An Integrated Hardware/Software Approach For Run-Time Scratchpad Management [*]

Poletti Francesco[†], Paul Marchal[‡], David Atienza[#],
Luca Benini[†], Francky Catthoor[‡], Jose M. Mendias[#]
[†] University of Bologna, DEIS, Viale Risorgimento 2, 40134 Bologna, Italy.
[‡] IMEC vzw, Kapeldreef 75, 3001 Heverlee, Belgium.
[#] DACYA/UCM, Avda. Complutense s/n, 28040 Madrid, Spain.
{ fpoletti@deis.unibo.it, marchal@imec.be, datienza@dacya.ucm.es
lbenini@deis.unibo.it, catthoor@imec.be, mendias@dacya.ucm.es }

ABSTRACT

An ever increasing number of dynamic interactive applications are implemented on portable consumer electronics. Designers depend largely on operating systems to map these applications on the architecture. However, today's embedded operating systems abstract away the precise architectural details of the platform. As a consequence, they cannot exploit the energy efficiency of scratchpad memories. We present in this paper a novel integrated hardware/software solution to support scratchpad memories at a high abstraction level. We exploit hardware support to alleviate the transfer cost from/to the scratchpad memory and at the same time provide a high-level programming interface for run-time scratchpad management. We demonstrate the effectiveness of our approach with a case-study.

Categories and Subject Descriptors

C.0 [**General**]: Hardware/software interfaces; C.3 [**Special-purpose and Application-based Systems**]: Real-time and embedded systems; C.4 [**Performance of Systems**]: Design studies

General Terms: Measurement Performance Verification.

Keywords: Scratchpad, DMA, Dynamic Allocation, AMBA AHB.

1. INTRODUCTION

Memories determine to a large extent the energy cost and performance in today's embedded systems. As a result, several architectural extensions and dedicated compilation techniques have been developed to optimally use the memory hierarchy (see [4], [9] and [5] for good overviews).

An important architectural innovation involves adding scratchpad memories next to hardware-controlled caches. Scratchpads are more energy-efficient than caches since they do not need complex tag-decoding logic. They can also reduce the number of conflict misses in the cache ([8]). To exploit the potential benefits of scratchpad memories, the designers should first carefully decide which data to assign to the scratchpad and secondly, they need to efficiently implement the assignment decisions.

In the simplest cases, the programmer determines at design-time which data to store in the scratchpad. At run-time no other data can be moved into the scratchpad. For dynamic applications (such as MPEG21, MPEG4) where tasks are either created at run-time or require a varying amount of data, this limitation prevents the effective usage of scratchpad memories. Run-time scratchpad management techniques are thus needed. Recently, researchers have proposed run-time scratchpad management techniques for pointer-based applications (see e.g.[12]). When an object is created in the scratchpad, the processor explicitly transfers data between the template of the object in the main memory and the instantiation in the scratchpad.

The copy cost can be reduced when dedicated data transfer hardware is used, such as a Direct Memory Access controller (DMA). Once the copy cost is reduced, more and better objects can be found to take advantage of the scratchpad. As an example, in DSP applications designers often tile large arrays to create locality. The task then accesses tiles of the original array. Copies of the accessed tiles can be assigned to the scratchpad, but need to be regularly updated with data from the original array ([14]). Existing DMAs and scratchpad memories are often programmed at the assembly level in a rather ad-hoc fashion. Without higher-level programming support, they are difficult to use in dynamic applications.

The contribution of this paper is to build an integrated hardware/software solution for managing scratchpad memories at run-time. Our hardware consists of scratchpad memories coupled to DMA engines to reduce the copy cost between scratchpad and main memory. More importantly, we provide a high-level programming interface which makes very easy to manage the scratchpads at run-time. We have integrated our solution in a cycle-accurate platform simulator (including the OS). Hence, we can present precise measurements of the scratchpad memory overhead.

This paper is organized as follows. First, we survey related work (Sect. 2). Then, we discuss the platform extensions necessary to manage the scratchpad at run-time (Sect. 3). Subsequently, we show, through a detailed case study and several additional experiments, how this environment is used to manage scratchpads for dynamic multi-threaded applications (see Sect. 4-Sect. 6).

