SDRAM-Energy-Aware Memory Allocation for Dynamic Multi-Media Applications on Multi-Processor Platforms

P. Marchal (*marchal@imec.be*, **) D. Bruni (*dbruni@deis.unibo.it*, ***)
J.I. Gomez (*gomezjo@imec.be*, *) L. Benini (*lbenini@deis.unibo.it*,***)
L. Piñuel (*lpinuel@dacya.ucm.es*,*) F. Catthoor(*catthoor@imec.be*,**)
H. Corporaal(*heco@imec.be*,****)

IMEC and K.U.Leuven-ESAT, Leuven, Belgium (**) DACYA U.C.M., Spain (*)
D.E.I.S. University of Bologna, Italy (***) IMEC and T.U. Eindhoven, Nederland (****)

Abstract

Heterogeneous multi-processors platforms are an interesting option to satisfy the computational performance of dynamic multi-media applications at a reasonable energy cost. Today, almost no support exists to energy-efficiently manage the data of a multi-threaded application on these platforms. In this paper we show that the assignment of data of dynamically created/deleted tasks to the shared memory has a large impact on the energy consumption. We present two dynamic memory allocators which solve the bank assignment problem for shared multi-banked SDRAM memories. Both allocators assign the tasks' data to the available SDRAM banks such that the number of page-misses is reduced. We have measured large energy savings with these allocators compared to existing dynamic memory allocators for several task-sets based on MediaBench[5].

1 Introduction

In the near future, the silicon market will be driven by low-cost, portable consumer devices which integrate multi-media and wireless technology. Applications running on these devices require an enormous computational performance (1-40$GOPS$) at a low energy consumption (0.1-2W). Additionally, they are subjected to time constraints, complicating their design considerably. The challenge to embed these applications on portable devices is enlarged even further because of user interaction. E.g. at any moment the user will be able to trigger new services, change the configuration of the running services or stop existing services. Heterogeneous multi-processor platforms with domain specific computational resources can potentially offer enough computational performance at a sufficiently low energy consumption. To store multi-media data these platforms need to be connected to large off-chip SDRAMs. They contribute significantly to the system's energy consumption (see [17]). The energy consumption of SDRAMs depends largely on how data is assigned to the memory banks. Since in our application domain the data which needs to be allocated is only known at run-time (at a particular time-instance), fully design-time based solutions as proposed earlier in the compiler and system synthesis cannot solve the problem (see Sect.3). Run-time memory management solutions as present in nowadays operating systems are too inefficient in terms of cost optimization (especially energy consumption). They are also not adapted for the time constraints. We present two *SDRAM-energy-aware memory allocators* for dynamic multi-tasked applications. We quantify the energy gains of both allocators with experimental results obtained with a multi-processor simulator. The results are based on task-sets derived from MediaBench[5]. They indicate that both allocators significantly reduce the energy consumption compared with the best known approach so far.

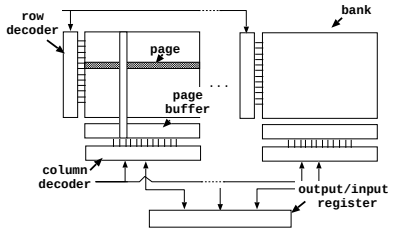

Figure 1. Multi-banked SDRAM architecture

2 Platform and SDRAM Energy Model

In the context of this paper we assume a platform that consists of a set of processor nodes (e.g. ARM [1]). Each processor is connected to a local memory and interacts with shared off-chip SDRAM modules. SDRAM memories are present on the platform because their energy cost per bit to store large data structures is lower than of SRAMs. They are used to store large infrequently accessed data structures. As a consequence, they can be shared among processors to reduce the static energy cost without a large performance penalty.

A simplified view of a typical multi-banked SDRAM architecture is shown in Fig.1. Fetching or storing data in an SDRAM involves three memory operations. An activation operation decodes the row address, selects the appropriate bank and moves an page/row to the page buffer of the corresponding bank. 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 output pins at the 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). 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. Similar to processor cores, SDRAMs nowadays support several energy states in which the SDRAM (see [16]) can be used. We model three energy states: *standby mode* (STBY), *clock-suspend mode* (CS) and *power down* (PWDN). Switching between the different energy states comes at a transition time penalty. We assume that in the future the energy states of each bank can be controlled independently [1].

