. Whenever the space in this pool is too limited, the processor allocates more memory from the shared memory pool. The shared pool is split in fixed sized blocks to simplify its management. The result is a memory manager which has very fast memory (de)allocations times.

In the context of multi-processors, the most scalable and fastest memory managers use a combination of private heaps combined with a shared pool [11][62]. These memory managers avoid typical multi-processor allocation problems such as blow-up of the required memory space, false sharing of cache-lines and contention of threads accessing the shared memory. However, they are unaware of the memory architecture and are complementary to our work. As we will show in section 6, we reuse the above techniques to manage the memory space at runtime, but we have to carefully control their allocation overhead.

We conclude from the above that:

1. no decision techniques cope with the underlying memory architecture, albeit a multi-banked SDRAM or the local memory layer.
2. no current runtime decision techniques optimize the memory bandwidth.
3. despite dynamic memory management is a well researched problem, limited support is available to integrate these decisions inside the code.

4 Memory bandwidth optimization for platform-based design

Many techniques optimize the memory bandwidth for a single thread (see section 3), but they break down when applied to multi-threaded applications. In this section, we illustrate how parallel accesses from different processing elements either to the shared memory (subsection 4.1) or the local memory layer (subsection 4.2) degrade the system's performance and increase its energy consumption. For each of them, we will introduce techniques to mitigate the problems. Our techniques exploit data assignment and scheduling to optimize the behavior of the memories.

4.1 The shared layer

As motivated, the shared layer usually is based on a multi-banked memory, such as an SDRAM. In this subsection, we first explain with an example why existing techniques breakdown (subsection 4.1.1). Then, we present how to overcome these limitations with data assignment and task scheduling (sections 4.1.2-4.1.3).

4.1.1 Multi-threading causes extra page-misses

Over the past years, several techniques have been proposed to eliminate page-misses inside a single thread (subsection 3.1), but they cannot cope the ones caused by parallel threads. A small example explains why (figure 9). It consists of task1 and task2, running on a different processor tile and accessing data stored in the shared SDRAM memory. As explained above, the more page-misses occur on the SDRAM, the more energy is consumed.[4] One way to minimize the number of page-misses is to carefully assign the tasks' data to the banks of the SDRAM. Current techniques optimize the assignment of a single task at a time. In case of our example, they generate layout A. If both tasks are sequentially executed, it results in only one page-miss for task1 (see sequential schedule). Also for task2, only three page-misses occur, because its data (b and c) are distributed across the two banks (see again the sequential schedule).

As soon as both tasks execute in parallel while using layout A, extra delays and many more misses occur, because the SDRAM interleaves accesses from both tasks. E.g., task1 fetches a while task2 reads simultaneously from b. With its single memory port, the SDRAM cannot access both data structures in parallel. Its interface has to serialize them, delaying the access to b with one cycle. Furthermore, every other access to a or b results in a page-miss, because they are stored on different pages in the same bank. The extra page-misses augment the energy cost and further delay the execution.

Interacting tasks on shared resources thus cause more delays and generate extra page-misses. Currently, no techniques can avoid this, because they optimize the data layout within a single task.

4.1.2 Optimizing the data assignment across the tasks' boundaries

We have proposed a technique for reducing page-misses across the tasks' boundaries [43]. It stores frequently accessed data structures with high access locality in separate banks as much as possible. To identify these data structures, we have developed a heuristic parameter called selfishness[5]. At design-time, each task is analyzed and profiled independently. Every relevant data structure of each task is characterized with a *selfishness factor*. A data structure's selfishness is the average time between accesses (tba) divided by the average time between page-misses (tbm). It is a measure of spatial locality of the data structure; we finally weight it with the data structure's importance by multiplying it by the number of accesses to the data structure.

At run-time, when we know which tasks will co-occur in time, we decide the assignment of the alive data. We have implemented a greedy algorithm that assigns data to the banks by decreasing order of selfishness. The higher the selfishness becomes, the more important it is to store the data in a separate bank. Our algorithm distributes the data across banks such that the *selfishness* of the banks is balanced. The selfishness of a bank is the sum of the selfishness of all data structures in the bank.

[4]For the clarity of our example, we only focus on their energy penalty, i.e. no performance penalty due to page-misses.

[5]for details how to measure selfishness we refer to [43]

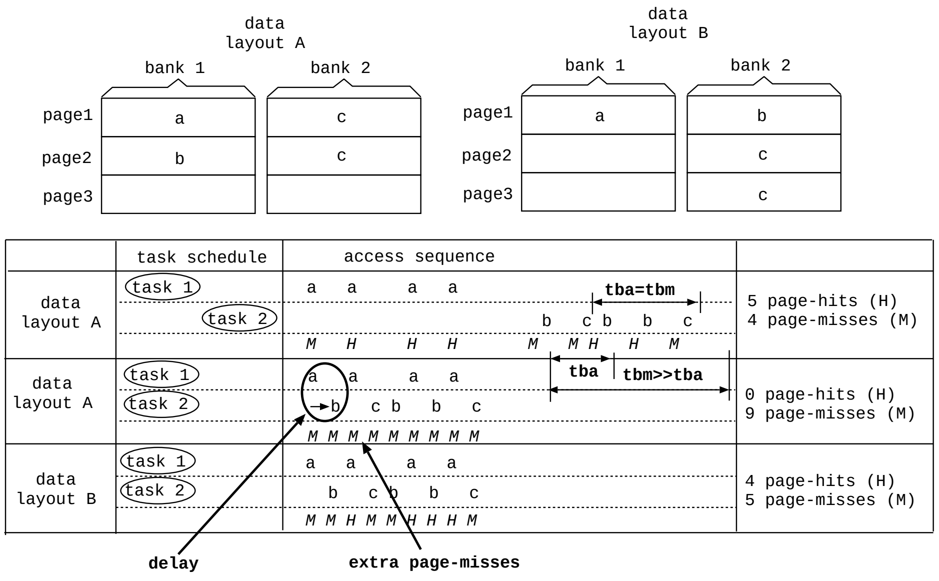


Figure 9: Interleaved accesses from different tasks cause page-misses and extra stalls

Consider the example in Fig. 9. The sequential schedule gives us the required information for each data structure. a and b both have the same spatial locality, because the time between misses equals the entire duration of the task and the time between accesses is similar. However, because b is less frequently accessed than a, its selfishness is slightly lower. The selfishness of c is much lower than both a and b, since for every access a page-miss occurs and it is less accessed. Therefore, when we schedule both tasks in parallel, our algorithm first separates the most selfish data structures a and b and then stores c with b (since the bank containing b is less selfish than the one with a).This corresponds with layout B in the figure. As a result, only five page-misses remain and the energy cost is significantly reduced compared with a naive layout: layout A just places one data structure after the other. This results in one page-miss per access.

4.1.3 Task ordering to trade-off energy/performance

Besides data assignment, also the task order heavily impacts the system's energy and performance. For instance, if we execute task1 and task2 sequentially, four page-misses occur. This is the most energy-efficient solution, but takes the longest time to execute. In contrast, when we execute both tasks in parallel, the execution time becomes shorter, but five page-misses occur, thus the energy cost increases.

Generally, by changing the task-order we can trade-off the energy/performance of the system. We have developed an algorithm to schedule a set of tasks in such a way that the energy consumption is minimized while meeting a pre-fixed time deadline. For a given application, we first define the most likely combination of tasks that will happen at run time (we call these combinations *scenarios*. See Sect. 5). For each scenario and an specific time constraint, we explore different schedule possibilities, trying to find the most energy efficient. Once the relative schedule is defined, we must allocate the data of the tasks. For that purpose, we reuse the ideas presented in the previous section. The obtained task schedule and data allocation represent a Pareto-optimal solution for performance and energy. We iteratively modify the time constraint in order to generate a set of solutions, automatically generating a set of Pareto-optimal solutions (called a *Pareto curve*. In Fig. 10 we depict the Pareto curve for our example). The designer can pick the operating point which best fits his needs from the generated trade-off points. All the details of this joined task schedule/data assignment technique are shown in [18].

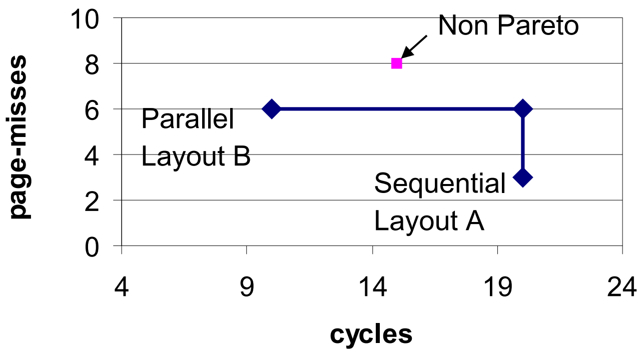


Figure 10: Energy/performance trade-off for task1 & task2

Memory aware task scheduling may be be also beneficial for performance. Assuming that it is always better to distribute the data across banks to reduce the number of page-misses, a conservative task schedule increases the assignment freedom. When the ratio *number of data structures number of banks* becomes high, insufficient banks are available to separate all energy critical data structures from each other. Data allocation alone does not suffice to decrease the number of page misses. In such a situation, task scheduling is a good way to enlarge the freedom of the allocation process. Generally, sequential schedules result in the lowest energy consumption, but they have the worst execution time. In general the trend is clear: the lower the execution time (scheduling more tasks in parallel), the higher the energy consumption. Of course, some schedules will not follow this tendency. Sometimes, too much parallelism is bad even for performance!! (if the number of page-misses increases too much and the applications are memory bounded, an aggressive parallel task schedule could increase the total execution time).

In Fig. 11 time and SDRAM energy consumption values are shown for four different schedules of the same task-set. Schedule D corresponds to a sequential scheduling: as expected, the longest execution time with the lowest energy consumption. Full parallel schedule is shown in part A of the figure. B and C are intermediate schedules, that trade-off performance and energy consumption. As well as illustrating our point, Fig. 11 also points out that the execution time of a task cannot be estimated independent of the other tasks running in parallel with it. The time penalty raised from sharing the SDRAM between several concurrent tasks reaches up to 300% for *CMP* (in schedule A), compared to its execution time without other tasks executing on the platform.

