Stack Frames Placement in Scratch-Pad Memory for Energy Reduction of Multi-task Applications

Gauthier, Lovic
System LSI Research Center,Kyushu University

Ishihara, Tohru
System LSI Research Center,Kyushu University

Takada, Hiroaki
Graduate School of Information Science, Nagoya University

出版情報：DAシンポジウム 2010 論文集, pp.171-176, 2010-08. 情報処理学会
バージョン：
権利関係：

Stack Frames Placement in Scratch-Pad Memory for Energy Reduction of Multi-task Applications*

LOVIC GAUTHIER[1], TOHRU ISHIHARA[1], AND HIROAKI TAKADA[2]

[1]System LSI Research Center,
3rd Floor, Institute of System LSI Design Industry, Fukuoka, 3-8-33 Momochihama,
Sawara-ku, Fukuoka 814-0001 JAPAN, Email: {lovic,ishihara}@slrc.kyushu-u.ac.jp

[2]Dept. of Information Engineering, Graduate School of Information Science,
Nagoya University, Furo-cho, Chikusa-ku, Nagoya 464-8603, JAPAN, Email hiro@ertl.jp

Abstract

Scratch-pad memories (SPM) are small on-chip memory devices whose access is much faster and consumes much less energy than off-chip memories. While SPM are usually too small for containing all the code or data of an application, significant energy consumption reductions can be achieved by assigning to them memory objects which are often accessed. The stack is one of the most frequently accessed data memory object, but its dynamic behavior makes it difficult to place into the SPM. This paper presents a simple and practical technique for placing frequently accessed parts of the stack into the SPM. The technique has been designed for multi-task environments where the SPM is shared among several tasks. Results show that the proposed technique achieves energy reductions which are comparable to ones obtain by other techniques supporting only single-task applications.

1 Introduction

Scratch-pad memories (SPM) are small on-chip memory devices. SPM are much smaller but also much faster and consume much less energy than off-chip memories. Accesses to SPM are done explicitly by the software as opposed to caches whose accesses are transparent to the software. This is why caches are often preferred for desktop or server applications. However, caches are poorly deterministic which favors SPM for real-time applications. Moreover, accesses to caches have longer delays and consume more energy than accesses to SPM. Hence, the latter are often preferred for embedded systems where energy constraints are tight.

When using the SPM for the data of a given task, the main idea is to place in it the memory objects (static variables, stack or heap) which are often accessed. While placing static variables into the SPM is not difficult since their size do not vary during the execution of the application, this is not the case for the stack nor for the heap and both require sophisticated techniques to benefit from the SPM. This get even more complicated with multi-task systems which share the SPM among several tasks because this requires to be cautious while changing the state of memory objects like a stack and because this reduces the energy consumption gain that can be expected at the task-level. Nonetheless, the stack is one of the most accessed memory object. For instance, with the applications of the MiBench benchmark suite [12], stack accesses represent about 60% of the data memory accesses.

This paper presents the implementation of a fully software technique which uses the SPM in order to reduce the energy consumption related to the stacks of multi-task applications. For each task, the technique manages three sub-stacks which are successively used while the program goes deeper into the call graph. The first and the third sub-stacks are located into the main memory (MM) while the middle one is located into the SPM and is meant to contain the frames which are the most frequently accessed. An integer linear programming (ILP) formulation is solved for selecting at compile time which sub-stack is to use at each function call in the program. This technique is simple enough to allow the operating system (OS) to efficiently share the SPM among the various memory objects of the tasks including their stacks. The management of the SPM by the OS is also taken into account into the formulation of the technique.

The rest of the paper is organised as follows: the next section presents some related works, section 3 explains how the stack and the multi-task management of the SPM are implemented in general then section 4 presents the stack management proposed in this paper. Section 5 gives an implementation for the proposed management before section 6 which presents some experimental results and section 7 which concludes the paper.

2 Related works

Several techniques exist for managing memory objects of single-task applications between the SPM and the MM in order to reduce the energy consumption or increase the speed. Among them, a majority are limited to static memory objects like [6, 7, 15, 14, 8]. A few works [1, 11, 4] do consider dynamic objects like the stack. In [3] we proposed a fully software technique for managing the stack between the SPM and the MM which supersets the techniques of [1, 4].

