Hindawi Publishing Corporation
Journal of Electrical and Computer Engineering
Volume 2012, Article ID 786943, 16 pages
doi:10.1155/2012/786943

Research Article

Processor Energy Characterization for Compiler-Assisted Software Energy Reduction

Lovic Gauthier[1] and Tohru Ishihara[2]

[1] *3rd Floor Institute of System LSI Design Industry, Kyushu University, Fukuoka, 3-8-33 Momochihama, Sawara-ku, Fukuoka 814-0001, Japan*
[2] *Department of Communications and Computer Engineering, Kyoto University, Yoshidahonmachi, Sakyo-ku, Kyoto 606-8501, Japan*

Correspondence should be addressed to Lovic Gauthier, lovic@soc.ait.kyushu-u.ac.jp

Received 15 July 2011; Revised 12 October 2011; Accepted 15 November 2011

Academic Editor: Ayse Kivilcim Coskun

Energy consumption is a fundamental barrier in taking full advantage of today and future semiconductor manufacturing technologies. The paper presents our recent research activities and results on characterizing and reducing the energy consumption in embedded systems. Firstly, a technique for characterizing the energy consumption of embedded processors during an application execution is presented. The technique trains a per-processor linear approximation model for fitting it to the energy consumption of the processor obtained by postlayout simulation. Secondly, based on the energy model mentioned above, the paper shows techniques for reducing the energy consumption by optimally mapping program code, stack frames, and data items to the scratch-pad memory (SPM) of the processor memory space.

1. Introduction

There is a wide consensus that the energy consumption is a fundamental barrier in taking full advantage of today and future semiconductor manufacturing technologies. The paper presents our recent research activities and results in two categories: estimating software energy consumption and reducing the energy required for accessing the memory subsystem. Firstly, the paper presents a technique to estimate the instantaneous energy consumption of embedded processors during an application execution. The technique trains a per-processor energy model which receives statistics from the processor instruction-set simulator (ISS) and gives the instantaneous energy consumption. Secondly, the paper presents techniques for reducing the energy consumption by optimally mapping program code, stack frames, and static data items to the scratch-pad memory (SPM) which is integrated on a processor chip.

In the rest of the paper, Section 2 presents the instantaneous energy estimation technique, Section 3 provides various techniques for reducing the energy consumption of the memory subsystem, and Section 4 summarizes and concludes the paper.

2. Software Energy Analysis

This section shows an overview of our energy characterization tool which helps designers in developing a fast and accurate energy model for a target processor-based system. Our tool uses a linear model for energy estimation and finds the coefficients of the model using multiple linear regression analysis. For more detailed information of our tool see [1].

2.1. Related Work. The most accurate and fastest approach to find the energy consumption of software running on a processor is to measure the power consumption of the actual chip. Tools like PowerScope [2] and Itsy [3] use computer-controlled multimeters or A/D converters to measure energy consumption. The major drawback of using PowerScope is that it cannot measure the energy consumption of individual subsystems (e.g., a memory system) separately. Although Itsy overcomes this issue, it cannot measure the energy consumed in a short period of time because the energy consumption is averaged out over the entire execution time. In recent years, many instruction-level energy modeling and analysis techniques have been proposed. The idea of instruction-level energy modeling by measuring the power consumption of

each instruction while executed in a loop was introduced in [4, 5]. The accuracy of these methods were improved by accounting for data dependencies, the effects of instruction and data addresses, register file addresses, and operand values [6]. The main drawback of these techniques is that they rely on exhaustive simulation to find the energy consumption of each instruction and the interinstruction effects on the power consumption. The efficiency of the characterization can be improved by performing measurements on a limited subset of instructions and instruction sequences [7, 8]. In [9, 10], the same average energy is assumed for all instructions and the average power consumption while running an application program is calculated using the operating voltage of the processor and the clock frequency. In [11], the authors measure the average energy consumed in each pipeline stage of a VLIW processor using a cycle-accurate simulator (e.g., Trimaran [12]) to improve the accuracy. The techniques estimate the average energy consumption over the entire program execution, while many software-level energy optimization techniques need cycle-accurate energy estimation. The technique described in [13] estimates the power consumption of the target processor cycle by cycle. However, this requires calculating the power consumption of every gate for each instruction which is very time consuming. Most of existing energy estimation techniques including the techniques presented in [4–11, 13–21] assume a linear approximation model for estimating the energy consumption of software running on a processor. However, none describes how to generate the linear approximation model for accurately characterizing the energy consumption of the processor. In [21], Tan et al. modeled the software energy consumption using a linear equation and discussed the parameters required to accurately estimate the energy consumption. However, they did not provide any method to find parameters, corresponding coefficients, and test benches required for the accurate energy modeling. Our energy characterization framework assists designers to find an accurate linear model for estimating the energy consumption of a processor-based embedded system. In our framework, designers can find a training bench suitable for the energy characterization of the target embedded system and can optimize the corresponding coefficients through multiple regression analysis. We use parameters which can be extracted through GNU C debugger which is provided for almost all types of commercial embedded processors. Therefore, our approach does not need a cycle-accurate ISS which is not provided for many types of embedded processors and is usually more expensive than a cycle-inaccurate one like a GNU debugger.

2.2. A Target System. We target a processor system which consists of a CPU core, an instruction cache, code and data on-chip scratch-pad memories (SPMs), and SDRAM as an off-chip main memory as shown in Figure 1.

The following three types of processors have been used in the experiments validating our approach in [1]: (1) M32R-II, a 32-bit RISC microprocessor originally developed by Renesas Technology Corporation, (2) SH3-DSP, a 32-bit RISC microprocessor of Renesas Technology Corporation,

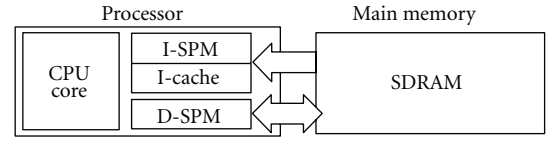


Figure 1: A target processor system model.

and (3) multiperformance processor [22, 23] which is based on Media embedded Processor (MeP) developed by Toshiba. All of the above three microprocessors have on-chip cache memories and scratch-pad memories (SPMs). The processors are synthesized using a 0.18 μm CMOS standard cell library and an SRAM module library. In this paper, for sake of concision, only (3), the multiperformance processor, is used in the results section.

2.3. Processor Energy Characterization. The energy consumption of a processor can be estimated using the following linear formula:

$$E_{\text{estimate}} = \sum_{i=0}^{N} c_i * P_i, \tag{1}$$

where P_i's, c_i's, and N are the parameters of the model, the corresponding coefficients, and the number of parameters, respectively. The first step for the modeling is to find the P_i's required for estimating the energy consumption of the target processor system. The P_i's should be parameters whose values can be easily obtained using a fast simulator like an ISS. For example, P_i's can be the number of load and store instructions executed, the number of cache misses, and so forth. Once the required set of parameters is obtained, the next step is to find a training bench for the energy characterization. Note that the number of cycles simulated for the training bench is much smaller than that for the target application programs. In our tool, the generated training bench is simulated during about 500,000 cycles only while the full simulation of the target application programs needs billions of cycles. More detailed explanation for our method to generate the training bench is presented in the following subsection. The final step is to find the coefficients, c_i's corresponding to the P_i's. This is done by using multiple linear regression analysis. The energy consumption E_{estimate} is then calculated using (1). Figure 2 shows an overview of our energy characterization flow.

To obtain the reference energy values, we simulate the processor system at gate level for a fixed number of instructions. We refer to this number of instructions for a test sequence as the instruction frame. The width is the same for all the instruction frames as shown in Figure 3. Since we perform gate-level simulation and calculate the energy consumption values for all the instruction frames, this step is time consuming. However, it needs to be done only once for a given processor system.

Next, we produce an instruction trace for each application program using an instruction-set simulator. The traces are divided into small segments each corresponding to one instruction frame. The parameters P_i's are obtained from

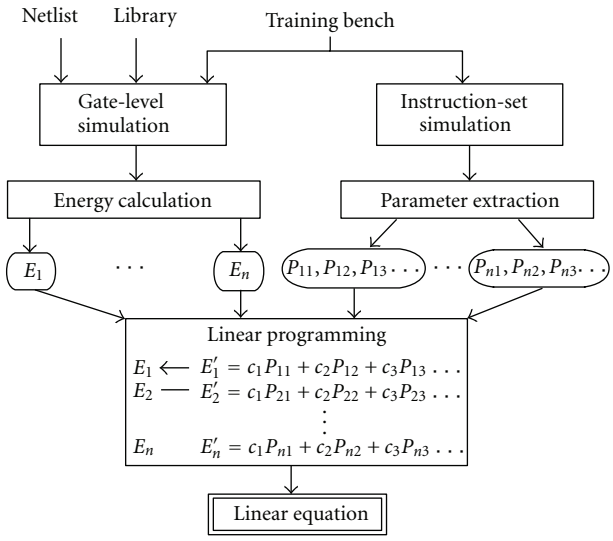


Figure 2: Overview of energy characterization.

