Run-time Scratch-pad Memory Management for Embedded Systems

Tiago Rogério Mück and Antônio Augusto Fröhlich
Software/Hardware Integration Lab
Federal University of Santa Catarina
Florianópolis, Brazil
{tiago,guto}@lisha.ufsc.br

***Abstract*—Scratch-pad memories (SPM) are being increasingly used in embedded systems due to their higher energy and silicon area efficiency in comparison to ordinary caches. However, in order to exploit all of its advantages, efficient memory allocation mechanisms must be provided. In this work we propose a run-time memory management approach for SPMs at OS-level that can be combined with other compile-time approaches. The operating system memory manager takes annotations inserted into the code by the programmer as hints to choose the most appropriate memory (i.e. main memory or SPM) for each allocation. Experimental results confirm the approach's efficiency when compared to a similar compile-time technique.**

I. Introduction

Contemporary embedded system applications are requiring faster processors and larger memories. Previous studies have shown that the memory subsystem is responsible for 50%–75% of the total system power consumption and occupies significant chip area [12], thus becoming an important optimization point. Most systems rely on cache-based memory hierarchies due to the capacity of caches to exploit the spatial and temporal locality of memory access. However, caches require additional memory for tag and address comparison logic, which may significantly increase their power consumption and silicon area. These often render cache inappropriate for embedded applications [19]. In addition, *worst case execution time* (WCET) must be overestimated due to the lack of predictability in many cache implementations [20], which may forbid their use on real-time embedded systems.

Software-controlled caches, often called *scratch-pad memories* (SPM), are emerging as alternatives to traditional caches. SPMs do not need the extra logic to map data and instructions because memory allocation is controlled by software, which makes them more power and area efficient than ordinary caches [19]. Also, considering that the contents of the SPM are known, tighter bounds on WCET prediction can be achieved [20]. Nevertheless, to exploit all the advantages of SPMs, an efficient allocation mechanism must be provided in software. Several software allocation approaches that have been proposed rely on profiling information to define the instructions and data to be allocated to the SPM at compile-time. These approaches have a major drawback. The use of profiling limits the scope of the mapping techniques not only because of the difficulty in obtaining reasonable profiles but also due to high space and time requirements for their generation [16]. This is especially true for dynamic applications in which the memory access patterns depend on the input data [8].

In this work we propose a run-time memory management approach for SPMs at OS-level which does not rely on compiler support, profiling or hardware support. We provide a framework that abstracts the SPM and parts of the main memory as operating system heaps. When the application dynamically allocates data, the operating system uses annotations, inserted into the code by the programmer, as hints to choose the most appropriate level in the memory hierarchy to allocate the data (i.e. the main memory heap or the SPM heap). If an allocation request on the preferred memory component fails, the operating system attempts to allocate on a less optimal one, thus handling exhaustion of a particular memory component. With this approach, the OS can take advantage of the developer's knowledge about the application to provide an efficient data allocation.

The remaining of this paper is organized as follows: section II presents a discussion about related work; section III presents our proposal; and sections IV and V show our experimental results and conclusions.

II. Related work

In the few last years, several SPM management approaches have been proposed. We have separated these approaches in the different categories shown bellow.

Static approaches are those in which the contents of the SPM are fixed prior to system deployment and never change. They can also be subdivided into two categories. In **compile-time techniques**, the allocation is defined during the compilation process [7][11]. In **post-compile techniques**, algorithms are applied at the final binary code to change the memory allocation [6]. These static allocation approaches either use greedy strategies to find an efficient solution, or model the problem as a *knapsack* problem or an *integer-linear programming* problem (ILP) to find an optimal solution. *Avissar et al* [7] proposed a memory allocation strategy that is optimal in relation to the profiling provided. In *Angiolini et al* [6] another optimal solution was proposed. This technique is applied in the application binary code and is optimal in relation to a certain set of execution traces. Despite being optimal, all

the static methods have the disadvantage of being dependent of an efficient application profile.