We model the timing behavior of the SDRAM memory with a state-machine similar to [11]. The timing parameters of the different state transitions have been derived from Micron 64Mb mobile SDRAM[16]. The energy consumption of the SDRAM is computed with the following formula:

$$E = \sum_{\forall i=1}^{N_{\text{banks}}} (E^i_{\text{static}} + E^i_{\text{dynamic}})$$
$$E^i_{\text{static}} = P_{\text{cs}}.t^i_{\text{cs}} + P_{\text{stby}}.t^i_{\text{stby}} + P_{\text{pwdn}}.t^i_{\text{pwdn}}$$
$$E^i_{\text{dynamic}} = N^i_{\text{pa}}.E_{\text{pa}} + N^i_{\text{rw}}.E_{\text{rw}}$$

where:

E_{pa} : energy of a precharge/activation
E_{rw} : energy of a read/write
N^i_{pa} : number of precharge and activations in Bank i
N^i_{rw} : number of reads and writes in Bank i

Our energy model is based on an SDRAM power estimation tool provided by Micron[16]. It decomposes the energy consumption in a static and a dynamic part. The static energy consumption is the standby power of the SDRAM. It depends on which energy states are used during execution. An energy state manager (see [9]) controls when the banks should transition to another power state. As soon as the bank idles for more than one cycle, the manager switches the bank to the CS-mode. When the bank is needed again it is switched back to STDBY-mode within a cycle. Finally, we switch off the bank as soon as it remains idle for longer than a million cycles [2]. The dynamic energy consumption depends on which operations are needed to fetch/store the data from/into the memory. The energy parameters are presented in Tab.1. The remaining parameters ($N^i_{\text{rw/pa}}$ and $t_{\text{cs/stby/pwdn}}$) are obtained by simulation (see Sect.6).

$E_{\text{precharge/activate}}$	14000 pJ/miss
$E_{\text{read/write}}$	2000 pJ/access
P_{stdby}	50 mW
$P_{\text{cs with self refresh}}$	10.8 mW
P_{pwdn}	0 mW

Table 1. Energy Consumption Parameters

3 Motivation and Problem Formulation

According to our experiments, on a multi-processor architecture the dynamic energy contributes on an average 68% to the total consumption of an SDRAM. The remaining static energy is usually not dominant because the SDRAM is shared by multiple tasks. As a consequence, it is more actively used compared to uni-processor architectures and it burns less static energy waiting between consecutive accesses. Moreover, even though in future technologies leakage energy is likely to increase, many techniques (at the technology, circuit and memory architecture level) are under development by DRAM manufactures to reduce the leakage energy during inactive modes (see [10]). Also existing hardware power state controllers (see Sect.4) can significantly decrease the static energy. Therefore, in this paper we focus on data assignment techniques to reduce the dynamic energy. At the same time, our techniques reduce the time of the SDRAM banks spent in the active state, thereby thus reducing the leakage loss in that state which is more difficult to control (see [10]). We motivate with an example how a good data assignment can significantly reduce the number of page-misses, thereby saving dynamic energy. The example consists of two parallel executing tasks, *Convolve* and *Cmp_threshold*.

Convolve

```
for (i = 0; i < K; i++)
  for (j = 0; j < K; j++)
  ... = imgin[i][j] *
        kernel[i*K][j];

imgout[r*col][] = ...
```

Cmp_threshold

```
for (i = 0; i < row; i++)
  for (j = 0; j < col; j++)
    tmp1 = img1[i][j];
```

