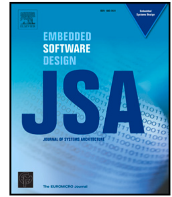
Journal of Systems Architecture 145 (2023) 103022

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc



COMPAD: A heterogeneous cache-scratchpad CPU architecture with data layout compaction for embedded loop-dominated applications

Tommaso Marinelli [a,c,*], José Ignacio Gómez Pérez [a], Christian Tenllado [a], Francky Catthoor [b,c]

[a] Universidad Complutense de Madrid, DACYA, Av. Séneca 2, Madrid, 28040, Spain
[b] Imec v.z.w., Kapeldreef 75, Leuven, 3001, Belgium
[c] KU Leuven, ESAT, Oude Markt 13, Leuven, 3000, Belgium

ARTICLE INFO

Keywords:
Cache
Compaction
Energy efficiency
Data reuse
Scratchpad

ABSTRACT

The growing trend of pervasive computing has consolidated the everlasting need for power efficient devices. The conventional cache subsystem of general-purpose CPUs, while being able to adapt to many use cases, suffers from energy inefficiencies in some scenarios. It is well-known by now in the academic literature that the utilization of a scratchpad memory (SPM) can help reducing the overall energy consumption of embedded systems. This work proposes a hybrid cache-SPM architecture with support logic for semi-transparent data management and spatial locality improvement. Selected data are transferred and stored in the SPM in a compact form using dynamic layout transformation. As a second major contribution, we introduce a methodology to identify memory access sequences that make an inefficient use of the cache, marking them as candidates to be moved to an SPM of constrained space. The methodology does not require access to the source code of the target applications, relying on binary instrumentation and offline profiling. The resulting mapping policies have been tested on a simulated system, showing a mean memory dynamic energy reduction of 43% and a mean speed gain of 13% with a representative benchmark set.

1. Introduction

Embedded microprocessors are employed in a large number of contexts in the contemporary era. Whichever the application field, there is a constant need for data collection, processing and transmission, and a trend to distribute the computation across the different nodes of a network, even with different capabilities. The emerging paradigms of Internet-of-Things and Edge Computing are excellent examples of the phenomena described above. In this scenario, application processors are flexible enough to perform a variety of computation tasks, but they must be energy efficient since they are usually battery-operated and/or employed in dense networks. For this reason they are a good target for optimizations that leverage the specific behavior of the executed applications.

Modern CPUs make use of integrated memory hierarchies to keep data copies closer to the processing elements and reduce the data access cost. However, traditional hardware-controlled caches present sources of energy inefficiency by design: first, several ways may be accessed in parallel to reduce latency, at the cost of a higher consumption. This typically happens with cache levels closer to the CPU, in case of a multi-level hierarchy. Secondly, large fixed-size cache blocks are used to exploit spatial locality, which entails wasting bandwidth, space and energy when most words of the block are not used before replacement.

In the last few decades, researchers have been exploring the possibility to adopt scratchpad memories (SPMs) as an alternative to CPU caches [1–3]: these are on-chip memories which are directly accessible by the processor – through specific load/store instructions – and do not need to be managed by the cache controller. Being intrinsically simpler, they have better latency and energy characteristics compared to caches of the same size [4,5]. This is also shown later in this article, with the data presented in Table 3, extracted with a widely employed memory modeling tool [6]. The drawback is that the complexity is transferred to the software, which must be programmed to perform explicit data management on the SPMs. For this reason, a hybrid solution (as in Fig. 1) can be beneficial, where one or more SPMs are added to the memory system and used only when convenient [7,8].

In this work, we propose a hardware solution for data management in hybrid cache-SPM memory subsystems, supporting data layout compaction for better spatial locality, aiming at an improved energy consumption. Details are provided on which components have be added to an existing processor design to achieve such goals. We call the

* Corresponding author at: Universidad Complutense de Madrid, DACYA, Av. Séneca 2, Madrid, 28040, Spain.
E-mail address: tommarin@ucm.es (T. Marinelli).

https://doi.org/10.1016/j.sysarc.2023.103022
Received 21 July 2023; Received in revised form 22 September 2023; Accepted 25 October 2023
Available online 29 October 2023
1383-7621/© 2023 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY-NC license (http://creativecommons.org/licenses/by-nc/4.0/).

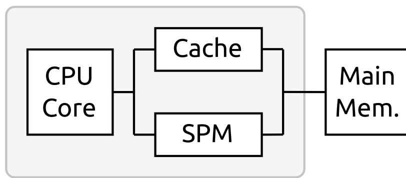


Fig. 1. Example of hybrid cache-SPM architecture.

resulting new architecture *COMPAD* (from the words *compaction* and *scratchpad*).

At the same time, we propose a profiling-based methodology to identify cache-inefficient access patterns in applications, and a placement policy to decide which data to serve from the SPM given its space constraints. Our methodology does not require access to the applications source code, and operates on traces generated after binary execution. Many works use analytical methods to identify access patterns from the code itself, enforcing data placement policies by adding new instructions and recompiling the program. However, in an industrial context it is a common scenario to employ third-party binaries and libraries, or legacy software where the source code is not available any longer. Our approach relies entirely on dynamic binary instrumentation [9] to get the needed information. The method we propose is dynamic in the sense that the data placement and movement are performed at runtime, but involves a first offline profiling pass to detect the memory access behavior of target applications and elaborate a placement policy for subsequent executions. The assumption is that the program behavior does not change significantly with different input data. This is true for a wide class of embedded applications, which is compatible with our focus. We believe that the SPM mapping policies enforcement can be applied to binaries as well, using static binary instrumentation, given the small amount of custom instructions to inject, but this is left for future work. Here, the effect of the policies is evaluated through high-level simulation (see Section 6).

The target of the proposed optimizations are loop-dominated applications. We can find that such kind of workloads are employed in a wide series of domains, like image processing [10,11], video decoding [12], scientific computation, statistical analysis, artificial intelligence, etc. In this work, the methodology has been tested on a representative series of numerical computation applications from the suite PolyBench/C [13]. We present a specific use case in Section 3, to show which characteristics can be exploited for our optimizations.

We have decided to center our analysis around the *heap* memory region, which contains dynamically allocated data, since it is the most relevant in applications where the core objects are created at runtime, which could be the case if their size depends on input data. In PolyBench/C, all the arrays used in the kernels computation are dynamically allocated by default.

The contributions of this work can be summarized as follow:

- Modeling of architectural support for data management in a hybrid cache-SPM architecture with layout compaction, to reduce cache inefficiencies with specific workloads.
- A methodology to analyze applications memory utilization and detect patterns in memory accesses.
- An algorithm for data partitioning between cache and scratchpad using data reuse and layout compaction information.
- An estimation of the impact of the proposed solution on a sample architecture through software simulation, with representative application benchmarks.

2. Related work

The idea of employing scratchpads in the memory hierarchy has been widely explored by the scientific community. This concept has been applied to instructions as well as data.

For very simple systems without a MMU, a methodology was developed by Egger et al. [14] to map code portions to the SPM and avoid continuous fetching from an external memory. Another work from Janapsatya et al. [1] used a SPM as a replacement for L1I (first level instruction) cache, mapping highly utilized code segments to the scratchpad for energy and performance improvements.

Works involving the mapping of data to scratchpads on embedded systems can focus on different memory regions, according to type of reference applications and the solutions employed. A part of the studies focused on *global* and *stack* data [2,15,16], while others covered data dynamically allocated in the *heap* [3,17], which size is not known at compile time.

A technical report from Banakar et al. [5] presented a size-varying comparative analysis of caches and scratchpads, using the SPM to store frequently used portions of both code and data. They generated an energy and area model for both devices and extracted performance data through simulation, showing that a scratchpad wins in every aspect if the memory size is big enough. Unlike our work, their analysis had a microcontroller as target, using an SRAM-based main memory and small benchmarks where changing the SPM content is never needed.