None of these papers studied the validity and the efficiency of their approach in the case of multi-task

*This work is supported by Toshiba and the CREST ULP program of JST.

applications. Moreover, only [3] discussed about implementation details but it still assumed that important modifications of the code will not significantly increase the resulting energy consumption (beside the energy taken into account for managing the stack) nor change the actual size of the stack's frames. Those weaknesses are specially addressed by the technique proposed here.

Several techniques also exist for sharing the SPM space among several tasks for their static memory objects. The usual ideas are to either share the SPM spatially, with the drawback of having less SPM space available for each task, or temporally, which provides the totality of the SPM space to each task but requires to update its content at context switches. Papers like [13, 9] studied these approaches and proposed more efficient hybrid spatial and temporal techniques. In this paper, we are using the hybrid technique proposed by [9].

3 Context

3.1 Default stack management

The stack is organised as a set of frames. Each function has its own frame for placing in it its local variables which could not be assigned to registers. Figure 1 shows how the stack is usually managed. In the figure, the content of the stack is shown evolving with a succession of functions, f_1 calling f_2 then f_2 returning to f_1. At the very beginning of each function a new frame is allocated on top of the stack by decreasing the stack pointer register (SP). Reciprocally, at the very end of each function, its frame is destroyed by subtracting its size from SP.

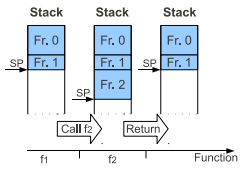
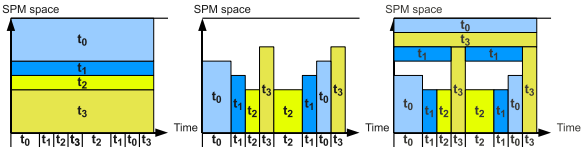


Figure 1: Standard evolution of a stack

For a majority of the processors, a function accesses its frame with register-relative addressing. Several registers can be used for these accesses, but the computations of their values are always rooted to SP which points to the start of the current frame. Additionally, when a function f calls another function g whose number of arguments is large so that some of them could not be assigned to registers, these extra arguments are conventionally stored into the frame of f. They are then accessed by g relatively to its *own* frame.

3.2 Multi-task management of the SPM

As introduced in section 2, the SPM can be shared among several tasks spatially, temporally or with an hybrid mix of both dimensions. Figure 2 illustrates these sharing approaches as presented in [9]. In the figure, the content of the SPM is shown evolving with the succession of the active tasks. In figure 2c, the upper part of the SPM is spatially shared, whereas the lower part is temporally shared. When the SPM or a part of it is temporally shared, the concerned memory objects need to be saved to the MM when their task is preempted. They are eventually restored when their task returns to execution. The hybrid technique of [9] is actually



(a) Space (b) Time (c) Hybrid

Figure 2: SPM sharing techniques

more sophisticated: it allows tasks of high priority to steal some spatially shared parts of the SPM from lower priority tasks as in can be seen in the figure for task t_3.

4 The proposed stack management

The technique proposed in this paper is based on two observations. First, the functions close to the root of the call graph of a task are not likely to be executed for a long time compared to the other functions. Therefore, their frames are not likely to be often accessed. Second, it is common that the frames of some functions which are close to the leaves cannot be put into the SPM. This is the case for recursive functions and for some library functions as it will be explained in section 4.1. Consequently, the functions whose frames worth to be placed into the SPM are mostly the ones located near the middle of the call graph.

In order to take advantage of this characteristic the simplest possible way, the stack of each task is split into three sub-stacks. The first one is located into the MM and contains the frames of the functions close to the root of the call graph, the second one is located into the SPM and contains the frames of the functions close to the middle of the call graph and the third one is located into the MM again and contains the frames of the functions close to the leaves of the call graph. In this paper, these three sub-stacks are called respectively *MM low*, *SPM* and *MM high*. With this approach, the only difference with the standard stack management is that SP is to be translated among the three sub-stacks just before calling some functions and just after returning from them. This translation is performed with four operations:

warp(SPM): translates SP to the start of the SPM sub-stack.

warp(MM high): translates SP to the start of the MM high sub-stack.

unwarp(SPM): translates SP back to the top of the SPM sub-stack.

unwarp(MM low): translates SP back to the top of the MM low sub-stack.

