Optimal Stack Frame Placement and Transfer for Energy Reduction Targeting Embedded Processors with Scratch-Pad Memories

Gauthier, Lovic
System LSI Research Center, Kyushu University

Ishihara, Tohru
System LSI Research Center, Kyushu University

https://hdl.handle.net/2324/18604

KYUSHU UNIVERSITY

Optimal Stack Frame Placement and Transfer for Energy Reduction Targeting Embedded Processors with Scratch-Pad Memories

Lovic Gauthier and Tohru Ishihara
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

Abstract—**Memory accesses are a major cause of energy consumption for embedded systems and the stack is a frequent target for data accesses. This paper presents a fully software technique which aims at reducing the energy consumption related to the stack by allocating and transferring frames or part of frames between a scratch-pad memory and the main memory. The technique utilizes an integer linear formulation of the problem in order to find at compile time the optimal management for the frames. The technique is also extended to integrate existing methods which deal with static memory objects and others which deal with recursive functions. Experimental results show that our technique effectively exploits an available scratch-pad memory space which is only one half of what the stack requires to reduce the stack-related energy consumption by more than 90% for several applications and on an average of 84% compared to the case where all the frames of the stack are placed into the main memory.**

I. INTRODUCTION

The most commonly used memory devices in electronic systems are static random access memories (SRAM) and dynamic random access memories (DRAM). Off-chip DRAMs are cheap in area but slow and very energy consuming while on-chip SRAMs are expensive in area but fast and far less energy consuming. Hence, their respective advantages and drawbacks are usually balanced by using both a large DRAM for storing data and a small SRAM for reducing the cost of memory accesses. The management of the data between the DRAM and the SRAM can be made in hardware if the SRAM is used for a cache device, or in software if the SRAM is used for a scratch-pad memory (SPM).

Caches are the preferred solution for desktop systems as they are transparent to the software. Yet, caches are poorly deterministic and are significantly larger in both area and energy consumption than SPMs [1]–[3]. Hence, SPMs tend to be favored for embedded systems even though it is required to modify the application code in order to exploit these memories.

The main idea for using an SPM in order to reduce energy consumption or execution time is to place in it frequently accessed data or code. Code and static data memory objects are fixed at compile time and can be easily allocated to the SPM during this stage. On the contrary, dynamic memory objects, which include stack and heap data, are more difficult to place

within the SPM since their allocation state evolves at run time. Dynamic memory objects are however important to treat, especially the stack whose data are often the most frequently accessed ones. For instance, stack accesses represent about 64% of the total data accesses on average for the MiBench [4] benchmark.

This paper presents a fully software technique which aims at reducing the energy consumption regarding the memory accesses to the stack by allocating to the SPM the stack frames which are known to be often accessed. When the SPM is full, the technique allows to move a part or the totality of a frame to the main memory (MM). Moved frames can then be moved back to the SPM if further numerous accesses are to be performed. The paper defines a set of operations to insert into the code in order to manage dynamically the frames between both memories. The optimal use of these operations is formulated as an integer linear programming (ILP) problem. Constants used in the formulation are either extracted from the application code (e.g., size of frames) and the target architecture configuration (e.g., size of the SPM) or estimated from profile informations (e.g., stack access rates).

The paper is organized as follows: the next section gives a motivating example then section III presents some related works. Section IV explains our management of the SPM for the stack, then section V describes the ILP formulation of the stack energy optimization and section VI gives some extensions of the technique for global variables and recursive functions. Finally, section VII presents some experimental results and the last section concludes the paper.

II. MOTIVATING EXAMPLE

Let us assume the example shown in figure 1: function f whose frame size is 40 bytes calls g, a function whose frame size is 28 bytes. f accesses its frame 123 times for reading and 201 times for writing before calling g which accesses 110 times its frame for reading and 118 times for writing. When returned, f accesses its frame 7 times for reading and 4 times for writing.

Assuming the power characteristics given in table I (sources [5], [6]), the energy consumption for accessing the

```
int f(int a) {
  ...
  r = g(i+5);
  ...
}

int g(int a) {
  ...
}
```

Fig. 1.   Motivating example code

stack will be about 1345nJ if both frames are allocated to the MM and about 82nJ if they are allocated to the SPM (accesses to the MM are supposed to be burst for this example).

Memory access type	Energy (nJ per word)
SRAM 4kb	0.145
SDRAM/read random	11.747
SDRAM/write random	10.397
SDRAM/burst read	3.373
SDRAM/burst write	1.659

TABLE I
ENERGY CONSUMPTIONS OF SRAM AND SDRAM ACCESSES

Now if we assume that only 64 bytes are available in the SPM during the execution of f, then both frames cannot be simultaneously in the SPM. If we choose to put only f's frame into the SPM then the energy consumption for accessing the stack will be about 615nJ and it will be about 812nJ if it is g's frame which is put into the SPM.

Yet, in the common case where f's frame is not accessed while g is executed, it can be moved from the SPM to the MM just before g is called. In this paper this transfer of a frame from the SPM to the MM is called a *store*. After returning from g the frame of f can either be left in the MM or moved back to the SPM; this operation is called a *load* in this paper. Both operations have a cost: based on table I and assuming burst transfers, the respective energy consumptions of the store and the load are about 72nJ and 141nJ. Therefore, when the store only is applied, the total energy consumption regarding the stack will be about 173nJ, and it will be about 294nJ if the load is also applied.

Finally, it is enough to store 4 bytes of f's frame in order for g's frame to fit in the SPM. With this partial store operation and the corresponding partial load operation after returning from g the energy consumption drops to about 103nJ. If the last load is not performed (assuming f will not access the part which have been stored), the energy consumption decreases to about 101nJ.

III. RELATED WORKS

Optimizing the usage of the limited space of the SPM has been a subject of research for several years. Some approaches like [1], [3], [7]–[10] are purely static: memory objects are allocated within the MM and the SPM at compile time and cannot be moved at run time. These methods are simple enough to allow ILP formulations like the ones given by [1], [3], [8]–[10]. However, static approaches are also quickly limited by the size of the SPM. They are opposed to dynamic approaches which do allow data to be transfered from one

memory to the other at run time. Dynamic approaches can utilize additional hardware features like an MMU [11] or be fully software like [12], [13]. Such approaches can support any kind of memory objects but suffer from significant overhead for the dynamic management.