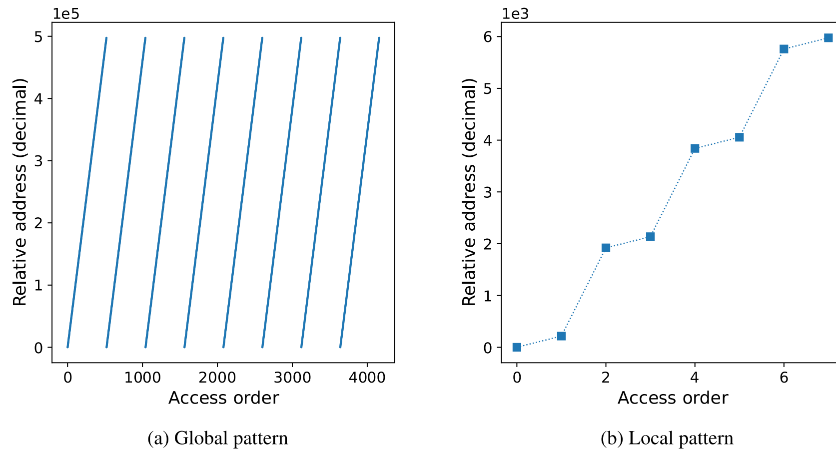
Besides embedded systems and single-core devices, scratchpads have also shown to be effective in more complex contexts. For example, is possible to find some proposals of compiler-assisted techniques to optimally exploit on-chip SPMs in many-core architectures with scientific and HPC workloads [18,19].

Emerging non-volatile technologies have also been considered as an option for scratchpads, usually mixed with conventional SRAM, with placement policies that take into account their characteristics [20–22]. Although emerging non-volatile memories show interesting properties, we believe that SRAM-based memories will still be dominant in commercial architectures for a long time, especially at the L1 level. For this reason, in this paper we have performed experiments based only on SRAM scratchpads. Still, our concepts are applicable to any L1-oriented memory technology, only the evaluation functions in the analysis methodology would have to change, e.g. to take into account asymmetric read/write latencies or endurance considerations.

Up to this point, we have only reported solutions where scratchpads are used as a replacement of caches. Chakraborty et al. [7] proposed an allocation heuristic for data in both fixed and reconfigurable-size cache-SPM systems, and a partitioning algorithm based on application requirements. Zhang and Wu [8] explored different on-chip cache and SPMs combinations, using a simple instruction/data mapping policy to improve energy consumption in an embedded context. These solutions are closer to what we propose, but do not address the inefficiencies of applications which do not make full use of the cache lines.

Several works are present in literature which try to solve the poor data spatial locality problem in hardware. Seshadri et al. [23] proposed a gathering-scattering enabled DRAM, which distributes the data of a strided pattern among different chips of the same DRAM module, to fetch them with a single command to a compacted cache line. A similar idea had already be explored in Impulse, a smart memory controller proposed by Carter et al. [24] that allows continuous-to-strided addresses mapping using shadow memory, providing efficient access and caching of compacted elements. Another work from Zhang et al. [25] aims at extending the AXI4 protocol to support bus packing in case of irregular workloads, integrating their solution into an existing RISC-V vector processor to improve its overall energy efficiency. These solutions do not involve the use of scratchpad memories, but only focus on data compaction rather than reuse. Additionally, the fact that no SPM is employed does not imply that these solutions do not require software modifications to operate. This is only avoidable by implementing a pattern/stride detection controller and dynamic remapping mechanism, which comes at a higher hardware complexity cost and possibly lower performance.

Instead of having separate devices, Komuravelli et al. [26] proposed a work where they combine aspects of the caches and scratchpads to



(a) Global pattern

(b) Local pattern

Fig. 2. Motivating example with exploitable characteristics (benchmark: *correlation*)

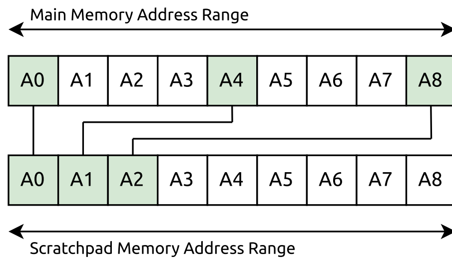


Fig. 3. Dynamic data layout compaction.

define a new memory organization and get "the best of both worlds" (direct addressing and compacted storage as in an SPM, global visibility and implicit data movement as in a cache). That work tries to address the same problems as ours, but focusing on GPUs. Moreover, there is not any proposed automatic analysis of access patterns, and manual programmer intervention is always required.

There are previous works that focus specifically on application analysis. A general methodology for data objects identification and pattern extraction was proposed by Ji et al. [27]. Our work follows the same steps for this part of the problem, while using Pin [9] also for the data objects detection pass.

In conclusion, to the best of our knowledge, none of the published hardware–software solutions properly addresses all the issues which we have identified in the introduction.

3. Motivating example

Since we want to reduce inefficiencies derived by the use of the cache, the main metrics in our analysis are the amount of data *reuse* and degree of possible *compaction* in the address space. Scratchpad memories generally have a better energy per access compared to caches of the same size – as shown later in Table 3 – so it is convenient to use them to provide data which are frequently accessed within a certain time frame (*reuse*). Furthermore, in the proposed solution the SPM can be contiguously filled with non-contiguous data from the main memory, with a granularity minor than a cache line, reducing unnecessary data exchange in the memory system with sub-optimal access patterns (*compaction*).

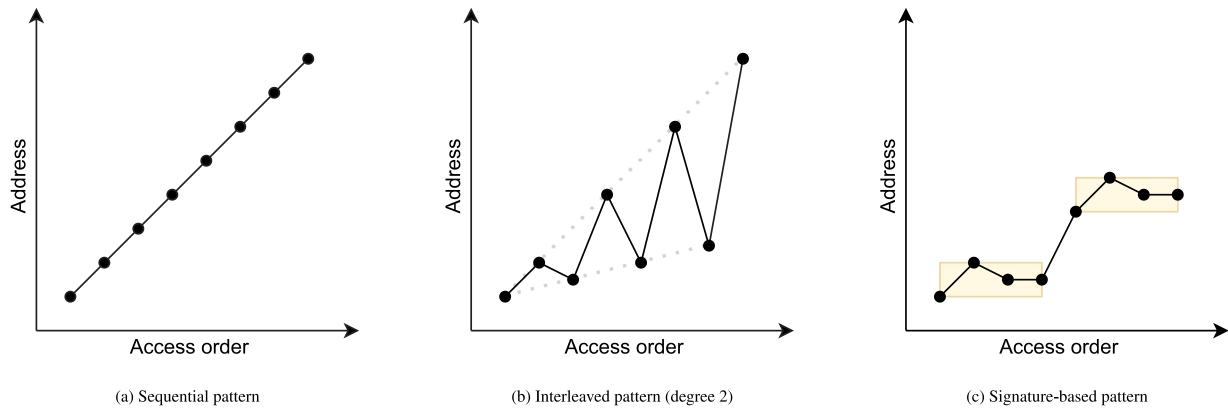
Before proceeding to a more formal description of the methodology, we would like to present a sample case where it proves to be effective. The example originates from one of the benchmarks used in the experimental part. In Fig. 2 we show some of the memory accesses targeting a certain data object of the sample application during one long phase of execution. The x axis represents the order of the accesses as they are emitted by the CPU, while the y axis represents the relative position in the data object address range, starting from zero. The global pattern of the analyzed phase (Fig. 2(a)) is a repetition of line-shaped accesses. Each one of the represented lines, which we call *intervals*, is a sequence of 520 read accesses to memory. If we look at Fig. 2(b), which simply depicts a closer view of an individual interval, we can see that the sequence is not really linear, but divided in couples of points. The stride between each first point of the couples is 1920, and their addresses remain constant between subsequent intervals, i.e. only the second element of the couples change. This already gives us an important information: half of the data can be recycled among different iterations, and only 260 words have to be transferred from the main memory, which means that there is a good amount of data reuse between intervals that can be exploited.

