A run-time memory management approach for scratch-pad-based embedded systems

Tiago Rogério Mück Antônio Augusto Fröhlich
Federal University of Santa Catarina (UFSC)
Laboratory for Software and Hardware Integration (LISHA)
{tiago, guto}@lisha.ufsc.br

Abstract

Software-controlled caches, often called scratch-pad memories (SPM), are being increasingly used due to their efficiency. And to exploit all the advantages of SPMs an efficient allocation must be done in software. In this work we propose a runtime operating system management approach for SPMs that do not require compiler support, application profiling or hardware support. The OS will use annotations, inserted into the code by the programmer, as hints to choose the most appropriate level in the memory hierarchy to allocate the data. The results showed that we were able to implement a run-time SPM allocation technique without adding any significant overhead to the system when compared with manual allocation.

1. Introduction

Contemporary embedded system applications are requiring even 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 [6], which shows that the memory subsystem is 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 an additional tag memory and address comparison logic which may significantly increase their power consumption and area cost, thus turning them inappropriate for most embedded applications [8].

An alternative to caches is the use of *software-controlled caches*, often called *scratch-pad memories* (SPM). The SPMs do not need the extra logic to map data and instructions to it because all of the memory allocation to the SPM is controlled by software, so SPMs are more power and area efficient than caches [8]. However, to exploit all the advantages of SPMs an efficient allocation must be done in software. Several software allocation approaches have been proposed. All of the proposed approaches are compile-based techniques, and most of them rely on profiling information to define the instructions and data to be allocated to the SPM at compile-time. These compile-based approaches have two big drawbacks. First, 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 to generate a profile [7]. This is especially true for dynamic applications where the memory access patterns depends on the application input data [4]. Second, this kind of allocation scheme leads to a more complicated software development and reduced portability because it requires very specific toolchains with modified compilers.

In this work we propose a runtime operating system management approach for SPMs that do not require compile support, profiling or hardware support. We assume that an embedded system targets a specific hardware platform and a specific application, so an embedded OS can receive information about the underlying memory hierarchy and about the kind of access to the application data in order to provide an efficient memory allocation. When an application requires a memory allocation, the OS will use annotations, inserted into the code by the programmer, as hints to choose the most appropriate level in the memory hierarchy to allocate the data. This way the OS can take advantage of the developer's knowledge about the application to provide an efficient data allocation without the drawbacks of the previously proposed techniques.

2. Related work

In the last years many SPM management approaches have been proposed. We separated this approaches in the different categories shown in figure 1.

The *static* approaches are those in which the contents of the SPM are fixed at compile-time and never change. They can be divided into two categories: *compile-time techniques* where the allocation is made by the compiler [3], and *post-compile techniques* where algorithms are applied at the final binary code to change the memory allocation [2]. These static allocation approaches either use greedy strategies to find an efficient solution, or model the problem as an *integer-linear programming* problem (ILP) to find an optimal solution.

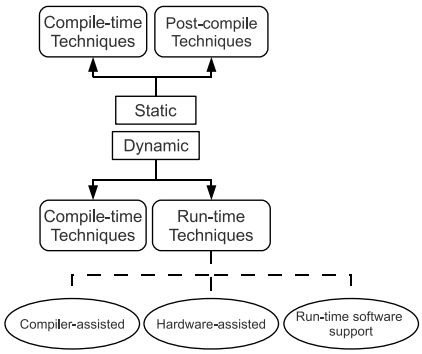

Figure 1. Classification of SPM allocation approaches

In the *dynamic* approaches the SPM contents change during the program execution. When the sets of code and/or data that will be moved to/from the SPM are defined at compile-time, we call it a *compile-time* approach [8]. *Verma et al* [8] is a good example of this kind of technique. The authors propose an overlay-based memory allocation approach for both code and data. The proposed solution uses ILP to find the optimal memory allocation and overlay points that minimizes energy consumption for a given profile. They also propose a first-fit algorithm to obtain a near-optimal solution.

