Power aware data and memory management for dynamic applications

P. Marchal, J.I. Gomez, D. Atienza, S. Mamagkakis and F. Catthoor

Abstract: In recent years, the semiconductor industry has turned its focus towards heterogeneous multiprocessor platforms. They are an economically viable solution for coping with the growing setup and manufacturing cost of silicon systems. Furthermore, their inherent flexibility 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 into account and do not deal with the dynamic behaviour of these novel applications. The main limitations of current techniques are outlined and an approach for dealing with them is introduced.

1 Design challenges of media-rich services

Business analysts forecast a 250 billion dollar market for media-rich, mobile wireless terminals [1]. These systems require an enormous computational performance: 40 giga-operations per second (GOPS). Even though current PCs could provide sufficient performance, they are too power-hungry (10–100 W). Mobile devices should consume at least two or three orders of magnitude less [2]. 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 Era of platform-based design

A platform is a fixed microarchitecture together with a programming environment that minimises mask-making costs and is flexible enough to work for a set of applications [3]. The production volumes can then remain high over an extended chip lifetime.

To cope with the energy constraints, platforms usually consist of multiple processors. Since power is cubic to the processing frequency, parallelism is an effective way to reduce it. Therefore, on most platforms two or more processors are integrated. 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 control-dominated applications. The DSP, however, provides the computational performance for audio and video processing, while keeping the energy cost bounded. Indeed, taking a look to the current market offers (e.g. ST Nomadik [4], Philips Nexperia [5], TI OMAP [6]), it is clear that heterogeneous multiprocessor platforms are conquering the world of low-power embedded systems.

1.2 Desire for media-rich services

Platforms also perfectly support the next wave of media rich, wireless applications, bound to flood the multibillion dollar consumer market. Typical applications are media-players such the MPEG4 IM1 player. We summarise their most important characteristics in Fig. 1:

- *multi-threaded:* The systems contain multiple tasks which can execute in parallel. The tasks can either be independent or dependent. The system of Fig. 1 contains two parallel tasks (T1 and T2).
- *closed system:* Even though we can only determine at runtime which tasks execute and when they start, their type is known at design time and their source code is available. We assume that no other tasks can be downloaded on the system (such as e.g. Java applets or other software agents). For our example of Fig. 1, this entails that no other types of tasks but T1 and T2 can occur at runtime.
- *time constraints:* Tasks within multimedia applications are usually bound to time constraints. The most common deadline is the frame rate (see above). For a fluid video display, the tasks of a thread-frame have to finish within a deadline imposed by the frame rate. In the first frame of Fig. 1, we use a high frame rate, i.e. a tight deadline for T1. Thereafter, a user event relaxes the frame rate. In the remainder of this text, we mainly focus on the frame rate, despite the fact that 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 run time.

IEE Proceedings online no. 20045077

doi: 10.1049/ip-cdt:20045077

Paper first received 9th July and in revised form 6th October 2004

P. Marchal and F. Catthoor are with IMEC, Kapeldreef 75, Leuven, Belgium and with Katholieke Universiteit Leuven J.I. Gomez and D. Atienza are with DACYA, Universidad Complutense de Madrid, 28040 Madrid, Spain

S. Mamagkakis is with the Democritus University of Thrace, Xanthi, Greece

E-mail: marchal@imec.be

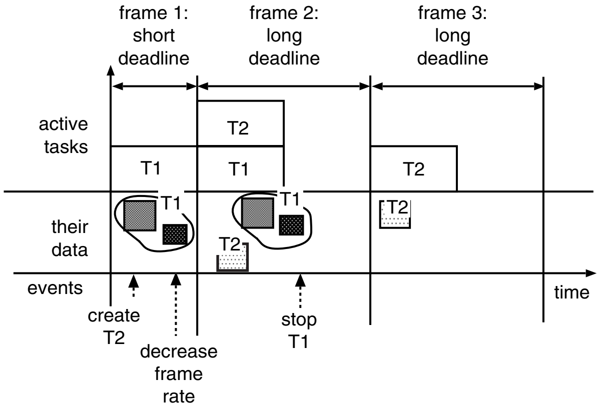

Fig. 1 *Characteristics of our application domain*

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.

- *data-dominated:* The tasks are data-dominated. As a result, the energy of the data memory architecture dominates the system cost. On multimedia systems, this assumption is particularly true after the cost of the instruction memory hierarchy is optimised (e.g. with [7, 8]). The data memory cost is then the remaining energy bottleneck. Consequently, optimising 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 in multimedia applications [9–11]. The most well known technique for improving the performance of the memory subsystem is to introduce a layered memory architecture [12]. Large memories used to store multimedia data have long access times. Therefore, they are too slow for feeding the processing elements at a sufficient rate. 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.

We focus on how to exploit a layered memory architecture. Particularly, we optimise the available bandwidth to the multiple memories/banks of each layer. This problem consists of detecting a data assignment and instruction schedule. It has to satisfy all time constraints while minimising the energy consumption. Although many bandwidth optimisation techniques already exist, they only improve the bandwidth within a basic block and assume that the memories are accessed by a single thread. Moreover, they require that the access pattern of the application can be analysed at the 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 characterise the access pattern at runtime. We will show that existing techniques break down under these circumstances, resulting in energy and performance loss.

In this paper, we overview the techniques which we have developed to overcome the above limitations. We have investigated on the one hand, design-time techniques for globally optimising the memory bandwidth, even across the tasks' boundaries. On the other hand, we have developed a combined design-time approach for dealing with the dynamic behaviour. It makes runtime decisions based on an extensive design-time analysis phase. Finally, we present how these runtime decisions can be energy-efficiently implemented at runtime. Before introducing our approach, we explain the memory architecture targeted throughout this paper and discuss the related work in more detail.

2 Target architecture

During our research we focus on a generic target architecture (Fig. 2). Different processing tiles contain multiple processing elements and are connected to a local memory layer. The processing elements are closely synchronised. A processing tile could be for instance a VLIW or a simple RISC processor (e.g. on a TI OMAP). The local memory layer on a processing tile comprises multiple scratchpad memories/banks. Again this closely resembles ST LX [13] or TI C6X [6] DSPs, where up to eight memories are included in the local layer. We do not directly exploit caches, but focus on scratchpad memories. These software-controlled memories do not require complex tag-decoding logic [10, 14]. Therefore, they have a lower energy cost per access compared to caches and also reduce the indeterminacy of the system. To further reduce their energy cost, we assume that they are heterogeneous. They can have different sizes, number of ports and access time.

Furthermore, the processing tiles share an offchip SDRAM (like on the TI OMAP or Philips Nexperia). We include the SDRAM in our overall target architecture, because it consumes up to 30% of the system energy cost of a PDA [15]. (This percentage is for a complete system including speakers, LCD, etc.)

We integrate a crossbar as communication architecture between the processing elements and the local layer as well as between the local layers and the shared SDRAM. Although a crossbar is not the most energy-efficient architecture, its energy cost is currently limited to only 10% of the global data transfer cost. (In principle, a more scalable communication architecture could be programmed or synthesised (such as e.g. [16]). However, research on

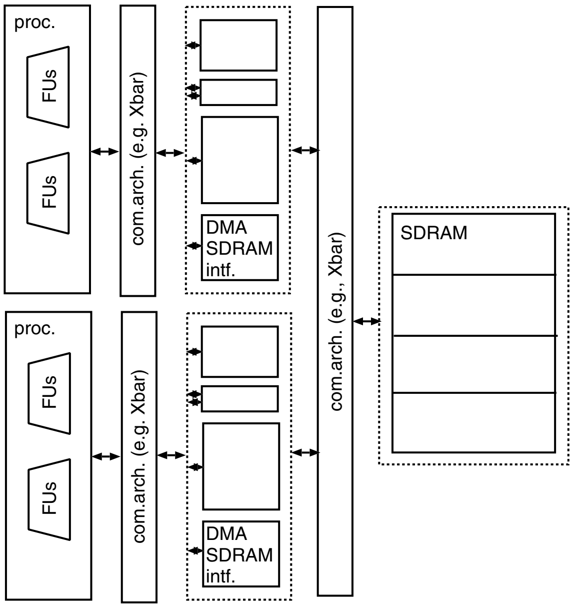

Fig. 2 *Target architecture for bandwidth optimisation*

advanced communication architectures fall outside the scope of this text.) Hence, not the communication architecture itself, but mainly the memory ports are potential bandwidth bottlenecks.

3 Surveying memory bandwidth optimisation techniques

Memory bandwidth optimisation is a widely researched topic It improves the bandwidth of a single memory layer. This layer can either consist of multiple SRAMs or of a large SDRAM memory (Fig. 2). We discern two methods which are commonly applied/combined to optimise the memory bandwidth; data layout transformations and instruction scheduling techniques. Their 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 first for SDRAMs and then 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 (Fig. 3). Each bank has a small buffer, called a 'page-buffer', that stores the last accessed datum together with its neighbouring data. The application can read these data elements at a low access latency. An access to another, non-neighbouring element, however, requires a much longer access time, because the data needs to be read from the memory plane. The latter is called a 'page-miss'. The less page-misses occur, the better the performance and energy of the SDRAM consumption become. Most methods below focus on transforming the application such that page-misses are avoided.