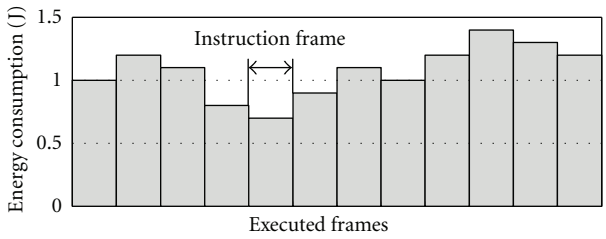


Figure 3: An example of instruction frame.

these instruction traces. Then, for a set of P_i's, we find the coefficients which minimize $\sum |E_{\text{estimate}}(i) - E_{\text{gate-level}}(i)|^2$, where $E_{\text{gate-level}}(i)$ and $E_{\text{estimate}}(i)$ are the energy consumption values obtained by gate-level simulation and through (1) for the ith instruction frame, respectively.

2.4. Training Bench Generation. As described in the previous subsection, the selection of training bench is the most important process for the energy characterization. Figure 4 shows a motivational example. Both (a) and (b) of Figure 4 show energy estimation results for JPEG encoder and MPEG2 encoder run on a target processor system. For the results of the left figure, a file compression program compress is used as a training bench and is executed for 500,000 instructions by gate-level simulation for the energy characterization. Through the characterization, a linear equation for estimating the energy consumption of a target processor system is obtained. Then the estimated energy consumption values are compared with the gate-level energy consumption results. As can be seen from the Figure, the energy estimation error is huge. The error is on an average 67% and more than 1000% for the worst case. There are two major reasons of the huge estimation error as follows.

(1) Standard deviations of some parameter values are too small. For example, the numbers of cache misses are constant for all the instruction frames. In this case, it is difficult to identify an impact of the cache misses on the energy consumption of the processor.

(2) Some parameters are strongly correlated to each other. For example, the numbers of cache misses and the numbers of branch misses have similar trends in an entire training bench. In this case, it is difficult to distinguish the impact of cache misses from that of branch misses.

If we carefully generate the training bench, the accuracy of the energy estimation can be improved drastically. The right of Figure 4 shows estimation results of our approach. The only difference between the left and right results in Figure 4 is the training bench. We generate the training bench considering the standard deviations of every parameter values and correlation factors between any two parameters as criteria for generating the training bench. As a result, the estimation error of our approach is on an average 2% and 16% even for the worst case. Figure 5 shows the flow of our approach for generating the training bench. The training bench generation starts from a template of training bench which consists of subroutines which execute power-hungry instructions like a data-access instruction repeatedly and produce many cache misses, many read-after-write hazards, and other pipeline stalls. The parameter values are extracted using ISS. This process takes a few seconds. Then the standard deviations of every parameter values and correlation factors of any two parameters are evaluated. If the standard deviations of some parameter values are lower than a specific value or if the correlation factors of some parameters are higher than a specific value, the initial training bench is modified so that the standard deviations and the correlation factors are improved. This process is repeated until those two criteria are satisfied.

2.5. Debugger-Based Energy Estimation. Once the energy model is developed, the energy consumption of software running on the processor system can be estimated using a cycle-*inaccurate* instruction-set simulator (ISS) whose speed is 350,000 instructions per second. We use our tool for characterizing the energy consumption of three commercial microprocessors with their on-chip caches, SPMs, and an off-chip SDRAM. Experimental results using three benchmark programs demonstrate that the error of our technique is on an average 5% compared to the postlayout gate-level estimation results. Figure 6 shows the energy consumption results estimated for an MPEG2 encoder program executed on MeP processor. The line chart represents the results of postlayout gate-level energy estimation. The colored portion of the graph represents the energy consumption estimated by our approach. The results show that the accuracy of our energy estimation model is very good.

Today's SoC chips are usually implemented with off-the-shelf processor IPs. Even for those SoC chips, our method can accurately model the energy consumption since it does not need to know the detailed internal architecture of the target processor. Another key point of our method is that it works very well even with a cycle-*inaccurate* simulator like a GNU debugger which is a de facto standard of software debugger.

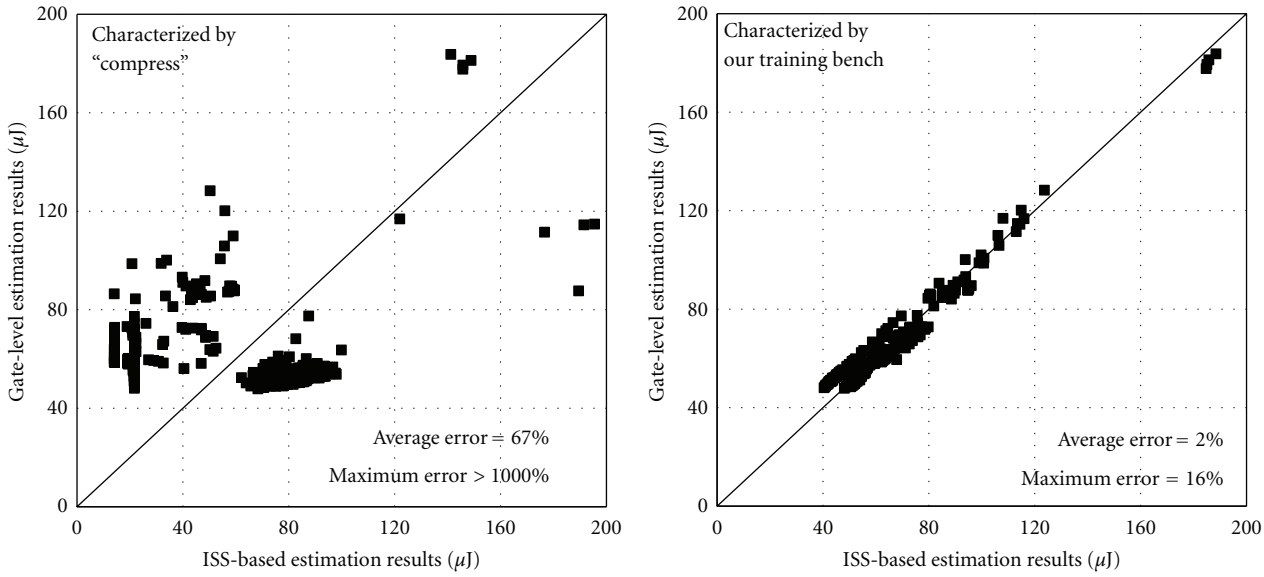


FIGURE 4: Impact of training bench selection.

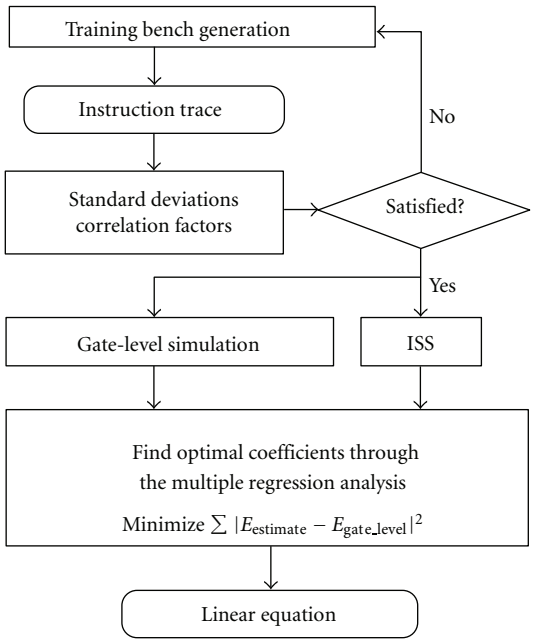


FIGURE 5: Training bench generation.

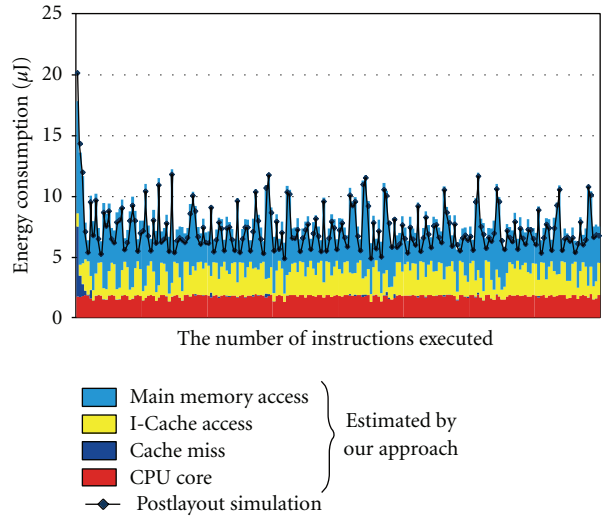


FIGURE 6: Results for MPEG2 encoder run on MeP.

This helps compilers or programmers to customize software codes to meet customers' needs for low power.

3. Memory Energy Reduction

SPMs are small on-chip memories which are much faster and consume much less energy than off-chip memories. Compared to a cache, an SPM requires an explicit software management but is more deterministic and consumes significantly less energy. In this context, two fully software techniques are presented which place into these SPMs the code and the data that are the most often accessed. Both techniques are then merged to achieve a more general and efficient management of the SPM.

The first technique, detailed in [24], is applied on single-task applications and assigns frames of the stack to the data SPM. This is motivated by the fact that the stack is usually one of the most often accessed data memory objects. For instance, with the MiBench [25] benchmark suite, stack accesses represent about 60% of the total data memory accesses. The second technique, detailed in [26], shares the spaces of the code and the data SPMs among the static memory objects of several tasks. This second technique is then merged with the stack technique for an efficient usage

of the SPMs with the majority of the memory objects, that is, the code, the static data, and the stack.

Both techniques utilize profile information from the target application for generating integer linear programming (ILP) formulations whose objectives to minimize model the energy consumptions related to memory accesses. For each technique, the solutions of the formulations give the optimal configurations of the respective managements for each SPM.