[*]This work is partially supported by the Spanish Government Research Grant TIC2002/0750 and by a grant from STMicroelettronics. Francky Catthoor is also professor at ESAT/K.U.Leuven-Belgium.

2. RELATED WORK

Scratchpad management techniques have been widely researched in the past (see [9] and [5] for an overview). We discern design-time and run-time techniques.

A large body of research exists in how to decide which data to assign, at design time, to the scratchpad memory. [8] discusses the combined effect of a scratchpad and a cache memory, and presents an algorithm to optimally generate a custom cache/scratchpad architecture. [3] explains how the scratchpad can be managed without the help of the linker. A special decoder is presented that detects for each access whether the corresponding data is in the main memory or in the on-chip scratchpad memory. In [1], an algorithm based on profiling information is presented to find which segments of the linked executable should be mapped in the scratchpad. [2] extends scratchpad management techniques to the context of heterogeneously sized memories. Finally, [14], [6] and [11] describe how to partition large data structures. The tiles of the original data structures can then be mapped onto the scratchpad memory, but require data transfers between the scratchpad and the main memory. Previous work fails to measure the overhead of these transfers on a real architecture with an DMA.

The main limitation of above design-time techniques is that they cannot cope with dynamic applications where only at run-time it is known which data needs to be assigned. Dynamic applications however are slowly becoming desirable in the context of embedded systems. Several authors have therefore started to research run-time scratchpad management techniques. For instance, [12] and [2] decide at design-time for each call-site to malloc/new to which memory the data should be assigned. They base their decision on simple criteria: object co-location to avoid conflict misses, object size and energy-efficiency. Upon object-creation the processor needs to explicitly transfer data between the main memory and scratchpad, which is rather costly. We provide therefore extra hardware (an DMA) to reduce the copy overhead. As a consequence, we can initiate more transfers between scratchpad and main memory, thereby enabling the assignment of different data objects (such as array tiles) to the scratchpad. Also, instead of relying on the existing memory managers that come with the OS, we provide a configurable memory manager, which can be customized to further reduce the cost. Finally, in contrast to prior work, we quantify the real scratchpad overhead on a cycle-accurate processor platform.

Compared to existing platforms available in industry (see e.g. ST LX or TI C6x), we provide a higher-level and more integrated environment to exploit the scratchpad. The details of our environment are outlined in the next section.

3. PLATFORM EXTENSIONS

In this section, we describe the hardware extensions and software support that we have integrated on our platform to alleviate the designer from the cumbersome details of exploiting a complex memory hierarchy.

3.1 Hardware Extension

On existing memory architectures (see Fig. 1 left) both the cache and scratchpad memory are connected to a local memory bus. Whenever new data needs to be moved in/from the cache, the processor enters a special state. Its Memory Management Unit (MMU) fills then the cache-lines by transferring small bursts over the bus-master port. To move data into(/out) of the scratchpad, the data is always first fetched in the cache and only then copied into the scratchpad (or main memory).

Compared to scratchpads used in earlier academic work, we place an DMA engine next to each processor. The DMA enables

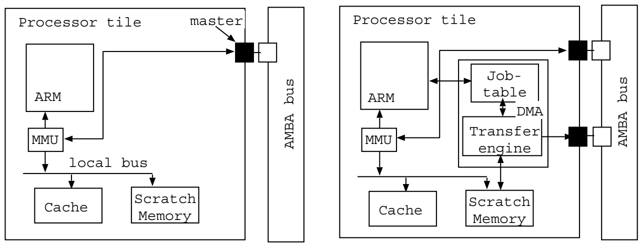

Figure 1: Hardware extensions for scratch-management: (left) original-(right) extended

memory transfers between the scratchpad and the main memory without any processor involvement (if possible). It consists of two parts: a controller and a transfer engine. The controller is the main interface between the processor and the DMA. The processor programs DMA jobs and enquiries on the status of DMA jobs by writing/reading into/from the controller's address space. An DMA job contains information on the source addresses and stride and the target addresses and stride. Every time the transfer engine is free, and the job queue is not empty, the DMA controller starts a new transaction on the transfer engine. In order to transfer a block of data, the transfer engine generates the necessary burst accesses to the bus and accesses to the scratchpad memory. The bursts to the bus can be of different size. The transfer engine contains a 64B queue to store incoming/outgoing data waiting to be forwarded to the scratchpad or to the main memory. Finally, it is connected to the memories outside of the processing tile through a master port [1] . Inside the processing tile a dedicated connection exists to the scratchpad.