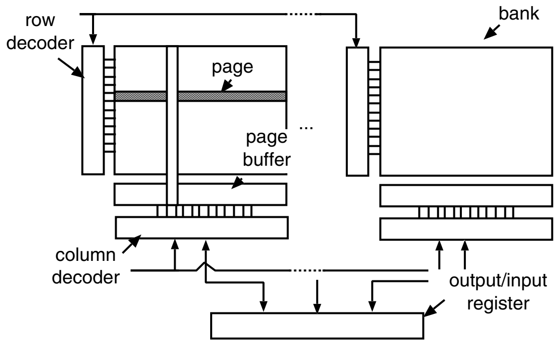

Fig. 3 *Multi-banked SDRAM architecture*

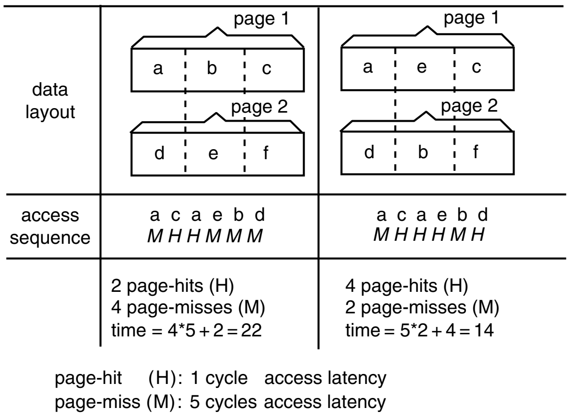

Fig. 4 *Different data layouts impact the number of page-misses*

3.1.1 Data layout transformations and data assignment techniques: The layout of the data in a memory bank defines how many page-misses occur (Fig. 4). To illustrate this, we map the scalars a, b, c, d, e, f in two different ways onto the pp. 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). For example, 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. For example, when we move e to the first page and b to the second one, only two page-misses remain (Fig. 4-right). Furthermore, it reduces 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 optimise it. In [17] arrays were partitioned 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 minimised. In [18] the number of page-misses were estimated based on a polyhedral description of the computation and a given layout. In [19], the scalar variables are laid out inside the program.

In contrast with the older DRAM architectures, most SDRAMs nowadays have more than one bank. For example, the Rambus' SDRAMs have up to 32 banks. Multiple banks provide an alternative way to eliminate page-misses. For instance, [20, 21] distribute data with a high temporal affinity over different banks such that page-misses are avoided. Their optimisations rely on the fact that the temporal affinity in a single-threaded application is analysable at the design time.

Thus, despite data assignment techniques existing for limiting the page-miss penalty, they are restricted to single-threaded, design-time analysable tasks. As we will motivate in Section 5.1, these techniques break down for dynamic multi-threaded applications.

3.1.2 Memory access reordering techniques: The access order also influences the number of page-misses (Fig. 5). In the left access order, every access causes a page-miss. However, when we slightly reorder the accesses, we reduce the number of page-misses (right). In research, but also in industry, both hardware and software techniques have been proposed to avoid page-misses in this way.

Hardware controllers to reorder the accesses are proposed in [22–26]. Typically, they buffer and classify memory

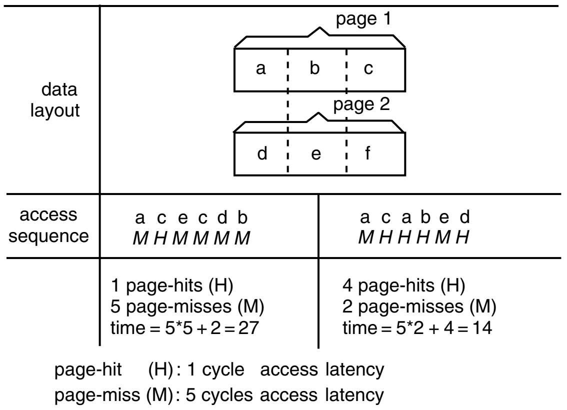

Fig. 5 *Access order impacts the number of page-misses*

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. Our focus, low-power design, is different. 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 (Section 5.2). In this way, we avoid the extra hardware which increases the energy consumption of all memory accesses.

Besides hardware approaches, several software ones have been presented too. Reference [27] exposes the special access modes of SDRAM memories to the compiler. As a result, their scheduler hides the access latency to the SDRAMs. The work was started in the context of system synthesis, but extended to VLIW compilers [28, 29]. Finally, [30] combines the scheduling technique of [27] with the memory energy model of [31] for reducing the static SDRAM energy.

The existing bandwidth optimisation techniques for SDRAMs rely on the fact that the access pattern to the data can be analysed for single-threaded applications. This is not the case in our application domain, where memory accesses from different threads are interleaved. Currently, no techniques analyse the access pattern across threads. The dynamic behaviour of our applications further complicates the problem, because the tasks' schedule is only known at runtime.

3.2 Bandwidth to the local memory layer

Complementary to the SDRAM layer, memory bandwidth optimisation has also been researched for the local memory layer. Their optimisation objective is mostly to reduce 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 synthesising a memory architecture which provides enough memory bandwidth, but is energy-or area-efficient too (e.g. [32–34]). 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).

DSPs have a local memory layer which consists of multiple memories (Fig. 6). This architecture has two single-ported memory banks (X, Y) which can be read in parallel. Most compilers model this memory layer as a monolithic memory with multiple ports. Under this assumption, as many memory operations can be scheduled in parallel as load/store units exist on the architecture. The compiler will even schedule accesses to the same memory bank in parallel. Although this simplifies the instruction scheduling, special hardware at runtime needs to serialise 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. After compilation, [35] maximises 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. Reference [36] optimises the register assignment and assignment of the arrays to the memory banks of a DSP together. The above techniques only focus on performance and do not try to optimise the access order.

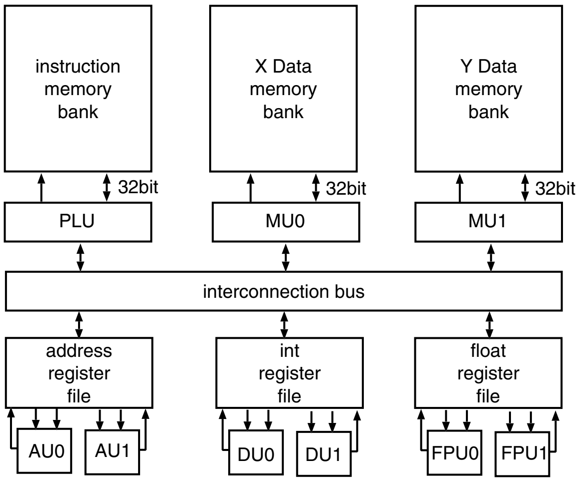

Fig. 6 *VLIW architecture [35]*

3.2.2 Access order: The order of the memory accesses has also an important impact on the performance (Fig. 7). Without changing the access schedule, the architecture 1 is the most energy-efficient. However, the dual-port memory remains an important energy bottleneck of this architecture. By rescheduling the memory accesses of the inner loop, we can eliminate the need for this memory (see architecture 7–2). It retains the same performance, but both data structures can now be mapped in a single-port memory, thereby reducing the energy cost from 0.23 mJ to 0.13 mJ. 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 optimise memory layout and access ordering together.

An example is [37]. It optimises 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. However, it only reorders the memory accesses within the scope of a basic block.

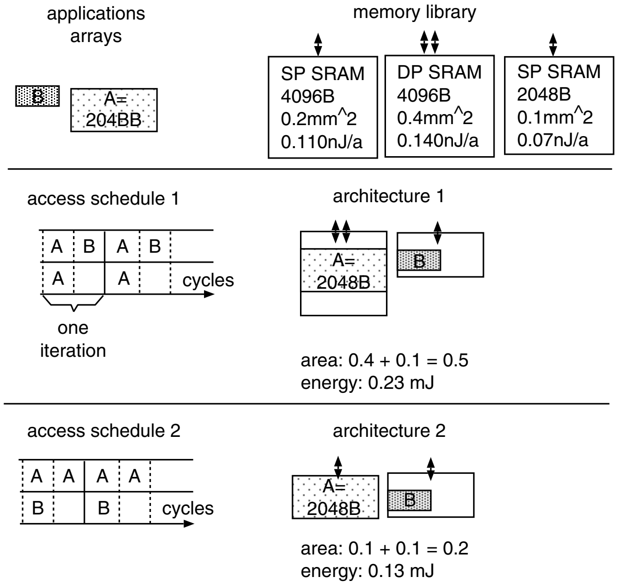

Fig. 7 *Reducing the memory cost with access ordering*

Several authors explored more global optimisation techniques to further improve the performance. In the high-performance community, several works tried to globally schedule instructions to parallelise code [38], but they do not focus on how to optimise the memory bandwidth. In the embedded processors context, [39] 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 bandwidth optimisation techniques for both the local and the shared memory layer are:

(i) *single-threaded applications*: they optimise the memory bandwidth for a single task at a time. As a result, we cannot directly use them in our context where we want to optimise the bandwidth across multiple tasks.
(ii) *static applications*: they obtain information on the locality for the data at the design time. The locality depends on which tasks are executing in parallel. Since in our application domain the actual schedule is only known at runtime, we can no longer extract it at the design time.
(iii) *no global optimisation*: the existing techniques only reorder the memory accesses within the scope of a basic block. No optimisations across the boundaries of the basic blocks are systematically applied, but this significantly reduces the potential performance gains and energy savings.