The second points in each of the couples in Fig. 2(b) have the same stride, but they have an offset compared to the first points which increases at each iteration, with increments of 8 bytes each. After the first few iterations, with the growth of the gap between the two points, each access ends up targeting (and possibly carrying to the cache) a different cache line, for a total of 520 individual lines. This even saturates the capacity of a 32KiB cache, which is able to store up to 512 lines. For these considerations we are assuming a standard cache line size of 64 bytes.

As illustrated in Fig. 3, our proposal is to compact the layout and fetch only the elements that will be accessed in the closest time frame, according to profiling information. In the case of the loop just described (in "steady state"), given that each access targets a 64-bits word, if data are stored in the SPM discarding the redundant elements in their cache lines, they will occupy 1/8 of the original space, i.e. 65 equivalent cache lines. Saving compacted data in the SPM has the additional positive effect of not polluting the cache and leave it free to serve other accesses.

4. Methodology

In this section we present the different steps of our systematic methodology, from the applications memory behavior profiling to the mapping decision algorithm. In essence, the goal is to understand how the application accesses memory, identifying patterns with "cache-inefficient" characteristics than can be mapped to a scratchpad memory to save energy. The analysis is entirely based on memory access metrics which are indirectly connected to the energy consumption. The outcome of the mapping policies, applied to a relevant benchmark set in a sample system, is shown in Section 7.



(a) Sequential pattern (b) Interleaved pattern (degree 2) (c) Signature-based pattern

Fig. 4. Example of identifiable access patterns.

4.1. Memory utilization

This phase consists in an offline profiling of the target application to learn the way it uses memory, identifying the totality of structures allocated in the heap and dumping a detailed access trace for each of them.

Many software applications, especially commercial ones, come in form of compiled binary, without any detail about the original source code. For this reason, it is not possible to obtain detailed information on memory utilization through code analysis, unless decompilation techniques are employed — which are effective only to a limited extent. The approach adopted for this phase is based on dynamic binary instrumentation (DBI). Instructions executed by the target application are modified at runtime to include custom analysis routines. For this task we developed a tool which relies on the Pin [9] instrumentation framework and is partially based on previous work from Chakraborty and Panda [28].

When an application is run under the Pin framework, the programmer can specify what analyses to conduct on it by means of a *Pintool*. There are different points where the execution can be stopped to inject the extra code. In this case, the instrumentation happens with a *trace* granularity. A *trace* represents a portion of code with a single entry point and multiple exit points. In each *trace*, any call to C library memory allocation functions - i.e. `malloc()`, `calloc()` and `posix_memalign()` - is detected, with the help of the symbol table. At this point, the found calls are surrounded by analysis routines to extract the function arguments and return value, then the execution is resumed. The relevant extracted parameters are the *start address* and *size*, which allow to determine the boundaries of the reserved memory area. This part of the tool was reused from Chakraborty's and Panda's code,[1] with minor modifications.

We define as *data object* each of the memory pools resulting from a dynamic memory allocation call. In case of more complex structures it is also possible to cluster multiple calls from the same code region – using information about the instruction pointer and call stack – and consider them as a single object, but this has not been needed for the applications used in this work.

Our *Pintool*, other than dumping information about the data objects, is able to keep track of the memory accesses targeting such structures. For each of the objects, a detailed access trace is generated, which is later fed to the pattern analysis tool. This is achieved by instrumenting each load and store instruction of the executed binary, performing a range check and – if positive – dumping basic information: request type, address and size.

For the rest of the steps of our methodology, we do not rely on third-party frameworks, but we employ custom utilities developed with an extensive use of scripting languages, mainly Python.

[1] https://www.cse.iitd.ac.in/~panda/spm-sieve/

4.2. Pattern analysis and classification

The following step consists in the analysis of the memory access traces, to identify patterns and properties that can be later used to select candidates to be moved to the SPM. For a better detection of the patterns, each data object is analyzed independently. The global view on the application is not lost though, as the resulting information is merged back at the end of this phase.

Any series of ordered memory accesses which fulfills some predetermined regularity criteria is classified as *interval*. The concept was already introduced in Section 3. Specifically, these are the interval types that the tool is able to detect:

- *Sequential*: a series of subsequent accesses that span the address space with a certain stride (Fig. 4(a)).
- *Interleaved*: a series of sequential patterns mixed together in time (Fig. 4(b)). These mixed patterns may or may not have the same stride. The patterns previously shown in Fig. 2 belong to this category.
- *Signature-based*: a generally irregular pattern that repeats itself over short periods of time (Fig. 4(c)).

Any other sequence is considered irregular and not classified. It is important to note that the presented categories are already able to cover cases that go beyond simple stride-based loop codes. In the benchmarks selected for the experimental evaluation, more than 99% of the accesses are classified and fit one of these pattern definitions.

Intervals can be of variable lengths, which depends on the starting and ending points of the regular behavior. Nonetheless, some minimum and maximum boundaries on the length are configurable in the tool, to avoid spurious detections or to fragment bigger intervals in smaller units, e.g. to guarantee that each one of them can fit the SPM size.

In this methodology, intervals are the minimum units that can be mapped to an SPM, and the scheduling of data movement is based on their boundaries. Operating with this granularity does not always result in optimal choices, since some reuse between phases can be missed and more data than necessary moved. For this reason, we perform grouping on contiguous intervals which share the same characteristics, if it brings some benefits. Given that the pattern detection mechanism has the ability to identify multi-level loops from contiguous intervals, such grouping can be done in different ways to match the loop dimensions. This way, it is possible to generate a reuse tree and perform a reuse analysis as in the methodology from Wuytack et al. [29]. We follow a similar approach, using our (macro)-intervals as *time frames*, but we base the analysis on the metrics defined in the following subsection and add constraints on the interval size, to make sure the amount of data does not exceed the fixed size of the SPM.

From now on, we will not make a distinction between an interval and a group of intervals, after each group has been defined.

4.3. SPM mapping algorithm

The output intervals of the pattern detection tool are then used as input for the SPM mapping algorithm. Here follows a definition of the metrics which the evaluation is based on.

Provided an interval *int*, defined as a family of memory addresses x_i corresponding to a set of indices I, which represents the interval access order in the data object trace:

$$int = (x_i \mid i \in I)$$

The intra-interval reuse can be computed as the complement of the ratio between the number of unique addresses (i.e. the cardinality of the address set) and the interval length (i.e. the cardinality of the indices set). Since it is possible to have repeated elements in the interval, the ratio will be always minor or equal to 1.

$$intra_reuse(int) = 1 - \frac{|\{x \mid x \in int\}|}{|I|}$$

The inter-interval reuse metric is relative to the previous interval. If we go back to Fig. 2(a) of the motivating example, and imagine the second line being the current interval (int), the previous interval ($prev_int$) would be represented by the first line. We define the previous interval as:

$$prev_int = (x_h \mid h \in H)$$

Where the set of indices H corresponds to its access order in the memory trace, as for the current interval. The inter-interval reuse is a measure of how many accesses of the current interval are to memory positions already accessed in the previous one:

$$inter_reuse(int) = \frac{|\{i \mid i \in I \wedge x_i \in prev_int\}|}{|I|}$$

Note that in this context we do not take into account the type of operation, thus not using the traditional definition of data reuse, and instead we refer to the re-accesses to the same memory position.

Finally, we calculate the data layout compaction ratio. This is represented as the complement of the ratio between the number of virtually needed lines, i.e. if data were compacted together, and the number of effective accessed lines. At the numerator, the number of unique addresses is multiplied by the data size and divided by the cache line size. The data size is assumed to be constant for the duration of the interval. At the denominator, a bit mask is applied to each address to generate the corresponding cache line address, then the cardinality of the resulting unique set is computed:

$$comp_ratio(int) = 1 - \frac{|\{x \mid x \in int\}| \ * \ data_size \ / \ cl_size}{|\{cl_addr(x) \mid x \in int\}|}$$

To decide which data to map in the limited space of the SPM, the intervals from all the data objects of one applications are analyzed together. For now, only *sequential* and *interleaved* intervals are considered as possible targets, to keep the implementation of the support hardware simple enough (address translator and gathering/scattering unit, described in Section 5). These types of intervals are dominant in the benchmarks we have tested, and many other applications.