A cycle-accurate model of both the DMA and scratchpad memory have been integrated in the MPARM simulator environment ([7]). MPARM is a full-system SystemC-based simulator which can track, in a cycle-accurate fashion, performance and power consumption of a multi-processor systems-on-chip. A complete operating system is integrated in the environment, thereby enabling research on the delicate interplay between software and hardware.

3.2 Configurable Dynamic Memory Manager

The memory space available at run-time to our applications (i.e., the heap). 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. To keep fragmentation of the memory space and the allocation overhead bounded, we have integrated a configurable DMM in our environment.

In order to customize the DMM, we divide the available memory space in segments and select an appropriate memory manager for each segment. Currently, we have integrated two memory managers, which we both borrowed from RTEMS[10]. The first one is a simple *partition* manager with low assignment overhead, the other one is a more complex *region* manager which uses more effectively the available memory space (see [13]).

The *partition manager* partitions the available memory in equal-sized chunks. It uses a single table to keep track of the free/used chunks, which limits the allocation overhead. The *region manager* uses a doubly linked list to keep track of the free-blocks. When a tasks requests memory space, a first fit mechanism is used and then memory splitting and coalescing techniques are used to return

[1]For a shared bus, like AMBA AHB, we could also reuse the master port of the MMU, saving energy area and obaining the same performance. With a complex connection it's something that can be interesting to explore.

function	arguments	description
SMcreateManager	startaddress managed_size type id chunk-size (opt)	initializes a memory manager for a part of the SM type is either a partition or a region manager
SMmalloc	size id	like a normal malloc id specifies the segment in which to allocate
SMfree	pointer id	like a normal free
DMAjob	width length M1 address M1 matrix width M2 address M2 matrix width direction waitfor (wait or go)	create a job in the DMA-Q
DMAwait	object	wait until the job of object is ready
DMAnewstate	object direction waitfor	reinitialize
DMAM1(2)add	object address	change M1 start-address
DMAfree	object	remove the object from the DMA-Q

Table 1: Scratchpad and DMA API

a tightly-fitting memory chunk and to reduce fragmentation. As a result of the maintenance structures for the multiple allocation sizes allowed, the allocation overhead is slightly higher in the latter manager.

3.3 Application Programmers Interface

Since we want to relieve the designer as much as possible of the cumbersome details of programming the DMA and the scratchpad, we provide several high-level functions. The main ones are shown in Tab. 1.

The scratchpad memory (SM) functions are used to assign data in the scratchpad at run-time. Before the scratchpad can be used at run-time, the custom run-time manager first needs to be initialized. For this purpose, *SMCreateManager* starts a memory manager in a segment of the scratchpad, specified by its start address and size. Depending on the type, either a region or a partition manager is created. In case a partition manager is used, the designer should also set the size of each partition. To call the memory manager in a specific segment, the designer should call *SMmalloc/free* together with the id of the manager.

The DMA routines are used to specify memory transfers between the scratch and the main memory. An DMA job is initialized using *DMAjob*. Its first two arguments (width and length) specify the shape of the block transfer. The following four arguments contain information on the start-address and size of the data structures between which data is exchanged. This information is needed by the transfer engine to generate the addresses for the burst/copy operations. The direction argument indicates which block is the source/destination of the transfer. When the last argument is set to one, the processor is stalled until the DMA transfer is done. Otherwise the processor continues its execution in parallel with the DMA. At all times, the processor can wait for an DMA transfer to finish by calling the *DMAwait* function. After the transfer is done, the same DMA object can be reinitialized to implement a different transfer. We provide *DMAnewstate* and *DMAM2add* for this purpose.

Our approach thus consists of special-purpose hardware to efficiently transfer data to/from the scratchpad and a high-level API which makes it possible to explore several scratchpad management solutions.