Several extensions are clearly needed for dealing with multiple threads and coping with the dynamic behaviour of our application domain. We discuss now techniques which optimise the memory hierarchy across the boundaries of a single task (Section 3.3) and overview the techniques for managing dynamic behaviour (Section 3.4).

3.3 Memory optimisation 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 parallelising applications while reducing the communication cost (e.g. the SUIF project [40–42] and the Paradigm compiler [43, 44]). However, they target an architecture which is very different from ours. For example, they rely on complex hardware to guarantee data coherency and consistency, which may come at an important energy penalty.

Also in the embedded systems' context, many authors have studied multi-threaded applications. For example, in [45] a top-down hierarchical scheduling heuristic maps regular DSP algorithms onto multiprocessors taking memory and time constraints into account. The authors take advantage of both temporal and spatial parallelism to optimise the throughput [46] follows a similar hierarchical approach, compiling code on a heterogeneous multiprocessor. The main disadvantage of these approaches 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, multiprocessor 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 [47, 48] for an overview). However, in recent years, the energy consumption has become an important bottleneck too. If energy is considered at all in task scheduling, the focus has been on the processing cores. Particularly, many techniques have been introduced on dynamic voltage scaling [49–51].

Unfortunately, only limited research exists in reducing the energy cost of the memory system. References [52, 53] both a describe a heuristic which does allocation, assignment, scheduling of multiple task-graphs and [54, 55] 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). These task-level approaches use a naive memory architecture model and hardly incorporate the real behaviour of the interconnections and memories.

Recently, [56, 57] discussed how many processors are required to execute code as energy-efficiently as possible. The task interaction is empirically accounted for (based on simulation), but that is not a scalable approach. Partitions the data space of a linked binary is partitioned in [58]. Each part is then mapped onto a memory bank. It selects the partition which optimises 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 these techniques for multi-threaded applications:

(i) They target an architecture which is either too different from ours or is not detailed enough. As a result, we cannot reuse them to optimise the interaction between parallel executing tasks.
(ii) Their program model is too limited for our application domain in which dynamic behaviour plays an important role too.

In the following Section, we review the current techniques for coping with the dynamic behaviour.

3.4 Runtime memory management

Dynamic applications are slowly becoming desirable in the context of embedded systems. Optimisations for dynamic applications require, on the one hand, runtime policies and, on the other hand, an efficient implementation of these policies (Fig. 8)

As indicated in the previous Sections, for memory bandwidth optimisation, 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 run time (step 2). In this

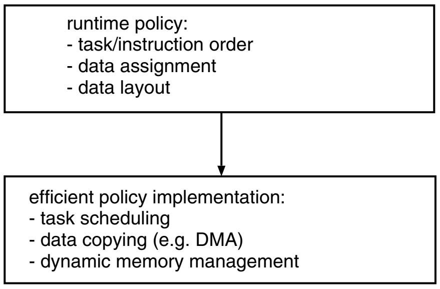

Fig. 8 *Runtime memory optimisation decomposed in two problems: decision taking and implementation*

Section, we overview both the runtime decision taking and implementation techniques.

3.4.1 Runtime policies: In the context of embedded systems, only a few techniques decide where to store the data at runtime taking the memory architecture into account For instance, [14, 59] decide at the 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 colocation 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. For example, 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 not only significantly reduces the available memory space, but also increases the allocation overhead. Several dynamic memory managers have been proposed for reducing fragmentation (see [61–63] for an overview). An important technique to eliminate fragmentation is adapting the dynamic memory manager to the allocation requests of the applications. The available memory in pools is split in [64]. 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 [65] 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 can reserve more memory in 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 multiprocessors, the most scalable and fastest memory managers use a combination of private heaps combined with a shared pool [62, 66]. These memory managers avoid typical multiprocessor 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 existing dynamic memory management techniques to manage, but we carefully control their allocation overhead.

We conclude from the above that:

(i) no decision techniques cope with the underlying memory architecture, albeit a multibanked SDRAM or the local memory layer
(ii) no runtime decision techniques optimise the memory bandwidth
(iii) despite dynamic memory management is a well researched problem, limited support is available to integrate these decisions inside the code.

4 Memory bandwidth optimisation for platform-based design

In this Section, we illustrate how parallel accesses from different processing elements either to the shared memory (Section 4.1) or the local memory layer (Section 4.2) degrade the system's performance and increase its energy consumption.

4.1 Shared layer

As motivated, the shared layer usually is based on a multibanked memory, such as an SDRAM. In this Section, we first explain, with an example, why existing techniques break down (Section 4.1.1). Then, we explain 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 (Section 4.1), but they cannot cope with ones caused by parallel threads. A small example explains why (Fig. 9). It consists of two tasks, 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. (For the clarity of our example, we only focus on their energy penalty, i.e. no performance penalty due to page-misses.) One way to minimise the number of page-misses is to carefully assign the tasks' data to the banks of the SDRAM. Current techniques optimise the assignment of a single task at a time (Section 3.1.1). In the 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. For example, 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 serialise 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 pp. 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 optimise the data layout within a single task.

4.1.2 Optimising the data assignment across the tasks' boundaries: We have built a technique for reducing page-misses across the tasks' boundaries [67]. It stores frequently accessed data structures with high access locality in separate banks. To identify these data structures, we have developed a heuristic parameter called selfishness (for details how to measure selfishness we refer to [67]). 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 weight it with the data structure's importance, i.e. we multiply it with the number of accesses to the data structure.

The higher the selfishness becomes, the more important it is to store the data in a separate bank. After analysing

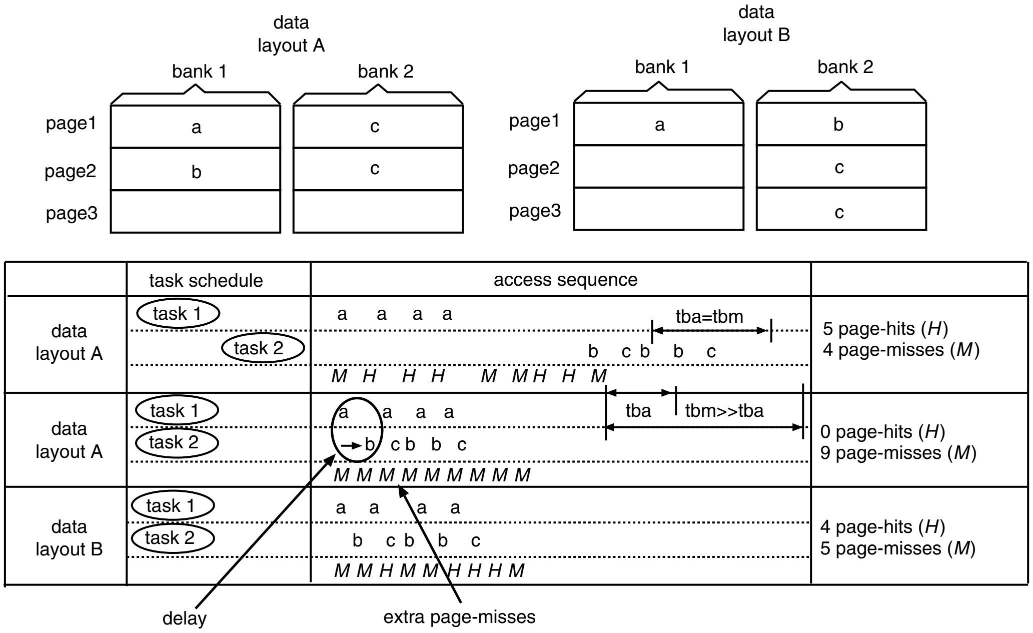

Fig. 9 *Interleaved accesses from different tasks cause page-misses and extra stalls*

the selfishness, our technique assigns the data to the banks, starting in order of decreasing selfishness. In our example, *a* and *b* both have the spatial locality, because the time between misses equals the entire duration of the task and the time between misses is the same. 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 even less accessed. Therefore, our algorithm first separates the most selfish data structures *a* and *b* and then decides to store *c* with *b* together in a single bank. As a result, only five page-misses remain and thus the energy cost is significantly reduced for the parallel execution.

4.1.3 Task ordering to trade-off energy/ performance:

Besides data assignment, 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 proposed a joined task ordering/data assignment technique [68]. It uses a genetic algorithm for quantifying the influence of task scheduling on the energy consumption and performance of the SDRAM. Its main flow is shown in algorithm 1 in Table 1.