If the data sets allocated in the SPM can change or are defined during the program execution, we call it a *run-time* approach. The run-time approaches generally follows one or more of the following three basic methodologies that we have defined: a run-time approach is *compiler-assisted* if actions are taken by the compiler to define the allocation. *Hardware-assisted* approaches are the ones that require special hardware support. Finally there are the approaches that require *run-time software support* provided by a standalone library or by an OS to perform the allocation. In *Shrivastava et al* [7] an approach is proposed that does not require profiling or hardware support. They manage the function's stack frames, allocating them on the SPM when they are in use, thus this approach handles only stack data which represents 65% of the memory access on multimedia applications, according to the authors. In *Cho et al* [4] an allocation scheme was proposed that integrates compiler, OS, and hardware. 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* [1]. This is a compiler-assisted approach that allocates fixed size parts of heap variables.

3. The proposed approach

In our approach we assume that an embedded system targets a specific hardware platform and a specific application, so an embedded OS receive information about the underlying memory hierarchy and about the kind of access to the application data in order to provide an efficient memory allocation. Each memory component is abstracted as a single dynamic heap. When an application requires a memory allocation, the OS 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.

We have implemented a framework for the C++ language that provides parameterized 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) [5]. EPOS relies on the *Application-Driven Embedded System Design Method* (ADESD) [5] to design and implement both software and hardware components that can be automatically adapted to fulfill the requirements of particular applications. Low overhead and high performance are achieved by a careful implementation that makes use of *generative programming* techniques, including *static metaprogramming*.

Our framework provides three different types of annotations:

- ALLOC_HIGH
- ALLOC_LOW
- 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 to indicate that the OS should decide the best place to allocate that object. The example in figure 2 shows how these annotations are passed to the *new* operators. In this example, four arrays of type *int* are allocated using the different annotations and without annotation. When annotations are not used, the OS assumes *ALLOC_NORMAL*. The usage of the *delete* operator doesn't change.

```
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 2. Examples of memory allocation requests using the priority annotation

Given an annotation, it may not be possible to allocate the data on the desired location. The pseudo-code in figure 3 describes the algorithm used to decide where the data will be allocated based on the annotation and the amount of space available on the different memories. If an allocation request can't be accomplished on the preferred memory component, the OS will attempt to allocate on a less optimal memory component. If it is left to the OS to decide where to allocate, the OS will attempt to allocate on the memory 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 will be deallocated on the main memory or on the SPM.

```
memory allocation (size, annotation)

    if annotation = ALLOC_HIGH
        if fits on spm (size)
            allocate on spm
        else
            allocate on main mem

    else if annotation = ALLOC_LOW
        if fits on main mem (size)
            allocate on main mem
        else
            allocate on spm

    else if 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 3. Algorithm that defines where the allocation will be done

4. Evaluation and results

We used the *Xilinx ML403 Evaluation Board* to build a platform for our preliminary evaluation. The ML403 board features a Xilinx Virtex 4 FPGA with an embedded PowerPC 405 processor that is implemented as a hard-core inside the FPGA (i.e. it is implemented directly in silicon instead of using FPGA logic resources). The ML403 board features an 64 Mb DDR SDRAM external memory that we connected to one of the *Processor Local Bus* (PLB) interface to use as our platform main memory. We implemented the SPM using the Virtex 4 internal block RAM connected to the *On-Chip Memory* (OCM) bus. Another block RAM peripheral is connected to the other PLB bus interface to store the bootloader software that loads the system into the main memory. This bootloader memory is idle after the system is loaded, so we used it as a SPM as well, although its performance may suffer since its PLB bus is shared with other IO peripherals. The PowerPC processor offers two internal 16Kb caches for instructions and data that can be used to cache any memory peripheral connected to the PLB buses. To keep a standard memory hierarchy, we connected only the external memory to the caches. The resulting memory hierarchy is shown in figure 4. Table 1 summarizes other parameters used on the experiments.