A few research works consider specifically the stack. Some of them utilize new or existing hardware extensions in order to reduce energy consumption while accessing the stack. For instance, [14] remarked that an important part of the stack accesses consisted of saving and restoring registers and return addresses when calling and returning from a function. Hence, they proposed to add a separate small memory for these data. Without adding any hardware, [15] assumes the existence of an MMU and uses a paging system in order to manage the stack between the SPM and the MM.

Fully software methods also exist for optimizing accesses to the stack, they can be static or dynamic, and they can utilize two levels of granularity: the frame level where the frames of the stack are treated as monolithic blocks and the variable level where the variables of the frames are treated independently of each other. [10] is a technique which utilizes an ILP formulation for optimizing the placement of static and stack data between the SPM and the MM. For the stack data, they consider both frame-level and variable-level optimizations and their conclusion is in favor of a hybrid approach using both the frame and the variable levels of granularity in order to balance the lack of precision of the first with the larger overhead of the second. The technique proposed by [16], [17] is dynamic: it inserts transfer points into the program where stack or global variables can be moved from the MM to the SPM or evicted back. They use profile information to determine which and when variables should be in the SPM. While this paper does not mention energy consumption, a second publication [18] does extend their work with this consideration. Compared to our work, they consider stack variables instead of stack frames but only use a non-optimal greedy algorithm.

The previous software approaches did not treat the case of recursive functions, their corresponding frames are simply left in the stack of the MM. Approaches supporting recursive functions include [19] and [20]. [19] proposes to use a circular buffer for managing frames within the SPM. When the SPM is full, the first frame (the oldest one) is moved to the MM and is reloaded when the program is back to its corresponding function. This technique naturally supports recursive functions, is simple, but is not optimal as the oldest frame eviction is systematic when the SPM is full even if the cost is larger than the gain. The technique of [20] consists of looking for the depths of recursion where there are numerous accesses to the stack using profiling informations. Frames corresponding to such depths are stored into the SPM whereas the others are stored into the MM. It is necessary for this approach to add a depth counter to the recursive functions' code in order to dynamically determine where the frames should be allocated. This second approach makes the strong assumption that depths where frames have numerous accesses can be identified statically which highly depends on the application

code and inputs.

IV. Managing the stack

A. Using the SPM for the stack

A usual approach for using the SPM in order to reduce the energy consumption related to the stack is to put its frequently accessed parts into the SPM while keeping the rest in the MM. Figure 2 illustrates this scheme: frames 0, 1, 2 and 3 are successively allocated in this order. First frame 0 then frame 1 are allocated to the stack of the MM, then frame 2 is allocated to the SPM and finally frame 3 is allocated again to the MM.
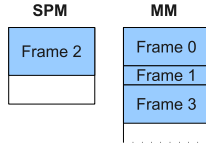

Fig. 2.   Frames in the SPM and the MM

As seen in the related works of section III, this general approach can be implemented with static or dynamic allocations and with a frame or a variable level of granularity.

With static approaches, the locations of stack data are fixed at compile time and cannot be changed at run time, while dynamic approaches allow to change them at run time. Even if static approaches can take into account the limited liveness of stack data as [10] does, dynamic approaches which can move stack data at run time can use the SPM space more efficiently.

The variable level of granularity is more precise than the frame level one hence can require a smaller SPM size for achieving the same energy reduction. However, this advantage is limited by several issues: first, there are several cases where variable-level techniques cannot be applied while frame-level ones are still usable. This happens for the frequent cases where accesses to a variable cannot be ascertained or where a function code (a library function for instance) cannot be modified. Second, access overheads are not the same: with the frame level, stack variables can be accessed traditionally, i.e., relative to the stack pointer, whereas the variable level requires either accesses relative to immediate addresses or to translate the stack pointer every time a stack variable is in a different memory. These two variable-level access modes induce important overheads.

B. The proposed technique

Overview: the management technique proposed in this paper is dynamic, in the sense that frames' allocation state can change at run time, and it works at the frame level of granularity. It follows the usual approach of trying to put frequently accessed frames into the SPM while keeping the rest in the MM as described in the previous section and illustrated by figure 2.

Specificities: the main idea of the technique is to move the stack pointer between the MM and the SPM depending on the allocation of the frame of currently executed function. This is based on the observation that a new frame is allocated by simply decreasing the stack pointer when entering a function so that it is enough to translate this pointer just before calling the function to change the memory which will contain the frame. When the called function returns, it is enough again to translate the stack pointer back to its initial memory. While this approach does require a frame level of granularity, it has a very low overhead (only one register to translate before and after calling a function and only in the case where its frame is in the other memory).

The second idea is to allow to move frames or part of frames from the SPM to the MM or from the MM to the SPM. As mentioned in section II these operations are respectively called *store* and *load*. The store operation is used to free memory from the SPM, whereas the load operation restores the previous state of the frame. It can be possible to insert load and store operations anywhere into the code but this is not pertinent: their goal is to manage the free space in the SPM when frames are allocated or deallocated. It only happens when entering or leaving a function so that it is enough to insert stores and loads respectively just before and just after call instructions.

Now, there are several issues to address so that such operations keep the consistency between the memory organization and the application code.

Frames consistency: as a frame can be partially stored or loaded, it happens that one of its half is within the SPM while the other is within the MM. As the technique has a frame level of granularity, a frame cannot be in such a state when it is accessed. The solution is to insert a symmetric partial load for each partial store before the corresponding frame is accessed.

When a store is full (i.e., the frame is fully stored), it is possible to leave the frame in the MM and update the following possible references to the frame instead of inserting a load, provided dependence analysis is able to explicit all the accesses to the frame that occur after the corresponding call instruction. If this is not the case, it is necessary to insert a symmetric full load in order to prevent accesses to the wrong memory.