Warp operations are used for translating SP to the next sub-stack and unwarp operations are used for the reverse translations. Since a frame cannot move, when a warp operation is inserted the opposite unwarp operation must be inserted after the same call. Figure 3 shows how these operations work for managing the stack. In the figure, the content of the sub-stacks is

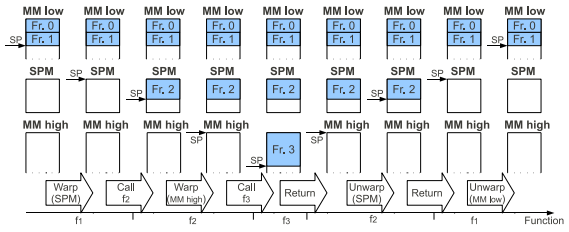


Figure 3: Evolution of the three sub-stacks

shown evolving with a succession of functions, f_1 calling f_2 which then calls f_3 before returning to f_2 then f_1. The frames of f_1, f_2 and f_3 are respectively put into the MM low, the SPM and the MM high sub-stacks, warp and unwarp operations being inserted accordingly.

The decision to insert or not the operations for each call of a program is taken from the resolution of an ILP whose objective to minimize models the stack-related energy consumption. The formulation is constrained so that the state of the sub-stacks is valid, i.e., the size of the SPM sub-stack must not exceed the space reserved into the SPM for it and the transition order between the sub-stacks must be from MM low to SPM and finally to MM high when going from the root to the leaves of the call graph.

For the multi-task point of view, it is considered in this paper that the SPM sub-stacks are put into the temporally shared part of the SPM since their $access\ number/size$ ratio is high. The costs of saving and restoring the SPM sub-stacks at context switches can then be taken into account in the objective of the stack management as a function of the current sizes of the SPM sub-stacks.

4.1 Difficulties

Stack-related difficulties. The stack management we propose in this paper and the other fully software ones proposed in [1, 11, 4, 3], require to modify the code for allocating some of the frames to the SPM and accessing them. This can only be done at assembly level since the stack is abstracted in higher level languages like C. In conventional programs, the stack is localized with SP. Yet, this does not forbid to access it relatively to other registers provided the computation of their value takes root from SP. Therefore, if translation operations are inserted in arbitrary places of the code, the value of several registers, and also some memory contents (e.g., in case of spill code) have to be updated at the same time. This requires both deep data dependency analysis to identify the places to update and important assembly code modifications. Both are complex to carry out, but more importantly, the energy cost of the code modifications can exceed the gain achieved by using the SPM. Moreover, such modifications are likely to change the size of the frames which could invalidate the inserted stack operations. Thus, the technique proposed in this paper has been defined so that such complex analysis and code modifications were not necessary. Indeed, warp and unwarp operations are inserted just before and just after call instructions that is to say in places where the sole reference to the current frame is SP. References to a frame are safe too since the computation of their address takes root from the stack pointer register. That would not be the case for a method which moves frames or part of them like [3] does. With this latter approach, such references which are passed as argument to another function can become invalid if the referenced frame is moved to a different memory.

Extra arguments which are present in some frames can jeopardize the validity of the stack management too. This is because a function using such arguments will access them into its caller's frame but relatively to the address of its own frame. This is incorrect if both frames are not contiguous. To avoid such an invalid case, the frame of a function using extra arguments must forced to be contiguous with the frame of the caller.

Another difficulty appears when a function f is called from several points into the program: it can happen that its frame is assigned to different memories (SPM or MM) depending on where f has been called from. This is not a problem as the stack operations are inserted into the calling function. But it becomes problematic if f calls another function g. Once again, g's frame might be assigned to different memories, but depending on where f (and not only g) has been called from. Such cases can be avoided by considering that the corresponding calls are identical when selecting at compile time the memory for each frame.