4. RUN-TIME SCRATCHPAD MANAGEMENT FOR DYNAMIC MULTI-TASKED APPLICATIONS

The goal of this case-study is to demonstrate our integrated approach on a small example and compare it with existing techniques.

TASK A

```
while(input){
  key[i] = malloc(4*32);
  ...
}
```

TASK B

```
  int X[N*N], Y[N*N], Z[N*N];
  int i,j,k;

  for (int i=0; i<N; i++) {
    for (int j=0; j<N; j++) {
      Z[i*N+j] = 0;
      for (int k=0; k<N; k++)
             Z[i*N+j] +=
               X[i*N+k] * Y[j+k*N];
   }}
```

OS-boot

```
SMcreateManager(scratch,2kB,
                region,Manager1)
```

TASK A

```
while(input){
  key[i] = SMmalloc(4*32, Manager1);
  ...
}
```

TASK B

```
  int X[N*N], Y[N*N], Z[N*N];
  int i,j,k;

  /* local memory */
  int *X_lr = SMmalloc(N*4,Manager1);
  object[0]= DMAJob(N,1,N,N,(uint)X_lr,(uint)X,2,true);

  for (int j=0; j<N; j++) {
    for (int i=0; i<N; i++) {
      DMAM2add((uint)((int*)X+i*N),object[0]);
      DMAnewstate('to',object[0],true);
      Z[i*N+j] = 0;
      for (int k=0; k<N; k++)
           Z[i*N+j]+= X_lr[k] * Y[j+k*N];
    }}
```

Figure 2: Motivational example: original code (left), integrated approach (right)

Fig. 2 left presents the original code of two independent tasks. In the context of this example, we assume that both tasks are executed on a single processor using a round-robin scheduler. In general, the tasks can be instantiated several times due to user-events. The energy and performance of the task-set running on a 4kB cache memory is shown in point 1 of Fig. 3.

Both the performance and the energy can be easily improved with a scratchpad memory of 2kB[2] (see e.g. points 2 and 4 which both exploit a scratchpad). Because applications are becoming dynamic, the scratchpad needs to be managed at run-time. As an example of dynamic behavior, remark in the original code of Fig. 2 left, that task A uses *malloc(4*32)* to reserve more memory space depending on *input*. Due to this dynamic behavior, it is hard to predict at design-time how much and when data should be assigned to the scratchpad. Hence, existing design-time techniques are suboptimal.

With our integrated approach, the limitations of the design-time approaches can be overcome. In Fig. 2 right, we show the code changes which are needed to manage the scratchpad at run-time. At boot-time of the OS, we instantiate a memory manager in the scratchpad. We call for this purpose *SMCreateManager* and initialize a *region*-manager in a 2kB segment of the scratchpad. Note that by replacing the third argument of *SMCreateManager* with *partition*, we could replace the *region*-manager with a *partition*-manager. To allocate data at run-time, the designer calls *SMmalloc* (see e.g. task A). This function has the same API as an ordinary malloc except that the dedicated manager is passed as a second argument (*Manager1*). Using above code changes, the designer can implement existing run-time scratchpad management techniques. As an example, we reuse the technique presented in [12] and malloc the dynamic data of Task A in the scratchpad. The energy and performance of this solution are shown in point 2.

In our approach, we introduce an DMA to reduce the overhead of data transfers between the memories. More transfers can then

[2]In this case we use a 2kB cache.

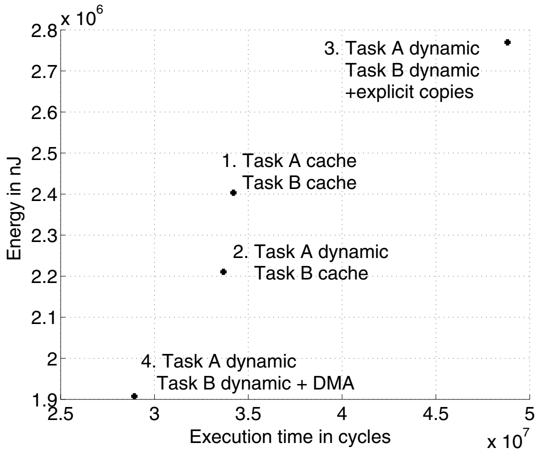

Figure 3: Case-study: even for dynamic applications a scratchpad with DMA outperforms a cache only solution