Dynamic approaches are those in which the SPM contents change during the program execution. Dynamic methods based on **compile-time techniques** change the SPM allocation based only on compile time decisions and profile information [19][18][15]. *Verma and Marwedel* [19] proposed an overlay-based memory allocation approach for both code and data. It uses ILP to find the optimal memory allocation and overlay points that minimizes energy consumption for a given profile. Compile-time dynamic methods are usually more efficient than static ones in exploiting the benefits of SPMs, but they are still dependent on good application profiles. Another issue of these methods is that they do not provide an efficient allocation when the accessed memory regions are correlated to the application's input data [14]. This issue is overcome by the **run-time techniques**. These approaches are generally composed by a run-time software or operating system which determines the SPM contents based on information inserted by the compiler and/or using hardware support.

In *Milidonis et al* [14] the global data structures are sliced into *tiles* which are allocated to the SPM by the compiler. A hardware component called *Data Type Unit* (DTU) works like a special cache that keeps references to the most accessed tiles and sends commands to a DMA unit to move the data between the SPM and main memory when necessary. This approach solves the problem of dynamic access patterns, but it still require an initial profiling to determine the possible tiles, and a very specific hardware support that forbids it application to fixed hardware platforms.

In *Shrivastava et al* [16] is proposed an approach that does not require profiling or hardware support. They manage function stack frames, allocating them to the SPM when they are in use. They proposed a software library which provides functions to manage the stack. The calls to the library are inserted by the compiler before and after function calls, using information provided by *Global Call Control Flow Graphs* (GCCFG). This approach have the advantage of not requiring profiling or hardware support, but it still require compiler modifications and focus on only one class of applications (multimedia applications).

In *Cho et al* [8] was proposed an allocation scheme that integrates compiler, OS, and hardware, which is very similar to *Milidonis et al* [14]. First, through profiling, the most accessed data sets are defined. Based on a cost analysis, the compiler define the optimal points to insert the operating system calls the reallocate the SPM data. When reallocating, the operating system verifies a hardware structure which keeps information about the most accessed data sets. The OS uses this information to define the new memory allocation. Since the data addresses change during run-time, they added a simplified MMU for address translation. The authors show that the technique is very efficient for multimedia applications, but it requires compiler, OS, and hardware support, thus it comes up with all the drawbacks of the dynamic run-time techniques discussed earlier.

All of the previous techniques handled only code and/or global/stack data. The only known technique that allocates heap data to the SPM was proposed in *Dominguez et al* [5]. The proposed technique divides the application code in regions in which the beginning and end of each region is defined by functions and loops bounds. A compile-time analysis is performed in these regions to define the heap variables that will be allocated to the SPM and where the code to make this allocation is to be inserted. Despite of being the first technique that allocates heap data to the SPM, it follows the **dynamic compile-time approach** discussed earlier, and suffers from the same problems. Another work that deals with the management of heap data was presented in *McIlroy et al* [13]. In this work, the authors just suppose a run-time system to manage SPMs as dynamic heaps, and propose a heap memory allocation algorithm optimized for very small memories.

III. Run-time SPM management

We have implemented a framework for the C++ language which provides annotated versions of the *new* and *delete* operators, allowing the programmer to easily insert allocation hints into the program. The framework was implemented on the *Embedded Parallel Operating System* (EPOS) [2]. EPOS relies on the *Application-Driven Embedded System Design* (ADESD) [10] methodology to design and implement both software and hardware components that can be automatically adapted to fulfill the requirements of particular applications. High reusability and low overhead are achieved by a careful implementation that makes use of object-oriented programming and *generative programming* [9] techniques, including *static metaprogramming*. A detailed description of the memory management framework and its implementation on EPOS is presented bellow.