Table 1: Algorithm 1 (main flow)

```
Main flow:
Input: Task set
Output: Pareto set of task scheduling/data assignment
  solutions
for all possible deadlines do
  Genetic algorithm(deadline, task-set)
end for
```

Our algorithm schedules the tasks of a given application such that the energy cost is optimised and a time constraint is met. The time constraint is given as a parameter to the algorithm and represents the deadline for the entire task-set. We run the algorithm several times for the same task-set but each time with a different deadline, thereby building a set of Pareto-optimal solutions. In Fig. 10, we depict the Pareto solutions for our example. The designer can pick the operating point which best fits his needs from the generated trade-off points. We will also use this Pareto set of solutions for our scenario-based runtime approach (Section 5).

4.2 Local memory layer

4.2.1 Access conflicts reduce the system's performance:

As indicated in Section 3.2, existing techniques only locally optimise the memory bandwidth. 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. 11-left), which are executed on a platform that consists of three memory ports fully connected to three single-port memories: two 4-kB ones (0.11 nJ/access) and a 2-kB one (0.06 nJ/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 now study the influence of loop fusion on the length of the memory access schedule and the energy cost.

During instruction scheduling, most compilers simply assume that any memory operation finishes after *n* cycles. When the executed operation takes longer than presumed,

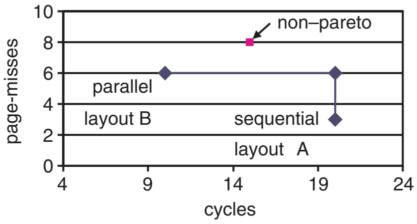

Fig. 10 *Energy/performance trade-off for task1 and task2 of Section 5.1*

```
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;
```

Fig. 11 *Motivational example: original code (left), code after fusion (right)*

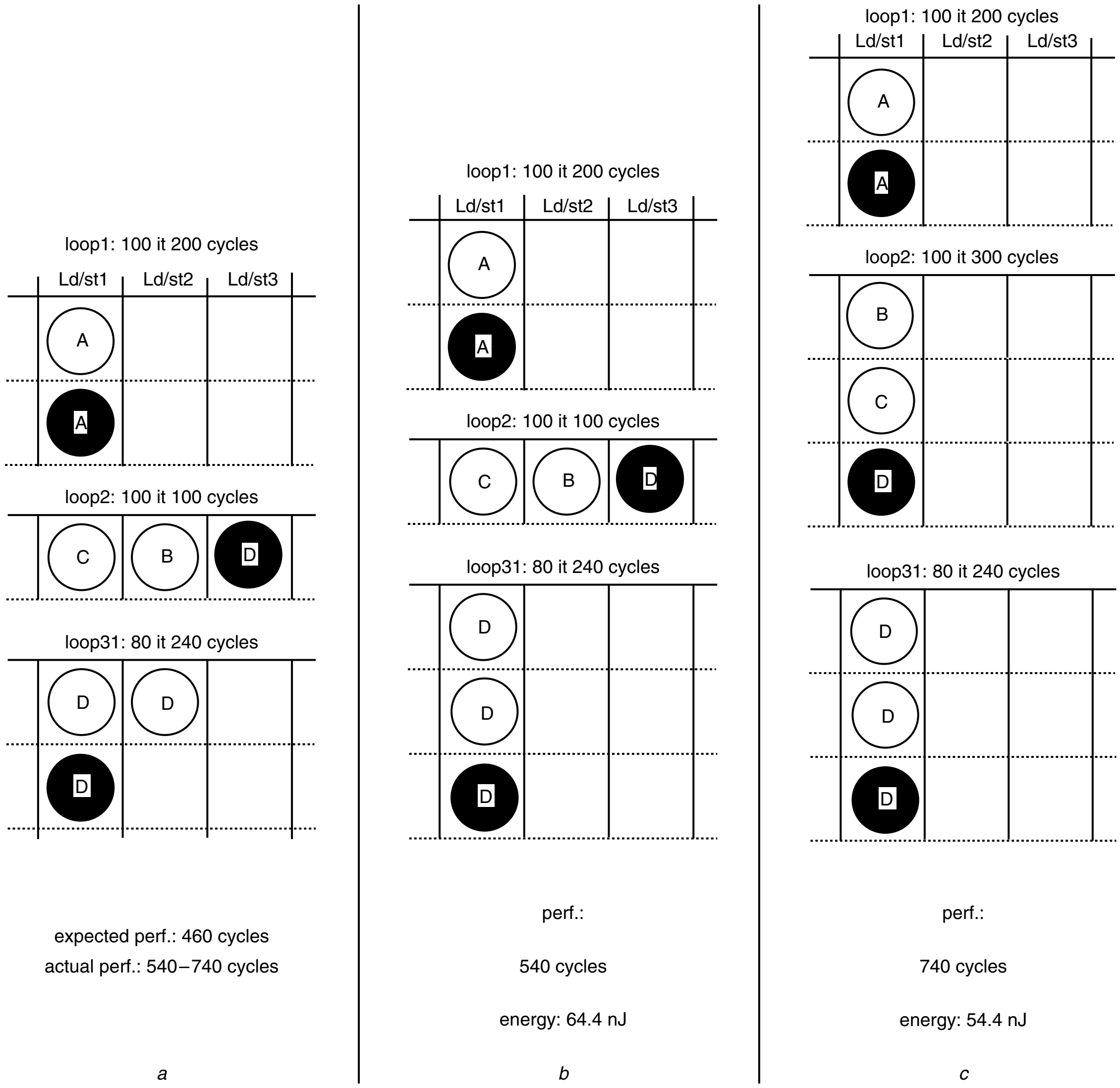

Fig. 12 *Empty issue slots in the memory access schedule of the inner-loops*

a Existing compiler
b With fastest partial data assignment
c With most energy-efficient partial assignment

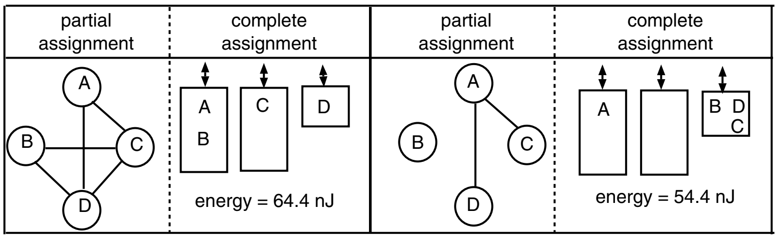

Fig. 13 *Partial assignment expressed with a conflict graphs*

(left) fast; (right) more energy-efficient

the entire processor is stalled. As a result, often a large difference exists between the expected and the effective performance of the processor. For instance, an existing modulo scheduler [69] generates a memory access schedule for the inner loops of 460 cycles (Fig. 12*a*). (Note how the modulo scheduler schedules read/write operations from the same instruction in the same cycle.) However, the actual performance varies between 540 and 740 cycles. The schedule takes longer than expected because the processor has to serialise 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 the way the linker assigns the data to the memories has such a large impact on the performance of the system, [37] optimises the data assignment and the memory schedule together. They impose restrictions on the assignment such that the energy is optimised, but still guarantee that the time-budget is met. The assignment constraints are modelled with a conflict graph (e.g. Figure 13-left). The nodes correspond to the data structures of the application. An edge between two data structures indicates that we need to store the data in different memories. Hence, the corresponding accesses to these data structures can be executed in parallel. For instance, the edge between *A* and *C* forces us to store both data structures in different memories. The schedule for this conflict graph takes 540 cycles (Fig. 12*b*). It consumes 64.4 nJ (We compute the energy consumption as follows: $\sum_{\forall m \in M} \sum_{\forall ds \in m} \mathrm{NrAccess}(ds) E^{m}_{access}$, because the conflict edges force us to store both *C* and *B* in large memory (see complete assignment in Fig. 13-left).

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 *D*–*C*, the data structures *B*, *D* and *C* can be assigned in the smallest, but most energy-efficient memory (Fig. 13-right). The energy consumption is then 54.4 nJ instead of the original 64.4 nJ. Less conflicts also imply that less memory accesses can execute in parallel. The code now takes 740 cycles (Fig. 12*c*). The energy savings thus come at a performance loss.