be executed for the same cost, thereby making possible to store data that needs to be frequently updated on the scratchpad. A well known example of such data in signal processing applications are array-tiles. After an application has been tiled, the tiles themselves are small enough to fit in the scratchpad, but need to be frequently updated with new information from the original array. We show in Fig. 2 again the code change to assign the tile to the scratchpad and update it with the DMA. We assign the tile (*X_1r*) to the scratchpad with *SMmalloc*. Then, we create an DMA transfer with *DMAJob*. The arguments to this function indicate the source (*X*) and target data structure (*X_1r*) and the shape of the block transfer. In this case, we move a rectangular tile of *N* by *1* elements. At the start of every *i*-iteration, the start address of the source tile is written in the DMA controller with *DMAM2add* and the transfer is initiated with *DMAnewstate*. The DMA transfer is then automatically executed by the transfer engine. Because the DMA transfers data more cost-efficiently, more and better data structures can be assigned to the scratchpad. Point 4 in Fig. 3 represents the performance and energy consumption of the solution with array-tiling and DMA. In contrast, when the tiles are assigned to the scratchpad but no DMA is used, the performance becomes worse, because of the extra processor overhead (point 3). Also note that in spite of the reduced number of cache misses, the energy consumption is in this case higher as a result of the intermediate read/write operations in the cache required to move the data to the scratchpad. With the help of our high-level API, the programming complexity to assign the tiles to the scratchpad and initiate the DMA transfers is reduced to a minimum.

From this example and the results of Sect. 6, we can conclude that our integrated management approach improves on existing solutions. However, even more space for improvement exists, as outlined in the next section.

5. DESIGN-SPACE EXPLORATION

In this section we explain several parameters which can be tuned to further optimize run-time scratchpad management. The most important ones are shown in Fig. 4 and will be discussed in the following sections.

5.1 Scratchpad candidates selection

A first important step is to decide which data structures are good candidates to map in the scratchpad. Typically, frequently accessed data structures are interesting candidates. Obviously, some frequently accessed data structures are too big to fit in the scratchpad. To increase freedom for run-time scratchpad management, we then try to build partial copies of these data structures that fit in the scratchpad. We reuse for this purpose techniques such as [14]. We also introduce the code to update the tiles whenever they will be assigned to the scratchpad.

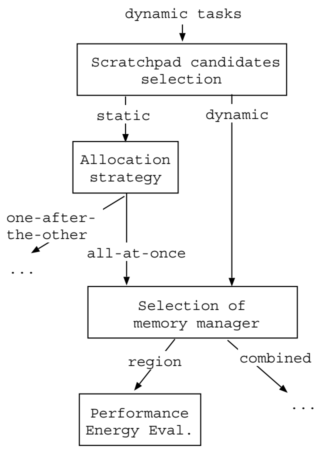

Figure 4: Design-space exploration for dynamic scratchpad management

The candidate data structures are either statically declared inside the task code or generated with a dynamic memory management routine (such as malloc). Two differences exist between them: (1) the static ones are assigned at the start-up phase of the task whereas the dynamic ones are assigned during the execution of the task; (2) for the static ones a fixed size needs to be assigned whereas for the dynamic ones the size may vary.

After the scratchpad candidate selection, we end up with a set of statically declared data structures a set of call-sites (new/malloc) which could benefit from the scratchpad.

5.2 Allocation order

Since all statically declared data is known at the start of a new task, we can apply different allocation strategies to reduce the assignment overhead. Particularly, we can vary the order in which space for the different static data is requested from the memory manager. In a first approach, we request space for all the static data with one assignment (*all-at-once* policy). The main idea is that when free space is available in the scratchpad, we only need a single call to the memory manager. If the assignment fails, we assign the least important object to the main memory. In our second approach (*one-after-the-other*), we try to successively assign the data in order of decreasing benefit (number of accesses divided by size). Like greedy knapsack heuristics, we thus assign the data which has the largest energy/size advantage first. In both the two case we continue until all the data is assigned to a memory (the main memory or the scratchpad).