A. *Placement* new *and* delete *operators*

The annotated versions of the *new* and *delete* operators were implemented using *placement new expressions*, which are part of the ISO C++ standard and are supported by any standard C++ compiler. *Placement new and delete expressions* provide a way to implement custom allocation strategies. Figure 1 shows the syntax of these expressions. Invocations of the type *new (expression-list) type_name* will require its respective overloaded version of the *operator new* function. The ISO C++ standard already defines a default implementation for the type *void**. Pointer placement new is necessary for hardware that expects a certain object at a specific hardware address [17].

```
(a)  new type_name (initializer-list);
(b)  void * operator new (size_t);

(c)  new ( expression-list ) type_name (initializer-list);
(d)  void * operator new (size_t, expression-list);

(e)  new ((void*)ADDRESS) type_name;
(f)  void * operator new (size_t, void *);
```

Figure 1. Default syntax (a) and function (b) of the *new* operator. Bellow, the generic syntax (c) and function (d) of the *placement new* is shown with an example of the default implementation (e) (f).

B. Memory management on EPOS

The memory mapping in EPOS is shown in figure 2a. The allocation of code and global data is defined at compile-time for both operating system and application. During system initialization, the initial stack used by operating system is allocated in the *system stack* region. All of the remaining free memory is distributed between the *system and application heaps*. These regions are used for dynamic memory allocation, and each one is managed by an instance of the *Heap* C++ class. The system heap (*System::heap*) is used to allocate the stacks for the application's and system's threads, while the application heap (*Application::heap*) is used by the C++ *new* and *delete* operators, which are used by the application programmer to dynamic allocate memory.

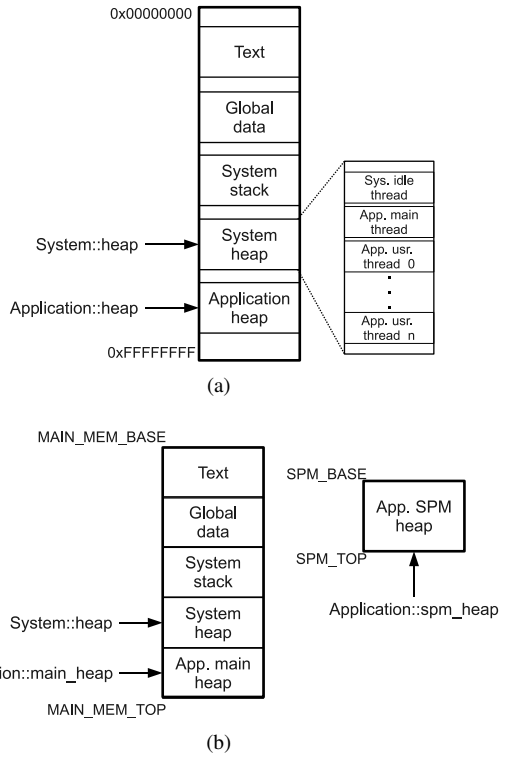

Figure 2. EPOS memory mapping before (a) and after (b) using the new framework.

C. The allocation framework

Figure 2b shows the modified memory mapping when using our SPM management approach. A new *Heap* instance is created to manage the SPM, which is memory-mapped. We overloaded the implementation of the *new* and *delete* operators to include a decision algorithm which decides if the data should be allocated using the *main_heap* or the *spm_heap*, based on annotations given by the programmer.

Three different types of annotations are supported: **ALLOC_HIGH**, **ALLOC_LOW**, and **ALLOC_NORMAL**. When the *ALLOC_HIGH* annotation is used, it means that particular object has a high memory access priority (i.e. performance/power consumption of read/write operations on it have a major impact over the system efficiency) and should be allocated on the more efficient level of the memory hierarchy (e.g. a SPM). The *ALLOC_LOW* have the inverse meaning. It means that the object has a low memory access priority and it can be allocated on the less efficient level of the memory hierarchy without a significant impact on the system efficiency. Finally, the *ALLOC_NORMAL* annotation is used when the programmer does not know or does not care about the physical location of the data. In this case, the OS decides the best place to allocate the object.