Then, we must consider how frames are allocated within the SPM. The natural approach is to manage a second stack structure within the SPM as illustrated in figure 3(a). The figure is a continuation the scenario of figure 2: after frame 3 is allocated to the MM, frame 4 is allocated to the SPM. This allocation scheme is compatible with the store and load operations as shown in figure 3(b): frame 5 requires also an allocation to the SPM but there is no room left so that a partial store is necessary. This figure seems to illustrate an apparent limitation of the technique: if we stick to the second stack structure, a store will concern frame 4 first even if storing another frame could have been more interesting (for example, frame 4 might be in a loop so that the corresponding store and load operations would be executed much more often that for frame 2). Yet, it is possible to store a frame ahead of the need as shown in figure 4 where frame 2 is stored before
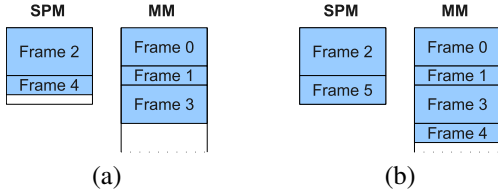
Fig. 3. Storing frame 4 for allocating frame 5, before (a) and after(b) frame 5 is allocated

function 3 is called even if there is still room in the SPM and even if frame 3 is to allocate to the MM. Actually, frame 2 is stored in prevision of the requirement of frame 5. The
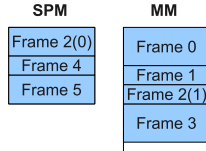

Fig. 4. Storing frame 2 ahead of the need

figure also illustrates a case of partial store: only the space required by frame 5 is stored from frame 2, frame 2(1) in the figure, while the remaining frame 2(0) is kept in the SPM. Additionally, it is enough to allow store and load operations to deal with current function's frame as long as it is not accessed by another function. For instance, with the example shown in figures 3 and 4, it is enough to allow storing or loading frame 2 during function 2, frame 4 during function 4 and so on. Indeed, as soon as a frame is not accessed, it is not necessary to keep it in the SPM so that if a store is necessary, it can be performed immediately. Furthermore, if a frame is not accessed by another function, it is enough to store it during the execution of its corresponding function only. The same reasoning goes for the load operations. In our technique, we do enforce each load and store to deal with the frame of current function only as it can be expressed with low complexity within the linear formulation of the problem. If no frame is larger than the SPM (or the space of the SPM dedicated to the stack), and if no function accesses a frame other than its current one, this model is optimal. The first restriction can be solved by splitting the frame as mentioned for future work in section VIII, and the second one is treated within the following paragraph.

Function arguments: if there is not enough registers to pass the totality of a function's arguments through, the usual convention is to push them on the top of the stack. We consider that this area is part of the frame of the called function so that the corresponding store (if any) have to be inserted before the arguments are written into the stack.

When one argument is actually a reference to a frame, the called function gains access to this frame. This is actually the only safe case where a function can have access to a frame different from its current one. In theory, it is possible to return a reference to a frame or to make a global variable point to a frame, but this is dangerous as when the function ends, its corresponding frame is deallocated. Yet, making a global

variable point to the frame is still valid as long as the variable is not used after the corresponding function ends. This last case can be considered as one where a pointer to current function's frame is passed as argument to all its descendant functions in the call graph. If dependence analysis cannot explicit all the accesses to a frame passed as argument or global variable, it must be either fully in the SPM or fully in the MM so that a partial store becomes impossible. Incidentally, it is only when a reference to a frame is passed as argument that it could have become interesting to delay the store of the frame. For instance, let us assume that function f's frame is referred by an argument of function g which does access frequently this frame before calling another function h. It could be preferable to store f's frame during g's execution, just before calling h.

Pointers to functions: with them, it is sometimes impossible to know at compile time which function is called. When this occurs there is no other choice but to leave the corresponding frame in the MM.

Library functions: it is common that compilers are provided with pre-compiled libraries. Although the functions of such libraries cannot be modified, it is still possible to integrate them into our optimization technique provided we know the maximum stack requirement for each of these functions. Without modifying the code of such a library function, its frame (or frames in the case it calls other functions) can always be allocated to the SPM instead of the MM as the stack pointer can be redirected before the library function is called. Store and load operations are also permitted just before and after the corresponding calling instructions.

Recursive functions: Recursive functions are also an important issue as it is often impossible to know at compile time how many instances of their frames will be allocated. Section VI-B describes how our technique is extended to support such functions.

C. Operations on the stack

In the standard managing of the stack, two operations are used: when entering a function, a first operation allocates a frame on the top of the stack by decreasing the stack pointer register by the frame's size. In this paper, we call this operation *grow*. When leaving a function, the second operation frees its corresponding frame from the top of the stack by increasing the stack pointer by the frame's size. We call this second operation *shrink*. Both operations take as argument the size of the frame to allocate as shown in their corresponding pseudo-code given in figure 5. In the figure, $sp is the stack pointer register.

For managing the stack between the SPM and the MM our technique utilizes the following new stack operations whose pseudo-code is also given in figure 5:

warp: makes $sp point to the SPM instead of the MM. The operation utilizes two variables, spm_top and mm_top, which contain the tops of the stack in respectively the SPM and the MM.

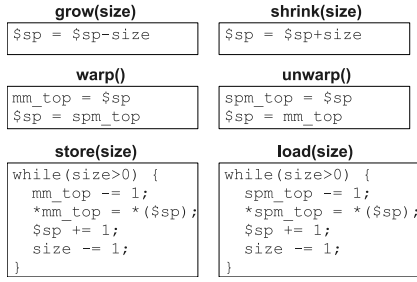unwarp: makes $sp point to the MM instead of the SPM. It is the opposite of the warp operation.

grow(size)	**shrink(size)**
`$sp = $sp-size`	`$sp = $sp+size`

warp()	**unwarp()**
`mm_top = $sp` `$sp = spm_top`	`spm_top = $sp` `$sp = mm_top`

store(size)	**load(size)**

`while(size>0) {`
`  mm_top -= 1;`
`  *mm_top = *($sp);`
`  $sp += 1;`
`  size -= 1;`
`}`
	`while(size>0) {`
	`  spm_top -= 1;`
	`  *spm_top = *($sp);`
	`  $sp += 1;`
	`  size -= 1;`
	`}`

Fig. 5.   Pseudo-code of the stack operations

store: copies `size` words from the top of the SPM stack pointed by `$sp` to the top of the MM stack pointed by the `mm_top` variable. Both `$sp` and `mm_top` are updated to continue to point to the respective SPM and MM tops. In the figure, the `*` operator is used to dereference pointers. This operation is possible only if `$sp` points to the SPM.

load: copies `size` words from the MM to the SPM. This operation is the opposite of the store as it can be seen for its pseudo-code given in figure 5. This operation is possible only if `$sp` points to the MM.