5.3 Run-time manager selection

Customizing the memory management routine for the scratchpad can significantly improve the allocation speed and reduce the fragmentation. We configure our memory manager by splitting the scratchpad memory space in segments. Each segment is managed with a different run-time policy. In the context of the case-study, we have explored the following two combinations: (1) we manage the entire scratchpad with a single region manager, which behaves like an ordinary malloc found in a C-library; (2) we use a combined region and partition manager. Half of the scratchpad is managed by the region manager, the other half by a partition manager.

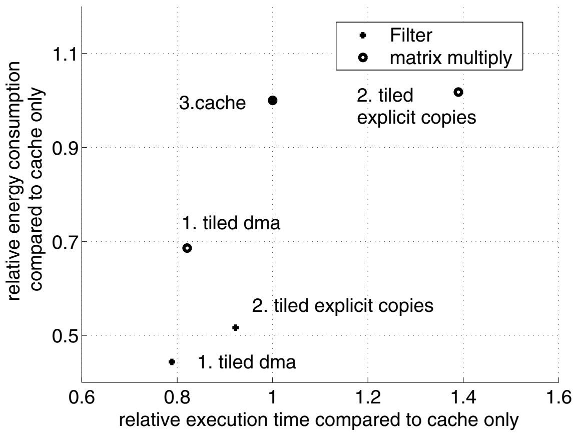

Figure 5: Explicit copying compared to DMA

The partition manager takes care of all calls to malloc/new inside the original code. The region manager assigns the statically declared data structures of the tasks. In the next section we compare the different management policies.

6. EXPERIMENTAL RESULTS

We present three different experiments. First, we quantify how much an DMA improves existing run-time scratchpad management techniques which use explicit copying. Secondly, we explore how much the allocation order and selection of the run-time manager influence the performance and energy cost. Finally, we provide an example of our approach in a multi-processor context. The energy parameters used during the MPARM simulations are presented in Tab.2. The ARM core runs at a frequency of 200 Mhz and the switching time of the Round Robin scheduler is 1ms.

	Size	**Energy**
Cache	Direct Mapped 8KB	energy 0.23nJ/access
	Direct Mapped 4kB	energy 0.21nJ/access
Scratchpad SRAM	4kB	energy 0.142nJ/access
L2 SRAM memory	256kB	energy 0.99nJ/access

Table 2: Platform parameters

An important advantage of our solution is that we use a DMA to avoid explicit memory copy operations to fill the scratchpad. We quantify this improvement on two applications which we have tiled (*filter* and *matrix multiply*). The copies of the tiles are assigned to the scratchpad memory. We measure the performance and energy consumption using on the one hand explicit copies and on the other hand the DMA to update the tiles (see Fig. 5). Explicit copying generates processing overhead and also extra accesses in the cache, which are needed to transfer the data to the scratchpad. As a consequence, the points without DMA (points 2) have a worse execution time and energy consumption compared to the points with DMA transfers (points 1). Remarkably, in case of *matrix multiply*, explicit copying causes so much overhead that its performance and energy cost are worse than the cache-only solution (point 3). Storing data tiles on the scratchpad can thus become inefficient when no DMA is present on the platform. Vice versa, an DMA creates more opportunities to exploit the scratchpad.

Our integrated scratchpad-manager is further validated with the help of a realistic execution trace. It models a sequence of 26 data-independent tasks which are started by some random events (representing a "typical" user). The tasks are executed on a single processor running a round-robin scheduling policy. They are either instances of static applications such as *image filter*, *matrix multiply* or *edge compression* derived from Mediabench, or they are instances of, a malloc-intensive network application borrowed from the NetBench benchmark suite, i.e. Deficit Round Robin (or DRR from now on).