Figure 3 shows how these annotations are used with the *new* operators. The annotations are defined as *enum constants* and a new *operator new* implementation is defined using the *placement new* syntax. In this example, four arrays of type *int* are allocated with and without annotations. When annotations are not used, the OS assumes *ALLOC_NORMAL*. The usage of the *delete* operator doesn't change.

```
//Placement type definitions
typedef enum {
    ALLOC_HIGH,
    ALLOC_LOW,
    ALLOC_NORMAL,
} alloc_priority;

//Annotations implementation using placement new
void * operator new(size_t bytes, alloc_priority p){
    ...
    //do memory allocation
    ...
}

//Anotated memory allocation examples
int *at_spm = new (ALLOC_HIGH) int[10];
int *at_main_mem = new (ALLOC_LOW) int[10];
int *somewhere = new (ALLOC_NORMAL) int[10];
int *somewhere_else = new int[10];//*equivalent to the
                                  statement above/*

delete[] at_spm;
delete[] at_main_mem;
delete[] somewhere;
delete[] somewhere_else;
```

Figure 3. Memory allocation request and implementation using annotations.

The pseudo-code in figure 4 describes the decision algorithm used to define where the data will be allocated. If an allocation request can't be accomplished on the preferred memory component, the operating system attempts to allocate on a less optimal memory component. If it is left to the operating system to decide where to allocate, it attempts to allocate on the heap with the highest percentage of free space, thus handling exhaustion of a particular memory component. The implementation of the memory deallocation is straightforward. The OS just checks the pointer address to decide if the memory is to be deallocated on the main memory or on the SPM.

IV. Evaluation

We have used the *Xilinx ML605 Evaluation Board* to build a platform for our evaluation. The platform is based on open source hardware IP-cores from *OpenCores* [4]. We have used the *aeMB* processor which is connected to a memory and several peripherals using a *Wishbone* bus. Figure 5 shows a

```
do memory allocation (size, annotation)
    case: annotation = ALLOC_HIGH
        if fits on spm (size)
            allocate on spm
        else
            allocate on main mem
    case: annotation = ALLOC_LOW
        if fits on main mem (size)
            allocate on main mem
        else
            allocate on spm
    case: annotation = ALLOC_NORMAL
        if spm free percentage >
                main mem free percentage
            if fits on spm (size)
                allocate on spm
            else
                allocate on main mem
        else
            if fits on main mem (size)
                allocate on main mem
            else
                allocate on spm
```

Figure 4. Algorithm which defines where the allocation will be done.

block diagram of our platform (for simplicity, it shows only memory-related blocks). It features a cache for instructions. The 512 Mb DDR3 SDRAM available in the ML605 board is used as the main memory; while a 16 Kb SPM is created using the FPGA's internal RAM blocks. We have implemented a *bus sniffer*, which is connected to both instruction (*IWB*) and data (*DWB*) sides of the Wishbone bus. This block contains performance counters, which are used to collect statistics about memory operations. The final system is generated using ISE 13.1 (for hardware synthesis) and GCC 4.1.1 (for software compilation), both provided by Xilinx.

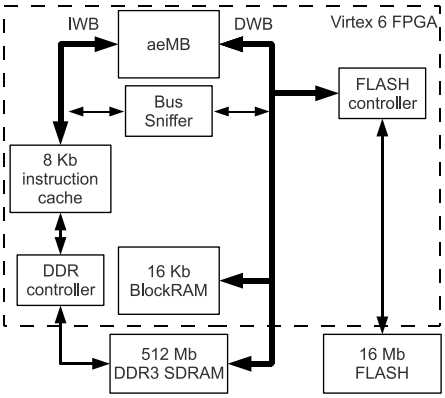

Figure 5. Block diagram of the evaluation platform. Only memory-related components are shown.

A. Benchmarks

In order to evaluate our approach, we have selected the following set of benchmarks from *MiBench* [3]:

Dijkstra: calculates the shortest path between nodes using Dijkstra's algorithm.

SHA: a secure hash algorithm.

Susan: implements a set of image recognition algorithms. In the following evaluation, *S-Smoothing* stands for the image smoothing algorithm, and *S-Corners* stands for the corner detection algorithm.

FFT: Fast Fourier Transform on an array of data.