Table 1. Configuration parameters used on the experiments

Parameter	Value
Synthesis tool	ISE/EDK 10.1
Compiler	GCC 4.0.2
FPGA clock	100 MHz
Microprocessor clock	100 MHz

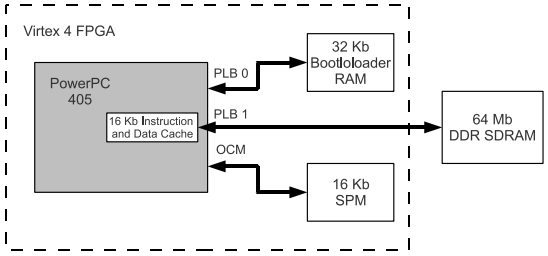

Figure 4. ML403 memory hierarchy

We have evaluated the performance of the *new* and *delete* operators for each different annotation and compared with the original allocation approach. The algorithm in figure 5 was used to evaluate these operations. With this algorithm we are able to measure the performance using different allocation patterns. We repeated these iterations for the original *new* and *delete* operations without using annotations and for each of the three possible annotations.

```
for (int i=0; i < 2 * BLK_SIZE; i++)
    char *p1 = new (ANNOTATION)
                   char[[ (i % BLOCK_SIZE) + 1];
    int *p2 = new (ANNOTATION) int;
    char *p3 = new (ANNOTATION) char[300];

    delete[] p1;
    delete[] p2;
    delete[] p3;
```

Figure 5. *new* and *delete* evaluation algorithm

Figure 6 shows the average time to complete a *new* and a *delete* operation. The results show that our new allocation approach added a negligible overhead to the original one. The higher allocation time when the *ALLOC_NORMAL* annotation is used is expected since extra logic is required to define the heap to be used.

We also measured the latency of read and write operations on the tree memory components available. Fig-

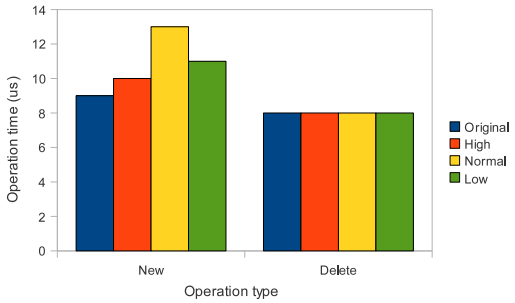

Figure 6. Performance of the original *new* and *delete* compared with the new parameterizable memory operations for different annotations

ure 7 shows the measured latency, which is the average time to read/write several blocks of 4096 words of 4 bytes. Surprisingly, the read/write times were very close for all memories.

Since the caches were disabled in this experiment, one explanation for the PLB based SPM is that it is connected to a bus that is shared with IO peripherals, which may have affected its performance. However, the other OCM based SPM is connected to a bus which is dedicated for on-chip memories and should have shown better performance. Another explanation is that the slow system clock frequency hides the higher latency of accessing an external DRAM memory. Nevertheless, it was not possible to achieve timing closure when synthesizing the system using higher clock frequencies.

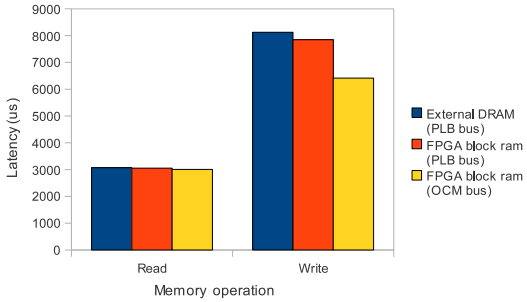

Figure 7. Average time to read/write a block of 4096 32-bit words in different memory devices

5. Conclusions and future work

We have proposed a runtime operating system management approach for SPMs that do not require compile support, profiling or hardware support. On the proposed approach, the OS will use annotations, inserted into the code by the programmer, as hints to choose the most appropriate level in the memory hierarchy to allocate the data. The results showed that we were able to implement a run-time SPM allocation technique without adding any significant overhead to the system compared with the standard allocation. However, real benchmarks must be used to determine the efficiency of the proposed approach in relation to a cache-based system and other approaches. An FPGA-based evaluation platform proved impractical. The results have shown the difficulties in achieving clock frequencies higher then the external memory devices. Also, FPGA-based platforms do not provide realistic energy consumption results, so a different platform or simulations based on energy consumption models are necessary.

In our approach we handle only user declared heap variables. Global variables allocated at compile-time can be easily converted to heap variables by declaring them in this way and allocating them manually during the application initialization. The automatic allocation of code and stack variables to the SPM at run-time without the help of a compiler or special hardware will be covered in future works.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] A. A. Fröhlich. *Application-Oriented Operating Systems*. PhD thesis, Technical University of Berlin, Berlin, 2001.

[6] 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.

[7] 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.

[8] 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.