Note: the actual implementation of the code given in figure 5 is done directly in assembly as it is only at this level that the stack pointer register and the real size of the frames can be accessed. In our current implementation, we use a small space in the SPM for the `spm_top` and `mm_top` variables and for saving the registers used by the stack operations.

D. Functions called several times

It often happens that a same function is called several times in the same program. Such a function will actually have several frames. Moreover, when the function is executed, the stack can be in several different states, so that depending on the case, different stack operations might be required before and after the call instructions of the function's code. Instead of finding the best compromise between several stack operations, our approach is to insert predicates which check from where the function has been called as illustrated in figure 6: `f` calls `g` so that before calling `g` a predicate is required to know if a store is to be inserted or not (symmetrically, the same kind of predicate controls the load). In the figure, the pseudo-code first checks
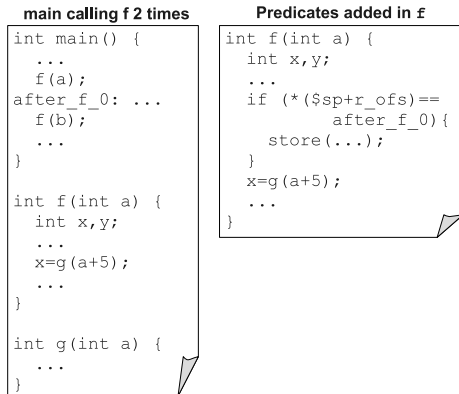
main calling f 2 times	**Predicates added in f**
```int main() { ... f(a); after_f_0: ... f(b); ... } int f(int a) { int x,y; ... x=g(a+5); ... } int g(int a) { ... }```	```int f(int a) { int x,y; ... if (*($sp+r_ofs)== after_f_0){ store(...); } x=g(a+5); ... }```

Fig. 6.   A predicate for checking from where function `f` is called

from which address function `f` has been called (it is assumed in the figure that the return address is stored into the last word of the frame, `r_ofs` being the corresponding offset relative to the stack pointer register during `f`). If it was from the place where, for instance, a later store is required for `g`'s frame, the corresponding block is executed before calling function `g`. Otherwise, function `g` is called without any operation on the stack.

When a function `f` does not call another function, no predicate is required as the corresponding load and store operations (if any) are located respectively before and after the call instruction and not within `f`'s body.

E. Additional optimizations

Store and load operations do consume energy so that reducing their number is preferable. In the case a function is called alone within a loop where there is no stack access, the corresponding store and load can be moved out of the loop.

Another optimization is possible if dependence analysis can show that the top part of a frame has not yet been accessed before a call: this part can then be omitted by the corresponding store and load operations.

Finally, if the target architecture includes a DMA, it can be used for the store and the load operations in order to reduce their costs.

V. PROBLEM FORMULATION

A. Initial definitions

We call *place* a point in the program where a new stack operation can be inserted. There are two kinds of places: the *store places* located just before the call instructions where store and warp or unwarp operations can be inserted, and the *load places* located just after the call instructions where load and warp or unwarp operations can be inserted. The optimizations mentioned in section IV-E are performed on the places: they are moved out of the loops when possible and are annotated with the part of the corresponding frame that can be omitted for storing and loading.

In order to formulate the problem, the control flow and call graph is divided into several sessions. We define a *session* as a tree in the graph whose root is a place (i.e., the corresponding basic block) and whose leaves are the next places encountered in sequence. In practice sessions are often simple paths within the graph, but in the case of calls nested within conditional blocks, the leaves can be multiple. As there are two kinds of places, there are two kinds of sessions: we call *grow session* a session starting from a store place (as it includes a grow operation) and *shrink session* a session starting from a load place[1]. Figure 7 gives an example of places and sessions in the call and control graph associated to a small program. In the figure, double arrows stand for call or return edges while plain arrows stand for control edges.

---

[1]This last session is built by connecting the return block to the corresponding calling block.
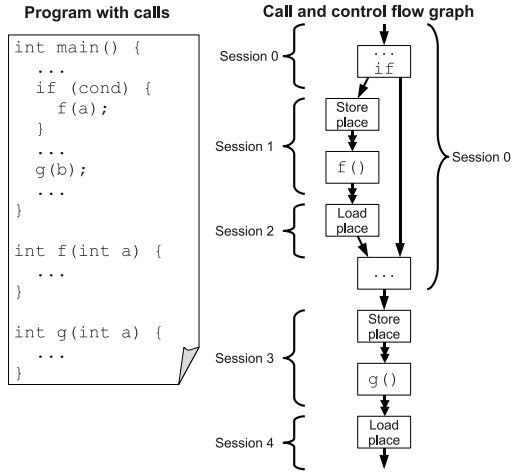
Fig. 7. Places and sessions in a control and flow graph

Note: when a function is called from several different points in the program, there will be equally several different sessions which will cover the same parts of its code.

Sessions are used to refer to potential new stack operations in the formulation given in the next section. They are also used for collecting the profile informations: stack access rates and executions counts are accumulated at the session level.

B. The ILP formulation

In order to decide which stack operations to insert, the energy consumed while accessing and transforming the stack is modeled by a linear function. This function is the objective to minimize in an ILP whose solution gives the evolution of the allocation states of the frames and the load and store operations to use during the execution of the program.

The objective function must account for:

- The reading and writing accesses to the frames located in the MM;
- The reading and writing accesses to the frames located in the SPM;
- The storing of frames;
- The loading of frames;

The costs of the warp and the unwarp operations and the cost of the predicates are initially neglected for simplifying the presentation.

The variables and the constants of the problem are indexed by the frame identifier (i) and by the session identifier (j). Additionally, we note the variables with lower case letters and the constants with heading upper case letters. With such conventions, the objective function is computed with the following variables:

$x_{i,j}$: is 1 when frame i is fully in the SPM for session j and 0 otherwise;

$st_{i,j}$: its value is the number of words of frame i stored at the beginning of session j;

$ld_{i,j}$: its value is the number of words of frame i loaded at the beginning of session j;

The constants used in the objective function are the followings:

$Cspm_{r/w}$: cost of a single word read (r) / write (w) access to the MM;

$Cmm_{r/w}$: cost of a single word read/write access to the MM.

Cst: cost for storing one word;

Cld: cost for loading one word;

$Na_{r/w,i,j}$: number of read/write accesses to frame i during session j, this constant is estimated by profiling the application;

Ne_j: number of executions of session j, this constant is estimated by profiling the application.

Note: the access costs given in this paper are averages and are used for sake of clarity. It is possible, instead, to use the exact energy consumption for the SPM, MM, load and store accesses related to each session.

Although the variables and the constants are indexed by all the frame and session identifiers they are not always defined:

- variables and constants are defined only when corresponding frames are allocated;
- storing a frame can be performed only before a call instruction from its corresponding function as mentioned in section IV-B;
- loading a frame can be performed only when going back to its corresponding function as mentioned section IV-B;

In order to simplify the notations, we consider that the undefined variables and constants have the default value 0.

With those conventions, the energy consumption related to the stack is modeled by the following expression:

$$\sum_{i,j} \Big(Na_{r/w,i,j} * \big(Cspm_{r/w} * x_{i,j} + Cmm_{r/w} * (1 - x_{i,j}) \big) + Ne_j * \big(Cst * st_{i,j} + Cld * ld_{i,j} \big) \Big) \quad (1)$$

This expression is to minimize but is not linear as it includes constant terms. Hence, the constant terms are removed to obtain the following function:

$$\sum_{i,j} \Big(Na_{r/w,i,j} * (Cspm_{r/w} - Cmm_{r/w}) * x_{i,j} + Ne_j * \big(Cst * st_{i,j} + Cld * ld_{i,j} \big) \Big) \quad (2)$$

This expression is the objective function to minimize in the ILP. As mentioned in section IV-B, we restrict the frames to be consistent when they are accessed, that is to say fully in the SPM or fully in the MM. This is translated to the following constraints:

$$\forall i,j \qquad A_{i,j} \Rightarrow xa_{i,j} = x_{i,j} * S_i \quad (3)$$

In equations (3), each constant $A_{i,j}$ is false if and only if the references to frame i are known and it is not accessed by its function, each constant S_i represents the total size of frame i and each variable $xa_{i,j}$ represents the number of words of frame i present in the SPM for session j. Therefore constraints (3) force each $xa_{i,j}$ to be of the size of frame i if $x_{i,j}$ is 1 or to be 0 if $x_{i,j}$ is also 0. Variables $xa_{i,j}$ are also

used to ensure that the stack data in the SPM does not exceed S_{spm} (the SPM size reserved to the stack) as checked by the following constraints:

$$\forall j \qquad \sum_i xa_{i,j} \leq S_{spm} \qquad (4)$$

The last constraints are there to apply the effects of stores and loads to the corresponding $xa_{i,j}$ variables (provided they are defined). Store constraints are the followings:

$$\forall i,j \quad \forall k \in Prev(j) \qquad xa_{i,j} = xa_{i,k} - st_{i,j} \qquad (5)$$

And load constraints are the followings:

$$\forall i,j \quad \forall k \in Prev(j) \qquad xa_{i,j} = xa_{i,k} + ld_{i,j} \qquad (6)$$

(Please notice that a load and a store cannot appear on a same session.) In the above equations, $Prev(j)$ is the set of indexes of the sessions which precede session j in the control and call graph. The last-in first-out (LIFO) allocation of the SPM stack is implicitly respected by the store and load variables as they are only defined when, respectively, calling and returning from a function. When lack of dependence information enforces load and store operations to be symmetric, the following constraints are added ($Safe_i$ is true when frame i is to be symmetrically stored and loaded):

$$Safe_i \Rightarrow ld_{i,j} = st_{i,Call(j)} \qquad (7)$$

In the above equations, $Call(j)$ is the index of the session which called the function returned to during session j.

While the sessions might be numerous, the total number of allocated frames for a session is rarely high in practice. Therefore a majority of the variables of the problem are actually independent with each other. Moreover, the access constraints force a lot of $xa_{i,j}$ variables to be actually pseudo-binary (having only 2 possible values) so that the ILP is of small complexity. For instance, for our experiments (section VII), the time for solving each problem by the ILOG-CLPEX [21] lp solver did not exceed 0.1 second. However, in the case where an application contains a deep call graph, the problem's complexity might quickly grow. If the solving time becomes too large, the complexity can be scaled down by formulating and solving the problem for the most frequently executed branches of the call graph first.

C. Additional costs

The cost of warps, unwarps and predicates can also be taken into account in the objective function if such a precision is required.

Warp and unwarp operations have a fixed cost and are necessary only when the top frame of the new session is in a different memory compared to one of the previous sessions. It is then enough to add the $Cw * w_j$ products to the objective function where Cw is the cost of a warp or an unwarp and where w_j is the binary variable telling if a warp or an unwarp is necessary for session j. It is constrained as follows:

$$\forall j \quad \forall k \in Prev(j) \qquad w_j \geq |x_{Fr(j),j} - x_{Fr(k),k}| \qquad (8)$$