These benchmarks were modified in order to be correctly compiled as EPOS C++ applications. The main modifications consisted of: replacing *malloc* and *free* C function calls by C++ *new* and *delete* operators, and changing the way the benchmark reads the input data; instead of reading from a file, it now reads from a FLASH memory (a file system is not available in the evaluation platform). Table I shows the resulting memory footprint of the benchmarks.

In our approach, only data which is dynamic allocated at run-time can be handed out to the SPM. However, this limitation can be softened by declaring global and stack data as heap data. The last two columns of table I (*Modified benchmarks* columns) show the footprint of the benchmarks when they are modified with this optimization. We have modified the benchmarks by redefining *only arrays* which were declared as global and stack data to heap data. Former global data are allocated before the application executes (e.g. at the beginning of *main* function). Former stack data are allocated and deallocated at the beginning and the end of functions. The *Susan* benchmarks did not require any modification since they originally relied on dynamic allocation.

B. Results

We evaluated both the original and the modified (global/stack arrays as heap arrays) benchmarks on three different configurations of our evaluation platform:

No SPM: our allocation framework is not used and all heap data is allocated in the external SDRAM.

SPM: our framework is enabled but no annotations are given. SPM allocation is fully handled by the operating system.

SPM-A: the **SPM** configuration with *ALLOC_HIGH* and *ALLOC_LOW* annotations added to *new* invocations. Since one of the main ideas of this work is to avoid profiling, the annotations where intuitively added based only on the authors knowledge about the benchmarks.

Figures 6a and 6b show the execution time of the original and the modified benchmarks, respectively. On the original benchmarks an average improvement of 5% is achieved when the SPM is fully managed by the operating system. Some benchmarks didn't show any significant improvement. This can be explained by the data access patterns shown in figure 7. Benchmarks on which most of the memory accesses were performed on global and stack data didn't benefit from our approach. By adding annotations we have improved the execution time in 7% (average), since *ALLOC_HIGH* annotations forced additional data to the SPM. The modified benchmarks showed an average improvement of 4% and 13% for the *SPM* and the *SPM-A* configurations, respectively. Some benchmarks have had a significant improvement when more data is allocated using our framework.

Table I
BENCHMARKS MEMORY FOOTPRINT (ALL VALUES IN BYTES). STACK DEPTH AND HEAP DEPTH REPRESENTS THE MAXIMUM AMOUNT OF DATA ALLOCATED IN THE STACK AND THE HEAP, RESPECTIVELY.

		Original benchmarks			Modified benchmarks		
Name	**Code**	**Global**	**Stack**	**Heap**	**Global**	**Stack**	**Heap**
Dijkstra	26292	42444	120	1980	1652	120	42788
SHA	29124	3660	568	0	1612	180	2472
Susan-Smoothing	69376	1612	336	377791	1612	336	377791
Susan-Corners	67480	1612	384	454400	1612	384	454400
FFT	49832	1604	248	16440	1604	192	16496

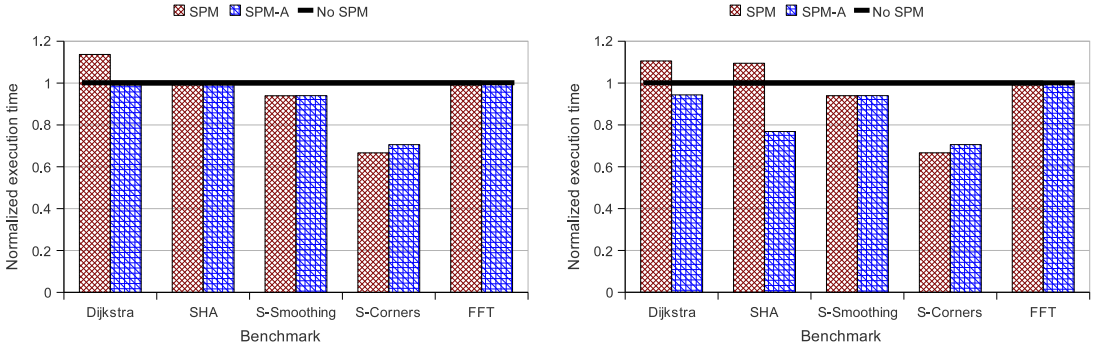

Figure 6. Original (a) and modified (b) benchmarks execution time comparison.