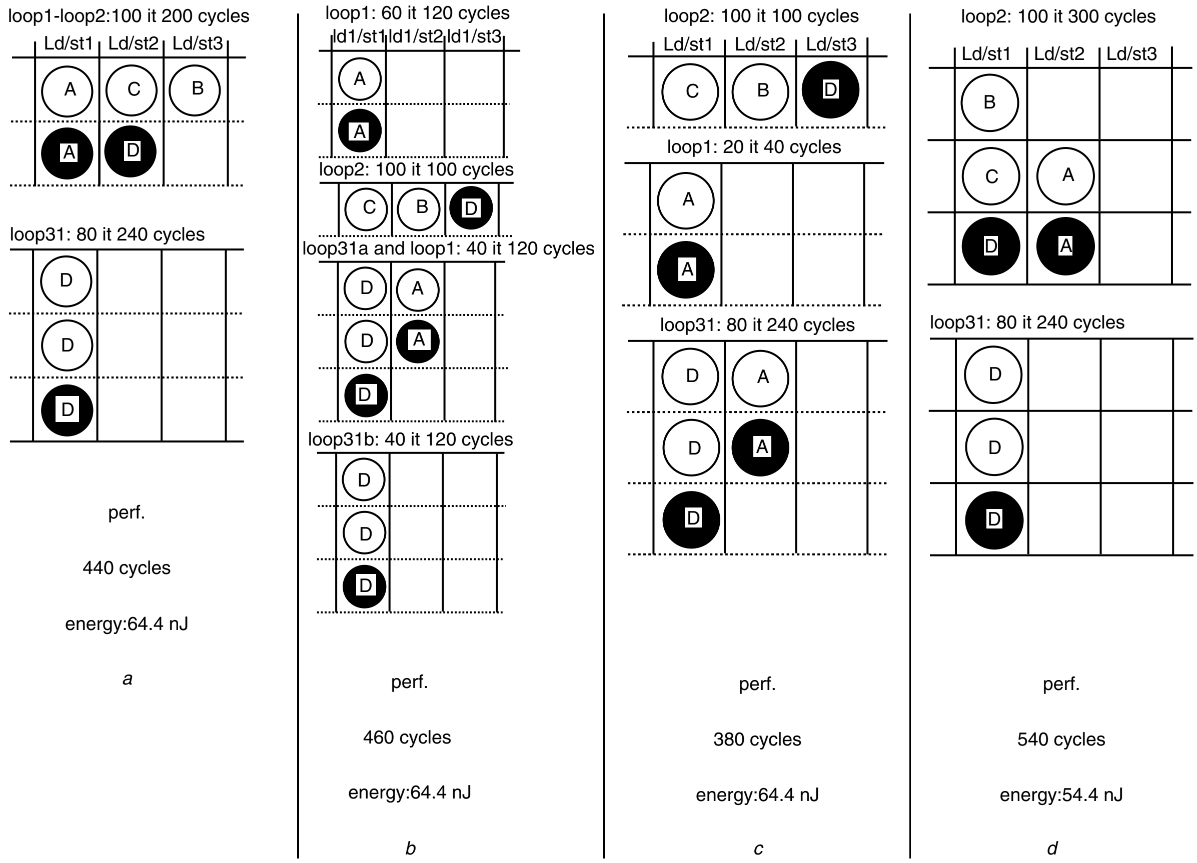

Fig. 14 *Loop fusion fills the issue-slots*

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

However, many memory access slots remain empty. This is mainly because: (i) interiteration dependencies; for instance the initiation interval of loop 1 is 2, because A depends on itself and, hence, only 30% of the available memory slots are used; (ii) we do not use power hungry multiport 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 Loop morphing: With more global optimisations, such as loop fusion [70], we can further compact the application's schedule. For instance, we can try to fuse the loops of our examples. We only consider loops between which no dependences exist. We can then, on the one hand, fuse loop 1 and loop 2. For the fastest conflict graph (Fig. 13-left), the resulting schedule takes 440 cycles (Fig. 14*a*). On the other hand, we can also fuse loop 1 and loop 31 (Fig. 14*b*). Because both loop nests are not conformable, we can in this case only fuse 40 iterations. This results in a schedule length of 460 cycles. However, if we combine loop fusion with strip mining and loop splitting, we can combine 80 iterations. As a result, the schedule length takes only 380 cycles (Fig. 14*c*). We show the fused code for this decision in Fig. 11-right.

In all three cases, the energy cost remains the same because we keep the same conflict graph. If we change the conflict graph, we need to take different fusion decisions. For example under the more energy-efficient conflict graph (Fig. 14*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. 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.

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

(i) we can overlap loops even with nonconformable loop headers. Otherwise, the number of overlapping iterations after fusion is limited. Therefore, we have proposed loop morphing [71]. It combines loop fusion, strip mining and loop splitting to increase the instruction level parallelism in as many iterations as possible.
(ii) we combine the loops which result in the largest performance gains. Therefore, we have presented a decision mechanism in [72]. It pairwise fuses loops, which considers memory size, number of ports, access latency and assignment constraints. Similar to our technique for the shared memory layer (Section 4.1.3), it generates a set of energy/performance optimal operating points.

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

5 Scenarios for coping with dynamic behaviour

5.1 Energy constraints necessitate for runtime decisions

Due to the dynamic behaviour 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 (Fig. 15). At the start of each frame, the user either executes task1 and/or task2. They are the same tasks as in Section 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. For example, 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.

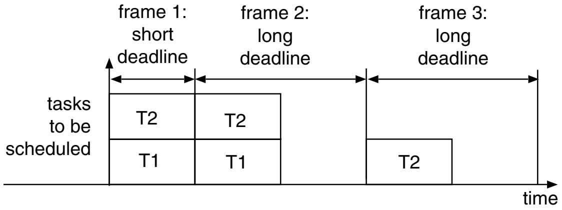

Fig. 15 *Dynamically created tasks with their deadline*

The optimal task order/data assignment decisions vary from frame to frame (see Fig. 16). For example, to satisfy the short deadline in frame1, we have to schedule both tasks in parallel. We obtain the least number of page-misses using layout B from Fig. 9. However, in frame2, only task2 is active. As indicated in Fig. 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 use layout A to eliminate most page-misses. So, in each frame, different scheduling/assignment decisions are optimal for energy. However, we can only take them at runtime.

With current approaches more page-misses always occur. Design-time techniques only select one task schedule and layout (Fig. 17*a*). This single design has to meet the

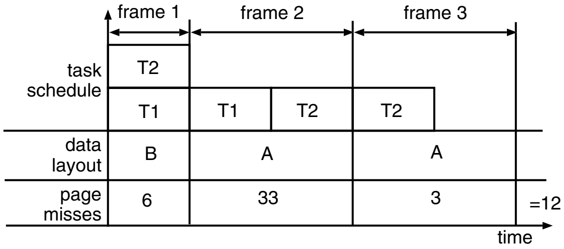

Fig. 16 *Our runtime task schedule/SDRAM assignment solution*

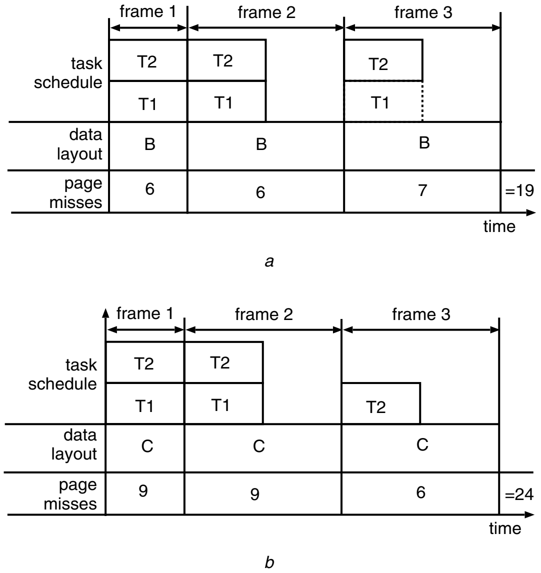

Fig. 17 *State-of-the-art task schedule/SDRAM assignment*
a Design-time
b Operating system based solution

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 (Fig. 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.

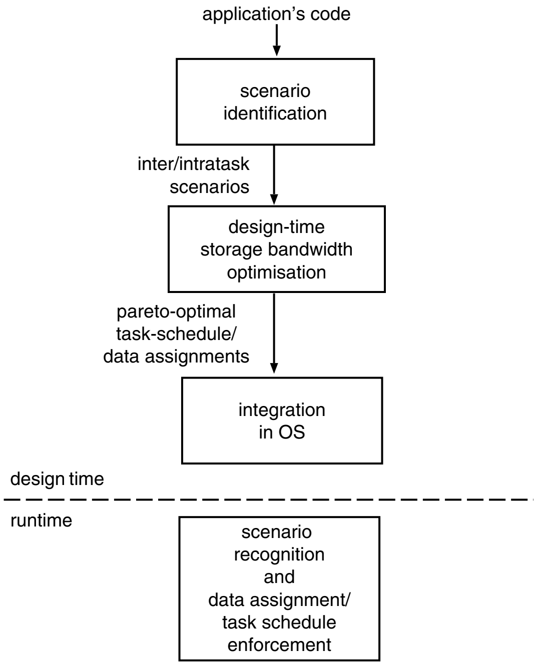
application's code
scenario identification
inter/intratask scenarios
design-time storage bandwidth optimisation
pareto-optimal task-schedule/ data assignments
integration in OS
design time
runtime
scenario recognition and data assignment/ task schedule enforcement

Fig. 18 *Scenario-based design-time/runtime approach*

Furthermore, current runtime approaches are far from optimal. A typical OS does not account for the specific behaviour 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 (Fig. 17*b*). 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.