In the above equations, $Fr(j)$ is the frame of the function which will be executed from session j.

Predicates are necessary when several sessions corresponding to a same piece of code (a function is called from several different locations) have different states. It is dealt with by adding the $Cp * p_j$ products to the objective function where Cp is the cost of a predicate and p_j is the binary variable telling if a predicate is required or not for session j. It is constrained as follows:

$$\forall i,j \quad \forall k \in Code(j) \qquad p_j \geq |xa_{i,j} - xa_{i,k}|/S_{spm} \qquad (9)$$

In the above equations, $Code(j)$ is the set of sessions whose code is the same as j's and S_{spm} is used to ensure that p_j is constrained by values smaller than 1.

Note: all these constraints are actually linear as $a \geq |b|$ is equivalent to $a \geq b$ & $a \geq -b$.

VI. EXTENDING THE APPROACH

A. Including static memory objects

Papers like [1], [3], [8]–[10] already shown how to reduce energy consumption by placing static objects (including code and data) in the SPM using an ILP formulation. Such approaches are fully compatible with ours so that a total static and stack optimization ILP formulation can be made as follows (obj_{stack} is the objective function given in equation (2)):

$$obj_{stack} + \sum_l Na_{r/w,l} * (Cspm_{r/w} - Cmm_{r/w}) * x_l \qquad (10)$$

In the above equation, the l indexes concern the same type of variables and constants as the ones used in the stack optimization part but refer to the static objects. The only thing to change in the constraints is to replace the S_{spm} constant (the size of SPM reserved to the stack) by the s_{stack} variable which represents the same size but in concurrence with the static objects. Hence, this new variable is constrained as follows:

$$s_{stack} + \sum_l S_l * x_l \leq S_{spm} \qquad (11)$$

In the above equation, constants S_l give the size of the corresponding static objects and constant S_{spm} becomes the size of the SPM reserved to the static objects and the stack.

Please notice that we gave here a simplified formulation for the static part in order to show clearly how such an extension can be performed. More complex ones are still compatible with our approach.

B. Including recursive functions

Section III mentioned two approaches able to manage recursive functions within the SPM. This section presents how these approaches are used to extend ours with support for such functions.

We restrict the approach with the circular buffer [19] to functions which do not pass any reference to their frames as arguments of calls or through global variables. If such a reference were to be passed, it would have been possible that a frame is accessed while it has been evicted from the buffer.

For each recursive function, a circular buffer is integrated into the model by adding to the objective function the cost of accessing the SPM for such frames plus the cost of moving oldest frames to the MM and, afterward, back to the SPM. These operations are actually close to our store and load operations. No cost for the MM accesses is required for such frames as they are always accessed while being in the SPM. The oldest frames of the buffer will start to be moved to the MM when the SPM is full. When this occurs, there will be exactly one store and one load per such additional frame. The resulting objective function is the following (obj_{stack} is the objective for non recursive functions):

$$obj_{stack} + \sum_{u,j} \Big(Na_{r/w,u,j} * (Cspm_{r/w} - Cmm_{r/w}) * x_{u,j}$$
$$+ (Ne_{u,j} - n_{u,j}) * Ccirc * S_u \Big) \quad (12)$$

In the above equation, u is the index of the frames of recursive functions, $n_{u,j}$ is the variable giving the number of times frame u's size is allocated to the SPM and $Ne_{u,j}$ is the constant giving the number of times the frame is allocated. In other words, $n_{u,v}$ represents the threshold of recursion before the oldest frames start to be stored. Constant $Ccirc$ represents the cost of replacing one word in the buffer, which corresponds to one store, one load and a residual cost for managing the buffer. As in section V-B, the $Ccirc*S_u$ product could be replaced by the actual cost measured for each buffer rotation operation. Indexes j still refer to the session numbers. Variables $n_{u,j}$, controlling the size of the buffers, are also linked to variables $xa_{u,j}$ in a modified version of equations (3):

$$\forall u, j \qquad A_{u,j} \Rightarrow xa_{u,j} = n_{u,j} * S_u \quad (13)$$

Please notice that the other constraints including $xa_{u,j}$ given by (4), (5), (6) and by (7) are still valid. A final set of constraints links variables $x_{u,j}$ to variables $n_{u,j}$ by forcing each $x_{u,j}$ to be 1 when the corresponding $n_{u,j}$ is greater than 0 and to be 0 when the corresponding $n_{u,j}$ is 0:

$$\forall j, u \qquad n_{u,j} * S_u / S_{spm} \leq x_{u,j} \leq n_{u,j} \quad (14)$$

In the equation, the value S_{spm}/S_u is guaranteed to be greater or equal to $n_{u,j}$.

The circular buffer for a recursive function is compatible with the LIFO allocation of the other functions because it is considered as a normal frame whose size is $n_{u,j} * S_u$. Even loads and stores can still be performed as the buffer is not used when another function called by the recursive one is executed.

For the approach where frames corresponding to a given depth of recursion are put into the SPM [20], it is enough to add a grow and a shrink session for the corresponding recursive function per depth of interest to be treated like a different non-recursive function.

VII. EXPERIMENTS

A. Experimental environment

We applied our technique on a MeP [22] processor configuration including a data SPM. We used the Toshiba's MeP Integrator (MPI) tool chain [22], [23] for compiling and simulating the applications. Compilations were performed with the $-O2$ level of optimization.