Figure 2. Extracts from *Convolve* and *Cmp_threshold*

The code of both tasks is presented in Fig.2. Page-misses occur when e.g. *kernel* and *imgin* of *Convolve* are assigned to the same bank. Each consecutive access evicts the open-page of the previous one, causing a page-miss ($2.K^2$-misses in total). Similarly, when data of different tasks are mapped to the same bank, (e.g. *kernel* of *Convolve* and *img1* of *Cmp_threshold*), each access to the bank potentially causes a page-miss.
The number of page-misses depends on how the accesses to the data structures are interleaved. When *imgin* and *imgout* are mapped in the same bank, an access to *imgout* is only scheduled after K^2 accesses to *imgin*. Therefore, the frequency at which accesses to both data structures interfere is much lower than for *kernel* and *imgin*. The resulting number of page-misses in this case is only two. The energy benefit of storing a data structure alone in a bank depends on how much spatial locality exists in the access pattern to the data structure. E.g. *kernel* is a small data structure (it can be mapped on a single page) which is characterized by a large spatial locality. When *kernel* is stored in a bank alone, only one page-miss occurs for all accesses to it.

From this, we conclude that the data assignment should (1) separate the most important data structures with a large spatial locality from the other data structures, since this results in large energy savings; (2) we should share the remaining data structures in such a way that the number of page-misses is minimized. The assignment problem is complicated by the dynamic behavior of modern multi-media applications. Only at run-time it is known which tasks are executing in parallel and which data needs to be allocated in the memory. A fully static assignment of the data structures to the memory banks is thus impossible.

[1] Although the energy-state of an SDRAM today can only be controlled at the granularity of an entire memory chip, many authors (e.g. [4] and [12]) make this assumption.

[2] similar to [4].

Dynamic memory allocators are a potential solution. However, existing allocators do not take the specific behavior of SDRAM memories into account to reduce the number of page-misses.

To solve above issues we introduce in this paper two dynamic memory allocators which reduce the number of page-misses. The first, a *best-effort* allocator shares the SDRAM banks between the tasks. However, it does not guarantee time-constraints. Therefore, when hard real-time requirements are an issue, banks should not be shared among tasks. The number of page-misses can still be reduced by cost-efficiently distributing the available banks to the tasks. This idea is implemented in our second, the *guaranteed performance* memory allocator.

4 Related Work

In the embedded system community the authors of [14] and [2] have presented assignment algorithms to improve the performance of SDRAM memories. Both algorithms distribute data with a high temporal affinity over different banks such that the number of page-misses is minimized. Their optimizations rely on the fact that the temporal affinity in a single threaded application is analyzable at design-time. This is not the case in our application domain (i.e. dynamic multi-threaded applications). The temporal affinity between tasks depends on their actual schedule which is only known at run-time. This renders these techniques not directly applicable in our context.

In [4], techniques are presented to reduce the static energy consumption of SDRAMs in embedded systems. The strategy of this paper consists of clustering data structures with a large temporal affinity in the same memory bank. As a consequence the idle periods of banks are grouped, thereby creating more opportunities to transition more banks in a deeper low-power mode for a longer time.

Several researchers (see e.g. [12], [4] and [9]) present a hardware memory controller to exploit the SDRAM energy modes. As soon as a memory bank remains idle for a predefined period, the memory controller transitions the bank to a low power state. The main advantage of hardware controlled policies is that SDRAM memories can be managed at a much finer level of granularity than pure software approaches.

The authors of [3] present a data migration strategy. It detects at run-time which data structures are accessed together, then it moves arrays with a high temporal affinity to the same SDRAM banks. By bringing these arrays together, the chances are increased that fewer banks need to be active in a given period of time. The research on techniques to reduce the static energy consumption of SDRAMs is very relevant and promising. Complementary to the static energy reduction techniques, we seek to reduce the dynamic energy contribution.

The most scalable and fastest multi-processor virtual memory managers (e.g. [15] and [6]) use a combination of private heaps combined with a shared pool to avoid memory fragmentation. They rely on hardware based memory management units to map virtual memory pages to the banks. These hardware units are unaware of the underlying memory architecture. Therefore, in the best-case memory managers randomly distributes the data across all memory banks (random allocation). In the worst-case, all data is concentrated in a single memory bank. We will compare our approach to both extremes in Sect.7.