These results indicate the potential benefits for a runtime technique, which considers the SDRAM behaviour and can generate the solutions of Fig. 16. 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 propose a mixed design-time/runtime approach for coping with the dynamic behaviour. The philosophy behind it is to take most scheduling/assignment decisions at the design time for all frequently occurring task-sets. In this way, we can reuse our bandwidth optimisation techniques multi-threaded applications and at the same time limit the runtime complexity. Only for the more seldomly occurring task-sets is a pure runtime decision taken as a backup solution. In the following paragraphs, we explain the main steps of our methodology (Fig. 18).

Scenario identification: Fixing as many decisions as possible at the design time comes at the risk of ignoring the actual behaviour and generating worst-case designs. For example, consider the code in Fig. 19. 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,

```
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);
     }
  ...
...
}
```

a

```
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);
    }
  ...
```

b

```
// 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);
    }
  ...
```

```
// 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++){
  ...
 }
```

c

Fig. 19 *Scenario-based approach*

a Data-dependent conditions as a limiting factor for bandwidth optimisation techniques
b Worst-case approach
c Scenario-based approach

optimising thus the design for the worst-case load. As a consequence, we heavily overestimate the required bandwidth and usually generate an over-dimensioned and energy-inefficient system.

To prevent this energy loss, we try to capture the dynamic behaviour with scenarios. First, we analyse which tasks-sets often co-occur at runtime. We call them 'intertask scenarios'. A similar but more restrictive concept is used by [73]. Secondly, we narrow down the data-dependent behaviour inside the tasks with introtask scenarios. An intratask scenario is an execution path through the task (or a combination of execution paths) for different data-dependent parameters [51, 74]. Both the intertask and intratask scenarios should be manually extracted by the designer (using profiling). For example, in the code of Fig. 19, we derive two intratask scenarios, one for *mode* equals true and another one for *mode* equals false. Research is ongoing in how to identify scenarios [75, 76].

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 optimisation at design time: In the second step, we optimise the storage bandwidth of each scenario. Our design-time techniques generate for each scenario a set of task ordering/data assignment solutions. Each solution optimises the energy cost for a given time-budget. From this set, we only retain the Pareto-optimal solutions. For example, for our example's scenario in which task1 and task2 are active (Section 5.1), we would generate the Pareto curve of Fig. 10. Finally, we integrate the Pareto set of each scenario into the operating system. We provide more details in [77].

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. For example, for frame 1 of our example (Section 5.1), our runtime mechanism 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 analysed at the design time, we use a backup solution. For example, 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. 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.

6 Dynamic memory management

The memory space available at runtime to our applications can be located in any physical memory of the system. It is managed with the help of a dynamic memory manager (DMM). The DMM keeps track of the unused memory blocks and has internal routines to find the best fitting free block for an allocation request. The DMM should use the memory space as efficiently as possible while keeping the allocation overhead low. The main difficulty is that the memory space can easily become fragmented because the requested data blocks have a different size and lifetime (see Fig. 20) Besides, data locality [78], sparseness [79] and ordering of blocks inside pools can be an issue.

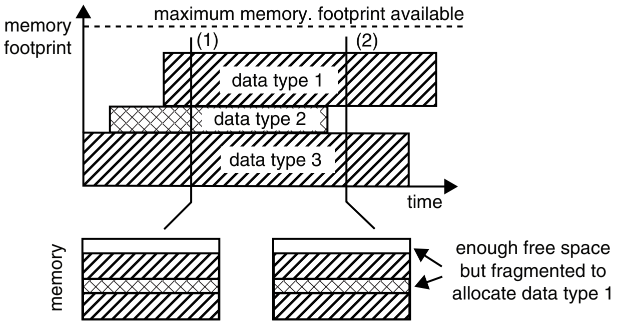
memory footprint
maximum memory. footprint available
(1)
(2)
data type 1
data type 2
data type 3
time
memory
enough free space but fragmented to allocate data type 1

Fig. 20 *Fragmentation in dynamic memory management*

To customise a DMM, we have classified the different design options in orthogonal decision trees. Together they compose the design space for dynamic memory management [80]. We give a brief overview of the design space in following subsection. It explains the design decisions taken in existing DMMs (Section 6.2), but also helps in matching the DMM with the application for reducing its energy consumption (Section 6.3).

6.1 DMM design space

In each dynamic memory manager three important decisions are made: (i) how to divide the memory space in pools; (ii) how to defragment the memory system; and (iii) how to select a free block.

The most well known way of preventing fragmentation is dividing the memory space into pools. Each pool contains several blocks of a similar size. When an application asks for space, a free block is returned from a pool which best matches the required size. If that block equals the requested size, then no memory space is wasted and no (internal) fragmentation occurs. By carefully selecting the number of pools, the size and the block size of each pool fragmentation can be avoided. This is possible in practice since usually less than ten different block sizes cover more than 80–90% of the allocation requests.

A second important decision for DMM is how to apply defragmentation techniques, such as coalescing/splitting (see Fig. 21). If an application needs a larger chunk than the size of any free memory block, we can coalesce two smaller blocks (left). In contrast, if the application requires less than the size of a free memory block, we can split the block

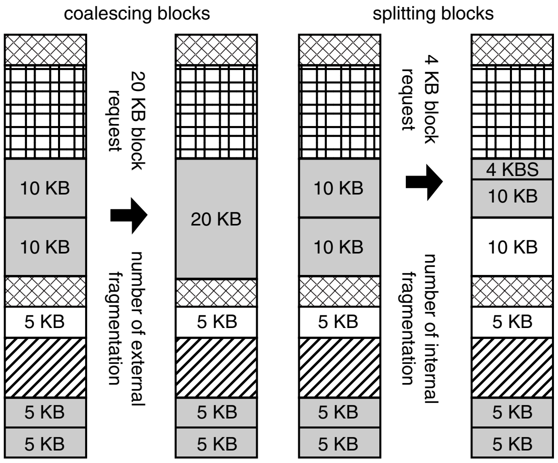

Fig. 21 *Coalescing and splitting techniques for coping with fragmentation*

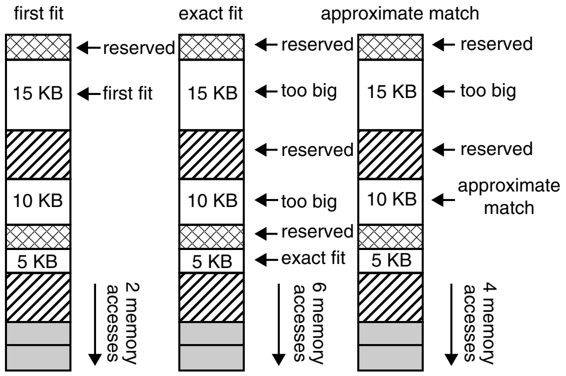

Fig. 22 *Fit policies*

a First fit
b Exact fit
c Approximate fit

(right). A later request can then use the remaining part of the free block.

Finally, the last decision is how we select a free block to satisfy a memory request. Three policies are widely applied: 'first fit', 'best fit' and 'exact fit' (see Fig. 22). The 'first fit' policy [63] returns the first free block that satisfies the memory request. Despite this policy being very fast, it easily fragments the memory space. In contrast, the 'exact fit' policy [63] reduces fragmentation. It iterates over all free blocks until it finds one with the same size as the memory request. Obviously, the execution time overhead of this policy is higher. Finally, the 'approximate match' policy [63] is a compromise between the 'first fit' and 'exact fit' policies. It searches for a block which can satisfy, but is not larger than n times the required size. If n is equal to one, this policy is the same as the 'best-fit' one. It has then a high execution time overhead. By increasing n, we can exchange the execution time overhead for more fragmentation.

6.2 Situating existing DMMs in the design space

A very popular allocator is the Lea Allocator [81], which is optimised for a low memory footprint. It uses several pools and frequently runs defragmentation routines. The latter, however, slow down the dynamic memory manager. It uses either a 'best fit' or a 'first fit' policy for selecting a free block. Compared to a 'first fit'-only policy, this further reduces fragmentation, but comes at an extra performance penalty. The Lea Allocator is not very energy-efficient due to the memory accesses necessary for managing the memory space.

A second well known memory manager is the Kingsley Allocator [63] that is mainly optimised for performance. It uses several pools with fixed-size blocks. To ensure fast allocation/deallocation, it use a relatively large memory space such that free blocks can always be found. Furthermore, it avoids running defragmentation routines that could slow down the memory manager. Finally, it selects a memory block in a pool with a 'first fit' policy, but uses 'best fit' to select a pool. This memory manager is again not very energy-efficient, because it requires more memory space and that increases the energy cost per access.

A third category of memory managers are custom DMMs. They are built for satisfying the needs of specific applications. An example is the Obstacks [63] custom DMM, which optimises a stack-like (de)allocation behaviour. Obstacks uses variable-sized pools, with no defragmentation support and applies a 'best fit' policy. This allocator works well for small-sized memory requests with a stack-like behaviour. Any other (de)allocation behaviour or bigger-sized requests can increase significantly the management overhead (in number of memory accesses) and the memory fragmentation.