3.1. Related Work. Several works use the SPM for reducing the energy consumption of memory accesses.

Usually, only the static data and the code memory objects are considered since they are allocated at compile time. Some techniques decide at compile time which of those objects are to place in the SPM, and which are to be left in the main memory (MM) [27–37]. In [38] the previous approaches are extended with the possibility to split some arrays so that only their often accessed parts are placed into the SPM. Recent works like [39] also consider the case where the size of the SPM is not known at compile time: they propose a technique which modify the code at run time, during an initialization phase, in order to take advantage of the discovered SPM size.

Dynamic memory objects like the stack and the heap are less often considered. Nevertheless, a few methods do exist for managing the stack between the SPM and the MM [40–43]. The main idea is to place in the SPM the most frequently accessed stack variables while leaving the others into the MM. Some of them require specific hardware features like [42, 43] which use a memory management unit (MMU) or [44] which presents an SPM controller device. Embedded systems with tight constraints often cannot afford such features, hence fully software methods like [24, 40, 41] are required. In [40], it is proposed to manage dynamically a circular buffer of stack frames inside the SPM. While interesting, especially because it supports recursive functions, this approach is suboptimal. By contrast [41] proposes to find at compile time an optimal allocation for the stack variables. Two levels of granularity are studied: the frame level, where the stack frames are considered monolithically, and the variable level where the stack variables are considered independently of each other. The implementation of the variable is questionable though, since when compiled, the local variables are usually assigned to registers, and the content of the frames corresponds actually to the spill code. This is why in [24] we prefer using the frame level only and get optimization opportunities by supporting moves of a part or the totality of frames during the execution of the application.

The heap remains the most difficult memory object to manage and place in an SPM. Usually, prefetch and additional hardware are the solutions [43, 44]. Among the recent works, [45] proposes a simple API for a semiautomatic management of the heap between the SPM and the MM for the case of a CELL processor.

Techniques which move at run time the objects between the SPM and the MM like [24, 43, 46–50] are complementary with the compile-time ones which can improve the usage of the limited SPM space. They can support code memory object [46, 50], static memory objects [48, 49], the stack [24], or all the memory objects [43]. However, these techniques have an overhead for moving the memory objects, which can surpass the gain of using the SPM. This overhead can be bounded though, when the displacement of the memory objects is fixed at compile time as it is the case with [24, 46, 48–50].

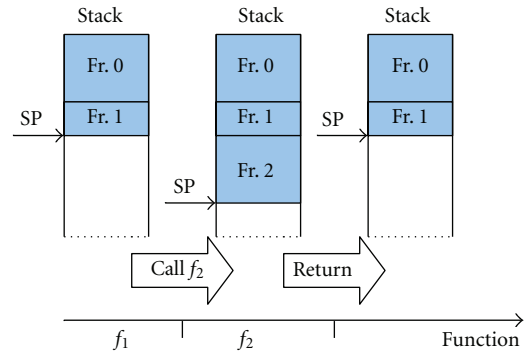


Figure 7: Standard management of the stack.

In multitask environments, the SPM is to be shared among several tasks for the placement of their memory objects. Several techniques exist for performing this sharing for static memory objects, that is, code and static variables [31–37]. Although they target multitask systems, some approaches like [31, 35] do not use the scheduling as a parameter for optimizing the usage of the SPM. Other approaches are targeting systems with a static scheduling [32, 33] or assume the knowledge at compile time of the fully execution flow of the tasks [34]. Finally, a few approaches target preemptive systems, which is also the target system in this paper. For instance, [26, 36, 37] optimize the SPM usage while using the properties of a static priority-based real-time preemptive system.

3.2. Stack Placement and Management. Contrary to the static memory objects (i.e., code or static variables), the size of the stack changes during the execution of the application. More precisely, when a function starts its execution, it allocates a new frame on top of the stack by decreasing the stack pointer register (SP). This frame is used for placing the function's local variables which are not assigned to registers. Eventually, just before returning, the function destroys this frame by increasing SP. Figure 7 illustrates this mechanism by giving the evolution of the content of the SPM during the execution of function f_1 which calls f_2, another function. A consequence is that the stack can grow larger than the SPM during the execution of the application. Furthermore, as frames are usually not equally accessed it may be inadequate to assign all of them to the SPM as the space they use could be more efficiently used by other memory objects.

In this context, the technique presented here manages two substacks, one into the MM and one into the SPM. The frequently accessed frames are allocated on top of the SPM substack while the others are allocated on top of the MM substack. The technique also supports moves of a part or the totality of frames during the execution of the application in order to free space from the SPM to be used by further

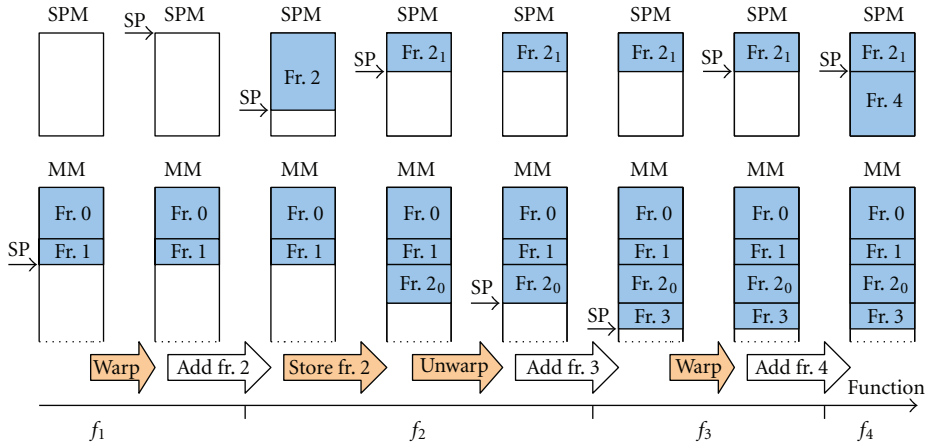


Figure 8: Proposed management with two substacks.

frames. It is this technique which is briefly presented in this section.

3.2.1. Overview of the Technique. The proposed management with two substacks is performed by new stack operations which are inserted at compile time into the assembly code of the application. For that purpose, four new stack operations are defined:

warp: translates SP to the top of the SPM substack,

unwarp: translates SP to the top of the MM substack,

store: copies a part or the totality of the top frame of the SPM substack to the top of the MM substack,

load: copies a part or the totality of the top frame of the MM substack to the top of the SPM substack.

Warp and unwarp operations are to be inserted just before and just after the call instructions where the substack of the coming frame is to be different from the current top one. This is enough because it is only when entering and leaving a function that the state of the stack changes. A store operation is to be inserted just before a call instruction (or if present, just before the unwarp operation preceding the call) when the top frame is to be moved (partly or fully) from the SPM to the MM. It is not necessary to insert store operations in other places in the program because it is only there that the stack grows. Symmetrically, a load operation is to be inserted just after a call instruction (or if present, just after the warp operation following the call instruction) when the top frame is to restore back to the SPM.

Figure 8 illustrates the proposed management with the new stack operations. In the figure, the evolution of the content of the SPM is shown while four functions, respectively, f_1, f_2, f_3, and f_4 are executed. Initially, frames 0 and 1 are assigned to the MM. Then a warp operation in f_1 allows to allocate frame 2 to the SPM. A store operation on frame 2 follows, executed during f_2, which frees some SPM space. This space is not immediately required since frame 3 is allocated to the MM but this space is indeed used by frame 4. This possibility to store a frame ahead of the need is the reason why the store and the load operations have been limited to act only on the respective top frames of the MM and the SPM.

3.2.2. Limitations. 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 into 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, and 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. Consequently, implementing a variable level approach like [41] is, at least, very difficult in practice.

This is why the technique proposed in this paper requires that, when accessed, a frame must be either fully into the SPM or fully into the MM. Furthermore, when a frame is stored it needs to be loaded symmetrically unless it is not accessed before the next call.

3.2.3. Difficulties. Even though the technique is simple, several cases require additional care. First, when the arguments of a function are too numerous, some of them cannot be assigned to registers and are instead placed into the caller's frame. Then they are accessed by the called function but relatively to its *own* frame. This requires both frames to be contiguous; that is, they cannot be assigned into different substacks. However, [24] can allow such frames to be in different substacks by moving the corresponding stack operations before the instructions which put the extra arguments into the stack.

Second, when references to the current frame are passed as arguments to another function, the frame cannot be stored since it would invalidate the reference.

Third, the user often cannot modify the code of library functions, thence cannot insert into them new stack operations. However, it is still possible to assign their frames to the SPM by considering each call to such a library function as a call to a large monolithic function whose frame size is the maximum stack requirement during its execution.

Fourth, when a function f is called from different points into the program, it can happen that its frame is assigned to different substacks depending on where it has been called from. This is not a problem yet as the substack is selected before calling f. However, if f calls another function g, g's frame too might be assigned to different substacks depending on where f has been called from. Thence, depending on where f has been called from, different stack operations might be required to be executed during f. A simple solution is to forbid stack operations for this frame. A more refined solution is presented in [24] where predicates computed from the return address of f are inserted for selecting the stack operations to execute.

Last, recursive functions are difficult to handle since a same frame can be assigned several times in sequence. If the number of recursions is known at compile time, a single large frame which will contain all the frames of the recursion can be assigned. Otherwise, the technique cannot support them and they must be assigned to the MM substack. In [24] a more efficient solution is described which allocates on top of the SPM substack a circular buffer (already presented in [40]) for the frames of a recursive function. The code of the function is then updated for managing this buffer so that, when full, its oldest frame is evicted to the MM.