5 Bank Aware Allocation Algorithms

We first present a best effort memory allocator (**BE**) which searches the most energy-efficient assignment for all the data in a task-set. The allocator can map data of different tasks in the same bank in order to minimize the number of page-misses. Hence, accesses from different tasks can interleave at run-time, causing unpredictable page-misses. We do not exactly know how much the page-misses will increase the execution-time of the tasks. As a consequence, the best effort allocator cannot be used when hard real-time constraints need to be guaranteed and little slack is available. The goal of the guaranteed performance allocator (**GP**) is to minimize the number of page-misses while still guaranteeing the real-time constraints.

5.1 Best Effort Memory Allocator

The BE (see Algo.1) consists of a design-time and a run-time phase. The design-time phase bounds the exploration space of the run-time manager reducing its time and energy penalty. At design-time we characterize the data structures of each task with a heuristic parameter: *selfishness* (line 4: $\mathrm{S}_{\mathrm{ds}}^{\mathrm{local}}$). It expresses the energy benefits of storing data alone in a bank. When accesses to a selfish data structure are not interleaved with accesses to other data structures in the same bank, page-misses are avoided. Selfishness of a data structure is calculated by dividing the average time between page-misses ($\tau_{\mathrm{ds}}^{\mathrm{misses}}$) with the average time between accesses ($\tau_{\mathrm{ds}}^{\mathrm{accesses}}$). This ratio expresses the available spatial locality and can be either measured or calculated[3] at design-time. We weigh it with the importance of the data structure by multiplying it with the number of accesses to the data structure ($\mathrm{N}_{\mathrm{ds}}^{\mathrm{accesses}}$). Finally, we

Algorithm 1 Best-Effort Memory Allocator

```
 1: Design-Time:
 2: for all T ∈ Task do
 3:   for all ds (= data structure) ∈ T do
 4:     Compute local selfishness:
        S_ds^local = N_ds^access · τ_ds^misses / τ_ds^accesses
 5:     Add S_ds^local to Tab_info
 6:   end for
 7: end for
 8: Run-Time:
 9: Initialize selfishness of all available banks: S_bank = 0
10: Initialize ordered queue Q
11: for all T ∈ ActiveTasks do
12:   for all ds ∈ T do
13:     Insert ds in Q according decreasing S_ds^local
14:   end for
15: end for
16: while Q is not empty do
17:   ds = head of Q
18:   Insert ds in bank with smallest selfishness
19:   Update selfishness of the bank:
      S_bank += S_ds^local
20: end while
```

[3] with a technique similar to e.g. [13]

add extra data structures to the source code for the design-time info needed at run-time (line 5: $\mathrm{Tab_{info}}$).

At run-time, when it is known which tasks are activated at the start of a new frame and thus which data needs to be allocated, the algorithm assigns the alive data to the memory banks[4]. The algorithm distributes the data among the banks such that *selfishness* of all the banks is balanced. The selfishness of a bank ($\mathrm{S_{bank}}$) is the sum of the selfishness of all data structures in the bank. The algorithm ranks the data structures according to decreasing selfishness (line: 11-15) and then greedily assigns the data to the banks starting from the most selfish one (lines: 15-20). Each data structure is put in the least selfish bank. This strategy puts the most selfish data structures in separate banks and clusters the remaining ones. The complexity of the algorithm at run-time is $\mathrm{O(nlog(n))}$ where n is the number of data structures in the task-set.

5.2 Guaranteed Performance Allocation

The time guarantees are only possible when all page-misses can be predicted, but that can be difficult due to the interference between accesses of different tasks. An obvious way to avoid interference is to assign the data of simultaneously active tasks to independent banks. This implies that at least one bank per task is required or extra task scheduling constraints need to be introduced. The following two degrees of freedom remain: how to partition the banks among the tasks and how to assign the data of each task to its partition.

The number of page-misses of a task heavily depends on the number of banks which are assigned to it (e.g. see tasks in Tab.2). The sensitivity of the number of page-misses to the number of banks varies from task to task. Some tasks benefit more than others from having extra banks assigned to it. Our guaranteed performance approach allocates more banks to those tasks which benefit most.