A pseudo-code description of the algorithm is reported below (Algorithm 1). The placement decision has been modeled as a Knapsack problem. Each interval is considered for mapping unless it overlaps with others. If this happens, and the space of the SPM is not sufficient to store all the intervals, the conflicting set is selected for further analysis, using the *getOverlapping* function. A value is computed for each of these intervals, including the original, using the *getValue* function. This value represents a weighted combination of the properties described above, i.e. inter/intra reuse and potential compaction. The intervals which have the lowest values are discarded, until the resulting set can fit the SPM. For convenience, we use the same weight for both reuse factors (*r_wgt*), while the compaction gets a weight of its own (*c_wgt*).

Algorithm 1 SPM Mapping Decision Algorithm

```
1:  function getValue(int)
2:      reuse ← max(
            calcInterReuse(int),
            calcIntraReuse(int)
        )
3:      compact ← calcCompRatio(int)
4:      return (reuse ∗ r_wgt + compact ∗ c_wgt)
5:  end function
6:
7:  discarded ← empty
8:  for int ∈ intervals do
9:      if int ∉ discarded then
10:         candidates ← getOverlapping(int) + int
11:         while sum(candidates.size) > spm_size do
12:             wst = min(getValue(∀ int ∈ candidates))
13:             discarded ← discarded + wst
14:             candidates ← candidates − wst
15:         end while
16:     end if
17: end for
18: selected ← intervals − discarded
```

For the experiments conducted in this work, we have chosen a balanced combination of weights, not giving any particular advantage to a metric respect to the other. Both weights have been set to 0.5. This choice was made after conducting a sensitivity analysis on the weights values, evaluating three different possibilities (one balanced and two unbalanced) and analyzing the impact on part of our application set. The balanced choice has the best average effect in terms of both latency and energy. That said, the behavior is application-dependent, as expected. We have registered relative performance improvements as high as 20% and energy reductions up to 25% while comparing the unbalanced policies to the balanced one, but at the cost of making the complementary metric worse.

Before the algorithm is executed, the intervals are pre-processed to discard all the ones which do not possess any exploitable reuse or compaction quality. We use the same *getValue* function as in the algorithm, setting our selection threshold to 0.3, as a trade-off between the quality of the selected intervals and the number of transfers. Both the selection threshold and the reuse/compaction weights can be tuned differently according to the context.

It is important to make a few final consideration about this methodology, highlighting some of the shortcomings that can be addressed in future work. The inter-reuse estimation is performed only with respect to the previous interval, and the window size is limited due to the size constraint of the SPM. This means that some of the reuse can be missed, which could be exploitable otherwise. Also, in case of complex loops having multiple phases within the same (outer) iteration, the tool may not be currently able to detect the subsequency of the intervals and perform any grouping. Finally, even if all the intervals are guaranteed to fit the SPM, there can be a sub-optimal use of the memory space in case of partial overlap, since there is no support for fractional selection, i.e. the entire interval is either selected or discarded. Even taking into account the described limitations, the current methodology is able to provide good results in most situations.

5. Architecture

In this section we introduce *COMPAD*, a potential architecture for semi-transparent scratchpad management with data layout compaction support. This novel architecture only differs from a traditional one by a minor number of additions/modifications, so it is easy to build from existing designs. The proof-of-concept here described is a single-core and single-scratchpad configuration, but some variants can be

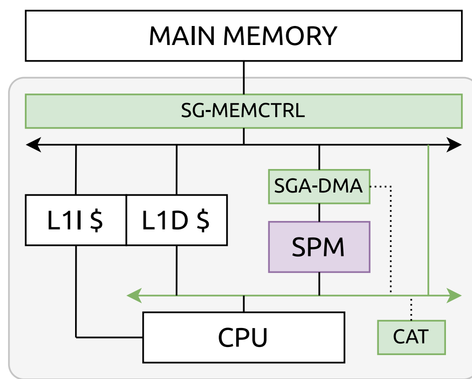


Fig. 5. COMPAD architecture.

developed to adapt to more complex scenarios, which we leave for future work. An overview of the architecture is depicted in Fig. 5.

We will now briefly illustrate the additions made to COMPAD with respect to a traditional CPU architecture. Apart from the additional scratchpad memory, we include the following elements:

- **SGA-DMA -** To allow for efficient data transfer between the main memory and the scratchpad, avoiding constant intervention from the CPU core, a Direct Memory Access (DMA) controller is added to the architecture. This controller must be able to send strided memory requests to the memory controller, in order to exploit the data layout compaction. Hence, we call this component *Scattering-Gathering Aware DMA Controller* (*SGA-DMA*). The SGA-DMA is implemented as a Memory-Mapped I/O (MMIO) device and can be programmed from software with writes to its configuration registers.
  Once the device has been programmed with the pattern information (type, length, stride, etc.) and the start command is received, the controller generates a list of addresses to access and splits the request in chunks of the size of a cache line, to maximize the system bus utilization. At the same time, it encodes into the request the elements access order, to be passed to the memory controller which processes it accordingly. This allows to leave the scattering/gathering task to the memory controller itself, which makes our proposal different from other strided DMA solutions [30–32] and closer to what happens in the *Address Calculation Accelerator* described by Yousefzadeh et al. [33]. Implementation-wise, this is compatible with the bus packing extension for strided requests proposed by Zhang et al. [25], or at least with a potential modification of it to support all the pattern types we cover.
  The SGA-DMA has an internal FIFO buffer, which is used to temporarily store data after fetching them from the origin device and before sending them to the destination.
  A problem that can occur when creating copies of data in the SPM is that such data can still be present in the caches, leading to unwanted data duplication and coherency issues when one of the copies is modified. To prevent this, all the cache lines corresponding to a SGA-DMA request are invalidated before the beginning of the transaction, and data therein contained are written back to the main memory, if modified. This flushing mechanism must keep into account the characteristics of the pattern, since in case of strided accesses the data may be non-contiguously distributed across different cache lines.
- **CAT -** Another key element in the proposed architecture is the *Compacted Address Translator* (*CAT*). This unit monitors the memory ranges of the objects selected to be mapped to the SPM at every phase of the application. If the load/store operation targets an address outside those ranges, the request is sent to the cache hierarchy. Otherwise, the unit checks if that specific address belongs to one of the points actually mapped in the SPM or one of the *holes*, which would be in main memory, and forwards the request to the correct device. Similarly to the SGA-DMA, the Compacting Address Translator is modeled as a MMIO device and can be programmed with specific instructions.
  We use the selected intervals from our methodology to decide in which moments to activate/deactivate the address translator and with which properties. The CAT activation always happens after the SGA-DMA data movement. Note that, since our methodology is based on profiling, we may fail in our predictions about the actual words that will be accessed at runtime, so a backup mechanism must be in place: detecting the *holes* during the execution and redirecting accesses to main memory is our backup mechanism.
  Given the low amount of objects mapped to SPM (always less than 8 in our experiments) and the characteristics of the patterns detected by our tool, the translation may be implemented efficiently, either analytically or by a simple sequence generator. As claimed by Komuravelli et al. in *Stash* [26], only six arithmetic operations are needed for a single-strided pattern analytical mapping. This might not be sufficient to cover more complex patterns. A primitive RTL implementation that employs the second approach is available in our GitHub repository.[2] Essentially, it works as an address tracker: whenever the expected address is encountered, the following one is computed according to the programmed characteristics and becomes the new expected address. The currently published version is not able to fully exploit internal reuse and does not allow flexibility in the access order, but it constitutes a proof-of-concept for the mechanism and can be improved in future work.
  Overall, the CAT component serves a dual purpose: first, it enables access to the new data layout by providing the effective address of data inside the SPM, and it does so knowing the characteristics of the mapped patterns. Secondly, it allows for transparent address translation after programming, so none of the original addresses has to be changed in the application code.
- **Translating Bus -** To handle the arbitration of the requests that come out of the CPU data port, a special bus is introduced. The bus is connected to the data cache, the scratchpad memory and has a shortcut to send requests to the memory controller bypassing the cache hierarchy, when needed (the *holes* mechanism explained before). Each access has to follow a lookup phase, where the address is checked before deciding its destination. This initial lookup happens in the critical path, but the impact in term of performance is negligible, since it is only a range check for a limited amount of mapped objects.
- **SG-MEMCTRL -** Finally, we include a custom memory controller with support for data scattering/gathering (*SG-MEMCTRL*). This is required to efficiently transfer data between the SPM and main memory without polluting the bus. When the memory controller receives a read request from the SGA-DMA, it translates it to a series of read commands for non-consecutive data in the main memory, gathering them into packets to be sent to the SPM. Vice versa, at the end of the lifetime of the interval, data are scattered to their original position in the main memory.
  Although not strictly necessary, this mechanism would provide the most benefit if combined with a special memory interface which allows for continuous strided access within the same line. Otherwise, standard DRAM interfaces can be used (e.g. when employing off-the-shelf DRAM chips), at the cost of some extra energy and performance cost. This is because typical burst sizes are higher than a single word, and it is not possible to access data with a finer granularity, thus wasting part of the accessed elements.