The characteristics of the tasks are shown in Tab. 3. The second column indicates the number of data structures which can be assigned to the scratchpad memory; the third column indicates the sum of the sizes of these data structures.

name	*nr. mallocs*	*max. allocated size*
image filter	3	960B
image filter light	3	480
matrix multiply	3	224B
matrix multiply light	3	112B
edge	3	256B
edge light	3	128B
deficit round robin	dynamic	dynamic

Table 3: Task characteristics

In Fig. 6 we compare the performance and energy consumption for the workload described above under six scratchpad management policies. As a reference, we run the workload on a cache-based architecture. This results in the slowest and most energy-inefficient solution (point 1). We also try to reuse existing design-time approaches. For this purpose we predict which tasks are most common at run-time and reserve space in the scratchpad for their statically declared data. As a consequence, we reduce the access cost (point 2) compared to cache-only. However, the more difficult it becomes to predict which data structures are active, the harder it becomes to apply this technique.

The points (3,4,5,6) of Fig. 6 show the results of our run-time approach, where both dynamic and static data are assigned to the scratchpad. In points 3 and 4, we use a single *region*-manager to manage the scratchpad. We respectively try to assign the statically declared data *all-at-once* (point 3) or assign the data in order of decreasing benefit *one-after-the-other* (point 4). Although, in principle, assigning the data all-at-once reduces the number of malloc-calls, it consumes more energy and executes slower. This is mainly due to a mismatch in the requested size of DRR and the other static applications. As a consequence, the scratchpad memory becomes fragmented and 60% of the allocation requests need to be directed to the main memory. Obviously, since the processor has to access these data through the cache, the energy consumption increases. By assigning the data one-after-the-other, a better match exists between the static tasks and DRR. Almost all data (99% of the requests) can be assigned to the scratchpad and less cache-misses occur. One-after-the-other (point 4) is therefore faster and more energy-efficient than all-at-once (point 3).

Another way is to reduce fragmentation is to logically split the scratchpad memory in two segments (points 5-6). Each segment is managed by an appropriate memory manager for the specific type of requests. We use a cheap partition manager for the equal-sized requests of the DRR applications and a region-manager for the variable size static data requests. No fragmentation occurs and the same data as in point 4 can be assigned to the scratchpad. This explains why points 4 and 5-6 have a similar energy consumption. The customized manager (points 5 and 6) executes slightly faster than point 4, because it uses a faster partition manager to assign the data of the DRR tasks. Almost no difference exists between assign-

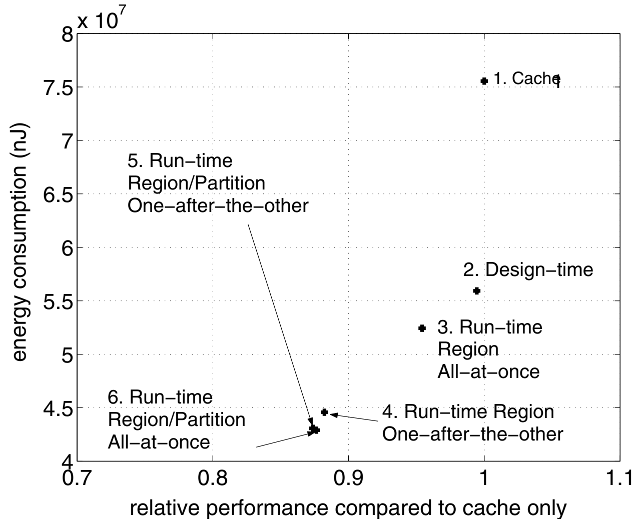

Figure 6: Energy/Performance for the multi-threaded execution workload under different scratch management policies

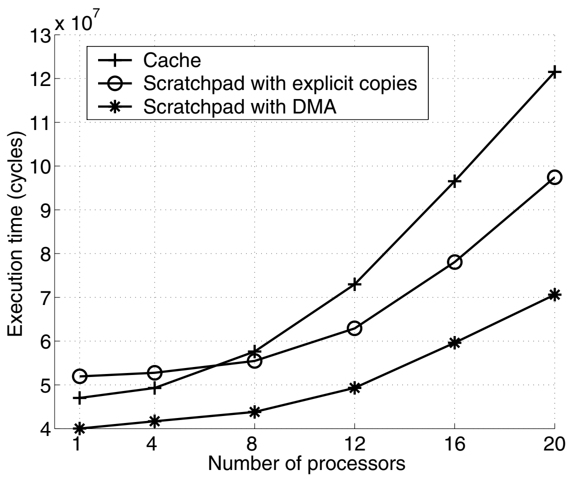

Figure 7: Execution time depends on bus congestion

ing the data all-at-once (point 5) or one-after-the-other (point 6) in case of a customized manager.