The above general purpose allocators are thus not energy-efficient. The custom allocators, however, are manually optimised for a specific allocation pattern. However, no systematic methodology exists for creating such a custom DMM.

6.3 Methodology for creating low-power DMMs

We have developed a methodology for creating low-power DMMs. We customise a DMM by trading-off memory fragmentation with the memory accesses necessary for managing the free space. It consists of two phases. First, we extensively profile the application to characterise its allocation pattern and to identify the most important dynamic data types [79]. Secondly, we use the gathered information for deciding the pool structure. For example, to prevent fragmentation and to speed up the DMM, we assign at least one fixed-sized pool for each dominant data type. We only defragment when the memory usages exceeds a high watermark (when 70–90% of the memory is used). Finally, we apply the 'exact fit' policy for the dominant data types and 'approximate fit' for the remaining data types.

Experimental results indicate that our custom DMMs may reduce the energy consumption up to 60% (e.g. [78]). The same design space can be explored for optimising different objectives, such as memory footprint (for more details on the methodology and extensive results see [80]).

7 Conclusions

Memory bandwidth has always been an issue for the energy/performance of multimedia systems. However, for the dynamic multi-threaded applications, several extensions are required to current bandwidth optimisation techniques.

Due to the concurrency, several tasks can access the same memory in parallel, thereby causing access conflicts. This delays the system and increases its energy cost. Existing techniques optimise the memory bandwidth inside a single task and cannot cope with these intertask conflicts. Therefore, we have therefore introduced several task-ordering/data assignment techniques that optimise the memory bandwidth across the tasks' boundaries.

The dynamic behaviour is the result of user events that start/stop tasks at runtime and data dependencies within the tasks. We have shown that neither design-time nor runtime techniques can effectively deal with it. Therefore, we have introduced a novel scenario-based memory bandwidth approach. It combines the best of the design-time and runtime techniques.

Finally, energy-efficiently managing the memory space at runtime, requires a custom organisation of the free space. We provide a heuristic for matching the DMM to the application. It generates a DMM based on a complete description of the DMM's design space.

8 Acknowledgments

This work is supported in part by the Spanish Government under grant CICYT TIC 2002–750 and the Flemish IWT.

9 References

1 Semicon.: www.semicon.org

2 De Man, H.: 'On nanoscale integration and gigascale complexity in the post.com world'. Proc. DATE, 2002

3 Vincentelli, A., and Martin, G.: 'A vision for embedded systems: platform-based design and software', *IEEE Des. Test Comput.*, 2001, **18**, (6), pp. 23–33

4 ST Nomadik.: www.st.com/stonline/prodpres/dedicate/proc/proc.htm

5 Philips.: www.semiconductors.philips.com/platforms/nexperia

6 Texas Instruments.: www.ti.com

7 Vander Aa, T., Jayapala, M., Barat, F., Deconinck, G., Lauwereins, R., Catthoor, F., and Corporaal, H.: 'Instruction buffering exploration for low energy vliws with instruction clusters'. Proc. ASP-DAC, Yokohama, Japan, Jan. 2004

8 Jayapala, M., Barat, F., OpDeBeeck, P., Catthoor, F., Deconinck, G., and Corporaal, H.: 'A low energy clustered instruction memory hierarchy for long instruction word processors'. Proc. PATMOS, Sept. 2002, pp. 258–267

9 Catthoor, *et al.*: 'Custom memory management methodology – exploration of memory organisation for embedded multimedia system design' (Kluwer Academic Publishers, Boston, MA, 1998)

10 Panda, P., Catthoor, F., Dutt, N., Danckaert, K., Brockmeyer, E., Kulkarni, C., Vandecappelle, A., and Kjeldsberg, P.G.: 'Data and memory optimizations for embedded systems', *ACM Trans. Design Autom. Electron. Syst.*, 2001, **6**, (2), pp. 142–206

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

12 Hennessy, J., and Patterson, D.: 'Computer architecture: a quantitative approach' (Morgan Kaufmann Publishers, San Fransisco, CA, 1996, 2nd edn.)

13 Faraboschi, P., Brown, G., and Fischer, J.: 'Lx: a technology platform for customizable VLIW embedded processing'. Int. Symp. Computer Architectures, 2000, pp. 203–213

14 Avissar, O., Barua, R., and Stewart, D.: 'Heterogeneous memory management for embedded systems'. Proc. CASES, Sept. 2001, pp. 34–43

15 Viredaz, M., and Wallacha, D.: 'Power evaluation of a handheld computer', *IEEE Micro*, 2003, **23**, (1), pp. 66–74

16 Rabaey, J.: 'Silicon architectures for wireless systems wireless systems: Part 2 configurable processors'. Tut. Hot Chips Conf., 2001

17 Hettiaratchi, S., and Cheung, P.: 'Mesh partitioning approach to energy efficient data layout'. Proc. DATE, Mar. 2003, pp. 11076–11081

18 Kim, H., Vijaykrishnan, N., Kandemir, M., Brockmeyer, E., Catthoor, F., and Irwin, M.J.: 'Estimating influence of data layout optimizations on SDRAM energy consumption'. Proc. ISLPED, Aug. 2003, pp. 40–43

19 Choi, Y., and Kim, T.: 'Memory layout techniques for variables utilizing efficient DRAM access modes'. Proc. 40th DAC, 2003, pp. 881–886

20 Panda, P.: 'Memory bank customization and assignment in behavioral synthesis'. Proc. ICCAD, Oct. 1999, pp. 477–481

21 Chang, H., and Lin, Y.: 'Array allocation taking into account SDRAM characteristics'. Proc. ASP-DAC, 2000, pp. 447–502

22 Ayukawa, K., Watanabe, T., and Narita, S.: 'An access-sequence control scheme to enhance random-access performance of embedded DRAM's', *IEEE J. Solid-State Circuits*, 1998, **33**, (5), pp. 800–806

23 Rixner, S., Dally, W., Kapasi, U., Mattson, P., and Owens, J.: 'Memory access scheduling'. Int. Symp. Computer Architectures, 2000, pp. 128–138

24 Macian, C., Dharnapurikar, S., and Lockwood, J.: 'Beyond performance: secure and fair memory management for multiple systems on a chip'. Proc. FPL, 2003

25 Gharsalli, F., Meftali, S., Rousseau, F., and Jerraya, A.: 'Automatic generation of embedded memory wrapper for multiprocessor SoC'. Proc. 39th DAC, Jun. 2002, pp. 596-601

26 Denali Software Databahn. www.denali.com

27 Panda, P., Dutt, N., and Nicolau, A.: 'Exploiting off-chip memory access modes in high-level synthesis'. Proc. ICCAD, Oct. 1997, pp. 333–340

28 Grun, P., Dutt, N., and Nicolau, A.: 'Memory aware compilation through timing extraction'. Proc. 37th DAC, Jun. 2001, pp. 316–321

29 Halambi, A., Grun, P., Ganesh, V., Khare, A., Dutt, N., and Nicolau, A.: 'EXPRESSION: a language for architecture exploration through compiler/simulator retargetability'. Proc. DATE, Paris, France, Mar. 1999, pp. 485–491

30 Lyuh, C., and Kim, T.: 'Memory access scheduling and binding considering energy minimization in multi-bank memory systems'. Proc. 41st DAC, 2004, pp. 81–86

31 Delaluz, V., Kandemir, M., Vijaykrishnan, N., Sivasubramaniam, A., and Irwin, M.: 'Hardware and software techniques for controlling DRAM power modes', *IEEE Trans. Comput.*, 2001, **50**, (11), pp. 1154–1173

32 Ramachandran, L., Gajski, D., and Chaiyakul, V.: 'An algorithm for array variable clustering'. Proc. EDAC, 1994, pp. 262–266

33 Schmit, H., and Thomas, D.: 'Synthesis of application-specific memory designs', *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 1997, **5**, (1), pp. 101–111

34 Huang, C., Ravi, S., Raghunathan, A., and Jha, N.: 'High-level synthesis of distributed logic-memory architectures'. Proc. ICCAD, 2002, pp. 564–571

35 Saghir, M., Chow, P., and Lee, C.: 'Exploiting dual data banks in digital signal processors'. Proc. ASPLOS, Jun. 1997, pp. 234–243

36 Sudarsanam, A., and Malik, S.: 'Memory bank and register allocation in software synthesis for ASIPs'. Proc. ICCAD, 1995, pp. 388–392

37 Wuytack, S., Catthoor, F., De Jong, G., and De Man, H.: 'Minimizing the required memory bandwidth in VLSI system realizations', *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 1999, **7**, (4), pp. 433–441

38 Lamport, L.: 'The parallel execution of do-loops', *Commun. ACM*, 1974, **17**, (2), pp. 83–93

39 Verhaegh, W., Aarts, E., van Gorp, P., and Lippens, P.: 'A two-stage solution approach for multidimensional periodic scheduling', *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, 2001, **10**, (10), pp. 1185–1199