[2] https://github.com/artecs-group/transpad

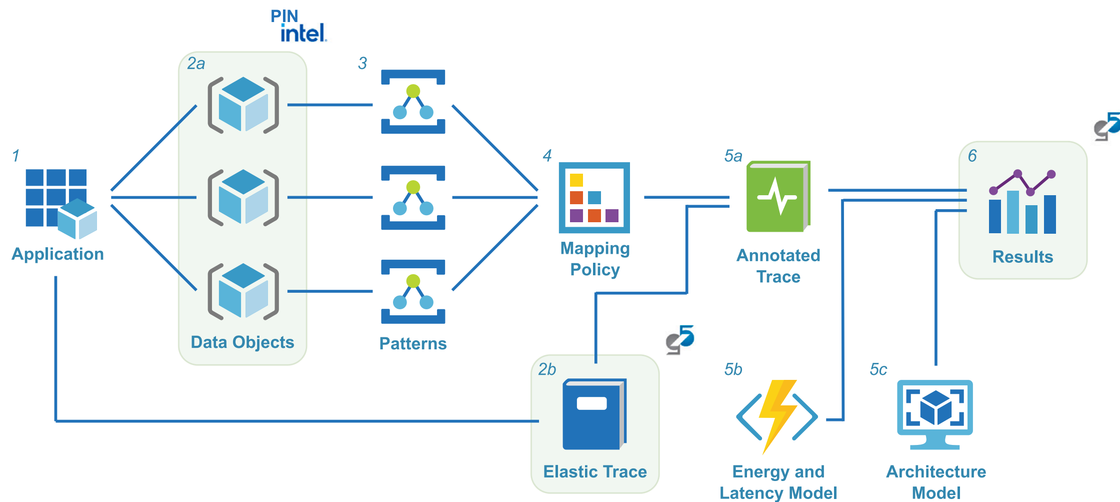


Fig. 6. Complete experimental analysis and simulation flow.

Table 1
Selected benchmarks (PolyBench/C Suite).

Benchmark	Domain	Description [13]
2 mm	lin. alg./kern.	2 Matrix Multiplications
3 mm	lin. alg./kern.	3 Matrix Multiplications
correlation	datamining	Correlation Computation
covariance	datamining	Covariance Computation
deriche	medley	Edge detection filter
doitgen	lin. alg./kern.	3 Multiresolution analysis kernel
durbin	lin. alg./solv.	3 Toeplitz system solver
gemm	lin. alg./blas	General Matrix Multiply
gemver	lin. alg./blas	Vector Mult. and Matrix Addition
gesummv	lin. alg./blas	Scalar, Vector and Matrix Mult.
symm	lin. alg./blas	Symmetric matrix-multiply
syr2k	lin. alg./blas	Symmetric rank-2k operations
syrk	lin. alg./blas	Symmetric rank-k operations
trmm	lin. alg./blas	Triangular matrix-multiply

The proposed modifications for the COMPAD architecture naturally entail an extra cost in terms of power, performance and area. Nevertheless, the support logic cost is typically a fraction of the memory one, and the additional SPM area can be compensated by reducing the amount of cache, as we do in the experimental part. In our simulation environment, described in the next section, we have developed a realistic performance and energy model for the complete system, with some reasonable simplification. For example, from the performance point of view, some of the latencies are assumed to be small enough to be negligible (more on this in Section 6.2). However, the additional data traffic between the SPM and main memory is completely accounted, which can have a negative impact in situations where the SPM does not allow a better exploitation of data locality compared to the cache. We show some of those cases in our experiments, in Section 7.

6. Experimental framework

In this section we describe the different tools used in the experimental part, providing information about the modeling and simulation framework. We also include a descriptive graph in Fig. 6, which represents the complete experimental flow. The graph spans from the application analysis - with the methodology described in Section 4 - to the effective results extraction in the simulated COMPAD architecture. Some of the steps are highlighted and accompanied by an icon, which indicates the third-party framework employed in the specific phase.

6.1. Representative benchmark set

The focus of this work is to reduce the energy per access in embedded devices executing fairly simple applications from some specific domains, whose access sequences are composed for the most part of *regular* and *interleaved* patterns, following the definitions presented in Section 4. We have selected a subset of benchmarks from the PolyBench/C suite [13] for our experiments: it is a collection of numerical computation applications from different domains, widely used in the research community. Most of the benchmarks are linear algebra kernels, which are often employed – for example – in machine learning applications. A few statistical analysis benchmarks from the suite have been included as well, which are essential for data science. All these benchmarks are loop-dominated, so they are very representative for our scope. Applications in PolyBench/C allow to specify a data set size at compile time: we have chosen the *medium* one, as a trade-off between simulation complexity and scenario realism. The list of target benchmarks is reported in Table 1.

PolyBench/C constitutes a good proof-of-concept for the methodology since its applications have a straightforward way of managing data. During the initial phase, a variable number of arrays is allocated. Some of them are initialized and fed to the algorithm as input, others are used to store the computation results. The default behavior is to allocate these arrays in the heap memory region (using `posix_memalign` calls), but it can be overridden to use the stack, which is not the case for this work.

6.2. Simulation environment

To evaluate the impact of the mapping policies we have used the gem5 architectural simulator [34], a widespread software in academia and industry. Our work is based on top of version 21.2. All the architectural modifications described in Section 5 have been modeled in terms of behavior and integrated into the simulation environment. The code is publicly available in our GitHub repository.[3]

In the software model, we account for the extra latencies associated with the address translator and DMA controller programming and setup, other than the data movement. The address translation cost is, instead, assumed to be negligible. In fact, the mechanism consists of a number of range checks, followed by the new address calculation if a correspondence is detected. Since we employ a limited number of CAT

[3] https://github.com/artecs-group/gem5-artecs/tree/compad

Table 2
Simulated system details (single core).

CPU	*Architecture:* x86-64 *Ordering:* In-Order *Frequency:* 1 GHz
Main Memory	*Size:* 512 MiB *Standard:* LPDDR3
L1I Cache	*Size:* 16 KiB 4-way set associative
L1D Cache	*Size:* 16 KiB/8 KiB 4-way set associative
SPM (Hybrid)	*Size:* 8 KiB

Table 3
CACTI caches and SPMs parameters.