3.2.4. Compile-Time Selection of the Stack Operations. The stack operations to insert are decided through an ILP whose objective to minimize models the energy consumed while accessing the stack and the extra energy consumed by the stack operations. The constraints ensure that the inserted operations maintain a valid state for MM and the SPM substacks.

For that purpose, a history of the calls is required to determine which frames exist at each point of the program where new stack operations can be inserted. This history is built from the call and control flow graph CCFG. This graph is partitioned into subgraphs we call *sessions*. Sessions are separated by the call instructions and are of two types: a *store session* is the subgraph of the CCFG starting just before a call instruction, and a *load session* is the subgraph of the CCFG starting just after a call instruction. By definition, a store operation can be inserted at the beginning of a store session and a load operation at the beginning of a load session.

The ILP formulation can then be built using the frames (u) and the sessions (v) for indexing the variables and the constants. The variables of the formulation are there to control the insertion of the stack operations and are the following:

$x_{u,v}$: this binary variable is 1 when frame u is fully into the SPM during session v;

$xn_{u,v}$: this integer variable is the number of bytes of frame u into the SPM during session v;

sn_v: this integer variable is the number of bytes stored at the beginning of session v;

ln_v: this integer variable is the number of bytes loaded at the beginning of session v.

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

S_{stack}: is the maximum size of the SPM substack;

$C\text{spm}_{u,v}$: it is the energy cost of the total accesses to frame u during the whole executions of session v if it is in the SPM;

$C\text{mm}_{u,v}$: it is the energy cost of the total accesses to frame u during the whole executions of session v if it is in the MM;

S_u: is the size of frame u;

C_{st}: is the energy cost for storing one byte;

C_{ld}: is the energy cost for loading one byte;

Ne_v: it is the number of executions for session v;

$A_{u,v}$: indicates if frame u is accessed during session v;

Forb_v: indicates if the load or the store operation of session v is forbidden;

Sym_v: indicates if the load operation of session v must be symmetric with the corresponding store operation.

The objective function is the sum of two parts: the first one, noted $\text{obj}_{\text{stack}}$, represents the energy consumed while accessing the stack, and the second one, noted obj_{ops}, represents the energy consumed by the stack operations inserted into the assembly code for managing the stack between the SPM and the MM.

The first part of the objective function is the sum of the energy consumed by the application during each of its sessions when accessing the stack. For each frame, the energy consumed depends on whether it is assigned to the MM or to the SPM. The $x_{u,v}$ variables are used for selecting the cost corresponding to the used memory, and $\text{obj}_{\text{stack}}$ is then computed as follows:

$$\text{obj}_{\text{stack}} = ((C\text{spm}_{u,v} - C\text{mm}_{u,v}) * x_{u,v}). \tag{2}$$

In the above equation, in order to keep the linearity of the objective, only the difference in energy consumption between the accesses to the MM and the SPM is actually represented.

The second part of the objective function is the sum of the store and the load costs (the warp and unwarp operations are neglected in this paper, please refer to [24] for details about how to take them into account):

$$\text{obj}_{\text{ops}} = (\text{Ne}_v * (C_{\text{st}} * sn_{u,v} + C_{\text{ld}} * ln_{u,v})). \tag{3}$$

Since the size of the SPM substack is bounded, constraints are required for limiting the assignment of frames to it. There is one such constraint per session:

$$\forall v \quad xn_{u,v} \leq S_{\text{stack}}. \tag{4}$$

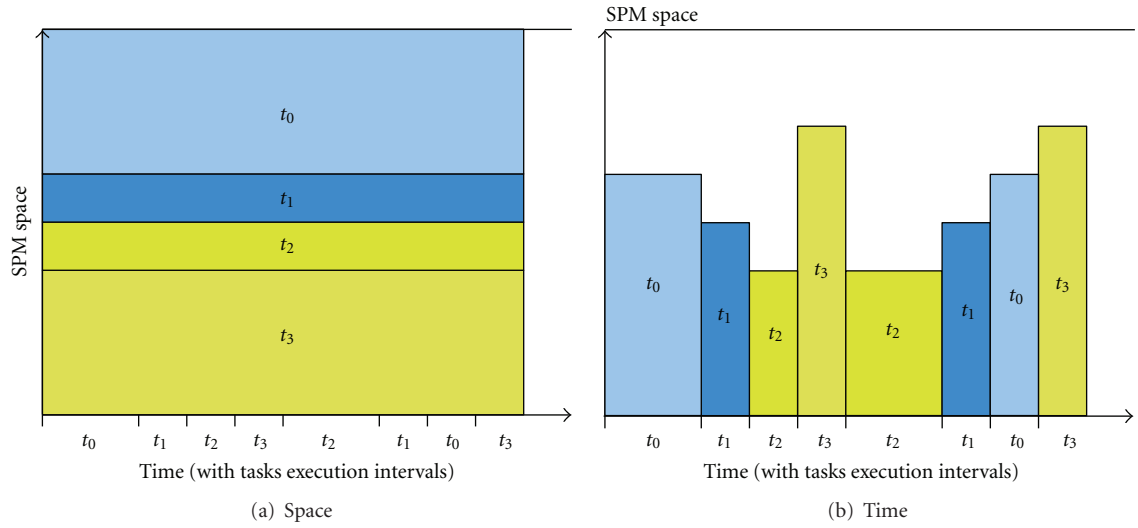


Figure 9: Basic SPM sharing among tasks.

When accessed during a session v, a frame u is required to be fully into the MM or fully into the SPM, which means that $xn_{u,v}$ must be either 0 or the size of the frame. This is ensured by the following constraint:

$$\forall u,\ v \quad A_{u,v} \Longrightarrow xn_{u,v} = x_{u,v} * S_u. \tag{5}$$

The load and store operations modify the size of current frame's part into the SPM, hence, they are constrained as follows where Store and Load are, respectively, the sets of the store and the load sessions and Prev(v) is the set of the sessions preceding v:

$$\begin{aligned} &\forall u \quad \forall v \in \text{Store} \quad \forall w \in \text{Prev}(v) \quad xn_{u,v} = xn_{u,w} - sn_{u,v} \\ &\forall u \quad \forall v \in \text{Load} \quad \forall w \in \text{Prev}(v) \quad xn_{u,v} = xn_{u,w} + ln_{u,v}. \end{aligned} \tag{6}$$

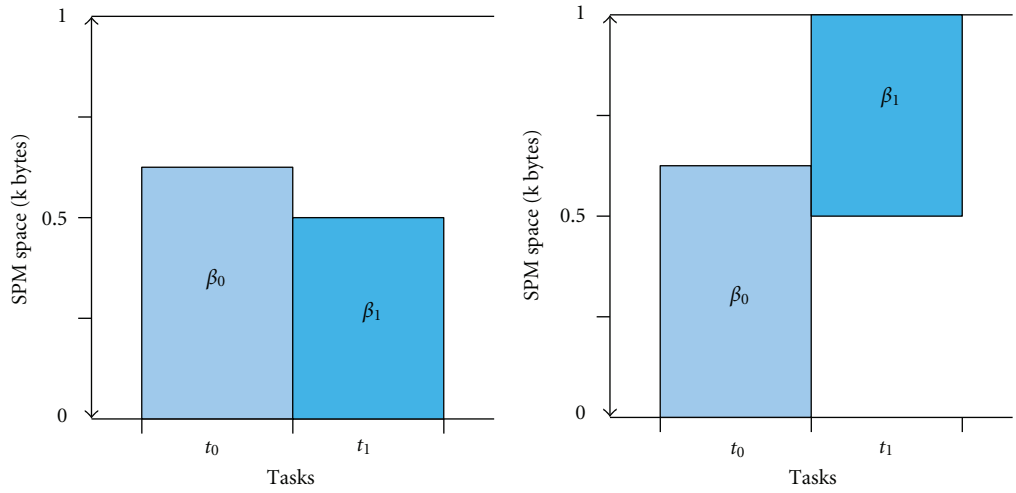
These operations are not always possible the validity of the assembly update can forbid them or force them to be symmetric. Both kind of limitations are converted to the following respective constraints where Call(v) is the session from which the function of frame u has been called:

$$\begin{aligned} &\forall u \quad \forall v \in \text{Store} \quad \text{Forb}_{u,v} \Longrightarrow sn_{u,v} = 0, \\ &\forall u \quad \forall v \in \text{Load} \quad \text{Forb}_{u,v} \Longrightarrow ln_{u,v} = 0, \\ &\forall u, v \quad \text{Sym}_{u,v} \Longrightarrow ln_{u,v} = sn_{u,\text{Call}(v)}. \end{aligned} \tag{7}$$

3.3. Code and Static Data Placement with SPM Sharing among Several Tasks. In multitask environments, the SPM is to be shared among several tasks when placing their memory objects. The sharing can be done among two dimensions [35, 37]: spatial and temporal. With the spatial sharing, each task is provided with an exclusive access to a part of the SPM for placing its memory objects. This sharing does not require any run-time processing, but each task has access to only a small part of the SPM. Figure 9(a) illustrates this sharing: whatever the executed task may be (i.e., t_0, t_1, t_2, and t_3), the assignments to the SPM space do not change. On the contrary, the temporal sharing provides each task the totality of the SPM space as seen in Figure 9(b). Consequently, this second sharing dimension requires to change the content of the SPM at each context switch: when a task is preempted, this content must be first saved to the MM (if it has been modified), then the content corresponding to the next active task must be restored from the MM to the SPM. These copies require time and consume energy which reduces the efficiency of this sharing. Works in [35, 37] presented hybrid approaches which use both sharing dimensions in order to achieve better results. Both approaches reserve a part of the SPM for the spatial sharing and the rest for the temporal sharing. In [26] we proposed an approach which also uses these dimensions but in a more uniform fashion since the full space of the SPM is used for both. It is this approach which is described in this section.