We follow an approach similar to the task-scheduling methodology of [8] to solve this problem. At design-time we generate a data assignment for every task and for any possible number of banks. The resulting assignments for each task can be presented in a Pareto curve which trades off the energy consumption of the task i.f.o. the number of banks. With each point in the curve thus corresponds an assignment, the number of banks required for the assignment and the energy consumption of the assignment. We also annotate each point with the run-time of the task executed with the corresponding assignment. The Pareto curves can be created with the best-effort approach based on selfishness. The approach consists then of assigning the data of a *single* task to the SDRAM banks. In this case no run-time information about other tasks is required. As a consequence, we can compute at design-time the selfishness of all data structures and generate the final data assignment.

At run-time we distribute the available banks of the platform among the active tasks using the Pareto curves. We select a point on the Pareto curve of each task such that the energy consumption of all tasks is minimized and that the total number of banks for all tasks is less or equals the available number of banks on the platform. We reuse for this purpose a greedy heuristic which we have developed in the context of task-scheduling (see [8]).

[4]We currently assume that tasks can only be started/deleted at predefined points in the program. However, this is not a severe limitation for most modern multi-media applications (see [8]).

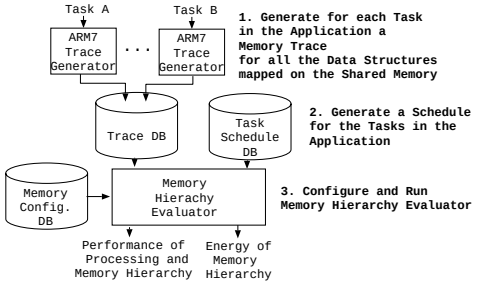

Figure 3. Simulation Environment

6 Evaluation Strategy

In this section we first present our simulation environment. The main goal of our simulation environment (see Fig.3) is to correctly study how multi-threaded applications should be mapped on a shared memory hierarchy. We simulate the processing elements and the memory architecture independently. This allows us to quickly explore different allocations of the data structures on the SDRAM memories while avoiding long simulation times for the processing elements. The processing elements and their performance are simulated using an adapted ARMulator[1]. This simulator dumps a memory access trace for each task in the parallel application. Each memory access in the trace is annotated with its relative issue-time. We input the memory traces together with the schedule of the corresponding tasks in the performance and energy evaluation script. This script combines the memory traces in a cycle-accurate way according to the issue-time of each access, the task schedule and the memory hierarchy. It outputs the total execution time of each task [5] and the energy consumption of the SDRAMs.

To evaluate the effectiveness of our assignment techniques, we have generated representative task-sets. The tasks have been extracted from MediaBench[5] (*Cmp_threshold*, *Convolve*, *Dct* and *Rawcaudio*). Two tasks *Convolve* and *Cmp_threshold* kernels are parts of *Edge_detection*. We have also added *Rgb2Yuv*, *Quick24to8*, *Fir*, *Lzw* and *Rinjadel* which are typical for many portable multi-media applications. In Tab.2 we enumerate the tasks and show how their total energy consumption i.f.o. the number of memory banks. The table contains measurements for the tasks executed on a ARM7 running at 100MHz. The results for this analysis were obtained with assignments based on the BE-approach.

7 Experimental Results

We verify the quality of our heuristics (best-effort **BE** and guaranteed performance **GP**) against a Monte-Carlo approximation of the best-possible assignment (**MA**). The results of

[5]including both the processor and memory delay.

		N_{banks}					
Task	N_{ds}	1	2	3	4	5	6
Cmp_threshold	3	4331	3293	993	-	-	-
Fir	3	1770	457	489	-	-	-
Lzw	3	6482	7004	7655	-	-	-
Rawcaudio	4	4202	2998	3234	4061	-	-
Convolve	4	19222	8173	8474	8474	-	-
Rinjadel	4	7037	6491	6870	7277	-	-
Rgb2Yuv	5	528	796	1064	1333	1593	-
Dct	6	2552	2845	2118	2540	3015	3485
Quick24to8	10	6930	5597	4215	4417	4620	4824