4.2 The local memory layer

4.2.1 Access conflicts reduce the system's performance

As indicated in section 3.2, existing techniques only optimize the memory bandwidth in the scope of a basic-block. As a result, a large room for improvement remains. We illustrate this with a small example that consists of three data-dominated loops (see code in Fig. 12-left) which are executed on a platform that consists of three single-port memories: two 4kB ones (0.11nJ/access) and a 2kB one (0.06nJ/access).

Because the applications are data dominated, the duration of the memory access schedule determines the performance of the loops. We assume that the remaining operations can be performed in parallel with the memory accesses or take only limited time. We will use the example presented in figure 12-left to study the the influence of memory access schedule together with data assignment in the resulting performance and energy consumption.

Current compilers are not aware of the final data to memory assignment. During instruction scheduling, most compilers simply assume that any memory operation finishes after n-cycles. When the executed operation takes

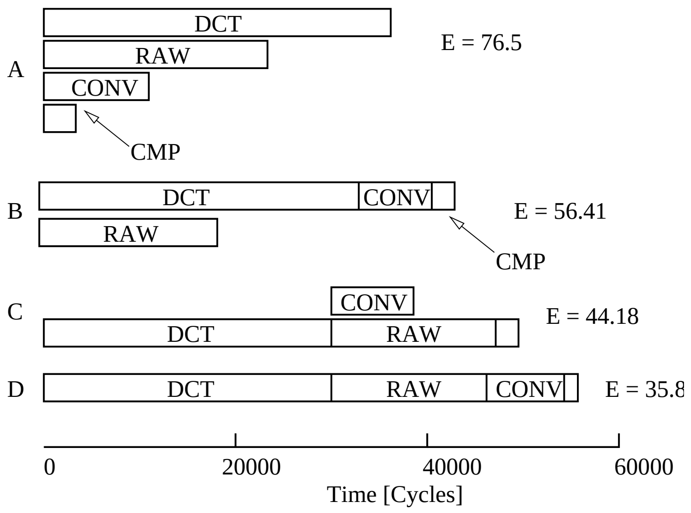


Figure 11: Scheduling outputs for four tasks

```
int A[301],int B[100];int D[100]
int C[100]; int U[2];
int i,j;

for (i=0; i<100; i++) // loop 1
 A[i+1] = A[i] + 1;

for (i=0; i<100; i++) // loop 2
 D[i] = C[i] + B[i];

for (i=0; i<2; i++){  // loop3
 for (j=0; j<40; j++)  // loop31
   D[j] = D[j-1]+ D[j];
 U[i] = D[39];
}
```

```
int A[300],int B[100];int D[100]
int C[100]; int U[2];

for (int i=0; i<100; i++) // loop 2
 D[i] = C[i] + B[i];

for (int i=0; i<2; i++){ // loop 1&3
 for (int j=0; j<40; j++){
  D[j] = D[j-1]+ D[j];
  A[40*i+j] = A[40*i+j-1] + 1;
 }
 U[i] = D[39];
}
// remainder of loop 1
for (int i=0; i<20; i++)
 A[i+80] = A[i-1+80] + 1;
```

Figure 12: Motivational example: original code (left), code after fusion (right)

longer than presumed, the entire processor is stalled. As a result, often a large difference exists between the expected and the effective performance of the processor. We use a typical modulo scheduler ([7]) to generate our memory access scheduling. Note that a modulo scheduler may schedule read/write operations from the same instruction in the same cycle. This is the case in our examples: the write operation belongs to the iteration i while the read operation comes from iteration $i+1$. Modulo scheduling may be applied when there are no data dependent conditions in the loop body. The scheduler generates a memory access schedule for the inner-loops of 460 cycles (Fig. 13-a). However, the actual performance varies between 540 and 740 cycles. The schedule takes longer than expected because the processor has to serialize the accesses to D in loop 31. Extra stalls occur depending on whether the linker has assigned the C, B and/or D to the same memory.

Because how the linker assigns the data to the memories has such a large impact on the performance of the system, it is better to optimize the data assignment and the memory schedule together ([57]). Our technique imposes restrictions on the assignment such that the energy is optimized, but still guarantee that the time-budget is met. The assignment constraints are modeled with a conflict graph (e.g., Fig. 14-left). The nodes correspond to the data structures of the application. An edge between two data structures indicates that we need to store them in different memories. Hence, the corresponding accesses to these data structures can be executed in parallel.

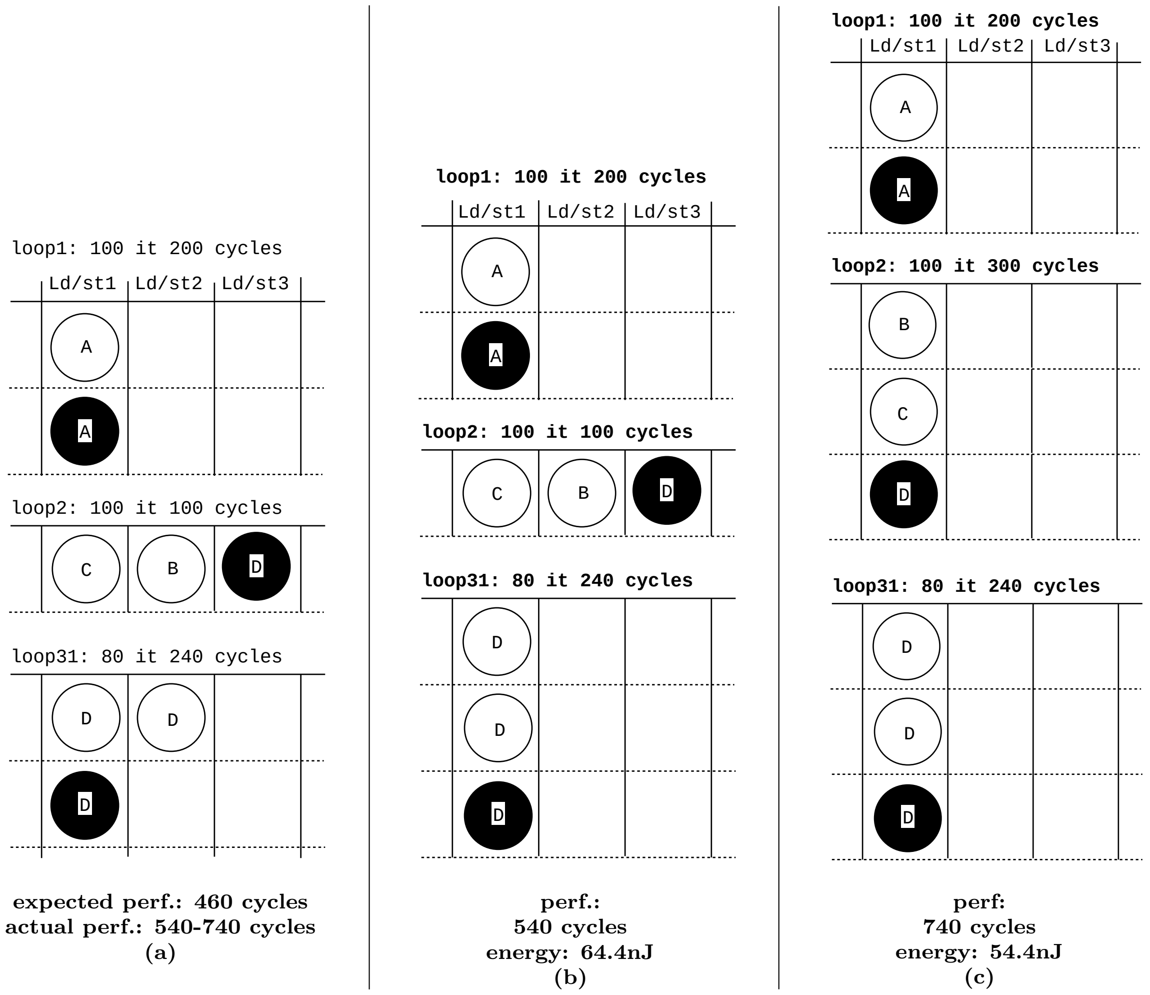


Figure 13: Empty issue slots in the memory access schedule of the inner-loops: (a) existing compiler; (b) with fastest partial data assignment; (c) with most energy efficient partial assignment

The assignment constraints imposed by the conflict graph prevents certain access to happen in parallel. This restricts the feasible schedules to those respecting the access restrictions. Thus, the conflict graph links data assignment and instruction scheduling: for a specific conflict graph we can determine the fastest schedule possible when using the least energy consuming data assignment solution. For instance the edge between A and C (Fig. 14-left) forces us to store both data structures in different memories. The fastest schedule for this conflict graph takes 540 cycles (Fig. 13-b). It consumes 64.4 nJ[6], because the conflict edges force us to store both A and B in large memory (see complete assignment in Fig. 14-left). Note that the energy consumed in the memories only depends on the conflict graph, not in the access schedule. For a fixed conflict graph, the energy dissipated in the memories will remain the same even with different instruction schedules.

We can decrease the energy cost of the above assignment by reducing the number of conflicts. After eliminating the edges between B-D, C-D and B-C, the small data structures B, D and C can be assigned in the smallest and most energy efficient memory (figure 14-right). The energy consumption is then 54.4nJ instead of the original 64.4nJ. Less conflicts also imply that less memory accesses can execute in parallel. The fastest feasible schedule now takes 740 cycles (Fig. 13-c). The energy savings thus come at a performance loss.