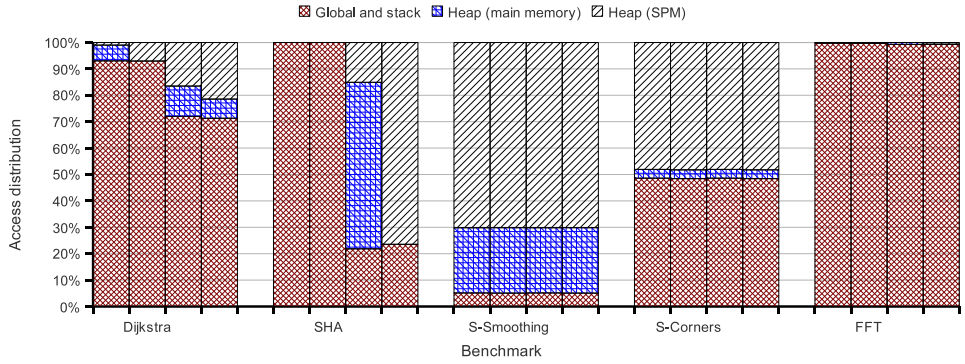

Figure 7. Distribution of memory operations on the different data types. Each bar represents a different benchmark configuration, from left to right: *SPM (original benchmark)*, *SPM-A (original benchmark)*, *SPM (modified benchmark)*, and *SPM-A (modified benchmark)*.

The evaluation of energy consumption was performed using memory models generated by CACTI [1]. The models for the cache and the SPM were generated for a $40nm$ technology, in order to match the one used on the Virtex 6 fabrication. For the external SDRAM, we extracted the parameters from the datasheet of the *Micron 512 MB MT4JSF6464HY-1G1* memory module used in the evaluation platform. Figure 8 shows the energy consumption of the memory hierarchy during the execution of the modified benchmarks. Average improvements of 20% (SPM) 34% (SPM-A) were achieved due to the power efficiency of the SPM when compared with the external SRAM memory.

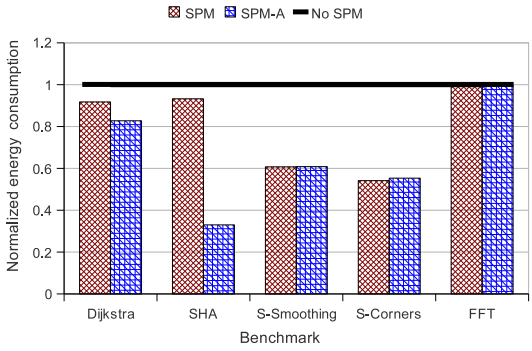

Figure 8. Modified benchmarks energy consumption comparison of the memory hierarchy.

We also compare our framework with *Dominguez et al* [5], in which the authors propose a compile-time approach for

handling heap data. In this work, the authors provide results showing the improvements of using their method for SPM placement versus placing data in DRAM. They report an average reduction of 39.9% in energy consumption. The comparison in figure 9 shows that our leveraging of the SPM benefits is comparable to a dynamic compile-time approach, but without the requirement of special compiler support and compile-time analysis (like the one proposed in *Dominguez et al*).

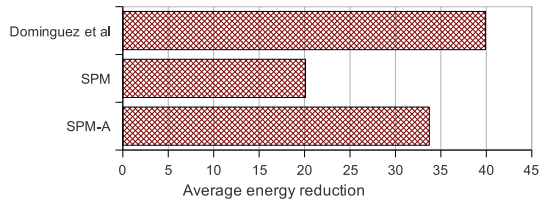

Figure 9. Average energy reduction. Comparison between our approach and *Dominguez et al*.