We evaluated our approach on applications coming from the EEMBC [24] and MiBench [4] benchmark suites. Informations related to these examples are shown in table II. In the table, the stack access rates are given relative to the total data memory accesses. For the energy consumption, we used the power

| Application | Benchmark | recursion | stack | |
			max size	access rate
AES	EEMBC	no	64 bytes	74%
DES	EEMBC	no	48 bytes	14%
rad2deg	MiBench	no	224 bytes	65%
Patrical	MiBench	yes	104 bytes	90%
mp2dec (part)	EEMBC	no	132 bytes	29%
MD5	EEMBC	no	320 bytes	57%
cubic	MiBench	no	1328 bytes	74%
FFT	MiBench	no	1012 bytes	62%

TABLE II
PROGRAMS USED FOR THE EXPERIMENTS

characteristics of memories given in table I (source [5], [6]).

B. Results

The technique proposed in this paper is compared with the circular buffer (*circular*) approach proposed by [19] and the static frame-level (*static*) approach proposed by [10]. The comparison has been made possible by reimplementing both approaches within our framework. The experiments' results are shown in figure 8. In the figure, each diagram displays, for the given application, the quotient between the stack-related energy consumption obtained after applying an optimization technique with the stack-related one obtained when the stack data are all within the MM. For each diagram, several SPM sizes are considered depending on the total size of the corresponding stack.

If we consider an SPM size which is only one half of the maximum stack size, as considered in the last diagram of figure 8, our technique achieves an energy reduction of 84% on average whereas with the circular buffer technique the reduction is about 64% and with the static technique it is about 79%. But these averages hide important disparities in efficiency between the circular and the static techniques while ours is always equivalent or more efficient than the best of the two. This is not surprising as both can be seen as subsets of our technique: they are part of the solution space which is explored while solving the ILP presented in this paper. This is obvious for the static frame-level technique, but this is also true for the circular buffer one even though the data structure used in the SPM is different. Indeed, the possibility to store frames ahead of the need includes the possibility to store the frames which would have been evicted from the circular buffer. Actually, even if partial loads and stores were not allowed, our technique would still be a superset of the two other techniques.

While the circular buffer approach can outperform the static frame-level one, some artifacts frequently occur which jeopardize the result. It is actually a consequence of the
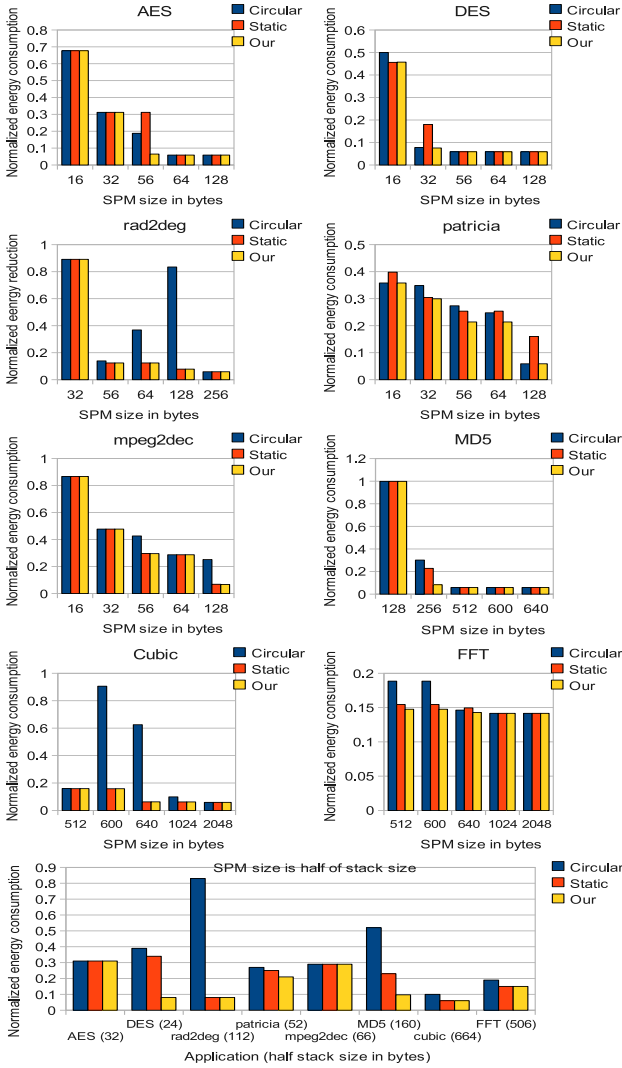
Fig. 8. Stack-related energy consumptions normalized relatively to the case where the stack is fully in the MM

Fig. 9. Stack-related energy reductions normalized relatively to the reduction obtained when the stack is fully in the SPM

systematic eviction of the oldest frames when the space left in the SPM for a new frame is too low. This leads sometimes to frame moves that are more expensive than the gain obtained by accessing the SPM. Cubic is a typical example for such artifacts, but they occur at lower degrees in the other applications.

The patricia application includes recursive functions and therefore the static approach gives poorer results than the circular buffer one and ours when, for a given input data set, there is enough space in the SPM for containing the totality of the stack. When the SPM is too small, our technique performs better than the pure circular buffer technique thanks to the optimal sizing of the buffer added into the ILP as presented in section VI-B.

When the SPM is large enough to include the totality of the stack, all the approaches obtain the same result (provided there is no recursive function). Hence, more than the absolute energy consumption reduction, it is the reduction rate compared to the reduction obtained when the stack is fully in the SPM which m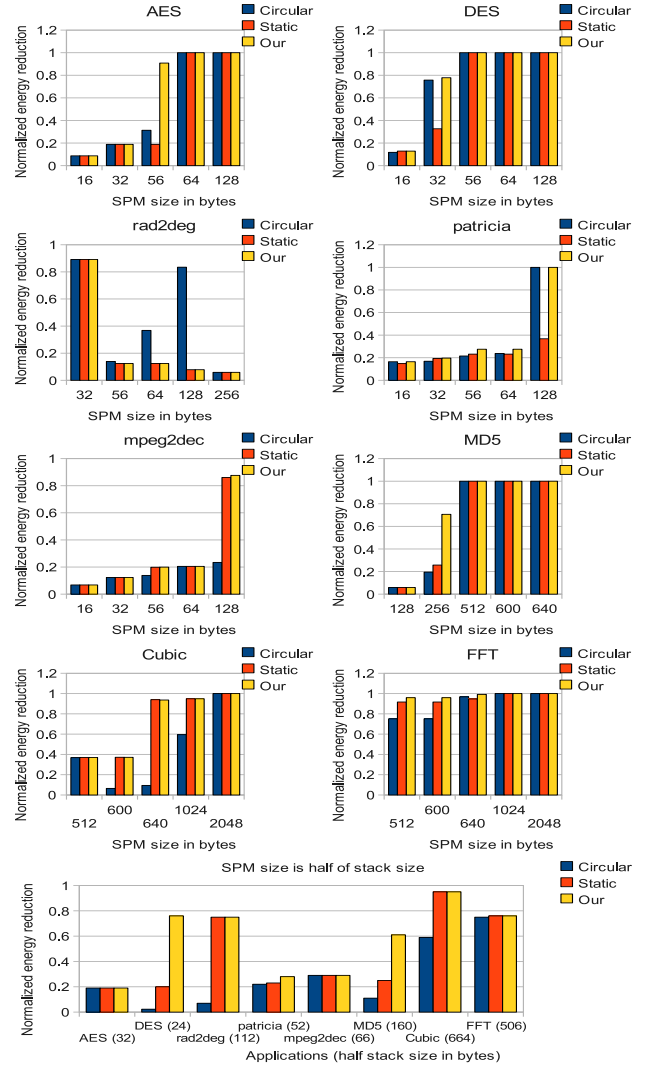akes sense. Figure 9 gives such rates for the previous applications and SPM sizes. If we consider an SPM size which is only one half of the maximum stack size, our technique achieves an energy reduction which represents 56% on average of the reduction obtained when the stack is fully allocated to the SPM. The circular buffer approach only achieves 27% and the static approach 41%.

The hybrid frame/variable-level mentioned by [10] is complex to formulate and implement in practice, but can be qualitatively compared to our technique. The low granularity of the frame level is usually compensated by partial loads and stores whose overhead can be compared favorably with the overhead of accessing the dispatched variables of a frame. Moreover, the dynamic evolution of the frames allocation state permitted by our approach allows it to surpass the hybrid method. Remains the case where a frame is larger than the SPM space available for the stack: if the variables of such a frame can be separated at low cost, the hybrid approach has an advantage as it can allocate a part of this frame to the SPM while our technique cannot as it requires it to be either fully in the SPM or fully in the MM when accessed.

We plan to address this limitation as future work by splitting frames when possible and pertinent during a preprocessing stage. Then each part of the split frames can be dealt with like any other frame by our current technique.

Finally, it can be mentioned that performance too is improved by using the scratch-pad memory. If we consider that the processor is running at 200Mhz, then the access latency of the main memory is about 24 cycles in sequential access for the SDRAM considered for our experiments [6] which mean than the ratio with the SPM latency (1 cycle) is even larger than the energy-related one.

VIII. CONCLUSION