Device	Acc. time	Read eng.	Write eng.	Leakage
Cache 8 KiB	0.44 ns	0.030 nJ	0.028 nJ	4.51 mW
Cache 16 KiB	0.48 ns	0.031 nJ	0.029 nJ	7.86 mW
Cache 32 KiB	0.53 ns	0.033 nJ	0.031 nJ	14.57 mW
SPM 8 KiB	0.18 ns	0.008 nJ	0.010 nJ	3.70 mW
SPM 16 KiB	0.25 ns	0.009 nJ	0.012 nJ	7.15 mW
SPM 32 KiB	0.40 ns	0.010 nJ	0.018 nJ	14.04 mW

entries – equal to 8, enough to cover all the cases in the experimental part – the comparisons can happen in parallel, and the new address can be easily pre-computed due to the benchmarks control flow regularity, so the translation can happen within the same cycle. We also operate some simplifications in the way we handle inter-interval reuse, but they do not impact the final results as at the interface between intervals the highest penalization is still given by data exchange with the main memory. As for the main memory part, no special gathering-scattering optimized interface has been employed – only modifications to the controller – to maintain compatibility with commercial DRAM modules, which is a conservative but pessimistic assumption.

The experiments have been conducted in *Syscall Emulation* mode, using elastic traces [35] to analyze the impact on the memory system without constantly re-simulating the detailed CPU model. This kind of traces essentially contain a dump of all the memory accesses performed by the CPU, with annotated order and register dependencies. The traces extracted from the benchmarks, generated with an out-of-order CPU model, have been modified to mimic an in-order processor, more in line with low-energy platforms requirements. Also, the control instructions for SPM management have been added to them at each interval boundary, according to the profiling and selection methodology results.

The platform chosen for the experiments is a generic low-power, in-order x86-64 processor with a single level cache design, whose characteristics are described in Table 2. It has been selected as a representative solution for our target domains due to its simplicity. The memory subsystem of the platform consists of a single-level cache, with an added SPM in the hybrid configuration. In order to make a fair comparison with respect to the baseline, the L1D size is reduced in this second case from 16 KiB to 8 KiB, to keep the total amount of memory constant. This is a slightly pessimistic assumption, since the area occupation of an SPM is comparatively lower to a cache, and a bigger quantity could be employed, which would limit the performance impact. The x86-64 ISA, while not as popular in the embedded domain, was favored to ARM or RISC-V counterparts because the adopted DBI framework does not support non-x86 binaries. However, this could be easily changed porting the memory analysis tool to a different framework, like DynamoRIO [36] or MAMBO [37].

6.3. Memory models

For a realistic estimation of cache and scratchpad memory parameters we have employed the latest version of CACTI [6], which is arguably the most widely used modeling tool for SRAM devices.

In Table 3 we provide a comparison table with the characteristics of memories of different sizes, extracted with the aforementioned tool, to show the advantages of using SPMs to the reader. The technology node is assumed to be 22 nm, with a low-power roadmap for the data array and high-performance for the tags and periphery. The design objective is the cycle time, with optimizations to minimize the energy-delay product. For caches modeling, we have selected a 4-way associativity. The tool was left free to decide the optimal memory organization (number of column and row partitions, number of sets per row, etc.). Reading the table it can be noted that scratchpad memories, which lack a tag array and do not need to fetch multiple blocks in parallel, have overall better parameters: if we compare, for example, the 32KiB cache numbers with an SPM of equal size we can notice an approximate 2x speedup and a 4x to 8x dynamic energy improvement.

For our experiments, we have adopted the CACTI energy parameters corresponding to the simulated configurations, i.e. 16 KiB cache for the traditional system and 8 KiB cache + 8 KiB SPM for the hybrid system. The access time has been assumed to be equal for the two devices, since the low frequency of the CPU does not allow to leverage the better latency of the SPM.

Regarding the main memory parameters estimation, we have employed one of the DRAM models already present in the gem5 simulator. Specifically, the simulated system includes a LPDDR3-1600 device with a 32-bits bus, whose latency and energy data were originally extracted from the datasheet of a commercial part.[4]

7. Results

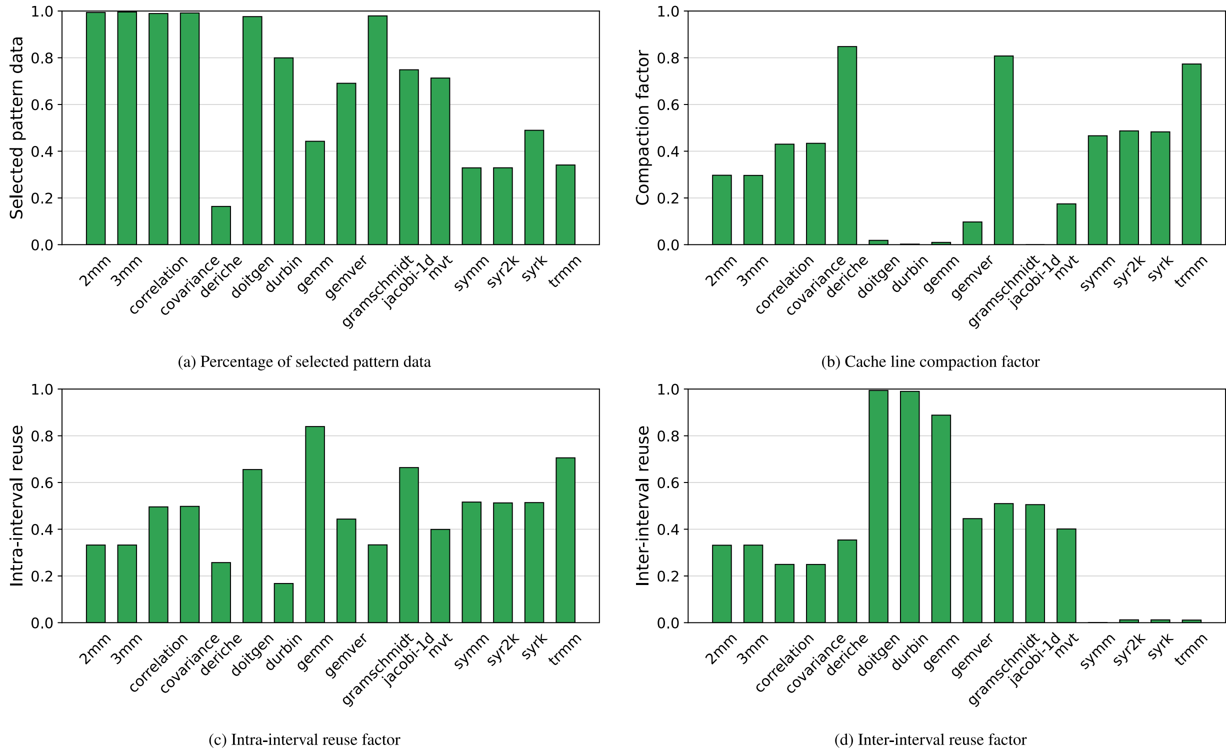
In this section we present and compare the simulation results, highlighting the impact of the methodology and taken mapping decisions for the selected benchmark set.

First, to better understand the results, in Fig. 7 we introduce some plots containing metrics resulting from the SPM mapping algorithm (Section 4.3). Fig. 7(a) shows how many data from detected patterns are effectively mapped to the SPM, due to the selection policies and SPM size constraint. The rest of the plots (Figs. 7(b), 7(c), 7(d)) represent the average values of the coefficients on which the selection is based, specifically the intra-interval reuse, inter-interval reuse and compaction factor, for the chosen set of patterns. Generally, there is a high variability of the metrics among the benchmarks, except for the ones that belong to the same class and perform similar operations, like *2mm and 3mm* or *correlation* and *covariance*.

The relative dynamic energy consumption of the combined data cache and SPM, with respect to the cache-only configuration, is represented in Fig. 8(a). The use of the additional SPM and custom data mapping strategy brings an energy improvement in all the presented cases, with some noteworthy differences. Several benchmarks show a high energy reduction, like *2* mm (53%), *3* mm (51%), *correlation* (56%), *covariance* (56%), *gramschmidt* (52%), and *jacobi-1d* (54%). Other benchmarks are less impacted, like *deriche* and *syr2k*, by 11% and 20% respectively. It appears that these last benchmarks are also the ones for which the least amount of intervals is selected, as in Fig. 7(a), which explains the limited beneficial effect. The geometric mean of the dynamic energy reduction is 43%.