However, many memory access slots remain empty (check again figure 13) . This is mainly due to: (1) inter-

[6]We compute the energy consumption as follows: $\sum_{\forall m \in M} \sum_{\forall ds \in m} \mathrm{NrAccess(ds)} E^{m}_{\mathrm{access}}$

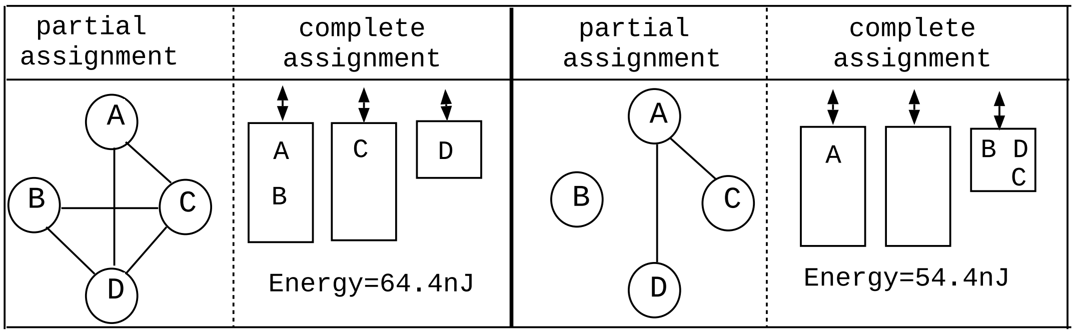
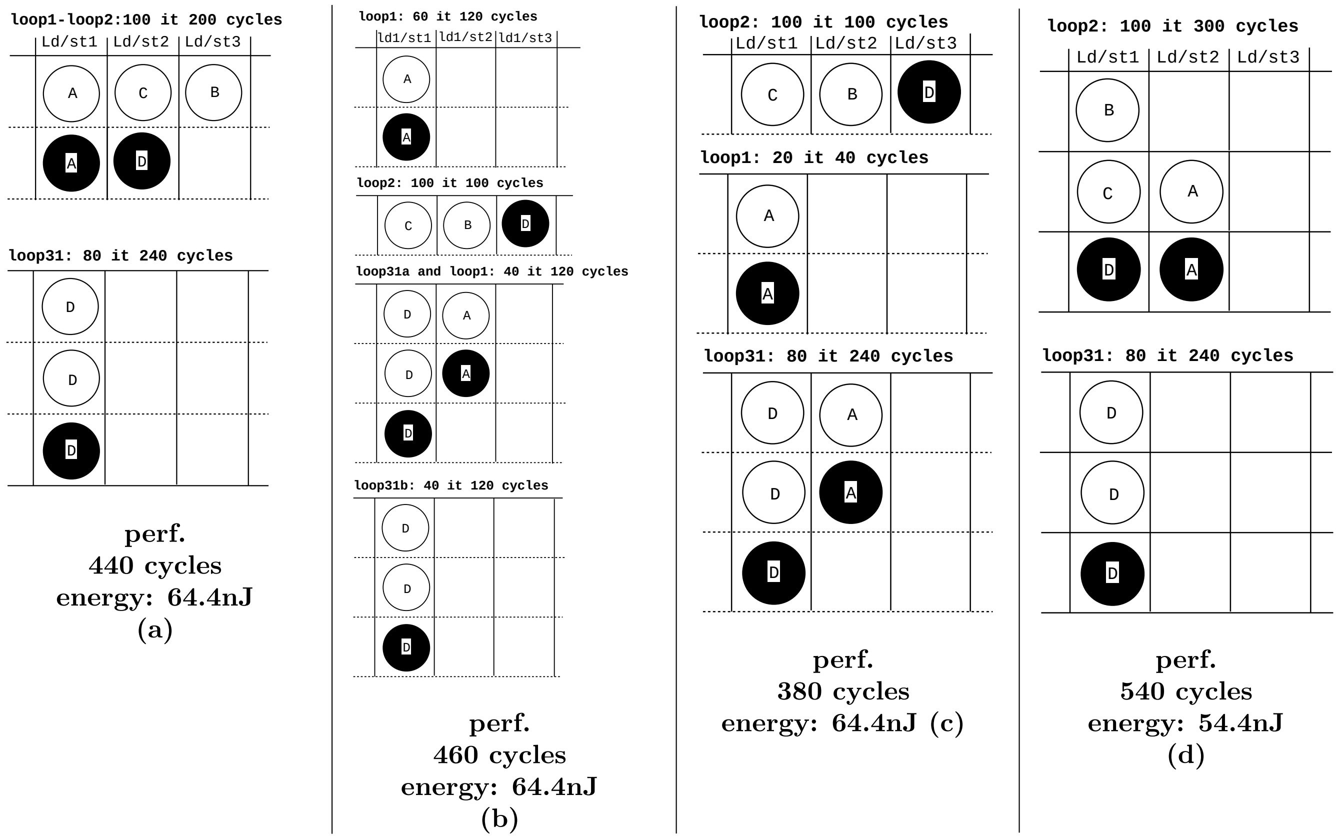


Figure 14: Partial assignment expressed with a conflict graphs: (left) fast; (right) more energy-efficient

iteration dependencies. For instance the initiation interval of loop 1 is 2, because A depends on itself. Hence, only 30% of the available memory slots are used; (2) we do not use power hungry multi-port memories. Consequently, we cannot schedule operations that access the same data in parallel. E.g., in loop 31 we cannot execute the accesses to D in parallel.

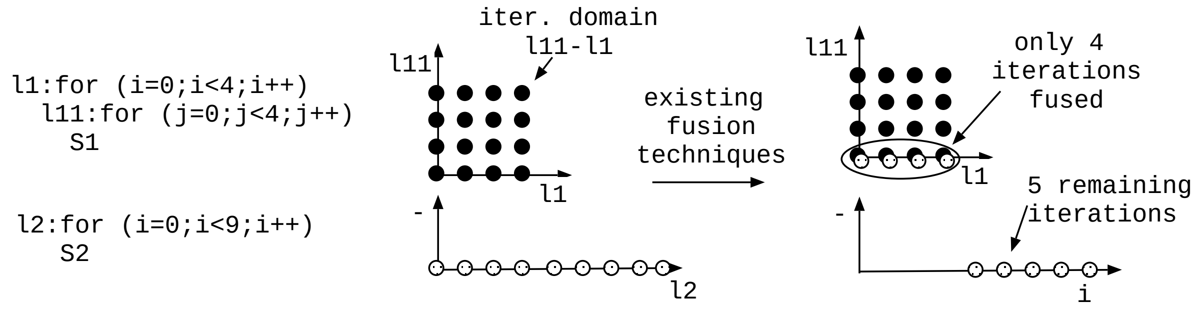
4.2.2 Global schedule: Loop morphing



loop1-loop2:100 it 200 cycles
Ld/st1
Ld/st2
Ld/st3
loop31: 80 it 240 cycles
perf. 440 cycles energy: 64.4nJ (a)
loop1: 60 it 120 cycles
loop2: 100 it 100 cycles
loop31a and loop1: 40 it 120 cycles
loop31b: 40 it 120 cycles
perf. 460 cycles energy: 64.4nJ (b)
loop2: 100 it 100 cycles
loop1: 20 it 40 cycles
loop31: 80 it 240 cycles
perf. 380 cycles energy: 64.4nJ (c)
loop2: 100 it 300 cycles
loop31: 80 it 240 cycles
perf. 540 cycles energy: 54.4nJ (d)

Figure 15: Loop fusion fills the issue-slots: (a)-(b) existing fusion techniques for the fastest partial data assignment; (c) fusion combined with loop splitting and strip mining; (d) best fusion for the energy-efficient partial assignment

With more global optimizations, such as loop fusion [38], we can further compact the application's schedule. However, existing fusion techniques can only overlap few iterations. Consider the loop nests in Fig. 16. We also show their iteration domains (each dot corresponds to a single iteration). Traditional techniques would only fuse four out of nine iterations, due to the different dimensions of the loop nests. This restriction limits the achievable performance gains.


```
l1:for (i=0;i<4;i++)
  l11:for (j=0;j<4;j++)
    S1

l2:for (i=0;i<9;i++)
   S2
```

Figure 16: Existing fusion techniques can only overlap a limited number of iterations

Let's apply loop fusion to our previous example (code in Fig.12). We first select one conflict graph, which prevents certain accesses to happen in parallel. Let's consider the fastest conflict graph (Fig. 14-left). If we choose to fuse loop 1 and loop 2, the resulting schedule would take 440 cycles (Fig. 15-a). Another option would be to fuse loop1 and loop3. In this case, the loop nests are not conformable (they do not have the same number of dimensions and their loop limits are different). However, current loop fusion techniques will just fuse 40 iterations, resulting in a schedule length of 460 cycles.

Loop morphing is a technique that enables loop fusion beyond conformability limits (details can be found in [45]). Once we have decided which loops to fuse, our algorithm gradually tries to make them as similar as possible. Loop splitting and strip mining are iteratively applied to obtain conformable loop nests. Fig. 17 shows the different steps to fully fuse the loops shown in Fig.16. First, we apply strip-mining to *l2* to fit the number of dimensions of the other loop. The resulting loop nest is split to avoid the if-condition in the body of the loop. A new loop nest, with just one iteration, is generated (*l3*). We now transform the longest loop *l1* such that it has the same length as *l2*. This transformation is accomplished through loop splitting. After that, we have two loop nests perfectly conformable, that we can easily fuse (Phase 2 of Fig. 17). Note that eight iterations have been fused, instead of only four in Fig. 16.

In our example on Fig. 12, we first split loop1. Two loops are generated: one with the first 80 iterations and a second one with the last 20 iterations. Strip mining is applied to the first of these two new loops. This way we obtain a two-level nested loop, similar to loop 3. Fusion is now straightforward. Finally the 20 remainder iterations from loop 1 may be considered for subsequent fusion decisions. The final code after morphing is shown in Fig. 12-right. Always using the fastest conflict graph, loop morphing enables a schedule length of only 380 cycles (Fig. 15-c).

From the above, we conclude that for a given conflict graph we may decide which are the best loops to fuse, and use morphing to maximize the iterations fused. However, if we change the conflict graph, we need to take different fusion decisions. E.g., under the more energy efficient conflict graph (Fig. 15-c), it is more beneficial to fuse loop 1 and loop 2. The execution time is then 540 cycles compared to 740 cycles for the non-fused code, for the same energy consumption. The fusion decisions and consequently, the performance of the application, heavily depend on the conflict graph. The more conflicts the higher the application's performance, but the more energy hungry it becomes.