3.3.1. Assumptions. As there is one code SPM and one data SPM in the target architecture, the technique is applied twice, once for the code and once for the data. In addition, when a task is preempted, the SPM management needs to save the parts of its memory objects assigned to the SPM which will be overwritten by other tasks only if they have been modified. For the code regions, it is assumed here they are never modified (which is often the case in practice for embedded systems). For the data regions, it is difficult to know if they have been modified or not. Hence, it is assumed here that they are always modified.

3.3.2. Overview of the Technique. The proposed technique assigns to each task a block in the SPM for placing its memory objects (code or data depending on the target SPM). Blocks can overlap but then, these overlap regions must be updated at context switches. In order to reduce the cost of such updates, the addresses of the blocks within the SPM are computed for reducing the overlaps among the blocks which often interfere. Figure 10 shows how the addresses of the



(a) Large overlap (b) Small overlap

Figure 10: Effect of the blocks' address on overlaps.

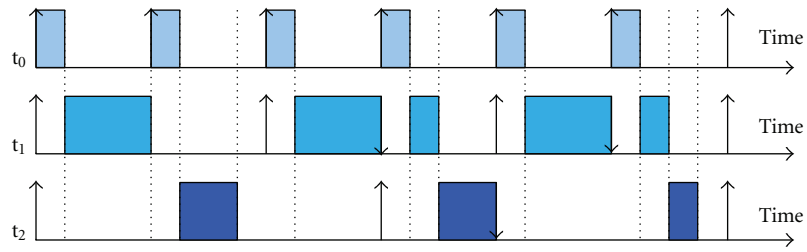


Figure 11: Example of scheduling with three tasks.

blocks change the overlaps' sizes between them for the case of two tasks. In Figure 10(a), both blocks have the same address and the overlap region is the totality of block β_1 whereas, in Figure 10(b), the overlap is minimized to 128 bytes.

3.3.3. Importance of the Scheduling. The scheduling is an important parameter for reducing the number of copies between the SPM and the MM during the context switches. For instance, if a task t_0 is never active while another task t_1 is ready, their respective blocks can overlap without any corresponding copy being required during the context switches for updating the content of the SPM. However if t_0 is active the overlap region between its block and t_1s will have to be saved to (if data) and restored from the MM. In order to take the scheduling into account, the following is defined.

> Preemption pattern: it is the set of the tasks executed between two consecutive active states of a ready task called the reference task.

Figure 11 gives an example of scheduling and the corresponding preemption patterns are given with their number of occurrence in Table 1. In the figure, the up arrows represent the firing of a task and the down ones represent preemptions.

During the execution of a given preemption pattern, the parts of the reference task's block to save (if data) and replace

Table 1: Example of preemption patterns.

Pattern	Reference task	Executed tasks	Occurrences
p_0	t_1	$\{t_0\}$	2
p_1	t_2	$\{t_0,t_1\}$	1

are the ones in overlap with blocks of the tasks of the pattern. When several overlaps of a pattern intersect, the common parts must still be counted only once for the corresponding context-switches update costs. Figure 12 illustrates such a case where $\delta_{0,1}$ is the overlap region between β_0 and β_1, but is also included into $\delta_{0,2}$ which is the overlap region between β_0 and β_2.

As an overlap region needs to be copied to the MM only once for a given pattern occurrence, $\delta_{0,1}$ should not be counted when computing the context switch cost. For that purpose we define the following.

> Effective overlap: it is an overlap region whose context switch cost is counted for a given scheduling pattern.

For a given pattern, the effective overlaps can be considered as the overlap regions which are not included into other ones obtained from other executed tasks in the pattern as it is shown in [26]. However, it can still happen that part of

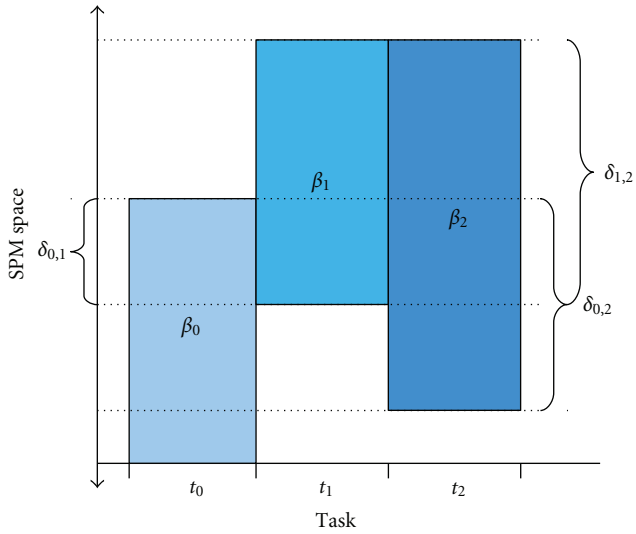


Figure 12: Intersection of overlaps.

overlaps are still over counted. For simplicity, these last over counts are neglected in this paper but [26] do take them into account.

3.3.4. Memory Objects Placement and Task's Blocks Computation at Compile Time. An ILP whose objective models the energy consumption related to the static memory objects' accesses is used for selecting for each task which memory object is to put into the SPM and for selecting the address of its SPM block. The variables and the constants of the formulation are indexed with i for the preemption patterns, j for the tasks and their respective blocks, and k for the memory objects. The variables are the following:

$x_{j,k}$: this binary variable is 1 when memory object k of task j is in the SPM when this task is active;

b_j: this integer variable gives the starting ("begin") address of block j;

e_j: this integer variable gives the ending ("end") address of block j;

$o_{j,j'}$: this integer variable is the size in bytes of the overlap between blocks j and j';

$o^{\text{sel}}_{j,j'}$: this binary variable is used for selecting the constraint defining the overlap $o_{j,j'}$;

$oe_{i,j,j'}$: this integer variable is the size in bytes of the effective overlap between blocks j and j' for pattern i;

$o^{\min}_{i,j,j'}$: this binary variable is used for selecting the constraint defining the effective overlap $\text{oe}_{i,j,j'}$.

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

Sspm: it is the total size of the (code or data) SPM;

R_i: it is the total number of occurrences for pattern i;

$C\text{spm}_{j,k}$: it is the energy cost of the total accesses to memory object k of task j if the object is in the SPM. This cost also includes the copy required when a task is fired;

$C\text{mm}_{j,k}$: it is the energy cost of the total accesses to memory object k of task j if the object is in the MM (which can be cached or not);

S_k: it is the size of the memory object k;

Ccxt: it is the average energy spent in context switches for one byte of data or code. When the target is the data SPM, this cost is the average energy required for the saving and restoring of one byte of data and when the target is the code SPM, this cost is the one required for restoring one byte of code.

The objective function is the sum of two parts: the first one, noted $\text{obj}_{\text{tasks}}$, represents the energy consumed by the tasks while accessing the memories. The second one, noted obj_{cxt}, represents the energy consumed while updating the SPM during the context switches.

The first part of the objective function is the sum of the energy consumed by each task while accessing its memory objects. For each memory object, the energy consumed depends on whether it is assigned to the MM or to the SPM. The $x_{j,k}$ variables are used for selecting the cost corresponding to the used memory, and $\text{obj}_{\text{tasks}}$ is then computed as follows:

$$\text{obj}_{\text{tasks}} = \left(C\text{spm}_{j,k} - C\text{mm}_{j,k}\right) * x_{j,k}. \tag{8}$$

In the above equation, in order to keep the linearity of the objective, only the difference in energy consumption between the accesses to the MM and the SPM is actually represented.

The second part of the objective function is the sum of the energy consumed by each scheduling pattern for saving and restoring parts of blocks pondered by their corresponding numbers of occurrences. For that purpose, the effective overlap $oe_{i,j,j'}$ are used:

$$\text{obj}_{\text{cxt}} = C\text{cxt} * \left(R_i * \left(oe_{i,j,j'}\right)\right). \tag{9}$$

The first constraints ensure that each block fits individually into the SPM. For that purpose the size of a block is computed as the sum of the sizes of each memory object of the corresponding task which are assigned to the SPM:

$$\forall i, j \quad e_j - b_j = x_{j,k} * S_k. \tag{10}$$

Finally, a block is forced to fit into the SPM by bounding its upper address as follows (for LP-solvers variables are positive by default, hence this is not necessary to bound b_j):

$$\forall j \quad e_j < S_{\text{spm}}. \tag{11}$$

Additional constraints are required in order to compute the effective overlaps. First of all, the (noneffective) overlap between two blocks, j, and j', is computed by considering the four cases of their relative positions when intersecting,

that is, j included into j', j's ending address included into j', j''s ending address included into j and j' included into j. The cases are select through the four mutually exclusive binary variables, $o^{\text{sel}}_{j,j}$, $o^{\text{sel}}_{j,j'}$, $o^{\text{sel}}_{j',j}$, $o^{\text{sel}}_{j',j'}$ as follows:

$$\begin{aligned} o_{j,j'} &\geq e_j - b_j - M_{j,j} * \left(1 - o^{\text{sel}}_{j,j}\right), \\ o_{j,j'} &\geq e_j - b_{j'} - M_{j,j'} * \left(1 - o^{\text{sel}}_{j,j'}\right), \\ o_{j,j'} &\geq e_{j'} - b_j - M_{j,j',j} * \left(1 - o^{\text{sel}}_{j',j}\right), \\ o_{j,j'} &\geq e_{j'} - b_{j'} - M_{j',j'} * \left(1 - o^{\text{sel}}_{j',j'}\right), \\ o^{\text{sel}}_{j,j} + o^{\text{sel}}_{j',j} &+ o^{\text{sel}}_{j',j} + o^{\text{sel}}_{j',j'} = 1\ . \end{aligned} \tag{12}$$

In the above equations the constants $M_{*,*}$ are used as "big Ms": they have to be large enough to ensure the right part of each equality to be negative when the corresponding $o^{\text{sel}}_{*,*}$ variable is 0. When there is no overlap, these equations are still valid as then, either $e_j - b_{j'}$ or $e_{j'} - b_j$ is negative so that $o_{j,j'}$ will be set to 0.