The same kind of problem arises with library functions and recursions. Library functions are often called from a large number of different points. Moreover, it is often impossible for the user to modify their code. A solution is to consider that each call to one of them is actually a call to a large monolithic function whose frame size is the sum of the sizes of its own frame and all the frames of the functions that are subsequently called. This size can be obtained from the specification of the library or from exhaustive profiling information, but if it cannot be ascertained, library functions' frames must be assigned to the MM. For the case of recursive functions, the same method can be employed but it is often impossible to bound the depth of recursion. [4, 3] actually proposed better solutions using circular buffers of frames inside the SPM in place of the monolithic frames proposed in this paper, but it does not work if references to frames are passed to functions as arguments.

Multi-task-related difficulties. A first difficulty is that a context switch which update the content of the SPM (including the parts dedicated to the stacks) can happen anytime during the execution of a task. Therefore, if we want to merge efficiently the stack management of the SPM into the sharing of this memory among several task, this management must be always consistent and known by the OS so that it can safely save and restore the only necessary number of bytes of the SPM space dedicated to a stack. Both the approach presented in the paper and the one of [3] manage the SPM like a real stack. Hence, the OS only needs to know the top address of the current SPM stack (the bottom address is fixed at compile time). The consistency of the stacks' state is also guarantied for the approach proposed in this paper because the

implementation of the warp and unwarp operations ensures that this state is always consistent as it will be shown in section 5.1. The consistency for the technique of [3] would be however guarantied only if the operations moving the frames are either consistent at each instruction of their implementation or non-interruptible.

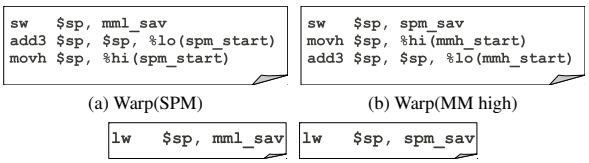
Finally, additional difficulties regard the access rights of the tasks. Typically, it is often forbidden for a task to modify directly an internal variable of the OS. Instead, expensive system calls are to be used. This strongly limits the possibility for informing at low cost the OS about the state of the stack management even with the case where the SPM is managed like an additional stack. In the proposed implementation, whole the necessary information can be found in SP so that no additional task-OS communication is required.

5 Implementation of the stack management

5.1 Implementation code

Warp and unwarp operations' code. Either operation translates SP from one sub-stack to another, but their implementations differ since the warps translate this pointer from a sub-stack containing frames to an empty one and the unwarps does it from an empty sub-stack to one with contents. Consequently, the warp operations require to save SP before updating it but it is enough for the unwarp operations to restore it for returning to the top of the former sub-stack. In our implementation, the top addresses of the MM low and the SPM sub-stacks are saved into two small areas in the SPM. If the system uses a 32-bit addressing range, the required space is 4 bytes for each that is to say 8 bytes in total.

If the processor includes instructions with immediate addressing, either warp and unwarp operations can be implemented without using any other register than SP. Figure 4 gives the implementation of these operations for the MeP [10] processor. In the figure, for each warp



(a) Warp(SPM) (b) Warp(MM high)

(c) Unwarp (MM low) (d) Unwarp (SPM)

Figure 4: Implementation for the MeP processor of the warp and unwarp operations

operation, the first instruction saves SP to the save area for the current sub-stack's top address (spm_sav and mml_sav). The two next instructions set SP with the address of the next sub-stack's start address ($start\_spm$ and $start\_mmh$). For the operation which warps to the SPM sub-stack, the least significant bits of SP are set first whereas for the warp to MM high, the most significant bits are set first. This way, when SP points to the SPM, its value is always consistent. Unwarp operations only need one instruction which loads SP from the save area for the corresponding sub-stack's top.