In [46], we present a heuristic to decide which loops to combine. The input of the algorithm is an initial description of the loops, their statements and iteration domains. As stated above, the decision also takes into account the current conflict graph. We compute the *fusion gain* of all possible pairwise fusions. The fusion gain is an estimation of the relative system's performance gains after fusion. We estimate the schedule length of every basic block with an

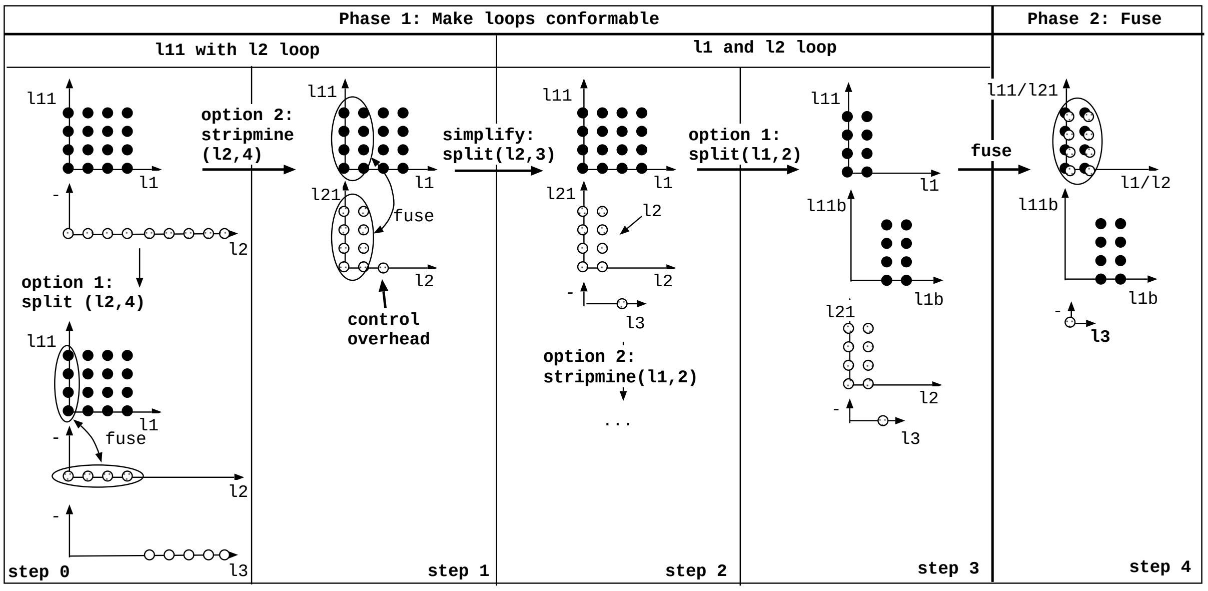


Figure 17: Morphing heuristic applied on an example

iterative modulo scheduler. The performance gain estimation is obtained by comparing the schedule length of the original loop nests and the fused version. The schedule takes the assignment constraints into account. We only schedule memory operations in parallel if a conflict exists between their corresponding data in the conflict graph.

After computing the fusion gains for all possible loop nest pairs, we fuse the loop pair with the highest gain. After the fusion step, we re-evaluate which loop pairs can be combined (data dependences may prevent some loops to be fused) and re-compute the fusion gains of the newly generated loops. This process is iteratively performed until the performance gain does not exceed a prefixed threshold. We finally obtain a fused version of the code which is corresponding conflict graph.

Thereafter, we generate information to decide which conflict edges (from the conflict graph) to remove first. This generates a more energy efficient conflict graph that triggers a new loop fusion process from the original code. This way, we may generate different versions of the code (i.e. the original code after different fusion decisions applied). Each of this versions have a conflict graph (and thus, and energy consumption) associated. Again, we have automatically trade-off performance and energy consumption, allowing the designer to choose the optimal point that meets the constraints with the lowest energy consumption. Fig. 18 shows this trade-off for different set of benchmarks. Details can be found in [46].

From this example, we conclude that fusion shortens the memory access schedule on condition that:

1. We overlap loops even with non-conformable loop headers. Otherwise, the number of overlapping iterations after fusion is limited. Therefore, we have proposed loop morphing, a technique that combines loop fusion, strip mining and loop splitting. Loop morphing fuses non-conformable loops while increasing the instruction level parallelism in as many iterations as possible. Besides its benefits for optimizing the memory bandwidth, it may be useful for different optimization objectives. Loop morphing has been presented in [45].

2. we combine the loops which result in the largest performance gains. Our technique pairwise fuses loops which considers memory size, number of ports, access latency and assignment constraints.

We have presented approaches which more globally optimize the memory bandwidth for both the local and shared memory layer. In the next section, we discuss to cope with the dynamic behavior of media-rich applications. Our approach for this problem relies on the above techniques.

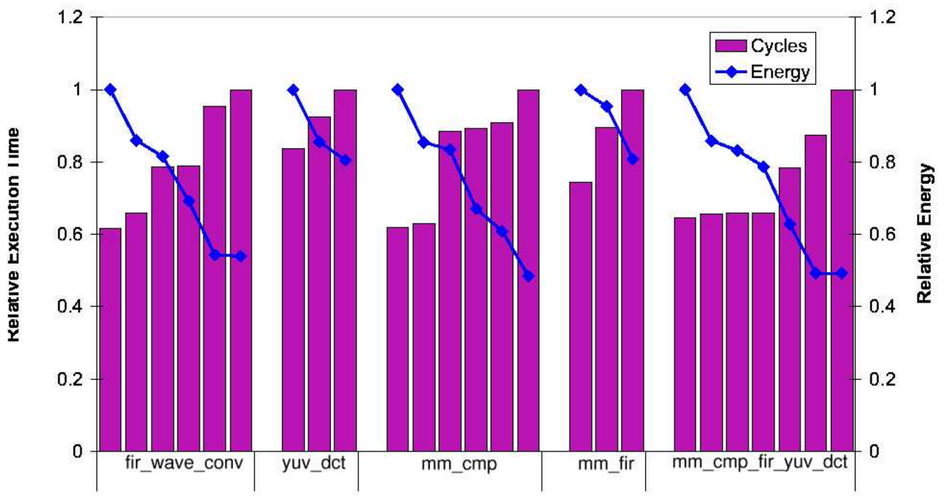


Figure 18: Energy vs. performance trade-off

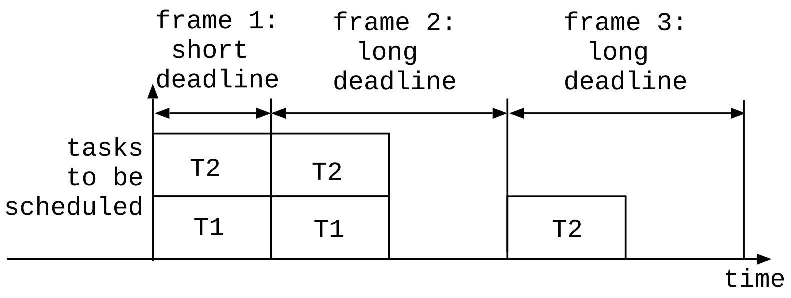


Figure 19: Dynamically created tasks with their deadline

5 Scenarios for coping with dynamic behavior

Dynamism introduces a certain degree of unpredictability in the system. Fully static optimizations cannot handle this non-deterministic behavior, and the solutions obtained may be far from optimal. On the other hand, moving all the optimizations to run time may introduce too much overhead. Scenarios lay in the middle of these two extremes. An exhaustive analysis is performed at design time, to gather all the relevant information about every task and their potential interactions. This information will be later used at run-time to quickly take a decision. In this section we will first examine the impact of this dynamism, focusing in the resulting energy consumption. Later, we will briefly explain the scenario approach.

5.1 Energy constraints demands for runtime decisions

Due to the dynamic behavior of our application domain, it is more energy-efficient to assign the data and schedule the tasks at runtime. An example in the context of the SDRAM layer explains why (figure 19). At the start of each frame, the user either executes task1 and/or task2. They are the same tasks as in subsection 4.1. We thus only know at runtime which tasks execute and which data they require. Also, note that the deadline varies from frame-to-frame. E.g., at the start of frame2, the user lowers the video quality, reducing the frame-rate by half. The system has then twice more time for each frame.

The optimal task order/data assignment decisions vary from frame to frame (see figure 20). E.g., to satisfy the short deadline in frame1, we have to schedule both tasks in parallel. We obtain the least number of page-misses

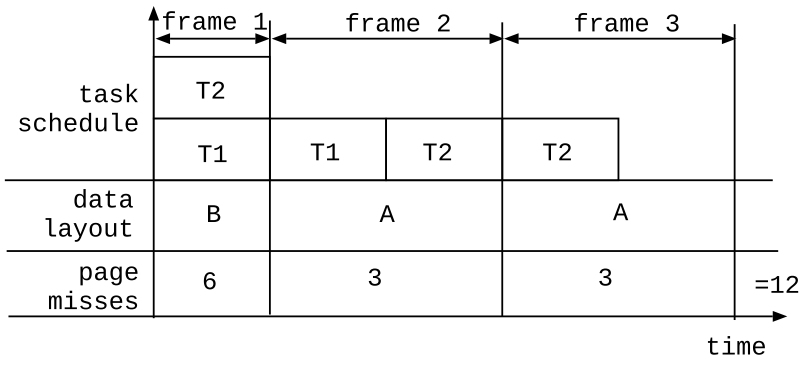


Figure 20: Our runtime task schedule/SDRAM assignment solution

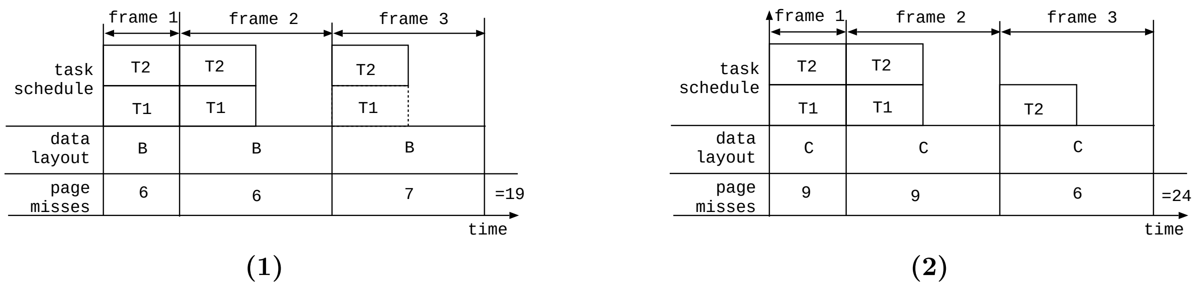


Figure 21: State-of-art task schedule/SDRAM assignment: a (1) design-time and (2) operating system based solution

using layout B from figure 9. However, in frame2, only task2 is active. As indicated in figure 9, layout A is then more energy efficient.