These overlaps are used for computing the effective overlaps as described previously. The formulation of each effective overlap $oe_{i,j,j'}$ is expressed using a binary variable $o^{\text{min}}_{i,j,j'}$ which is 1 when $o_{j,j'}$ is smaller than $o_{j',j''}$ and $o_{j,j''}$ and therefore included into them [26] (constants $M^{\text{min}}_{*,*,*}$ and $Me_{*,*}$ are also "big Ms"):

$$\begin{aligned} \left(1 - o^{\text{min}}_{i,j,j'}\right) * M^{\text{min}}_{j,j',j''} &\geq o_{j,j'} - o_{j,j''} + \epsilon_{j,j'} - \epsilon_{j,j''}, \\ \left(1 - o^{\text{min}}_{i,j,j'}\right) * M^{\text{min}}_{j,j',j''} &\geq o_{j,j'} - o_{j',j''} + \epsilon_{j,j'} - \epsilon_{j',j''}, \end{aligned} \tag{13}$$

$$oe_{i,j,j'} \geq o_{j,j'} - Me_{j,j'} * o^{\text{min}}_{i,j,j'}. \tag{14}$$

In the previous equations, constants $\epsilon_{*,*}$ ensure that at least one effective overlap is not set to 0 when several overlaps are equal. These positive constants are strictly smaller than 1 and are all different from one another. This artificially forces the subtraction of overlaps of (13) to be different from 0.

3.3.5. Run-Time Management of the SPM. The content of the SPM must be updated at each context-switch. For that purpose a function is added to the context-switch procedure of the operating system. This function uses two tables for determining which region of the SPM is to save (if data) and replace. These tables describe the content of the SPM partitioned into the set of the areas which are separated by the starting and ending addresses of the blocks. Figure 13 gives an example of such a partition where the blocks are noted β_* and the areas α_*.

By definition, areas are contiguous to each other, do not overlap and each block is a set of consecutive areas. The first table, fixed at compile time, is indexed by the blocks and its two entries are the first and the last of the areas which are covered by the block. The second table evolves at run time and is indexed by the areas. Its entries are for each area, its start address in the SPM, its size, and the block currently occupying it. At each context-switch, the SPM update function iterates on the areas (fetched from the first table) of the next active task. For each of these areas, the second table is looked up for determining which block is occupying it. If it is a block different from the one of the next active task, the area is saved to the MM (if data) and the corresponding content of the new block is copied back to the SPM.

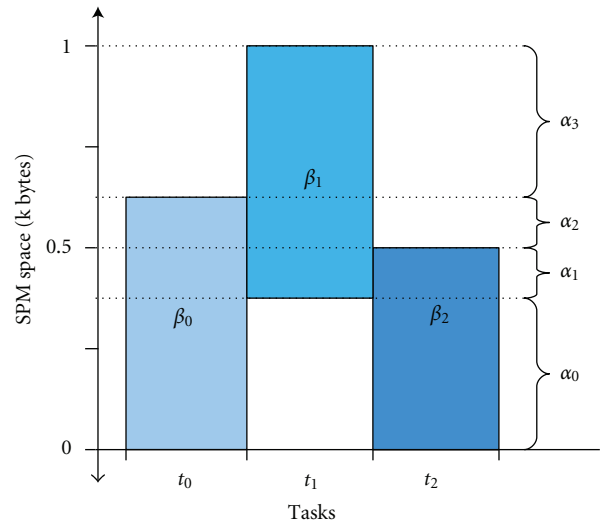


Figure 13: SPM partitioned into areas.

3.4. SPM with Code, Static, and Stack Data in Multitask Systems. Since the stacks are dynamic memory objects, they cannot be handled natively by the proposed technique for sharing the SPM among several tasks (nor, to our knowledge, by any of the existing sharing techniques). Yet, if the maximum size of the stacks can be known at compile time, they can still be considered as usual static objects whose size is this maximum.

The same can be done with the proposed stack management technique by considering each SPM substack as a static memory object whose access cost is the resulting value of the corresponding objective function and whose size is chosen small enough to leave room into the SPM for the other data memory objects. However, the efficiency of the stack management depends highly on the size of the SPM substack as it can be seen from the results of the next section. Fortunately, solving the ILP formulation for this technique proved to be very fast (less than one second for all the applications of our experiments) and reasonably fast for the case for the SPM sharing technique (15 seconds were required for the worst cases with [51]). But the number of iteration to perform is exponential with the number of sizes for the SPM stacks. For instance, with five SPM substack sizes and nine tasks, 9^5 iterations are required. It is difficult in practice to afford such a huge number of iterations of both techniques for finding the best sizes for each SPM substack. Yet, since it is affordable to iterate several times with the stack technique alone, we propose to extend the SPM sharing technique by adding several substacks of different size per task. Each of these substacks is considered as a static memory object whose access cost is the value of the

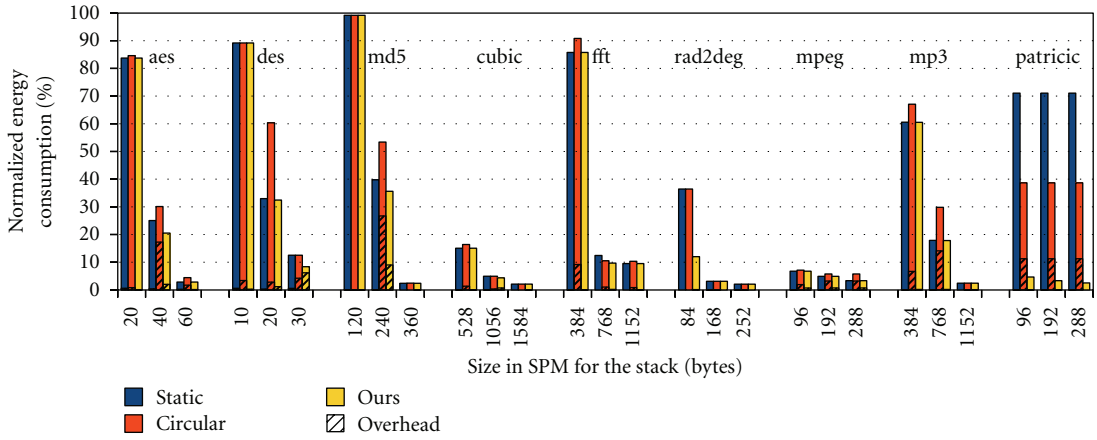


Figure 14

objective function of the respective stack management's ILP formulation. Equation (8) is therefore modified as follows:

$$\text{obj}_{\text{tasks}} = \left(\text{Cspm}_{j,k} - \text{Cmm}_{j,k}\right) * x_{j,k} + C_{\text{stk},m} * x_{\text{stk},j,m}. \tag{15}$$

In the equation, m is the index for the SPM substack case and $C_{\text{stk},m}$ is its corresponding energy cost. Besides, variable $x_{\text{stk},j,m}$ is 1 when the SPM substack case m is used and 0 otherwise.

As one and only one SPM substack is required for each task, the following new constraints are added:

$$\forall j \quad x_{\text{stk},j,m} = 1. \tag{16}$$

Finally, the sizes of the substacks must also be taken into account and (10) becomes

$$e_j - b_j = x_{j,k} * S_k + S_{\text{stk},m} * x_{\text{stk},j,m}. \tag{17}$$

As a result, the solution of this new formulation includes for the content of each block the selected SPM substack in addition to the static memory objects.

While the number of variables increases, the constraints of (16) are efficiently used by LP-solvers so that the solving time is not importantly increased: it took 20 seconds for our LP-solver [51] to solve the problem with the largest task set of our experiments.

3.5. Experimental Results. We applied our technique on a multiperformance processor (based on the MeP) which includes an 8 KB data SPM, an 8 KB code SPM, and a 4-way instruction cache. Energy characteristics of this architecture are given in Table 2.

We used the Toshiba's MeP Integrator (MPI) tool chain [52] for compiling and simulating the applications. Compilations were performed with the −02 level of optimization. Executions were performed for a static priority-based rate monotonic preemptive system with a processor utilization of about 60%. Both multitask and stack techniques were applied on the tasks of the sets given in Table 3. Tasks come from the EEMBC [53] and the Mibench [25] benchmarks.

Table 2: Average energy consumptions for The multiperformance processor.