Table 2. Energy for Benchmark Tasks @100MHz

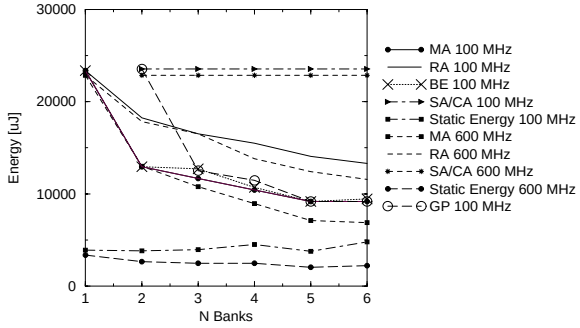

Figure 4. Comparison of Allocation Strategies for *Convolve* **and** *Cmp_threshold*

the latter were obtained by measuring 100 different data assignments. We compare our memory allocators against three existing policies. The first reference policy, random allocation (**RA**) randomly distributes the data structures across the memory banks similar to architecture-unaware allocators (see Sect.4). We show the average energy consumption after 100 runs of the RA policy. In the second reference we do not share the SDRAMs among the processors. Each processor owns an equal number of memory banks. On each processor a local memory allocator manages the private banks (sequential allocation **SA**). Finally, we compare our results with a static energy reduction technique ([4]). This technique clusters the data structures such that the number of active banks is minimized. In the most extreme case, all data is clustered in a single bank (clustered allocation **CA**).

In Fig.4[6] we visualize the energy consumption of the different allocators for the *Convolve* and *Cmp_threshold* task-set. Similar results for other tasks-sets are presented in Tab.3.

The energy consumption of all allocators, except for CA and SA, first decreases when the number of banks is increased. The allocators distribute the data across more banks, thereby reducing the number of page-misses and the dynamic energy. At the same time, the static energy consumption slightly reduces since less misses results in a shorter execution time (see the static energy of the example in Fig.4 executed at 600 MHz.).

However, when extra banks do not significantly reduce the page-miss rate anymore, the dynamic energy savings become smaller than the extra static energy needed to keep the banks in $CS/STBY$-mode. The total energy consumption increases then again. A nice illustration of this is *Quick24to8* in Tab.2. The total energy consumption decreases up to three banks and then increases again due to the extra static energy. Also, in Fig.4 the total energy consumption increases again when more than five banks are used. From these examples, we see that an optimal number of active banks exists. The optimal number of banks depends on the ratio of static versus dynamic energy. When the banks become more active (e.g. because more tasks are activated or the processor frequency is increased), the dynamic energy becomes more important than the static energy and the optimal number of banks increases. E.g. in Fig.4 the optimal number of banks increases from five to six when the processor frequency changes from 100MHz to 600MHz. We plan to further explore this trade-off between static and dynamic energy in the future.

CA clusters the data in as few banks as possible to limit the static energy of memories, but it comes at the cost of extra page-misses and thus more dynamic energy. Therefore, CA increases the total energy consumption when the energy is dominated by the dynamic energy (see Fig.4).

SA also performs poorly under these conditions. It cannot exploit idle banks owned by other processors to reduce the number of page-misses. The difference between SA and MA (an approximation of the best-possible assignment) is large (more than 300% for the *Rgb2Yuv*/*Cmp_threshold* task-set with 6 banks), indicating that sharing SDRAM memories is an interesting option for heterogeneous multi-processor platforms. It increases the exploration space such that better assignments can be found. When the banks are not too heavily used, even no performance penalty is present (see below).

We also observe in Fig.4 that existing commercial multi-processor memory allocators (RA) perform badly compared to MA. This suggests that a large headroom for improvement exists. When only one bank is available, obviously all memory allocation algorithms produce the same results. With an increasing number of banks the gap between RA and MA first widens as a result of the larger assignment freedom (up to 55 % for *Rgb2Yuv* and *Cmp_threshold* with four banks). However, the performance of the RA improves with an increasing number of banks: the chances increase that RA distributes the data structures across the banks which significantly reduces the energy consumption. Therefore, when the number of banks becomes large the gap between RA and MA becomes smaller again (50 % for *Rgb2Yuv* and *Cmp_threshold* with six banks). For higher processor frequencies the static energy consumption decreases and the potential gains become larger. E.g. for *Convolve* and *Cmp_threshold* the gap increases from 26% to 34%.