Finally, in frame3, both tasks are started again. However, since the frame-rate is lower now, we can execute them sequentially and eliminate most page-misses with layout A. So, in each frame, different scheduling/assignment decisions are optimal for energy and we can only take these decisions at runtime.

Current approaches always generate more page-misses occur. Design-time techniques only select one task schedule and layout (figure 21-1). This single design has to meet the deadline for the worst-case load, i.e. task1 and task2 executed within the short deadline (frame1). The most energy-efficient design for this load is executing both tasks in parallel and using layout B (figure 9). This operating point is not optimal for both frame2 and frame3. It results in seven more page-misses than the above approach. A pure design-time technique is thus not energy-efficient.

Furthermore, also current runtime approaches are far from optimal. A typical OS does not account for the specific behavior of SDRAMs. As long as enough processors are available, it schedules all tasks in parallel and assigns the data to the first available free space (figure 21-2). By storing all the data in a single bank and scheduling the tasks in parallel, twelve more page-misses occur than in the optimal case. Again this solution is not energy-efficient.

These results indicate the potential benefits for a runtime technique which considers the SDRAM behavior and can generate the solutions of figure 20. Since it should make complex task scheduling/data assignment at runtime, the main difficulty is restricting its overhead. The local memory layer requires a similar approach, but we restrict ourselves to the shared SDRAM layer.

5.2 Our scenario-based approach

We have proposed a mixed design-time/runtime approach for coping with the dynamic behavior (section 6). The philosophy behind it is to take most scheduling/assignment decisions at design-time for all frequently occurring task

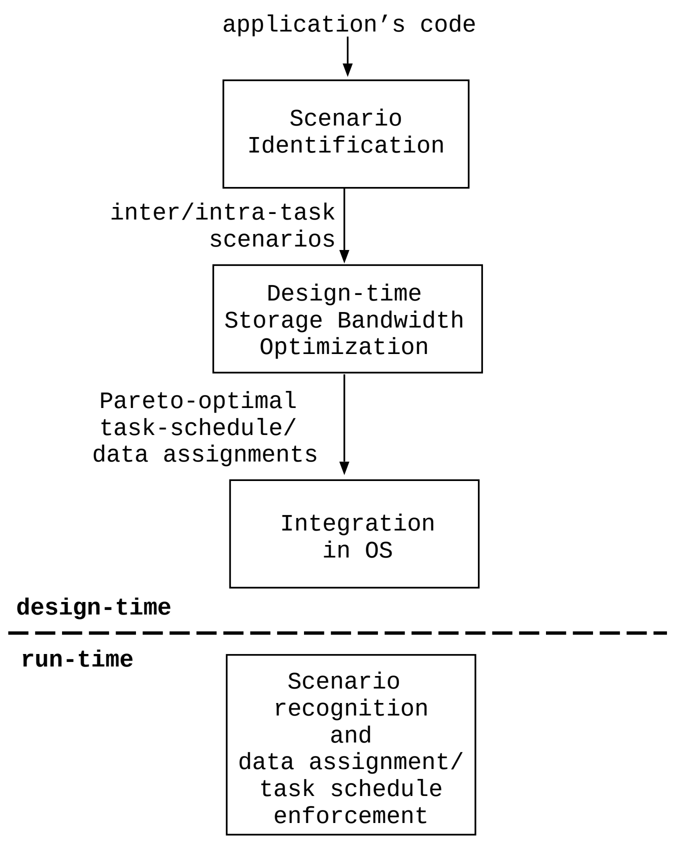

application's code
Scenario Identification
inter/intra-task scenarios
Design-time Storage Bandwidth Optimization
Pareto-optimal task-schedule/ data assignments
Integration in OS
design-time
run-time
Scenario recognition and data assignment/ task schedule enforcement

Figure 22: A scenario-based design-time/runtime approach

```
for (y=0; y<9; y++){
  ...
    for (n=0;n<8;n++){
    ...
      for (l=0;l<8;l++)
       tmp += prev_frame[];
    ...
   }
}
for (y=0; y<9; y++){
  if(mode)
    for (n=0;n<8;n++){
      p1 = sub_frame2[];
      if (ctrl)
       p2=0;
      else
       p2=prev_sub2_frame[];
      dist+=abs(p1-p2);
     }
  ...
...
}
```
(1)

```
for (y=0; y<9; y++){
  ...
    for (n=0;n<8;n++){
    ...
      for (l=0;l<8;l++)
       tmp += prev_frame[];
    ...
    }
}

for (y=0; y<9; y++){
  // mode == true
    for (n=0;n<8;n++){
      p1 = sub_frame2[];
      // ctrl ==  true
       p2=0;
      // ctrl ==  false
       p2=prev_sub2_frame[];
      dist+=abs(p1-p2);
    }
  ...
}
```
(2)

```
//scenario 1: mode=true
for (y=0; y<9; y++){
  ...
    for (n=0;n<8;n++){
    ...
      for (l=0;l<8;l++)
       tmp += prev_frame[];
    ...
    }
}

for (y=0; y<9; y++){
    for (n=0;n<8;n++){
      p1 = sub_frame2[];
      // ctrl ==  true
       p2=0;
      // ctrl ==  false
       p2=prev_sub2_frame[];
      dist+=abs(p1-p2);
    }
  ...
}
```
(3-1)

```
//scenario 2: mode=false
for (y=0; y<9; y++){
  ...
    for (n=0;n<8;n++){
    ...
      for (l=0;l<8;l++)
       tmp += prev_frame[];
    ...
    }
}

for (y=0; y<9; y++){
  ...
}
```
(3-2)

Figure 23: (1) data dependent conditions as a limiting factor for bandwidth optimization techniques; (2) worst-case approach; (3) scenario-based approach

sets. In this way, we can reuse our bandwidth optimization techniques for multi-threaded applications and at the same time limit the runtime complexity. Only for the more seldomly occurring task-sets a pure runtime decision is taken as a backup solution. In the next paragraphs, we explain the main steps of our methodology (figure 22).

Scenario identification Fixing as many decisions as possible at design time comes at the risk of ignoring the actual behavior and generating worst-case designs. E.g., consider the code in figure 23. Even though parts of the code are conditionally executed (e.g., *mode* and *ctrl*-conditions in the second loop nest), design-time techniques assume that both branches are executed, optimizing thus the design for the worst-case load. As a consequence, we heavily over-estimate the required bandwidth and usually generate an over-dimensioned and energy-inefficient system.

To prevent this energy-loss, we try to capture the dynamic behavior with scenarios. First, we analyze which tasks-sets often co-occur at runtime. We call them *inter-task scenarios*. A similar but more restrictive concept is used by [23]. Secondly, we narrow down the data-dependent behavior inside the tasks with *intra-task scenarios*. An intra-task scenario is an execution path through the task (or a combination of execution paths) for different data dependent parameters [49][50]. Both the inter- and intra-task scenarios should be manually extracted by the designer (using profiling). E.g., in the code of figure 23, we derive two intra-task scenarios, one for *mode* equals true and another one for *mode* equals false. Even though research outside IMEC is ongoing in how to identify scenarios [48][34], a more automated approach is still needed.

After identifying the scenarios, we can represent each one with a data-flow graph on which we can easily apply our design-time techniques.

Storage bandwidth optimization at design time In the second step, we optimize the storage bandwidth of each scenario. Our design-time techniques generate for each scenario a set of task ordering/data assignment solutions. Each solution optimizes the energy cost for a given time-budget. From this set, we only retain the Pareto-optimal

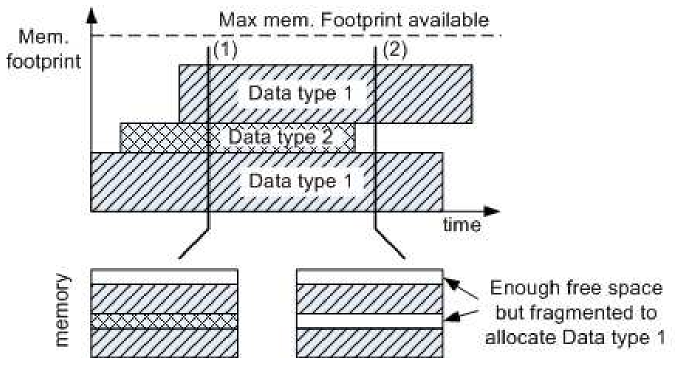

Max mem. Footprint available
Mem. footprint
(1)
(2)
Data type 1
Data type 2
Data type 1
time
memory
Enough free space but fragmented to allocate Data type 1

Figure 24: Fragmentation in dynamic memory management

solutions. E.g., for our example's scenario in which task1 and task2 are active (subsection 5.1), we would generate the Pareto curve of figure 10. Finally, we integrate the Pareto set of each scenario into the operating system. We provide more details in [44].