Memory access type	Energy (nJ per word)
I-SPM read	0.3774
I-Cache hit	1.433
I-Cache miss	57.897
D-SPM read	0.3774
D-SPM write	0.5053
SDRAM data read	42.6466
SDRAM data write	18.3986

Table 3: Tasks-set used for the experiments.

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

Figure 14 gives the results of the proposed stack management technique (*ours*) applied on monotask cases compared to a management which does not store nor load frames (*static*) [42] and to one which uses a circular buffer of frames (*circular*) [40]. As ours supersets static and circular, it is not surprising that it always achieves better results than them. On average, for the respective 1/3 and 2/3 SPM substack sizes, static achieves about 39% and 84% of stack-related energy consumption reduction, circular about 41% and 74%, and ours does better in both cases with about 49% and 85%.

The variable level of granularity mentioned by [41] 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 run-time partial loads and stores whose overhead can be compared favorably with the variable-level overhead for accessing the dispatched variables of a frame. However, for the cases where a frame is larger than the remaining SPM space, even after store

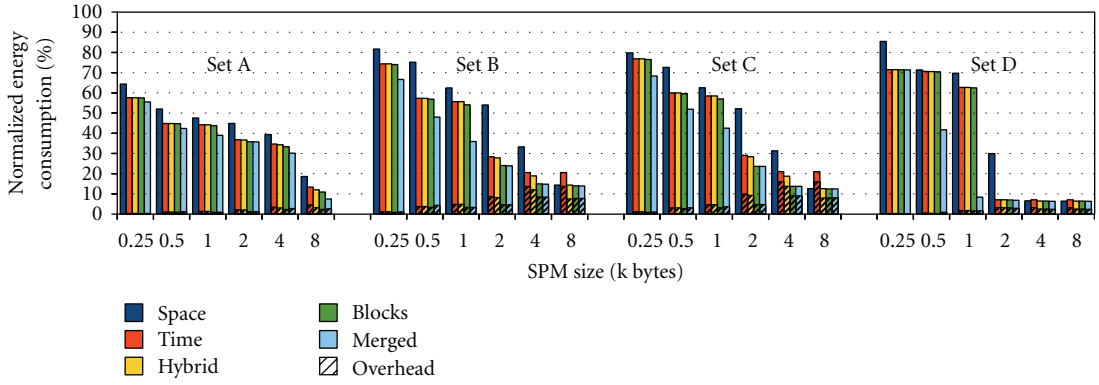


Figure 15

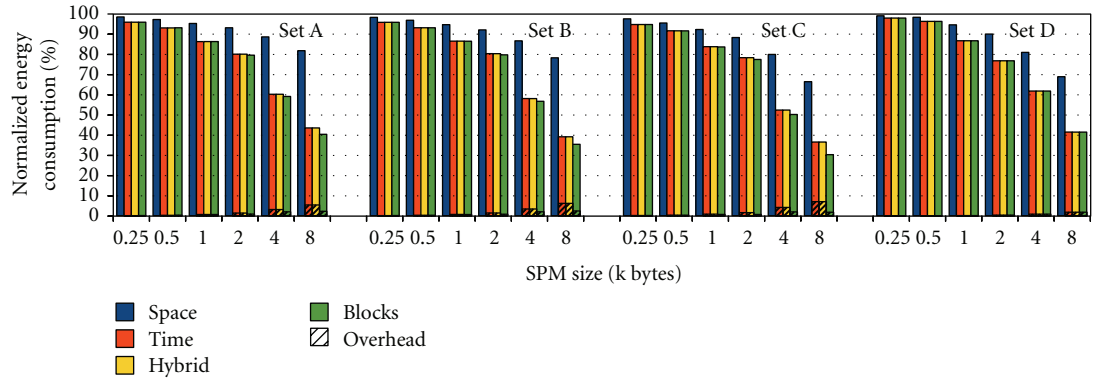


Figure 16

operations, there would indeed be more energy reduction if the often accessed variables of such a frame could be separated from the others at reasonable cost.

Figure 15 gives the results obtained when sharing the data SPM among the tasks of each set of Table 3 with the technique proposed in this paper (*block*) and with state-of-the-art techniques including a fully off-line spatial sharing one (*space*), a temporal sharing one (*time*) (both are described in [35]), and a hybrid spatial and temporal one (*hybrid*) [37]. The figure also includes the results obtained when the sizes of the SPM substacks are selected by the merging technique presented in Section 3.4 (*merged*). For fairness of the comparison, the techniques other than merged also include the stacks, but considered as static memory objects whose sizes are the corresponding maximum stacks' requirements. For the case of a 1 KB SPM, on average, space achieves about 40% of energy consumption reduction, time about 45% hybrid about 45%, block about 46%, and merged achieves much better with about 69%. If the SPM size is 4 KB, these techniques achieve, respectively, about 72%, 79%, 81%, 83%, and 84% of average energy reduction. In all the cases, merged achieves better or equal than block, block better or equal than hybrid, and hybrid better or equal than time and space. The systematic advantage of hybrid, block, and merged over space and time is due to the fact that the two latter are part of the solutions explored by the three former techniques. Moreover these three techniques also include the overhead in their ILP formulations so that it remains under control.

Figure 16 gives the results of the same experiments for the code SPM (the MM is then cached), with space, time, hybrid, and block (for the code, merged is identical to block). As the cache consumes much less energy than the external memory, more SPM is required for achieving important energy consumption reduction. For the case of a 4 KB SPM, on average, space achieves about 16% of energy consumption reduction, time about 42%, hybrid about 42%, and block about 43%. And for the case of an 8 KB SPM, these techniques achieve, respectively, about 26%, 60%, 60%, and 63%.

While not being the focus of the paper, the execution speed also benefits from the proposed techniques because accessing the MM requires much more time than accessing the SPM. With the processor configuration used for the experiments, accessing the MM requires about 20 cycles on average and there is no data cache. Therefore, the speedup related to the data memory accesses is significant. For instance, when the SPM size was 2/3 of what the stack required, the proposed stack management technique (ours) achieves 77% of speedup related to the memory accesses (including the overhead of the management). Similarly, the proposed technique for sharing the SPM among tasks (block) achieves about 80% of data-access-related speedup (including the overhead) with a 4 KB SPM. For the code

accesses, there is a cache though, but using the SPM reduces the miss rate so that there is actually a slight gain: 4% of code access-related speedup (including the overhead) with a 4 KB SPM.

4. Conclusions

The paper presented our recent research activities and results on characterizing and reducing the energy consumption in embedded systems. Our main focus is on software-directed approaches for estimating and reducing the energy consumption of embedded real-time systems. As the demands of system integration, performance, and low power operation have pushed chip vendors down to 65 nm or below, NRE (nonrecurring engineering) costs and design complexity have increased significantly. A remedy for the NRE explosion is to reduce the number of developments and sell tens of millions of chips under a fixed hardware design. In such a situation, embedded software plays much more important role than today. This paper covered our approach for fast and accurate energy estimation of software on a given processor system and software-directed techniques for reducing the energy required for accessing the memory subsystems in the embedded processor.

As future work, we plan to place also parts of the heap into the SPM for achieving more energy consumption reduction. We also plan to refine the stack approach proposed in this paper with a grain finer than the frame level while still being implementable in practice.

Acknowledgments

This work is supported by Toshiba Semiconductor and VDEC, the University of Tokyo with the collaboration of Renesas Technology, STARC, Panasonic, NEC Electronics, Toshiba, ROHM, Toppan Printing, Cadence Design Systems, Synopsys, and Mentor Graphics. This work is also supported by JST CREST program and JSPS NEXT program.

References

[1] D. Lee, T. Ishihara, M. Muroyama, H. Yasuura, and F. Fallah, "An energy characterization framework for software-based embedded systems," in *Proceedings of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia (ESTIMEDIA '06)*, pp. 59–64, IEEE Computer Society, Washington, DC, USA, 2006.

[2] J. Flinn and M. Satyanarayanan, "PowerScope: a tool for profiling the energy usage of mobile applications," in *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA '99)*, pp. 2–10, February 1999.

[3] W. R. Hamburgen, D. A. Wallach, M. A. Viredaz et al., "Itsy: stretching the bounds of mobile computing," *IEE Computer*, vol. 34, no. 4, pp. 28–36, 2001.

[4] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: a first step towards software power minimization," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design(ICCAD '94)*, pp. 384–390, November 1994.

[5] M. T. C. Lee, V. Tiwari, S. Malik, and M. Fujita, "Power analysis and low-power scheduling techniques for embedded DSP software," in *Proceedings of the 8th International Symposium on System Synthesis (ISSS '95)*, pp. 110–115, September 1995.

[6] N. Chang, K. Kim, and H. G. Lee, "Cycle-accurate energy consumption measurement and analysis: case study of ARM7TDMI," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED '00)*, pp. 185–190, August 2000.

[7] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto, "A instruction-level functionality-based energy estimation model for 32-bits microprocessors," in *Proceedings of the 37th Design Automation Conference (DAC '00)*, pp. 346–351, June 2000.

[8] A. Sama, M. Balakrishnan, and J. F. M. Theeuwen, "Speeding up power estimation of embedded software," in *Proceedings of the International Symposium on low Power Electronics and Design (ISLPED '00)*, pp. 191–196, July 2000.

[9] A. Sinha and A. P. Chandrakasan, "JouleTrack: a web based tool for software energy profiling," in *Proceedings of the 38th Design Automation Conference (DAC '01)*, pp. 220–225, June 2001.