The Fig.4 shows how BE outperforms RA. Results at the top part of Tab.3 suggest an improvement up to 50% (see *Rgb2Yuv* and *Cmp_threshold* with four banks). Moreover, BE often comes close to the MA results. The difference between BE and MA is always less than 23%. When the number of tasks in the application becomes large (see last task-set in Tab.3 which consists of 10 tasks), we note a small energy loss of BE compared to RA for the first task-set and eight banks are used. In

[6]Note that the SA and GP curves only start from two banks, which is the minimum number of banks needed by these policies.

		N_{banks}					
Task(100MHz)	N_{ds}	1(CA)	2	3	4	5	6
Fir.+Conv.(SA)	7	-	20993	20993	20993	20993	20993
Fir.+Conv.(MA)	7	20958	9858	9269	8641	8641	8641
Fir.+Conv.(RA)	7	20958	15712	13834	12381	12436	10990
Fir.+Conv.(GP)	7	-	20993	9943	8641	8641	8641
Fir.+Conv.(BE)	7	20958	10250	9269	9515	8942	8964
R2Y.+Cmp_t.(SA)	8	-	4859	4859	4859	4859	4859
R2Y.+Cmp_t.(MA)	8	4986	3832	1877	1521	1521	1521
R2Y.+Cmp_t.(RA)	8	4986	4362	3733	3407	3368	3209
R2Y.+Cmp_t.(GP)	8	-	4859	3821	1521	1521	1521
R2Y.+Cmp_t.(BE)	8	4986	4041	2031	1553	1821	2089
R2Y.+Cmp_t.+Conv.(SA)	12	-	-	24082	24082	24082	24082
R2Y.+Cmp_t.+Conv.(MA)	12	23531	13872	12456	11392	10109	10060
R2Y.+Cmp_t.+Conv.(RA)	12	23531	19730	17005	15444	14977	14533
R2Y.+Cmp_t.+Conv.(GP)	12	-	-	24082	13034	11987	9695
R2Y.+Cmp_t.+Conv.(BE)	12	23531	13769	13468	11515	9907	10206
R2Y.+Cmp_t.+Conv.+Dct (SA)	18	-	-	-	26647	26647	26647
R2Y.+Cmp_t.+Conv.+Dct (MA)	18	26195	16684	15165	13665	13132	12514
R2Y.+Cmp_t.+Conv.+Dct (RA)	18	26195	21896	19332	17947	17517	17696
R2Y.+Cmp_t.+Conv.+Dct (GP)	18	-	-	-	26647	15598	14551
R2Y.+Cmp_t.+Conv.+Dct (BE)	18	26195	16212	16143	14224	13405	12758
Task(600 MHz)	N_{ds}	1(CA)	2	3	4	5	6
Fir.+Conv.(SA)	7	-	20305	20305	20305	20305	20305
Fir.+Conv.(MA)	7	20480	9261	7582	6600	6494	6473
Fir.+Conv.(RA)	7	20480	14938	11664	11421	10115	9196
Fir.+Conv.(GP)	7	-	20305	8395	7842	6494	6473
Fir.+Conv.(BE)	7	20480	9526	8219	7623	6494	6473

Task	N_{ds}	8	16	32
2Quick.+2R2Y+2Rin.+Cmp_t.+2Lzw(SA)	50	-	48788	48788
2Quick.+2R2Y+2Rin.+Cmp_t.+2Lzw(RA)	50	49525	41872	45158
2Quick.+2R2Y+2Rin.+Cmp_t.+2Lzw(MA)	50	47437	37963	41610
2Quick.+2R2Y+2Rin.+Cmp_t.+2Lzw(GP)	50	-	38856	34236
2Quick.+2R2Y+2Rin.+Cmp_t.+2Lzw(BE)	50	51215	40783	35371
Static Energy	50	10227	9045	8489

Table 3. Energy Comparison of Several Allocation Strategies

this case 50 data structures are allocated in a small amount of banks. As a result, a limited freedom exists to reduce the number of page-misses, i.e. the energy gap between maximum and minimum energy consumption is small. Note that currently BE cannot detect the optimal number of banks. When the banks are not actively used, its energy consumption increases (compare e.g. *Cmp_threshold* and *Convolve* for five and six banks), but it remains lower than existing dynamic allocation policies.