Runtime phase Then, at runtime, after identifying which scenario is activated and which is its deadline, we simply select the best prestored operating point on the Pareto curve and enforce its task ordering and data assignment decisions. E.g., for frame 1 of our example (subsection 5.1), our runtime mechanism then selects the leftmost operating point, scheduling both tasks in parallel with layout B. In contrast, for frame2 with the relaxed deadline, it implements the rightmost operating point. If the scenario was not analyzed at design time, we use a back up solution. E.g., we simply use an existing dynamic memory manager for assigning the data. Note that our approach leverages current design-time techniques, but requires that scenarios can be identified inside the application. Obviously, this partly restricts the applicability of our technique.[7]

6 Dynamic Memory Management

The memory space available at run-time to our applications can be located in any physical memory of the system. It is managed with the help of a Dynamic Memory (DM) manager. DM Management basically consists of two separate tasks, i.e. allocation and deallocation. Allocation is searches for a memory block big enough to satisfy the request of a given application and deallocation returns this block to the available memory of the system in order to be reused later. In real applications, memory blocks with various sizes are requested and returned in a random order, creating "holes" among used blocks (see Fig 24). These holes are known as memory fragmentation [15]. We talk about *internal fragmentation* when the wasted space is inside allocated partitions. Otherwise, we will name it external fragmentation. Internal fragmentation happens when allocated memory is larger than requested memory and not being used. (e.g. a free block of 5 KB is used for a request of 4 KB, hence one KB is wasted). If we suffer from external fragmentation, total memory space exists to satisfy the request, but it is not contiguous . (e.g. if a request asks for 6 KB and there are several non-contigous free blocks of 3 KB, the memory request cannot be satisfied). The DM manager has to take care of fragmentation issues.

We have classified all the important design options in different orthogonal decision trees, which can compose the design space of DM management. Based on these orthogonal trees, we can construct custom DM manager from its basic building elements. The most critical parts of the design search space are overviewed in the next paragraphs.

[7]Another approach could be to start from existing operating systems and make them account for the energy cost of the underlying memory architecture. All decisions are then made at runtime without design-time preparation. Even though we did not investigate this, we expect that such an approach causes too much energy overhead and violates more deadlines, but more research is still needed.

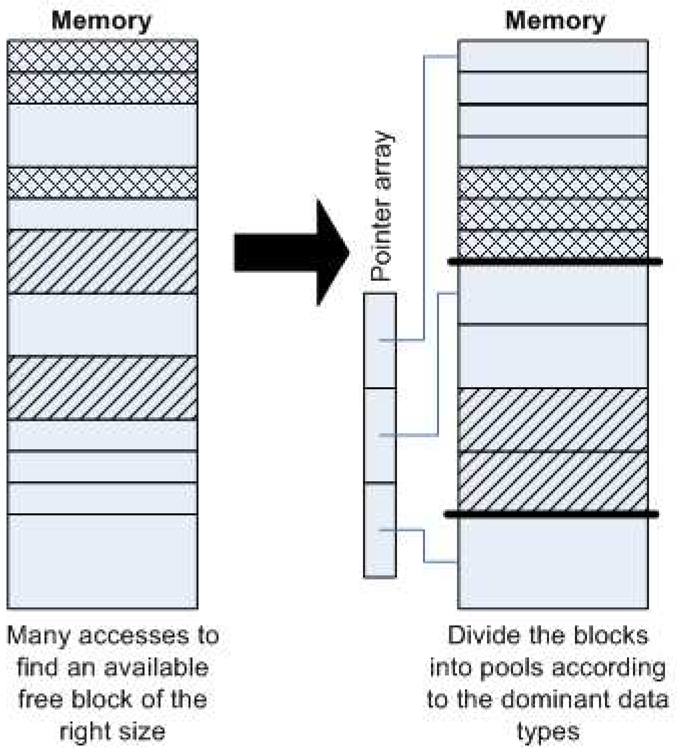

Memory
Memory
Pointer array
Many accesses to find an available free block of the right size
Divide the blocks into pools according to the dominant data types

Figure 25: Assigning the memory blocks to multiple pools

For a complete description about the design space and how it can be used to build custom DM managers see [2, 3]. Then, we analyze the most commonly used DM managers with the use of the design space. Finally, we explain our approach and why it is more energy-efficient than other state-of-the-art approaches to construct DM managers.

6.1 A brief summary of the dynamic memory management design space

The most important part is trying to prevent fragmentation. In Figure 25, we illustrate the most commonly used technique to prevent memory fragmentation. This technique assigns memory blocks to different memory segments (also called pools), which are then accessed by one pointer array that keeps track of the initial position of the pool with free blocks. The goal is to try to anticipate the size of the memory requests of the application. If every memory request is met with a memory block with the same size, then no memory space inside the block gets wasted. This means less internal fragmentation and a quick allocation of the requested block.For example, if an application usually employs blocks of 8 KBytes and 1024 KBytes 50% of its total memory allocation requests, its DM management will be simplified significantly if two memory pools of these sizes exist, then just 2 accesses are needed to return one block (i.e. one access to the pointer array and another one to update the first memory block pointer available to the next one).

The second most important part of the DM management design space is about trying to deal with fragmentation. This means using de-fragmentation functions (i.e., coalescing and splitting memory blocks - see also 26). On the left, we can see the coalescing of memory blocks, which is the way to deal with external fragmentation. If a requested size (e.g. 20 KBytes) is bigger than the size of the available memory blocks (e.g. 10 KBytes), then it is possible to coalesce the two smaller blocks. On the right, it is shown the splitting of a memory block, which is the way to deal with internal fragmentation. If a requested size (e.g. 4 KBytes) is smaller than the size of the available memory block (e.g. 10 KBytes), then it is possible to split it into two smaller blocks. In this way, the remaining 6 KBytes

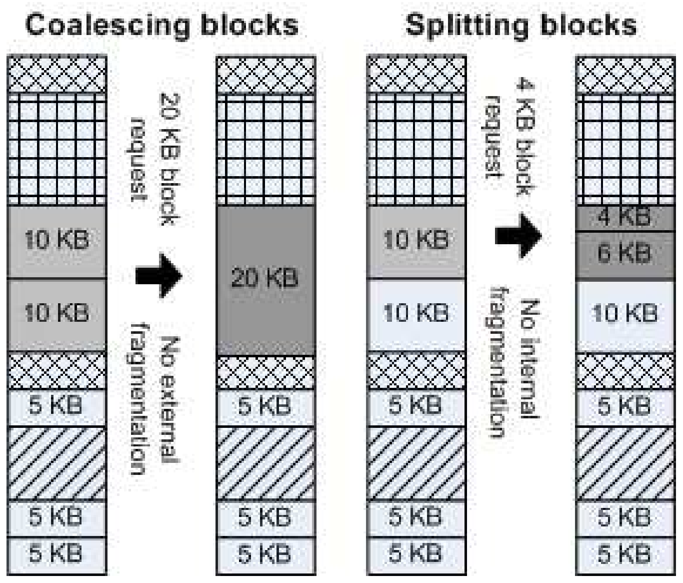


Figure 26: Dealing with external fragmentation by coalescing blocks and dealing with internal fragmentation by splitting blocks

are not wasted on internal fragmentation and can be re-used for a later request.

Finally, the third most important part of the design space is about trying to select the correct block to fulfill the memory request. This is done by the ”fit policies”, some of which can be seen in Figure 27. On the left, it is shown how ’first fit’ policy works. This policy satisfies the memory request with the first block that it finds, that is not reserved and has enough or more space than the request. Needless to say, this policy is fast, but produces big amounts of internal fragmentation since blocks of large sizes can be used for allocation requests that are small. In the middle, the ’exact fit’ policy is depicted. This policy will not stop looking for a block, unless it finds a block with the same size as the request. This policy eliminates internal fragmentation, but it is very slow due to the large amount of blocks that are needed to find the best candidate if the list of free blocks is extensive. Finally on the right side, we can see the ’approximate match’ policy. This policy satisfies the request according to a parameter, defined by the designer, which is a threshold that states how much it should look for a suitable block. In the case of Figure 27, this parameter states that the assigned block cannot be bigger than twice the size of the request. This is a more balanced approach than the two previous ones, neither wasting too much memory space nor slowing down the DM manager too much. However, the two previous options are also found in the literature in extreme cases where one of the two metrics (i.e. minimization of memory space or performance) is much more important than the other one.

6.2 Existing memory managers focus on different parts of the design-space

All the DM managers include these previous decisions of the design space in one way or the other in their designs. In the following we describe the main types of state-of-the-art DM managers.

First of all, one of the most popular DM manager, namely the Lea Allocator [26], is designed to optimize memory footprint by eliminating fragmentation, while preserving a reasonable speed. It is very frequently used in Linux-based systems. More specifically, it uses a very complex pool architecture, which prevents memory fragmentation and speeds up DM management. Then, it tries to defragment as much as possible, thus reducing even more the memory fragmentation, but slowing down DM management in a very significant percentage. Finally, it uses a combination of the previous best fit and first fit policies, which in total does not improve the speed a lot, but manages to preserve

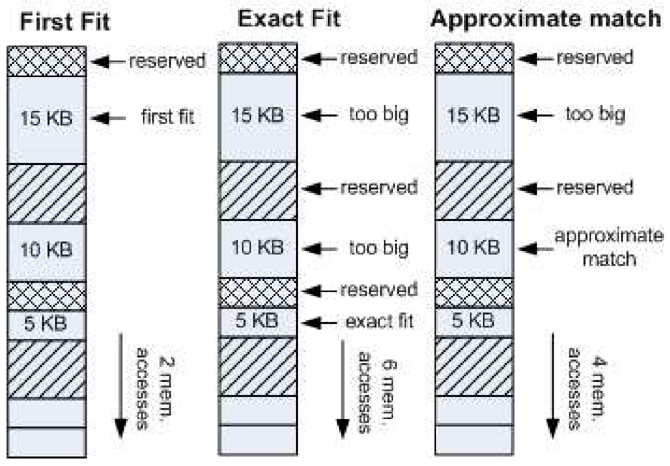


Figure 27: Fit policies example for a 5 KB block allocation

a reasonable low fragmentation level in a general context. The problem with the Lea Allocator, in the context of energy efficiency, is that it uses far too many memory accesses trying to defragment. All these memory accesses cause the power consumption to increase extensively.