40 Hall, M., Anderson, M., Amarasinghe, S., Murphy, B., Liao, S., Buignon, E., and Lam, M.: 'Maximizing multiprocessor performance with SUIF', *Computer*, 1996, **29**, (12), pp. 84–89

41 Anderson, J., Amarasinghe, S., and Lam, M.: 'Data and computation transformations for multiprocessors'. Proc. Symp. on Principles and Practice of Parallel Programming, July 1995, pp. 166–178

42 Fang, J.Z., and Lu, M.: 'An iteration partition approach for cache or local memory memory thrasing on parallel processing', *IEEE Trans. Comput.*, 1993, **42**, (5), pp. 529–546

43 Banerjee, P., Chandy, J., Gupta, M., Holm, J., Lain, A., Palermo, D., Ramaswamy, S., and Su, E.: 'Overview of the PARADIGM compiler for distributed memory message-passing multicomputers', *Computer*, 1995, **28**, (10), pp. 37–47

44 Gupta, M., Schonberg, E., and Srinivasan, H.: 'A unified framework for optimizing communication in data-parallel programs', *IEEE Trans. Parallel Distrib. Syst.*, 1996, **7**, (7), pp. 689–704

45 Hoang, P.D., and Rabaey, J.M.: 'Scheduling of DSP programs onto multiprocessors for maximum throughput', *IEEE Trans. Signal Process.*, 1993, **41**, (6), pp. 2225–2235

46 Li, Y., and Wolf, W.: 'Hierarchical scheduling and allocation of multirate systems on heterogeneous multiprocessors'. Proc. DATE, 1997, pp. 134–139

47 Ramamritham, K., and Stankovic, J.A.: 'Scheduling algorithms and operation systems support for real-time systems', *Proc. IEEE*, 1994, **82**, (1), pp. 55–67

48 El-Rewini, H., Ali, H.H., and Lewis, T.: 'Task scheduling in multiprocessing systems', *Computer*, 1995, **28**, (12), pp. 27–37

49 Jha, N.: 'Low-power system scheduling and synthesis'. Proc. ICCAD, San Jose, CA, USA, Nov. 2001, pp. 259–263

50 Quan, G., and Hu, X.: 'Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors'. Proc. 38th DAC, Jun. 2001, pp. 828–833

51 Yang, P.: 'Pareto-optimization based run-time task scheduling for embedded systems'. PhD thesis, Catholic University Leuven, 2004

52 Dave, B., Lakshminarayana, G., and Jha, N.: 'COSYN: Hardware-software co-synthesis of heterogeneous distributed embedded systems', *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 1999, **7**, (1), pp. 92–104

53 Meftali, S., Gharsalli, F., Rousseau, F., and Jerraya, A.: 'An optimal memory allocation for application-specific multiprocessor system-on-chip'. Proc. ISSS, Sept. 2001, pp. 19–24

54 Madsen, J., and Jorgensen, P.: 'Embedded system synthesis under memory constraints'. Proc. CODES, Rome, Italy, May 1999, pp. 188–192

55 Szymanek, R., and Kuchcinski, K.: 'A constructive algorithm for memory-aware task assignment and scheduling'. Proc. CODES, Apr. 2001, pp. 147–152

56 Kandemir, M., Zhang, W., and Karakoy, M.: 'Runtime code parallelization for on-chip multiprocessors'. Proc. DATE, 2003, pp. 10510–10515

57 Kadayif, I., Kandemir, M., Vijaykrishnan, N., Irwin, M., and Sivasubramaniam, A.: 'EAC: A compiler framework for high-level energy estimation and optimization'. Proc. DATE, Paris, France, Mar. 2002, pp. 0436–0441

58 Patel, K., Macii, E., and Poncino, M.: 'Synthesis of partitioned shared memory architectures for energy-efficient multi-processor SoC'. Proc. DATE, 2004, pp. 700–701

59 Tomar, S., Kim, S., Vijaykrishnan, N., Kandemir, M., and Irwin, M.: 'Use of local memory for efficient Java execution'. Proc. ICCD, 2001

60 Delaluz, V., Sivasubramaniam, A., Kandemir, M., Vijaykrishnan, N., and Irwin, M.: 'Scheduler-based DRAM energy management'. Proc. 39th DAC, 2002, pp. 697–702

61 daSilva, J.L., Ykman, C., Miranda, M., Croes, K., Wuytack, S., de Jong, G., Catthoor, F., Verkest, D., Six, P., and De Man, H.: 'Efficient system exploration and synthesis of applications with dynamic data storage and intensive data transfer'. Proc. 35th DAC, Jun. 1998, pp. 76–81

62 Berger, E., McKinley, K., Blumofe, R., Wilson, P.: 'Hoard: A scalable memory allocator for multithreaded applications'. Proc. 8th ASPLOS, Nov. 1998, pp. 117–128

63 Wilson, P.R., *et al.*: 'Dynamic storage allocation: a survey and critical review DDR'. Technical report, Department of Computer Science, Univ. Texas, 1995

64 Vo, K.-P.: 'Vmalloc: a general and efficient memory allocator', *Softw.-Pract. Exp.*, 1996, (26), pp. 1–18

65 Shalan, M., and Mooney III, V.: 'A dynamic memory management unit for embedded real-time systems-on-a-chip'. Proc. CASES, San Jose, CA, USA, Nov. 2000, pp. 180–186

66 Vee, V., and Hu, W.: 'A scalable and efficient storage allocator on shared-memory multiprocessors'. Int. Symp. Parallel Architectures, Algorithms and Networks, Jun. 1999, pp. 230–235

67 Marchal, P., Bruni, D., Gomez, J.I., Benini, L., Pinuel, L., Catthoor, F., and Corporaal, H.: 'SDRAM-energy-aware memory allocation for dynamic multi-media applications on multi-processor platforms'. Proc. DATE, Munich, Germany, Mar. 2003, pp. 10516–10523

68 Gomez, J.I., Marchal, P., Bruni, D., Benini, L., Prieto, M., Catthoor, F., and Corporaal, H.: Scenario-based SDRAM-energy-aware scheduling for dynamic multi-media applications on multi-processor platforms. Workshop on Application Specific Processors (in conj. with MICRO), 2002

69 Rau, B.: 'Iterative modulo scheduling'. Technical report, HP Labs, 1995

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

71 Marchal, P., Gomez, J.I., Pinuel, L. Verdoolaege, S., and Catthoor, F.: 'Loop morphing to optimize the memory bandwidth'. Proc. IEEE ASAP, Galvestone, TX, USA, Sept. 2004

72 Marchal, P., Gomez, J.I., Pinuel, L. Verdoolaege, S., and Catthoor, F.: 'Optimizing the memory bandwidth with loop fusion'. Proc. ISSS, Stockholm, Sweden, Sept. 2004

73 Leijten, J.A.: 'Real-time constrained reconfigurable communication between embedded processors'. PhD thesis, Technishe Universiteit Eindhoven, Nov. 1998

74 Yang, P., Marchal, P., Wong, C., Himpe, S., Catthoor, F., Patrick, D., Vounckx, J., and Lauwereins, R.: 'Cost-efficient mapping of dynamic concurrent tasks in embedded real-time multimedia systems', in 'Multiprocessor systems on chip' (Elsevier Science & Technology, 2004), pp. 46–58

75 Poplavko, P., Pastrnak, M., Basten, T., van Meerbergen, J., and de With, P.: 'Mapping of an MPEG-4 shape-texture decoder onto an on-chip multiprocessor'. PRORISC 2003, 14th Workshop on Circuits, Systems and Signal Processing, Nov. 2003, pp. 139–147

76 Pastrnak, M., Poplavko, P., de With, P., and van Meerbergen, J.: 'Resource estimation for MPEG-4 video object shape-texture decoding on multiprocessor network-on-chip'. PROGRESS 2003, 4th Seminar on Embedded Systems, Oct. 2003, pp. 185–193

77 Marchal, P., Gomez, J., Bruni, D., Benini, L., Pinuel, L., and Catthoor, F.: 'Integrated task-scheduling and data-assignment to enable SDRAM power/performance trade-offs in dynamic applications', *IEEE Des. Test Comput.*, 2004, **11**, (1), pp. 1–12

78 Mamagkakis, S., Atienza, D., Catthoor, F., Soudris, D., and Mendias, J.M.: 'Custom design of multi-level dynamic memory management subsystem for embedded systems'. Proc. Signal Processing Symp. (SiPS), Austin, TX, USA, October 2004

79 Daylight, E., *et al.*: 'Memory access aware data structure transformations for embedded software with dynamic data Accesses', *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 2004, **12**, (3)

80 Atienza, D., Mamagkakis, S., Catthoor, F., Mend ias, J.M., and Soudris, D.: 'Dynamic memory management design methodology for reduced memory footprint in multimedia and wireless network applications'. Proc. Design, Automation and Test in Europe (DATE), Paris, France, February 2004

81 Lea, D.: 'The lea 2.7.2 dynamic memory allocator', 2002. http://gee.cs.oswego.edu/dl/