Finally, in Fig. 8(b) we show the relative execution time between the two configurations, highlighting how the hybrid system behaves with respect to the traditional one. Contrarily to the energy, in this case the effect is not always positive, resulting in a penalization in some benchmarks, like *durbin* (40%), *jacobi-1d* (25%), *syr2k* (19%), and *trmm* (27%). On the other hand, other benchmarks are highly benefited, especially *correlation* and *covariance* (both with a 65% speedup), and *mvt* (47% speedup). The overall effect is still favorable, with a mean geometric speedup of 13%. The positive result of *correlation* should

[4] Micron EDF8132A1MC



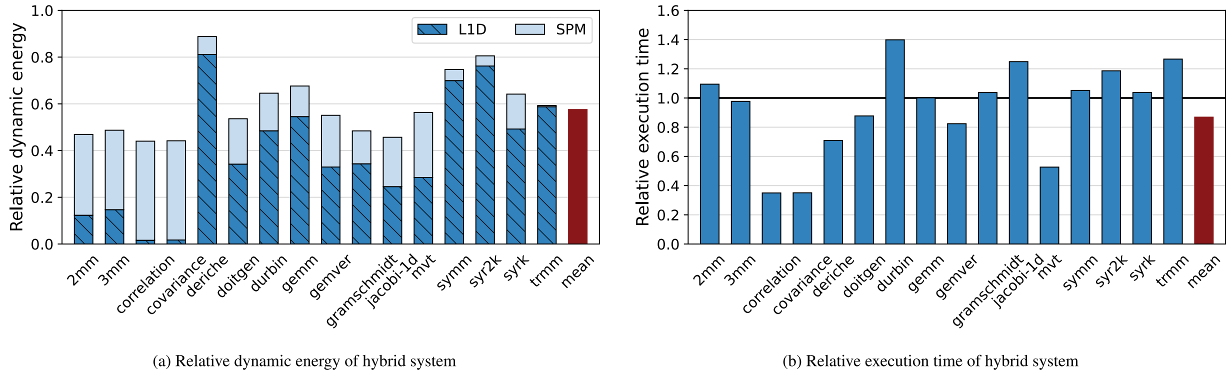
(a) Percentage of selected pattern data

(b) Cache line compaction factor

(c) Intra-interval reuse factor

(d) Inter-interval reuse factor

Fig. 7. Analysis metrics.



(a) Relative dynamic energy of hybrid system

(b) Relative execution time of hybrid system

Fig. 8. Simulation results.

not surprise, given that it is the application considered as motivating example for this work (see Section 3).

We can make some hypotheses about why some benchmarks are behaving worse than the baseline. Memory accesses in *durbin* and *jacobi-1d* exhibit a good spatial locality, as it can be seen in Fig. 7(b) since the compaction ratio is very low. This means that most of the data carried to the cache are actually accessed. In these cases, the cache is already behaving well enough, and the additional data transfers and reduced cache size are impacting the execution. This is also testified by the fact that in the traditional system the amount of data cache misses is extremely low for these benchmarks, with most of them likely being compulsory misses. The miss rate increases in the hybrid system. It is important to mention that in similar cases, when there are almost no capacity/conflict misses and no compaction is possible, we cannot have performance gains with the hybrid solution, provided that the cache and SPM access times are comparable. Additionally, these two benchmarks present composite patterns which cannot be grouped due to current limitations of our framework, leading to an increased frequency of data transfers. Likewise, the absence of inter-interval reuse in *symm*, *syr2k*, *syrk* and *trmm* (Fig. 7(d)) can make the impact of the SPM data transfers not negligible and affect their execution time.

If we compare *durbin* with *doitgen*, we see that they possess similar characteristics in terms of inter-interval reuse (close to one) and compaction ratio (close to zero), but the former is worse in terms of intra-interval reuse. This fact, together with the previous considerations on *durbin*, can explain their difference in relative performance. Energy-wise, *doitgen* has a slightly better relative consumption, also thanks to the higher percentage of mapped intervals.

As a last task, we have computed for all benchmarks the product between the energy and performance data described above, evaluating once again the relative trends between the traditional and hybrid configuration. This allows to better understand the overall impact of the new architecture and mapping policies. Like in the energy results, the product decreases in all the analyzed cases, with a geometric mean of 50% for the relative values. The worst value is found in *syr2k* (95%), while the best results are achieved by *correlation* and *covariance* (15%).

We believe that it is possible to obtain a better energy-slowdown trade-off for some of the benchmarks by improving the heuristics and tuning the pattern selection parameters, which is left for future work.

8. Conclusions

In this work we have shown that traditional cache subsystems may not be optimal for embedded platforms with low energy requirements, and some gains could be achieved identifying problematic memory patterns in applications and moving the corresponding data to a software-controlled device, which can serve such accesses more efficiently. We have illustrated our profiling-based methodology to identify these patterns, provided details of a possible modified architecture to exploit data layout compaction in the SPM and conducted some experiments with a software simulator to show the potentiality of our proposal. Using a selection of loop-dominated benchmarks, we have obtained some substantial gains in terms of dynamic energy, with a mean value of 43% compared to a traditional cache-only solution. Performance results indicate that the execution time can be either positively or negatively affected depending on the benchmark, nonetheless there is an overall 13% improvement across the application set, with respect to the baseline.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Part of the code has been shared through links within the text. Rest of the code and data will be available on request.

Acknowledgments

This work has been supported by grant PID2021-123041OB-I00 funded by MCIN/AEI/10.13039/501100011033 (*Spanish Ministry of Science and Innovation*) and by "ERDF A way of making Europe" (*European Regional Development Fund*), and by the CM (*Community of Madrid*, Spain) under grant S2018/TCS-4423.

References

[1] A. Janapsatya, S. Parameswaran, A. Ignjatovic, Hardware/software managed scratchpad memory for embedded system, in: IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004, 2004, pp. 370–377, http://dx.doi.org/10.1109/ICCAD.2004.1382603.

[2] H. Cho, B. Egger, J. Lee, H. Shin, Dynamic data scratchpad memory management for a memory subsystem with an MMU, in: Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES '07, Association for Computing Machinery, New York, NY, USA, 2007, pp. 195–206, http://dx.doi.org/10.1145/1254766.1254804.

[3] A. Dominguez, S. Udayakumaran, R. Barua, Heap data allocation to scratch-pad memory in embedded systems, J. Embedded Comput. 1 (4) (2005) 521–540.

[4] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, P. Marwedel, Scratchpad memory: a design alternative for cache on-chip memory in embedded systems, in: Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No.02TH8627), 2002, pp. 73–78, http://dx.doi.org/10.1145/774789.774805.

[5] R.M. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, P. Marwedel, Comparison of cache-and scratch-pad based memory systems with respect to performance, area and energy consumption, Technical report #762, Universität Dortmund, 2001, URL https://api.semanticscholar.org/CorpusID:2889002.

[6] S. Wilton, N. Jouppi, CACTI: an enhanced cache access and cycle time model, IEEE J. Solid-State Circuits 31 (5) (1996) 677–688, http://dx.doi.org/10.1109/4.509850.

[7] P. Chakraborty, P.R. Panda, S. Sen, Partitioning and data mapping in reconfigurable cache and scratchpad memory–based architectures, ACM Trans. Des. Autom. Electron. Syst. 22 (1) (2016) http://dx.doi.org/10.1145/2934680.

[8] W. Zhang, L. Wu, Exploiting hybrid SPM-cache architectures to reduce energy consumption for embedded computing, in: 2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS), 2014, pp. 340–347, http://dx.doi.org/10.1109/HPCC.2014.59.

[9] Intel Corporation, Pin 3.27, 2023, https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.