Second, another very popular type of DM managers is the one which uses many simple fixed-sized pools to allocate memory. This style is used by Kingsley Allocator [15]. These allocators are very fast, but they deal poorly with fragmentation, thus use a big memory footprint. More specifically, they use a more straightforward definition of fixed-sized pools where only one size can fit in each pool perfectly and many lists of different sizes are pre-allocated during the initialization of the DM manager, which try to prevent as much fragmentation as possible by placing the memory requests in the correct pools but can result in a large portion of memory wastage if all the memory pools of different sizes are not used. However, the truth is that the main goal of these kind of managers is quick allocations and de-allocations and not limiting memory footprint consumption. As a result, since they use fixed-sized memory pools, they rarely (or even never) coalesce and split memory blocks thus ignoring defragmentation. This makes fragmentation usually even worse, but makes the DM manager even quicker. Finally, they also use a combination of best fit and first fit policies, which in total do not affect much speed, but unfortunately maintain the very high fragmentation level. The problem with this kind of allocators, in the context of energy efficiency, is that high fragmentation affects the energy per access, because we have to assign data with more fragmentation to bigger, more power-hungry memories.

Finally, several custom DM managers exist to satisfy the needs of specific types of applications and their memory requests. An example is the Obstacks [15] custom DM manager, which is used to optimize a stack-like behavior. Obstacks uses variable-sized pools, with no defragmentation support and an exact fit policy. This allocator is very fast and works well when many consecutive small-sized block allocations occur and finally one deallocation can be used to deallocate all the memory blocks at the same time (i.e. stack-like behavior). Any other behavior or big-sized requests, make the fragmentation grow extensively, thus reducing its energy efficiency because bigger memories are needed to store the data (as explained in the previous paragraph). The main limitation of these custom allocators is that they rely on a specific dynamic behaviour to work well and are manually designed and optimized. Unfortunately, no systematic methodology exists to create a custom DM manager from scratch to match a specific dynamic behavior of an application.

6.3 Our approach to create a low-power custom dynamic memory managers

Our approach consists of the study of the complete design search space. Also, we profile extensively the applications to define its dynamic behaviour and its dominant dynamic data allocation sizes. The combination of these two elements produces a systematic methodology to create custom DM managers. In contrast with the previous approaches we address the whole design space and not focus on small subsets that may work well for a certain number of applications and a concrete metric (e.g. performance as Kingsley-based managers). Thus, we create custom DM managers with reduced power consumption, because we try at the same time to have few memory accesses and keep a low fragmentation level. The key point of our energy-efficient approach is the study of the trade-offs between low memory accesses to improve speed and low memory fragmentation to reduce the use of memory footprint (for more details see [2, 3]). Moreover, all our custom DM managers are implemented in the middleware on top of the Operating System, hence platform independent. This means that they do not require any hardware changes, as opposed to [54].

As the basic template used in our energy-efficient DM managers, using the profiling information obtained in each concrete application, at least one fixed-sized pool is assigned to each of the sizes of the dominant data types. The sizes that are good candidates are those that imply at least 20% of the total amount of allocation requests. This approach prevents most of the memory fragmentation and speeds up the DM manager by a significant factor (i.e. between 10% - 50%) compared to those using only general-pools with many sizes. Then, we defragment only when this is absolutely needed to carefully balance the overhead of memory accesses required in the coalescing and splitting mechanisms. Mostly, we defragment when we are close to the high watermark of our memory space to try to avoid the allocation of the next requested memory block in a larger memory. Finally, a first fit policy is used only for dominant data types, for which we have already provided pools with their corresponding block size thus it is equivalent to exact fit. The remaining data types must make concessions and sacrifice some memory footprint using an "approximate fit" policy for preserving a certain degree of performance, because the number of memory accesses explodes if the DM manager searches the pools exhaustively to find the exact fit, as we indicated in the previous section.

We illustrate our technique with a real-life illustrative example: the Deficit Round Robin (DRR) application taken from the NetBench benchmarking suite [39]. It is a buffering and scheduling algorithm implemented in many wireless network routers today. Using the DRR algorithm the router tries to accomplish a fair scheduling by allowing the same amount of data to be passed and sent from each internal queue. It requires the use of DM because the input can vary enormously depending on the network traffic. The DRR algorithm has 3 types of dynamic data: the Internet packets, the packet headers and the list that accommodates the internal queues. It works in 3 phases when it is receiving packets. First, it checks the packet header; secondly it traverses the list of internal queues and, finally, it traverses the correct internal queue and stores the Internet packet in a FIFO order. Then, it forwards the packets in 3 additional phases. First, it traverses the list of internal queues, secondly, it picks up the first Internet packet and, finally, checks the packet header to forward it.

After analyzing the DM behavior and allocation sizes of this example with our approach, we decide to create two fixed-sized memory pools for the maintenance fields of the list of internal queues and the headers of the packets. Then, a general pool using approximate fit is used to store the internal variable-sized packets coming from the network that need to be forwarded. This choice was based on the dynamic sub phases found with our approach in the algorithm, which indicate that the list of internal queues has 85% of the total accesses on average and that this list is small enough to fit inside a small memory pool of 4 KB (or 8 KB in the worst case), which can increase significantly the locality when all the blocks are placed together and reduce enormously the complexity of DM management compare to other DM managers (i.e. 40% less energy than Lea and 45% less memory footprint than Kingsley-based DM managers).

Other experimental results in real-life embedded applications from the multimedia and wireless network domain show a significant reduction (up to 60%) in power consumption with the use of our approach [29]. Also, when other factors are really limiting (e.g. memory footprint), trade-offs can be made to achieve the desired results (i.e. reduction of 60% on average in memory footprint [2]) by and exhaustive exploration of the design space using our plug-and-play approach (see [3] for more details). Finally, run-time behavior profiling is embedded within our custom DM managers, so that real-time performance restrictions can be observed and deadlines met, using trade-offs between power consumption and performance [3].

7 Conclusion

In dynamic multi-threaded applications dealing with dynamic data and tasks is crucial. The memory bandwidth is both an issue at the shared SDRAM memory as well as on the local memory layer.

Modern multi-media applications contain multiple tasks and/or benefit from task parallelization. However, tasks running on multiple processors can access the same memory in parallel. This causes access conflicts that delay the system and increase its energy cost. Since existing techniques also optimize the memory bandwidth inside a single task, they cannot cope with inter-task conflicts. A need thus exists for techniques that optimize the memory bandwidth across the tasks' boundaries. We have therefore introduced several task-ordering/data assignment techniques for both the local memory layer as the SDRAM.

Multimedia applications consists of multiple tasks which are started/stopped at run-time due to user events. Also the tasks themselves have become data dependent. We have shown that design-time nor run-time techniques can effectively deal with this dynamic behavior. Therefore, we have introduced a novel scenario-based memory bandwidth approach. It combines the best of the design-time and run-time techniques.

Finally, run-time memory management requires an efficient management of the free space. We have introduced a methodology which customizes the dynamic memory managers for this purpose.

In future work, we want to automatically extract the inter-thread frame scenarios form the user's behavior.

References

[1] T.Vander Aa, M.Jayapala, F.Barat, G.Deconinck, R.Lauwereins, F.Catthoor, and H.Corporaal. Instruction buffering exploration for low energy vliws with instruction clusters. In *Proc. ASP-DAC*, pages x–x, Yokohama, Japan, Jan. 2004.

[2] David Atienza, Stylianos Mamagkakis, Francky Catthoor, José M. Mendías, and Dimitrios Soudris. Dynamic memory management design methodology for reduced memory footprint in multimedia and wireless network applications. In *In the Proceedings of Design, Automation and Test in Europe (DATE '04)*, Paris, France, February 2004. IEEE Press.

[3] David Atienza, Stylianos Mamagkakis, Jose Manuel Mendias, Francky Catthoor, Luca Benini, and Dimitrios Soudris. Efficient System-Level Prototyping of Power-Aware Dynamic Memory Managers for Embedded Systems. *Integration-The VLSI journal - Special Issue on Low Power Design*, Oct. 2004.

[4] A.Vincentelli and G.Martin. A Vision for Embedded Systems: Platform-Based Design and Software. *IEEE Design and Test Special of Computers*, 18(6):23–33, Nov. 2001.

[5] O. Avissar, R. Barua, and D. Stewart. Heterogeneous Memory Management for Embedded Systems. In *Proc. Cases*, pages 34–43, Sept. 2001.

[6] P. Banerjee, J. Chandy, M. Gupta, J. Holm, A. Lain, D. Palermo, S. Ramaswamy, and E. Su. Overview of the PARADIGM Compiler for Distributed Memory Message-Passing Multicomputers. *IEEE Computer*, 28(10):37–37, Mar. 1995.

[7] B.Rau. Iterative Modulo Scheduling. Technical report, HP Labs, 1995.

[8] C.Huang, S.Ravi, A Raghunathan, and N.Jha. High-level synthesis of distributed logic-memory architectures. In *Proc. Iccad*, pages 564–571, 2002.

[9] C.Lyuh and T.Kim. Memory Access Scheduling and Binding Considering Energy Minimization in Multi-Bank Memory Systems. In *Proc. 41th Dac*, pages 81–86, 2004.

[10] B. Dave, G. Laksshminarayana, and N. Jha. COSYN: Hardware-Software Co-Synthesis of Heterogeneous Distributed Embedded Systems. *IEEE Trans. VLSI Systems*, 7(1):92–104, Mar. 1999.

[11] E.Berger, K.McKinley, R.Blumofe, and P.Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proc. 8th Asplos*, pages 117–128, Nov. 1998.

[12] Hesham El-Rewini, Hesham H.Ali, and Ted Lewis. Task Scheduling in Multiprocessing Systems. *IEEE Computer*, 28(12):27–37, December 1995.

[13] Catthoor et al. *Custom Memory Management Methodology - Exploration of Memory Organisation for Embedded Multimedia System Design.* Kluwer Academic Publishers, Boston MA, 1998.

[14] Catthoor et al. *Data Access and Storage Management for Embedded Programmable Processors.* Kluwer Academic Publishers, Boston MA, 2002.

[15] P.R.Wilson et al. Dynamic storage allocation: a survey and critical review DDR. Technical report, Department of computer science, Univ. of Texas, 1995.

[16] P. Faraboschi, G. Brown, and J. Fischer. Lx: a Technology Platform for Customizable VLIW Embedded Processing. In *Int. Sym. Computer Architectures*, pages 203–213, 2000.