[10] A. Sinha, N. Ickes, and A. P. Chandrakasan, "Instruction level and operating system profiling for energy exposed software," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, no. 6, pp. 1044–1057, 2003.

[11] M. Sami, D. Sciuto, C. Silvano, and V. Zaccaria, "Instruction-level power estimation for embedded VLIW cores," in *Proceedings of the 18th International Workshop on Hardware/Software Codesign (CODES '00)*, pp. 34–38, May 2000.

[12] Trimaran, http://www.trimaran.org/.

[13] C. T. Hsieh, L. S. Chen, and M. Pedram, "Microprocessor power analysis by labeled simulation," in *Proceedings of the Design, Automation and Test in Europe (DATE '01)*, pp. 182–189, March 2001.

[14] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "Design and use of SimplePower: a cycle-accurate energy estimation tool," in *Proceedings of the 37th Design Automation Conference (DAC '00)*, pp. 340–345, June 2000.

[15] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *Proceedings of the 27th International Symposium on Computer Architecture (ISCA '00)*, pp. 83–94, June 2000.

[16] J. T. Russell and M. F. Jacome, "Software power estimation and optimization for high performance, 32-bit embedded processors," in *Proceedings of the IEEE International Conference on Computer Design (ICCD '98)*, pp. 328–333, October 1998.

[17] S. Steinke, M. Knauer, L. Wehmeyer, and P. Marwedel, "An accurate and fine grain instruction-level energy model supporting software optimizations," in *Proceedings of the Int"l Workshop Power and Timing Modeling, Optimization and Simulation (PATMOS '01)*, pp. 3.2.1–3.2.10, September 2001.

[18] T. Li and L. K. John, "Run-time modeling and estimation of operating system power consumption," in *Proceedings of the Int'l Conference on Measurements and Modeling of Computer Systems*, pp. 160–171, June 2003.

[19] G. Contreras and M. Martonosi, "Power reduction for intel xscaler processors using performance monitoring unit events," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED '05)*, pp. 221–226, August 2005.

[20] W. L. Bircher, M. Valluri, J. Law, and L. K. John, "Runtime identification of microprocessor energy saving opportunities," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED '05)*, pp. 275–280, August 2005.

[21] T. K. Tan, A. Raghunathan, G. Lakshminarayana, and N. K. Jha, "High-level software energy macro-modeling," in *Proceedings of the 38th Design Automation Conference (DAC '01)*, pp. 605–610, June 2001.

[22] C. Zang and T. Ishihara, "An implementation of energy efficient multi-performance processor for real-time applications," in *Proceedings of the 1st International Conference on Green Circuits and Systems (ICGCS '10)*, pp. 211–216, June 2010.

[23] T. Ishihara, S. Yamaguchi, Y. Ishitobi et al., "AMPLE: an adaptive multi-performance processor for low-energy embedded applications," in *Proceedings of the IEE Symposium on Application Specific Processors (SASP '08)*, pp. 83–88, 2008.

[24] L. Gauthier and T. Ishihara, "Optimal stack frame placement and transfer for energy reduction targeting embedded processors with scratch-pad memories," in *Proceedings of the 7th IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia '09)*, pp. 116–125, October 2009.

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

[26] L. Gauthier, T. Ishihara, H. Takase, H. Tomiyama, and H. Takada, "Minimizing inter-task interferences in scratch-pad memory usage for reducing the energy consumption of multitask systems," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '10)*, pp. 157–166, ACM, New York, NY, USA, 2010.

[27] P. R. Panda, N. D. Dutt, and A. Nicolau, "Efficient utilization of scratch-pad memory in embedded processor applications," in *Proceedings of the European Design and Test Conference (EDTC '97)*, pp. 7–11, IEEE Computer Society, Washington, DC, USA, 1997.

[28] J. Sjödin and C. von Platen, "Storage allocation for embedded processors," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '01)*, pp. 15–23, ACM, New York, NY, USA, 2001.

[29] L. Wehmeyer, U. Helmig, and P. Marwedel, "Compiler-optimized usage of partitioned memories," in *Proceedings of the 3rd Workshop on Memory Performance Issue (WMPI '04)*, pp. 114–120, ACM, New York, NY, USA, 2004.

[30] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '02)*, pp. 409–415, IEEE Computer Society, Washington, DC, USA, 2002.

[31] B. Egger, J. Lee, and H. Shin, "Scratchpad memory management in a multitasking environment," in *Proceedings of the 7th International Conference on Embedded Software (EMSOFT '08)*, pp. 265–274, ACM, New York, NY, USA, October 2008.

[32] V. Suhendra, C. Raghavan, and T. Mitra, "Integrated scratchpad memory optimization and task scheduling for MPSoC architectures," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '06)*, pp. 401–410, ACM, New York, NY, USA, October 2006.

[33] V. Suhendra, A. Roychoudhury, and T. Mitra, "Scratchpad allocation for concurrent embedded software," in *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CODES+ISSS '08)*, pp. 37–42, ACM, New York, NY, USA, October 2008.

[34] R. Pyka, C. Fassbach, M. Verma, H. Falk, and P. Marwedel, "Operating system integrated energy aware scratchpad allocation strategies for multiprocess applications," in *Proceedings of the 10th International Workshop on Software & Compilers for Embedded Systems (SCOPES '07)*, pp. 41–50, ACM, New York, NY, USA, 2007.

[35] M. Verma, K. Petzold, L. Wehmeyer, H. Falk, and P. Marwedel, "Scratchpad sharing strategies for multiprocess embedded systems: a first approach," in *Proceedings of the 3rd Workshop on Embedded Systems for Real-Time Multimedia (ESTImedia '05)*, vol. 2005, pp. 115–120, IEEE Computer Society, Los Alamitos, Calif, USA, 2005.

[36] H. Takase, H. Tomiyama, and H. Takada, "Partitioning and allocation of scratch-pad memory in priority-based multi-task systems," *IPSJ Transactions on System LSI Design Methodology*, vol. 2, pp. 180–188, 2009.

[37] T. Hideki, T. Hiroyuki, and T. Hiroaki, "Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems," in *Proceedings of the Design, Automation and Test in Europe (DATE '01)*, pp. 1124–1129, IEEE Computer Society, Washington, DC, USA, 2010.

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

[39] N. Nguyen, A. Dominguez, and R. Barua, "Memory allocation for embedded systems with a compile-time-unknown scratch-pad size," *ACM Transactions in Embedded Computing Systems*, vol. 8, no. 3, pp. 21:1–21:32, 2009.

[40] A. Kannan, A. Shrivastava, A. Pabalkar, and J.-E. Lee, "A software solution for dynamic stack management on scratch pad memory," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '09)*, pp. 612–617, IEEE, Piscataway, NJ, USA, 2009.

[41] S. Udayakumaran, A. Dominguez, and R. Barua, "Dynamic allocation for scratch-pad memory using compile-time decisions," *ACM Transactions on Design Automation of Electronic Systems*, vol. 5, no. 2, pp. 472–511, 2006.

[42] B. Egger, J. Lee, and H. Shin, "Dynamic scratchpad memory management for code in portable systems with an MMU," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 2, pp. 1–38, 2008.

[43] Z. H. Chen and A. W. Y. Su, "A hardware/software framework for instruction and data scratchpad memory allocation," *ACM Transactions on Architecture and Code Optimization*, vol. 7, pp. 2:1–2:27, 2010.

[44] H. Wang, Y. Zhang, C. Mei, and M. Ling, "Energy-oriented dynamic SPM allocation based on time-slotted cache conflict graph," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '10)*, pp. 598–601, European Design and Automation Association, Leuven, Belgium, 2010.

[45] K. Bai and A. Shrivastava, "Heap data management for limited local memory (LLM) multi-core processors," in *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CODES+ISSS '10)*, pp. 317–325, ACM, New York, NY, USA, 2010.

[46] B. Egger, C. Kim, C. Jang, Y. Nam, J. Lee, and S. L. Min, "A dynamic code placement technique for scratchpad memory using postpass optimization," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '06)*, pp. 223–233, ACM, New York, NY, USA, 2006.

[47] M. Verma, L. Wehmeyer, and P. Marwedel, "Dynamic overlay of scratchpad memory for energy minimization," in *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and Systems Synthesis*

(CODES+ISSS '04), pp. 104–109, ACM, New York, NY, USA, 2004.

[48] D. Cho, I. Issenin, N. Dutt, J. W. Yoon, and Y. Paek, "Software controlled memory layout reorganization for irregular array access patterns," in *Proceedings of theInternational Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '07)*, pp. 179–188, ACM, New York, NY, USA, 2007.

[49] L. Li, J. Xue, and J. Knoop, "Scratchpad memory allocation for data aggregates via interval coloring in superperfect graphs," *ACM Transactions on Embedded Computing Systems*, vol. 10, no. 2, pp. 28:1–28:42, 2010.

[50] M. A. Baker, A. Panda, N. Ghadge, A. Kadne, and K. S. Chatha, "A performance model and code overlay generator for scratchpad enhanced embedded processors," in *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CODES+ISSS '10)*, pp. 287–296, ACM, New York, NY, USA, 2010.

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

[52] A. Mizuno, H. Uetani, and H. Eichel, "Design methodology and system for a configurable media embedded processor extensible to VLIW architecture," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD '02)*, pp. 2–7, IEEE Computer Society, Washington, DC, USA, 2002.

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