GP performs equally well for a sufficiently large number of banks. The main advantage of this technique is that the execution times of the different tasks can be predicted and guaranteed. Moreover, it will never use more than the optimal number of banks, but its performance breaks down when only few banks are available per task. In this case, it maps similar to SA all data structures of each task in a single (or few) banks. It then consumes more energy than RA (29% for *Convolve* and *Cmp_threshold* with two banks).

Timing measurements indicate that in 42 out of 47 cases the execution time of the task-sets improves with BE/GP (up to 10%) on top of the energy gain. In the remaining 5 cases we have measured a slowdown up to 0.42% (BE) and 8.93% (GP). This data-assignment technique combined with task-scheduling can significantly improve the performance of a task-set. Or, for a given deadline, the joined approach can be used to reduce the energy consumption of the application (see [7]).
The run-time overhead of both BE and GP is limited: usually, large time-intervals exist between successive calls to the run-time manager, which allows to relax its time- and energy-overhead over a large period. Also, in the context of run-time task-scheduling (see [8]), we have shown that the time and energy overhead of a comparable run-time scheduler is low.

8 Conclusions and Future Work

Low-power design is a key issue for future dynamic multimedia applications mapped on multi-processor platforms. On these architectures off-chip SDRAM memories are big energy consumers. A crucial parameter which controls the energy consumption of SDRAMs is the number of page-misses. This paper presents two dynamic memory allocators for bank assignment: a best-effort and a guaranteed performance allocator. Both allocators assign the arrays of dynamically created/deleted tasks to the memory banks, thereby reducing the number of page-misses and thus the energy consumption. Experimental results obtained with a multi-processor simulator are very promising. The allocators significantly reduce the energy consumption of SDRAMs compared to existing dynamic memory managers. In the future, we would like to further explore the trade-off between static and dynamic energy consumption. Furthermore, we want to study how cache memories affect the energy consumption of SDRAMs.

References

[1] ARM. www.arm.com.

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

[3] V. Delaluz, M. Kandemir, and I. Kolcu. Automatic Data Migration for Reducing Energy Consumption in Multi-Bank Memory Systems. In *Proc. 39th Dac*, pages 213–218, 2002.

[4] 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.

[5] C.Lee et al. MediaBench: a tool for evaluation and synthesizing multimedia and communication systems. In *Int. Symp. Microarchitecture*, 1997.

[6] E. Berger et al. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proc. 8th Asplos*, Oct. 1998.

[7] J.I. Gomez et al. Scenario-based SDRAM-Energy-Aware Scheduling for Dynamic Multi-Media Applications on Multi-Processor Platforms. In *WASP (in conj. with MICRO)*, 2002.

[8] P. Yang et al. Managing Dynamic Concurrent Tasks in Embedded Real-Time Multimedia Systems. In *Proc. Isss*, Oct. 2002.

[9] X. Fan, C. Ellis, and A. Lebeck. Memory Controller Policies for DRAM Power Management. In *Proc. Islped*, pages 129–134, 2001.

[10] K. Itoh. Low-Voltage Memories for Power-Aware Systems. In *Proc. Islped*, pages 1–6, Monterey, CA, Aug. 2002.

[11] Y. Joo, Y. Choi, and H. Shim. Energy Exploration and Reduction of SDRAM Memory Systems. In *Proc. 39th Dac*, pages 892–897, 2002.

[12] A. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power Aware Page Allocation. In *Proc. 9th Asplos*, Nov. 2000.

[13] K. McKinley, S. Carr, and C. Tseng. Improving Data Locality with Loop Transformations. *ACM Trans. PLS*, 18(4):424–453, July 1996.

[14] P. Panda. Memory Bank Customization and Assignment in Behavioral Synthesis. In *Proc. Iccad*, pages 477–481, Oct. 1999.

[15] 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.

[16] Calculating Memory System Power For DDR. Technical report, Micron.

[17] M. Viredaz and D. Wallach. Power Evaluation of a Handheld Computer: A Case Study. Technical report, WRL, Compaq, May 2001.