Handling of the stack state by the OS. When a task is preempted, all its objects which are in the time-shared part of the SPM must be copied to the MM. While with static memory objects it is enough for the OS to check SPM allocation tables built at compile time, doing the same with a SPM sub-stack area which is not full will induce copies of non-utilized space. This overhead can be avoided by simply using the value of SP. First it is checked which sub-stack SP is pointing at. This is done by looking at the most significant bits of this register in order to keep the consistency with the warp operations. When it points to the MM low sub-stack, the content of the SPM sub-stack is not to be saved at all. When SP points to the SPM sub-stack, its content is to be saved from this register. Finally, when SP points to the MM high sub-stack, the totality of the SPM sub-stack area is to save. With the last case, several bytes might be saved while not being used. Even so, they are not numerous in practice and this approximation allows the technique to work without requiring the tasks to modify any internal variable of the OS.

5.2 ILP for optimal frame placement

The variables of the formulation are there to control the insertion of the stack operations. These variables are indexed with i for the frames and j for the call instructions and are the followings:

$x_{i,j}$: this binary variable is 1 when frame i is into the SPM sub-stack for call j;
$y_{i,j}$: this binary variable is 1 when frame i is not into the MM low sub-stack for call j;
$z_{i,j}$: this binary variable is 1 when frame i is into the MM high sub-stack for call j[1];
w_j : this binary variable is 1 when warp and unwarp operations are inserted around call j.

The parameters used for the formulation include characteristics of the processor, metrics extracted from the application code and from profiling information. They are represented by the following constants:

S_{stk} : it is the maximum size of the SPM sub-stack;
$Cspm_{i,j}$: it is the energy cost of the total accesses to frame i if it is in the SPM during the executions of the function called from j;
$Cmm_{i,j}$: it is the energy cost of the total accesses to frame i if it is in the MM during the executions of the function called from j;
Cw : it is the energy cost of one warp plus one unwarp;
S_i : it is the size of the frames of function i;
$Ncxt_{i,j}$: it is the total number of context switches which happen when frame i, j is allocated;
$Ccxt$: it is the energy cost for saving and restoring a byte of the SPM during a context switch.

The objective function models the energy consumption related to the stack. It is made of three parts: the first one, noted obj_{stk}, represents the energy consumed while accessing the stack, the second one, noted obj_{op}, represents the energy consumed by the stack operations

[1] By definition, $z_{i,j}$ is necessarily inferior to $y_{i,j}$.

inserted into the assembly code and the last part, noted obj_{cxt} represents the energy consumed during the context switches for saving and restoring the SPM sub-stack.

The first part of the objective function is the sum of the energy consumed when accessing each frame. For a frame, the corresponding energy depends on whether it is assigned to the MM or to the SPM. Therefore obj_{stk} is computed using the x variables as follows:

$$obj_{stk} = \sum_{i,j} \Big((Cspm_{i,j} - Cmm_{i,j}) * x_{i,j} \Big) \quad (1)$$

In the above equation only the difference in energy consumption between the accesses to the MM and the SPM is actually represented. The x variables are computed from the y and z using their definition presented earlier:

$$\forall i,j \qquad x_{i,j} \leq y_{i,j} \quad (2)$$
$$x_{i,j} \leq 1 - z_{i,j} \quad (3)$$
$$x_{i,j} \geq y_{i,j} - z_{i,j} \quad (4)$$

Equations (2) and (3) ensure that $x_{i,j}$ is 0 when the current sub-stack is respectively MM low and MM high and equation (4) ensures that this variable is 1 when the current sub-stack is in the SPM. For preventing a frame to be moved, the value of the corresponding y and z variables need to be fixed as long as the frame exists. This is done as follows where $Called(j)$ is the index of the function called by j:

$$\forall i,j \qquad y_{i,j} = y_{i,Called(j)} \quad (5)$$
$$z_{i,j} = z_{i,Called(j)} \quad (6)$$

Finally, the definition of both y and z variables requires the following constraint:

$$\forall i,j \qquad y_{i,j} \geq z_{i,j} \quad (7)$$

The second part of the objective function is the sum of the energy consumed by each inserted warp and unwarp operation. These operations are inserted on both sides of a call instruction anytime the sub-stack of the coming frame is different from the current one. Hence, obj_{op} can be formulated as follows:

$$obj_{op} = \sum_{j} Cw * w_j \quad (8)$$

Variable w_j is 1 if the frame of the function called by j is in a different sub-stack from the current one. It can be computed by comparing the corresponding x variables as follows where $Current(j)$ is the index of the current frame before call j and $Frame(j)$ is the index of the frame which is allocated at call j:

$$\forall j \qquad w_j \geq x_{Current(j),j} - x_{Frame(j),j} \quad (9)$$
$$w_j \geq x_{Frame(j),j} - x_{Current(j),j} \quad (10)$$