In this paper, a software technique has been proposed to reduce the energy consumption of stack accesses by managing its frames between a scratch-pad (SPM) and the main memory (MM). The optimization which includes the possibility of moving partially of fully frames from one memory to the other is formulated as an integer linear programming problem. Then it has been shown how the technique is extended with support for static memory objects and recursive functions.

Experimental results showed that the approach gives similar or better results than a static ILP-based approach and a dynamic approach managing the SPM like a circular buffer of frames. With an available SPM space which is only one half of what the stack requires, the technique allows to benefit from an energy reduction of 84% on average, which represents 56% of what is obtained when the stack can fully fit in the SPM.

While the approach allows partial loads and stores, it still requires the frames to be fully in the SPM or fully in the MM when accessed. Less energy consumption could be expected if this last constraint could be relaxed. For future work we consider to add a step which split frames. Each part of a split frame could then be allocated, stored or loaded independently of the others.

Another future work is to include the heap in our global memory optimization technique. This last enhancement is however much more complex to model as heap objects are allocated and freed in a completely random manner. A track could be to use a pool-based memory allocation system.

REFERENCES

[1] L. Wehmeyer, U. Helmig, and P. Marwedel, "Compiler-optimized usage of partitioned memories," in *WMPI '04: Proceedings of the 3rd workshop on Memory performance issues*. New York, NY, USA: ACM, 2004, pp. 114–120.

[2] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," in *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*. New York, NY, USA: ACM, 2002, pp. 73–78.

[3] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction," in *DATE '02: Proceedings of the conference on Design, automation and test in Europe*. Washington, DC, USA: IEEE Computer Society, 2002, p. 409.

[4] University of Michigan, "MiBench benchmark suite." [Online]. Available: http://www.eecs.umich.edu/mibench/

[5] Samsung Semiconductor, "K4X51163PC Mobile DDR SRAM."

[6] Micron Technology, Inc., "MT48H8M16LF Mobile SDRAM."

[7] P. R. Panda, N. D. Dutt, and A. Nicolau, "Efficient utilization of scratch-pad memory in embedded processor applications," in *EDTC '97: Proceedings of the 1997 European conference on Design and Test*. Washington, DC, USA: IEEE Computer Society, 1997, p. 7.

[8] J. Sjödin and C. von Platen, "Storage allocation for embedded processors," in *CASES '01: Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*. New York, NY, USA: ACM, 2001, pp. 15–23.

[9] M. Verma, S. Steinke, and P. Marwedel, "Data partitioning for maximal scratchpad usage," in *ASP-DAC '03: Proceedings of the 2003 Asia and South Pacific Design Automation Conference*. New York, NY, USA: ACM, 2003, pp. 77–83.

[10] O. Avissar, R. Barua, and D. Stewart, "An optimal memory allocation scheme for scratch-pad-based embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 1, no. 1, pp. 6–26, 2002.

[11] B. Egger, J. Lee, and H. Shin, "Dynamic scratchpad memory management for code in portable systems with an mmu," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 2, pp. 1–38, 2008.

[12] E. G. Hallnor and S. K. Reinhardt, "A fully associative software-managed cache design," *SIGARCH Comput. Archit. News*, vol. 28, no. 2, pp. 107–116, 2000.

[13] C. A. Moritz, M. Frank, and S. P. Amarasinghe, "Flexcache: A framework for flexible compiler generated data caching," in *IMS '00: Revised Papers from the Second International Workshop on Intelligent Memory Systems*. London, UK: Springer-Verlag, 2001, pp. 135–146.

[14] M. Mamidipaka and N. Dutt, "On-chip stack based memory organization for low power embedded architectures," in *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2003, p. 11082.

[15] S. Park, H.-w. Park, and S. Ha, "A novel technique to use scratch-pad memory for stack management," in *DATE '07: Proceedings of the conference on Design, automation and test in Europe*. San Jose, CA, USA: EDA Consortium, 2007, pp. 1478–1483.

[16] S. Udayakumaran and R. Barua, "Compiler-decided dynamic memory allocation for scratch-pad based embedded systems," in *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*. New York, NY, USA: ACM, 2003, pp. 276–286.

[17] R. Barua and S. Udayakumaran, "Compiler-decided dynamic memory allocation methodology for scratch-pad based embedded systems," U.S. Patent 7 367 024, 2008.

[18] S. Udayakumaran, A. Dominguez, and R. Barua, "Dynamic allocation for scratch-pad memory using compile-time decisions," *ACM Trans. Embed. Comput. Syst.*, vol. 5, no. 2, pp. 472–511, 2006.

[19] 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: Proceedings of the 2009 Asia and South Pacific Design Automation Conference*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 612–617.

[20] A. Dominguez, N. Nguyen, and R. K. Barua, "Recursive function data allocation to scratch-pad memory," in *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*. New York, NY, USA: ACM, 2007, pp. 65–74.

[21] ILOG, "CPLEX LP solver." [Online]. Available: http://www.ilog.com/products/cplex

[22] Toshiba, "MeP processor." [Online]. Available: http://www.semicon.toshiba.co.jp/eng/product/micro/mep/document/index.html

[23] A. Mizuno, H. Uetani, and H. Eichel, "Design methodology and system for a configurable media embedded processor extensible to vliw architecture," in *ICCD '02: Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02)*. Washington, DC, USA: IEEE Computer Society, 2002, p. 2.

[24] Embedded Microprocessor Benchmark Consortium, "EEMBC benchmark suite." [Online]. Available: http://www.eembc.org/home.php