V. Conclusions

We have proposed a runtime operating system management approach for SPMs which do not require compiler support, profiling or hardware support. On the proposed approach, annotations, inserted into the code by the programmer, are used by the operating system to allocate the data in the most appropriate level of the memory hierarchy. The results have shown that, even by only handling user declared heap variables, we were able to improve the execution time and significantly decrease the total energy consumption. Yet simple, our solution yielded results comparable to a dynamic compile-time approach.

We also have shown how some kinds of global and stack variables allocated at compile-time can be easily converted to heap variables. However, this conversion was done by hand, and, depending on the application, it may not be feasible or natural. Nevertheless, our flexible software solution is clearly orthogonal to existing approaches, thus it can be easily used to handle heap data, while a different compile-time solution can be used handle instructions, global, and stack data. The efficient allocation of all instruction and data types in a heterogeneous memory hierarchy is a topic which will be covered in future works.

Acknowledgments

The authors deeply acknowledge the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper. This work was partially supported by the *Coordination for Improvement of Higher Level Personnel* (CAPES) grant, projects RH-TVD 006/2008 and 240/2008.

References

[1] Cacti, April 2011. http://www.hpl.hp.com/research/cacti/.

[2] Embedded Parallel Operating System, May 2011. http://epos.lisha.ufsc.br/.

[3] Mibench suite, April 2011. http://www.eecs.umich.edu/mibench/.

[4] Opencores, April 2011. http://opencores.org/.

[5] S. U. Angel Dominguez and R. Barua. Heap Data Allocation to Scratch-Pad Memory in Embedded Systems . *J. Embedded Comput.*, 1(4):521–540, 2005.

[6] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri. A post-compiler approach to scratchpad mapping of code. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 259–267, New York, NY, USA, 2004. ACM.

[7] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Trans. Embed. Comput. Syst.*, 1(1):6–26, 2002.

[8] D. Cho, S. Pasricha, I. Issenin, N. Dutt, M. Ahn, and Y. Paek. Adaptive Scratch Pad Memory Management for Dynamic Behavior of Multimedia Applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(4):554 –567, april 2009.

[9] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[10] A. A. Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, Aug.

[11] J. D. Hiser and J. W. Davidson. EMBARC: an efficient memory bank assignment algorithm for retargetable compilers. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 182–191, New York, NY, USA, 2004. ACM.

[12] M. Kandemir and A. Choudhary. Compiler-directed scratch pad memory hierarchy design and management. In *DAC '02: Proceedings of the 39th annual Design Automation Conference*, pages 628–633, New York, NY, USA, 2002. ACM.

[13] R. McIlroy, P. Dickman, and J. Sventek. Efficient dynamic heap allocation of scratch-pad memory. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 31–40, New York, NY, USA, 2008. ACM.

[14] A. Milidonis, V. Porpodas, N. Alachiotis, A. Kakarountas, H. Michail, G. Panagiotakopoulos, and C. Goutis. Low-power architecture with scratch-pad memory for accelerating embedded applications with run-time reuse. *IET Computers & Digital Techniques*, 3(1):109–123, 2009.

[15] O. Ozturk, G. Chen, M. Kandemir, and M. Karakoy. Compiler-Directed Variable Latency Aware SPM Management to CopeWith Timing Problems. In *CGO '07. International Symposium on Code Generation and Optimization, 2007*, pages 232 –243, 11-14 2007.

[16] A. Shrivastava, A. Kannan, and J. Lee. A Software-Only Solution to Use Scratch Pads for Stack Data. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(11):1719 –1727, nov. 2009.

[17] B. Stroustrup. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[18] S. Udayakumaran, A. Dominguez, and R. Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.*, 5(2):472–511, 2006.

[19] M. Verma and P. Marwedel. Overlay techniques for scratchpad memories in low power embedded processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(8):802–815, 2006.

[20] L. Wehmeyer and P. Marwedel. Influence of Memory Hierarchies on Predictability for Time Constrained Embedded Software. *Design, Automation and Test in Europe Conference and Exhibition*, 1:600–605, 2005.