The third part of the objective function is the sum of the energy consumed during each context switch for saving and restoring an SPM sub-stack. The x variables are used for computing the size of the SPM sub-stack at each context switch and the formulation of obj_{cxt} is then the following:

$$obj_{cxt} = \sum_{i,j} \Big((Ccxt * Ncxt_{i,j}) * S_i * x_{i,j} \Big) \quad (11)$$

Any value are not possible for the x variables since the size of the SPM sub-stack is bounded. This is constrained as follows:

$$\forall j \qquad \sum_{i} S_i * x_{i,j} \leq S_{stk} \quad (12)$$

Finally, the frames which include the arguments of a called function must be contiguous to the frame of this function. Furthermore, the frames of some recursive or library functions cannot be put into the SPM. For both cases correspond the respective predicates Arg_j and Out_j which are used as follows:

$$\forall j \qquad Arg_j \Rightarrow x_{Current(j),j} = x_{Frame(j),j} \quad (13)$$
$$\forall j \qquad Out_j \Rightarrow x_{Current(j),j} = 0 \quad (14)$$

6 Experiments

6.1 Experimental environment

We applied our technique on a MeP [10] processor configuration including an 8kb data SPM. We used the Toshiba's MeP Integrator (MPI) tool chain [5, 10] for compiling and simulating the applications. Energy characteristics of this architecture are given in table 1. Compilations were performed with the $-O2$ level of optimization. Executions were performed for a static priority-based rate monotonic preemptive system with a processor utilization of about 60%.

Memory access type	Energy (nJ per word)
SPM/data read	0.3774
SPM/data write	0.5053
SDRAM/data read	42.6466
SDRAM/data write	18.3986

Table 1: Energy consumptions of memory accesses for a MeP processor

Both the multi-task and stack techniques where applied on the tasks of the sets given in table 2. Tasks were taken from the EEMBC [2] and the Mibench [12] benchmark suites. The size dedicated to the SPM sub-stacks have been selected for maximizing the $energy\ gain/size$ ratio for the stack accesses.

Set	Tasks
Set A	aes, des, md5, cubic, fft, rad2deg, mpeg, mp3, patricia
Set B	aes, des, md5, cubic, fft, rad2deg, patricia
Set C	aes, des, md5, cubic, fft
Set D	cubic, fft, rad2deg

Table 2: Tasks-set used for the experiments

6.2 Results

The results of the stack management technique proposed in this paper are shown in figure 5. The technique, noted *our*, is compared with the stack management technique of [1] noted *free*. Additionally, *cxt* gives the results where the context switch costs are taken into account when optimizing with our method for the case of set A. The results are given for each application with SPM sub-stack sizes which are 1/3, 2/3 and the totality of the stack memory usage. On average, for the

respective 1/3 and 2/3 sizes, the *free* approach achieves about 22% and 78% of stack-related energy reduction, *our* achieves quasi identical results (differences are not visible in the averaged figures) which shows that the heuristic with three sub-stacks is efficient. Finally, *cxt* achieves respectively 22% and 77% which is only slightly less than the other cases. The results of the patricia application are not good because it contains recursive functions which were not handled by any of the stack management techniques used here.

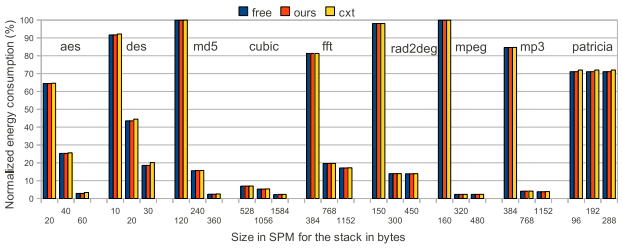


Figure 5: Stack-related energy consumption normalized to the case where the stack is fully kept into the MM