[10] T. Van Achteren, M. Ade, R. Lauwereins, M. Proesmans, L. Van Gool, J. Bormans, F. Catthoor, Transformations of a 3D image reconstruction algorithm for data transfer and storage optimisation, in: Proceedings Tenth IEEE International Workshop on Rapid System Prototyping. Shortening the Path from Specification To Prototype (Cat. No.PR00246), 1999, pp. 81–86, http://dx.doi.org/10.1109/IWRSP.1999.779035.

[11] C. Lezos, G. Dimitroulakos, I. Latifis, K. Masselos, A locality optimizer for loop-dominated applications based on reuse distance analysis, ACM Trans. Des. Autom. Electron. Syst. 25 (6) (2020) http://dx.doi.org/10.1145/3398189.

[12] L. Nachtergaele, F. Catthoor, B. Kapoor, S. Janssens, D. Moolenaar, Low power storage exploration for h.263 video decoder, in: VLSI Signal Processing, IX, 1996, pp. 115–124, http://dx.doi.org/10.1109/VLSISP.1996.558310.

[13] Ohio State University, PolyBench/C 4.2.1, 2016, https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

[14] B. Egger, S. Kim, C. Jang, J. Lee, S.L. Min, H. Shin, Scratchpad memory management techniques for code in embedded systems without an MMU, IEEE Trans. Comput. 59 (8) (2010) 1047–1062, http://dx.doi.org/10.1109/TC.2009.188.

[15] M. Verma, L. Wehmeyer, P. Marweclel, Dynamic overlay of scratchpad memory for energy minimization, in: International Conference on Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004, 2004, pp. 104–109.

[16] S. Udayakumaran, R. Barua, Compiler-decided dynamic memory allocation for scratch-pad based embedded systems, in: Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CASES '03, Association for Computing Machinery, New York, NY, USA, 2003, pp. 276–286, http://dx.doi.org/10.1145/951710.951747.

[17] R. McIlroy, P. Dickman, J. Sventek, Efficient dynamic heap allocation of scratch-pad memory, in: Proceedings of the 7th International Symposium on Memory Management, ISMM '08, Association for Computing Machinery, New York, NY, USA, 2008, pp. 31–40, http://dx.doi.org/10.1145/1375634.1375640.

[18] X. Tao, J. Pang, J. Xu, Y. Zhu, Compiler-directed scratchpad memory data transfer optimization for multithreaded applications on a heterogeneous many-core architecture, J. Supercomput. 77 (12) (2021) 14502–14524, http://dx.doi.org/10.1007/s11227-021-03853-x.

[19] L. Alvarez, L. Vilanova, M. Moreto, M. Casas, M. Gonzàlez, X. Martorell, N. Navarro, E. Ayguadé, M. Valero, Coherence protocol for transparent management of scratchpad memories in shared memory manycore architectures, in: 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), 2015, pp. 720–732, http://dx.doi.org/10.1145/2749469.2750411.

[20] M. Shekarisaz, A. Hoseinghorban, M. Bazzaz, M. Salehi, A. Ejlali, MASTER: Reclamation of hybrid scratchpad memory to maximize energy saving in multi-core edge systems, IEEE Trans. Sustain. Comput. (2021) 1, http://dx.doi.org/10.1109/TSUSC.2021.3049447.

[21] L.A. Bathen, N. Dutt, HaVOC: A hybrid memory-aware virtualization layer for on-chip distributed ScratchPad and non-volatile memories, in: Proceedings of the 49th Annual Design Automation Conference, DAC '12, Association for Computing Machinery, New York, NY, USA, 2012, pp. 447–452, http://dx.doi.org/10.1145/2228360.2228438.

[22] J. Hu, Q. Zhuge, C.J. Xue, W.-C. Tseng, E.H.-M. Sha, Management and optimization for nonvolatile memory-based hybrid scratchpad memory on multicore embedded processors, ACM Trans. Embed. Comput. Syst. 13 (4) (2014) http://dx.doi.org/10.1145/2560019.

[23] V. Seshadri, T. Mullins, A. Boroumand, O. Mutlu, P.B. Gibbons, M.A. Kozuch, T.C. Mowry, Gather-scatter DRAM: In-DRAM address translation to improve the spatial locality of non-unit strided accesses, in: Proceedings of the 48th International Symposium on Microarchitecture, in: MICRO-48, Association for Computing Machinery, New York, NY, USA, 2015, pp. 267–280, http://dx.doi.org/10.1145/2830772.2830820.

[24] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, T. Tateyama, Impulse: building a smarter memory controller, in: Proceedings Fifth International Symposium on High-Performance Computer Architecture, 1999, pp. 70–79, http://dx.doi.org/10.1109/HPCA.1999.744334.

[25] C. Zhang, P. Scheffler, T. Benz, M. Perotti, L. Benini, AXI-pack: Near-memory bus packing for bandwidth-efficient irregular workloads, in: 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2023, pp. 1–6, http://dx.doi.org/10.23919/DATE56975.2023.10137243.

[26] R. Komuravelli, M.D. Sinclair, J. Alsop, M. Huzaifa, M. Kotsifakou, P. Srivastava, S.V. Adve, V.S. Adve, Stash: Have your scratchpad and cache it too, SIGARCH Comput. Archit. News 43 (3S) (2015) 707–719, http://dx.doi.org/10.1145/2872887.2750374.

[27] X. Ji, C. Wang, N. El-Sayed, X. Ma, Y. Kim, S.S. Vazhkudai, W. Xue, D. Sánchez, Understanding object-level memory access patterns across the spectrum, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2017.

[28] P. Chakraborty, P.R. Panda, SPM-sieve: A framework for assisting data partitioning in scratch pad memory based systems, in: 2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013, pp. 1–10, http://dx.doi.org/10.1109/CASES.2013.6662527.

[29] S. Wuytack, J.-P. Diguet, F. Catthoor, H. Man, Formalized methodology for data reuse: exploration for low-power hierarchical memory mappings, Very Large Scale Integr. (VLSI) Syst., IEEE Trans. 6 (1999) 529–537, http://dx.doi.org/10.1109/92.736124.

[30] Xilinx, Channelized direct memory access and scatter gather, 2010, https://docs.xilinx.com/v/u/en-US/chan_dma_sg.

[31] Lattice Semiconductor, Scatter-gather direct memory access controller IP core user guide, 2015, https://www.latticesemi.com/-/media/LatticeSemi/Documents/UserManuals/RZ/ScatterGatherDirectMemoryAccessControllerIPCoreUsersGuide.ashx?document_id=24824.

[32] Intel Corporation, Scatter-gather DMA controller core, 2018, https://www.intel.com/content/www/us/en/docs/programmable/683130/21-4/scatter-gather-dma-controller-core.html.

[33] A. Yousefzadeh, J. Stuijt, M. Hijdra, H.-H. Liu, A. Gebregiorgis, A. Singh, S. Hamdioui, F. Catthoor, Energy-efficient in-memory address calculation, ACM Trans. Archit. Code Optim. (2022) http://dx.doi.org/10.1145/3546071, Just Accepted.

[34] J. Lowe-Power, A.M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, S. Bharadwaj, G. Black, G. Bloom, B.R. Bruce, D.R. Carvalho, J. Castrillón, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, M. Fariborz, A.F. Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S.A.R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T.M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L.E. Olson, M.S. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M.D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, Z. Wang, N. Wehn, C. Weis, D.A. Wood, H. Yoon, É.F. Zulian, The gem5 simulator: Version 20.0+, 2020, CoRR arXiv:2007.03152, URL https://arxiv.org/abs/2007.03152.

[35] R. Jagtap, S. Diestelhorst, A. Hansson, M. Jung, N. When, Exploring system performance using elastic traces: Fast, accurate and portable, in: 2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), 2016, pp. 96–105, http://dx.doi.org/10.1109/SAMOS.2016.7818336.

[36] MIT and Hewlett-Packard, DynamoRIO, 2001, https://dynamorio.org.

[37] C. Gorgovan, A. d'Antras, M. Luján, MAMBO: A low-overhead dynamic binary modification tool for ARM, ACM Trans. Archit. Code Optim. 13 (1) (2016) http://dx.doi.org/10.1145/2896451.