[17] F.Gharsalli, S.Meftali, F.Rousseau, and A.Jerraya. Automatic Generation of Embedded Memory Wrapper for Multiprocessor SoC. In *Proc. 39th Dac*, pages 596–601, Jun. 2002.

[18] J.I. Gomez, P. Marchal, D. Bruni, L. Benini, M. Prieto, F. Catthoor, and H. Corporaal. Scenario-based SDRAM-Energy-Aware Scheduling for Dynamic Multi-Media Applications on Multi-Processor Platforms. In *Workshop on Application Specific Processors (in conj. with MICRO)*, 2002.

[19] M. Hall, M. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Buignon, and M. Lam. Maximizing Multiprocessor Performance with SUIF. *IEEE Computer*, 29(12):84–89, Dec. 1996.

[20] H.Chang and Y.Lin. Array Allocation Taking into Account SDRAM Characteristics. In *Proc. ASP-Dac*, pages 447–502, 2000.

[21] H.Schmit and D.Thomas. Synthesis of application-specific memory designs. *IEEE Trans. VLSI Systems*, 5(1):101–111, Mar 1997.

[22] Texas Instruments. www.ti.com.

[23] J.A.Leijten. *Real-Time Constrained Reconfigurable Communication between Embedded Processors.* PhD thesis, Technishe Universiteit Eindhoven, Nov. 1998.

[24] J.Madsen and P.Jorgensen. Embedded System Synthesis under Memory Constraints. In *Proc. Codes*, pages 188–92, Rome, Italy, May 1999.

[25] K.Patel, E.Macii, and M.Poncino. Synthesis of Partitioned Shared Memory Architectures for Energy-Efficient Multi-Processor SoC. In *Proc. Date*, pages 700–701, 2004.

[26] Doug Lea. The lea 2.7.2 dynamic memory allocator. http://gee.cs.oswego.edu/dl/, 2002.

[27] Y. Li and W. Wolf. Hierachical Scheduling and Allocation of Multirate Systems on Heterogeneous Multiprocessors. In *Proc. Date*, pages 134–139, 1997.

[28] L.Lamport. The parallel execution of do-loops. *Communications of ACM*, 17(2):83–93, Feb. 1974.

[29] Stylianos Mamagkakis, David Atienza, Francky Catthoor, Dimitrios Soudris, and Jose Manuel Mendias. Custom design of multi-level dynamic memory management subsystem for embedded systems. In *Proceedings of Signal Processing Symposium (SiPS)*, Austin, Texas, USA, October 2004. IEEE Signal Processing Society and IEEE Circuits and Systems Society.

[30] H.De Man. On nanoscale integration and gigascale complexity in the post .com world. In *Proc. Date*, 2002.

[31] S. Meftali, F. Gharsalli, F. Rousseau, and A. Jerraya. An Optimal Memory Allocation for Application-Specific Multiprocessor System-on-chip. In *Proc. Isss*, pages 19–24, Sept. 2001.

[32] M.Jayapala, F.Barat, P.OpDeBeeck, F.Catthoor, G.Deconinck, and H.Corporaal. A low energy clustered instruction memory hierarchy for long instruction word processors. In *Proc. PATMOS*, pages 258–267, Sept. 2002.

[33] M.Kandemir, W.Zhang, and M.Karakoy. Runtime Code Parallelization for On-Chip Multiprocessors. In *Proc. Date*, pages 10510–10515, 2003.

[34] M.Pastrnak, P.Poplavko, P.de With, and J.van Meerbergen. Resource Estimation for MPEG-4 Video Object Shape-Texture Decoding on Multiprocessor Network-on-Chip. In *PROGRESS 2003, 4th seminar on embedded systems*, pages 185–193, Oct. 2003.

[35] M.Saghir, P.Chow, and C.Lee. Exploiting Dual Data Banks in Digital Signal Processors. In *Proc. ASPLOS*, pages 234–243, Jun. 1996.

[36] M.Verma, L.Wehmeyer, and P.Marwedel. Dynamic Overlay of Scratchpad Memory for Energy Minimization. In *Proc. Isss*, pages 104–110, Setp. 2004.

[37] M.Viredaz and D.Wallacha. Power Evaluation of a Handheld Computer. *IEEE Micro*, 23(1):66–74, Jan. 2003.

[38] M.Wolf. Improving locality and parallelism in nested loops. Technical report, Technical report CSL-TR-92-538, Stanford Univ., CA, USA, Sep. 1992.

[39] Netbench. http://www.veritest.com/benchmarks/netbench.

[40] ST Nomadik. www.st.com/stonline/prodpres/dedicate/proc/proc.htm.

[41] P.Grun, N.Dutt, and A.Nicolau. Memory Aware Compilation through Timing Extraction. In *Proc. 37th Dac*, pages 316–321, Jun. 2001.

[42] Philips. www.semiconductors.philips.com/platforms/nexperia.

[43] P.Marchal, D.Bruni, J.I.Gomez, L.Benini, L.Pinuel, F.Catthoor, and H.Corporaal. SDRAM-Energy-Aware Memory Allocation for Dynamic Multi-Media Applications on Multi-Processor Platforms. In *Proc. Date*, pages 10516–10523, Munich, Germany, Mar. 2003.

[44] P.Marchal, J.Gomez, D.Bruni, L.Benini, L.Pinuel, and F.Catthoor. Integrated task-scheduling and data-assignment to enable SDRAM power/performance trade-offs in dynamic applications. *IEEE Design and Test of Computers*, 11(1):1–12, Sept. 2004.

[45] P.Marchal, J.I.Gomez, L.Pinuel, S.Verdoolaege, and F.Catthoor. Loop morphing to optimize the memory bandwidth. In *Proc. IEEE ASAP*, Galvestone, TX, Sept. 2004.

[46] P.Marchal, J.I.Gomez, L.Pinuel, S.Verdoolaege, and F.Catthoor. Optimizing the memory bandwidth with loop fusion. In *Proc. Isss*, Stockholm, Sweden, Sept. 2004.

[47] P.Panda, N.Dutt, and A.Nicolau. Exploiting Off-Chip Memory Access Modes in High-Level Synthesis. In *Proc. Iccad*, pages 333–340, Oct. 1997.

[48] P.Poplavko, M.Pastrnak, T.Basten, J.van Meerbergen, and P.de With. Mapping of an MPEG-4 Shape-Texture Decoder onto an On-chip Multiprocessor. In *PRORISC 2003, 14th Workshop on Circuits, Systems and Signal Processing*, pages 139–147, Nov. 2003.

[49] P.Yang. *Pareto-optimization based Run-time Task Scheduling for Embedded Systems.* PhD thesis, Catholic University Leuven, 2004.

[50] P.Yang, P.Marchal, C.Wong, S.Himpe, F.Catthoor, D.Patrick, J.Vounckx, and R.Lauwereins. *Multi-Processor Systems on Chip*, chapter Cost-efficient mapping of dynamic concurrent tasks in embedded real-time multimedia systems, pages 46–58. Elsevier Science & Technology, 2004.

[51] J. Rabaey. Silicon Architectures for Wireless Systems Wireless Systems:Part 2 Configurable Processors. In *Tut. Hot Chips Conference*, 2001.

[52] Marcus T. Schmitz, Bashir M. Al-Hashimi, and Petru Eles. Cosynthesis of Energy-Efficient Multimode Embedded Systems with Consideration of Mode-Execution Probabilities. *IEEE Trans. Computer Aided Design of Integrated Circuits and Systems*, 24(2):153–169, Feb 2005.

[53] Semicon. www.semicon.org.

[54] M. Shalan and V. Mooney III. A Dynamic Memory Management Unit for Embedded Real-Time Systems-on-a-Chip. In *Proc. Cases*, pages 180–186, San Jose, CA, Nov. 2000.

[55] S.Hettiaratchi and P.Cheung. Mesh Partitioning Approach to Energy Efficient Data Layout. In *Proc. Date*, pages 11076–11081, Mar. 2003.

[56] S.Rixner, W.Dally, U.Kapasi, P.Mattson, and J.Owens. Memory Access Scheduling. In *Int. Sym. Computer Architectures*, pages 128–138, 2000.

[57] S.Wuytack, F.Catthoor, G.De Jong, and H.De Man. Minimizing the required memory bandwidth in VLSI system realizations. *IEEE Trans. VLSI Systems*, 7(4):433–441, Dec. 1999.

[58] R. Szymanek and K. Kuchcinski. A Constructive Algorithm for Memory-Aware Task Assignment and Scheduling. In *Proc. Codes*, pages 147–152, Apr. 2001.

[59] S. Tomar, S. Kim, N. Vijaykrishnan, M. Kandemir, and M. Irwin. Use of Local Memory for Efficient Java Execution. In *Proc. Iccd*, pages x–x, 2001.

[60] V.Delaluz, A.Sivasubramaniam, M.Kandemir, N.Vijaykrishnan, and M.Irwin. Scheduler-Based DRAM Energy Management. In *Proc. 39th Dac*, pages 697–702, 2002.

[61] V.Delaluz, M.Kandemir, N.Vijaykrishnan, A.Sivasubramaniam, and M.Irwin. Hardware and Software Techniques for Controlling DRAM Power Modes. *IEEE Trans. Computers*, 50(11):1154–1173, Nov. 2001.

[62] V. Vee and W. Hu. A Scalable and Efficient Storage Allocator on Shared-Memory Multiprocessors. In *Int. Symp. Parallel Architectures, Algorithms and Networks*, pages 230–235, Jun. 1999.

[63] W. Verhaegh, E. Aarts, P. van Gorp, and P. Lippens. A Two-stage Solution Approach for Multidimensional Periodic Scheduling. *IEEE Trans. Computer Aided Design of Integrated Circuits and Systems*, 10(10):1185–1199, Oct. 2001.

[64] Kiem-Phong Vo. Vmalloc: A general and efficient memory allocator. *Software Practice and Experience*, (26):1–18, 1996.

[65] W.Wolf and M.Kandemir. Memory system optimization of embedded software. *Proc. IEEE*, 91(1):165–182, 2003.

[66] Y.Choi and T.Kim. Memory Layout Techniques for Variables Utilizing Efficient DRAM Access Modes. In *Proc. 40 Dac*, pages 881–886, 2003.