Figure 6 shows how effective our proposed management is when merged into a technique which shares the SPM among several tasks. In the figure, *static* shows the data memory access-related energy consumption when sharing the SPM among several tasks for the technique presented in [9] while *ours* shows the results of the same technique but merged with our stack management approach. For fairness of the comparison, we considered that for *static*, the stack of each task were a static object whose size is its maximum memory requirement. It can therefore be put into the SPM without any specific management provided it is not too large. It can be seen

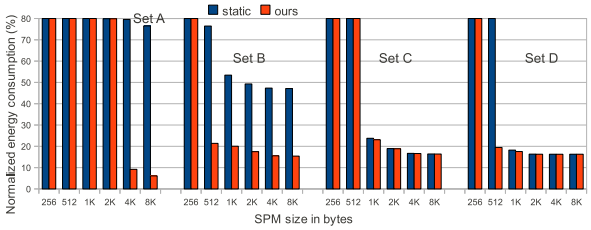


Figure 6: Data memory access-related energy consumption normalized to the case where the SPM is not used

in the figure that the proposed merged approach achives often much better results than *static*. On average, *static* achives about 56% of energy reduction and *ours* achieves about 65% for the case of a 1Kb SPM.

7 Conclusion

This paper presented a technique for reducing the energy consumption of the accesses to the stacks of tasks in a multi-task embedded system. The technique has been carefully implemented in order to be efficiently supported by an OS which shares the SPM among several tasks. It also takes into account the cost of this sharing while determining, at compile time which frame is to allocate to which memory.

Experimental results show that even though simple, the technique achieves results comparable to a more refined technique which is limited to single-task applications. When taking into account the multi-task, the technique achieves about 65% of data access-related energy reduction on average with a 1Kb SPM.

As future work we plan to enhance the technique with finer controls of the frames, like allowing to move them, while still keeping a reasonable feasibility of its implementation and the compatibility with an efficient OS-level sharing of the SPM.

References

[1] 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.

[2] Embedded Microprocessor Benchmark Consortium. EEMBC benchmark suite. `http://www.eembc.org/home.php`.

[3] L. Gauthier and T. Ishihara. Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems. In *ESTIMedia '9*, Grenoble, France, 2009.

[4] A. Kannan, A. Shrivastava, A. Pabalkar, and J.-e. Lee. A software solution for dynamic stack management on scratch pad memory. In *ASP-DAC '09*, pages 612–617, Piscataway, NJ, USA, 2009. IEEE Press.

[5] A. Mizuno, H. Uetani, and H. Eichel. Design methodology and system for a configurable media embedded processor extensible to vliw architecture. In *ICCD '02*, page 2, Washington, DC, USA, 2002. IEEE Computer Society.

[6] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *EDTC '97*, page 7, Washington, DC, USA, 1997. IEEE Computer Society.

[7] J. Sjödin and C. von Platen. Storage allocation for embedded processors. In *CASES '01*, pages 15–23, New York, NY, USA, 2001. ACM.

[8] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *DATE '02*, page 409, Washington, DC, USA, 2002. IEEE Computer Society.

[9] H. Takase, H. Tomiyama, and H. Takada. Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems. In *DATE '10*, Washington, DC, USA, 2010. IEEE Computer Society.

[10] Toshiba. MeP processor. `http://www.semicon.toshiba.co.jp/eng/product/micro/mep/document/index.html`.

[11] 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.

[12] University of Michigan. MiBench benchmark suite. `http://www.eecs.umich.edu/mibench/`.

[13] M. Verma, K. Petzold, L. Wehmeyer, H. Falk, and P. Marwedel. Scratchpad sharing strategies for multiprocess embedded systems: a first approach. In *Estimedia '05*, volume 0, pages 115–120, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[14] M. Verma, S. Steinke, and P. Marwedel. Data partitioning for maximal scratchpad usage. In *ASP-DAC '03*, pages 77–83, New York, NY, USA, 2003. ACM.

[15] L. Wehmeyer, U. Helmig, and P. Marwedel. Compiler-optimized usage of partitioned memories. In *WMPI '04*, pages 114–120, New York, NY, USA, 2004. ACM.