Finally, we quantify the benefits of our approach for multi-processors. Our multi-processor architecture consists of several ARM processors connected to a shared L2 memory with an AMBA bus. The L2 memory has an access latency of three cycles. Each processor runs an independent application, thus no problem of coherency between shared memory is present. In Fig. 7, we compare three architectures: one only with caches, another one with scratchpads and finally one with scratchpads and DMA. The architecture with DMA outperforms both the cache and scratchpad alone. It also scales better with an increasing number of processors. Also notice that the execution time of the scratchpad one is first worse (one processor) than cache-only, but becomes better when the number of processors is increased (more than eight). When only one processor is used, the access latency to the L2 memory is relatively small. Therefore, the reduction in cache misses due to the scratchpad is outweighed by the extra processing overhead to fill the scratchpad. However, when the number of processors increases, traffic on the bus becomes congested. Consequently, the access latency increases and any reduction in the number of cache misses results then in a measurable performance improvement.

7. CONCLUSION

In this paper, we have presented an integrated software/hardware approach to take advantage of energy-efficient scratchpad memories for dynamic applications. Our experimental results show that even in dynamic applications scratchpads are energy-efficient and improve performance. The DMA next to the scratchpad enables the use of the scratchpad for more data structures. An important decision for scratchpad management is the selection of run-time memory manager. Designers can trade/off fragmentation and assignment overhead by customizing the memory manager. A good way is to cluster allocations requests of equal size and map them onto independently managed segments. Finally, changes in the application code for run-time scratchpad management can be limited with the help a high-level API to the DMA and scratchpad. In future work, we will like build a methodology to automate the selection of the scratchpad candidates and the selection of an appropriate memory manager.

8. REFERENCES

[1] F. Angiolini, L. Benini, and A. Caprara. Polynomial-Time Algorithm for On-Chip Scratchpad Memory Partitioning. In *Proc. CASES*, 2003.

[2] O. Avissar, R. Barua, and D. Stewart. Heterogeneous Memory Management for Embedded Systems. In *Proc. CASES*, 2001.

[3] L. Benini, A. Macii, and E. Macii. Increasing Energy Efficiency of Embedded Systems by Application Specific Memory Hierarchy Generation. *IEEE Design and Test*, 17(2):74–85, 2000.

[4] L. Benini and G. De Micheli. System-level power optimization: techniques and tools. *ACM TODAES*, 5(2):115–192, 2000.

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

[6] M. Kandemir, J. Ramanujam, et al. Dynamic Management of Scratch-pad Memory Space. In *Proc. DAC*, 2001.

[7] Loghi M., Angiolini F., Bertozzi D., Benini L., and Zafalon R. Analyzing Chip Communication in a MPSoC Environment. In *Proc DATE* , 2004.

[8] P. Panda, N. Dutt, and A. Nicolau. Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications. In *Proc. DATE*, 1997.

[9] P.Panda, F.Catthoor, et al. Data and Memory Optimizations for Embedded Systems. *ACM TODAES)*, 6(2):142–206, Apr. 2001.

[10] RTEMS. www.rtems.com.

[11] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing Energy Consumption by Dynamic Copying of Instructions onto Onchip Memory. In *Proc. ISSS*, 2002.

[12] S. Tomar, S. Kim, N. Vijaykrishnan, M. Kandemir, and M. Irwin. Use of Local Memory for Efficient Java Execution. In *Proc. ICCD*, 2001.

[13] P. Wilson, M. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *Proc. Int. Workshop. Memory Management*, 1995.

[14] S. Wuytack, J. Diguet, F. Catthoor, and H. De Man. Formalized methodology for data reuse exploration for low-power hierarchical memory mappings. *IEEE Trans. VLSI Systems*, 6(4):529–537, Dec. 1998.