Estimating the Impact of Architectural and Software Design Choices on Dynamic Allocation of Heterogeneous Memories

Tristan Delizy[1], Stephane Gros[2] Kevin Marquet[1], Matthieu Moy[3], Tanguy Risset[1], Guillaume Salagnac[1]
1: Univ Lyon, INSA-Lyon, Inria – Villeurbanne France `firstname.lastname@insa-lyon.fr`
2: eVaderis – Montbonnot France `stephane.gros@evaderis.com`
3: Univ Lyon, Inria, CNRS, ENS de Lyon, LIP – Lyon, France `matthieu.moy@univ-lyon1.fr`

Abstract—**Reducing energy consumption is a key challenge to the realization of the Internet of Things. While emerging memory technologies may offer power reduction, they come with major drawbacks such as high latency or limited endurance. As a result, system designers tend to juxtapose several memory technologies on the same chip. This paper studies the interactions between dynamic memory allocation and architectural choices regarding this heterogeneity. We provide cycle accurate simulations of embedded platforms with various memory technologies and we show that different dynamic allocation strategies have a major impact on performance. We demonstrate that interesting performance gains can be achieved even for a low fraction of heap objects in fast memory, but only with a clever data placement strategy between memory banks.**

I. Introduction

In a low power system-on-chip the memory hierarchy is traditionally composed of Static RAM (SRAM) and NOR flash. The main feature of SRAM is a fast access time, matching the clock speed of the processor. Its downsides include low integration density and continual power consumption. Flash memory on the other hand is dense, and also non-volatile i.e. it does not require power to retain data. But writing to flash is costly both in time and energy, and also suffers from a very limited write endurance. As a result, flash memory is mostly used in a read-only fashion (e.g. for code) and the amount of SRAM is kept to a minimum in order to lower leakage power.

Emerging memory technologies exhibit different trade-offs and more heterogeneity. Non-Volatile RAM (NVRAM) technologies like MRAM or RRAM open new perspectives on power-management since they can be switched on or off at very little cost. Their characteristics are very dependent on the technology used, but it is now widely known that they will provide a high integration density and fast read access time to persistent data. NVRAM is usually not as fast as SRAM and some technologies have a limited endurance hence are not suited to store frequently modified data. In addition, most NVRAM technologies have asymmetric access times, writes being slower than reads. For a given family of NVRAM technologies, changing design parameters allows different trade-offs, between density and performance for instance.

In high performance computing systems, these technologies allow for novel, attractive trade-offs [1], [2] when designing the storage hierarchy. In the context of embedded systems, the hardware architecture is evolving towards a model where different memory banks, with different hardware characteristics, are directly exposed to software, as it has been the case for scratchpad memories (SPM). This raises questions including:

- What is the expected performance impact of adding fast memory to a system based on NVRAM? In particular: will the addition of a small amount of fast memory result in significant performance improvement?
- How should one adapt and optimize their software memory management to leverage these new technologies?

Other issues like write endurance and (non-)volatility are also relevant but left out of the scope of this paper: we consider only memories with enough endurance to be used as working memory, and keep the non-volatility for future work. We focus on the performance variability between memory banks, and take into account the potential asymmetry between read and write performance.

These issues have already been extensively studied for static memory allocation [3] (code and globals), and important improvements have been obtained with a small portion of fast memory. The problem is however still open for dynamic allocation in the heap, on which we focus in this paper.

The goal of this paper is to provide answers to these questions, through the following contributions:

- We expose the problem of dynamic allocation in multiple heaps located in heterogeneous memories.
- We propose a set of reference software dynamic memory allocation strategies for multi-heap dynamic memory management, we also provide an upper and a lower bound on performance to be reached.
- A cycle accurate simulation environment with precise heterogeneous memory models is used to estimate the performance impact of a change in memory architecture and the impact of the above proposed strategies on performance.

We show that using fast memory for a small portion of the heap (5 to 25 %) can result in a significant performance improvement, almost as high as changing the complete memory into a fast memory. However we also show that this

978-1-5386-7557-1/18/$31.00 ©2018 IEEE

performance gain can only be achieved with a clever dynamic memory allocation strategy.

II. Context and Related Work

A. Non-Volatile Embedded Systems

One of the motivations for this work is the new challenges for power-management raised by the Internet of Things (IoT) [4]. Ultra low-power systems must switch off energy-consuming components (including memories) when inactive to extend the battery life or allow working without battery [5]. An obvious major contribution of NVRAM for these systems is to allow the persistence of the data and state of the program.

Of course handling non-volatility is not that easy [6]–[9], but still many research teams are exploring the design of micro-controller containing NVRAM at various levels [10], [11]. These works are all considering that the future low power sensors will include a mix of different memory technologies: SRAM, possibly different NVRAM technologies, maybe even embedded-DRAM et cetera. This is the motivation for our paper: this memory *heterogeneity*, that we explain below, will require many new optimization techniques in run-time or compile-time memory allocation.

B. Non-Volatile Memory Technologies

One of the first NVRAM technologies to gain commercial market is the FeRAM [12], based on a ferro-electric component, they have been deployed on TI MSP430 FRAM series for several years. Magnetic RAM (MRAM, STT-MRAM) are based on spintronics and should provide a very good endurance and power consumption. Resistive RAM technologies, (RRAM, such as OxRAM for instance) are based on resistivity of isolating materials. Phase Change Memory RAM (PCM) use crystal conductivity [13]. The design of these technologies is not mature yet but the important point for computer scientists is that these technologies vary from one to another along various criteria: cost, endurance, read and write speed, power consumption, retention. Depending on the characteristics of the technologies used and of the type of application executed (memory intensive or compute-intensive), the performance may vary a lot.

Another important research topic is the memory architecture: will these new memories be close or far from the processor? Exploring *heterogeneous memory architecture* is an important open research problem today. We are willing to contribute to this research by evaluating the impact of dynamic heterogeneous memory management.

C. Memory Management

We refer to *memory heterogeneity* when several memory banks with different performance characteristics co-exist on the same system. This is not to be confused with *memory hierarchy* which has been studied a lot in non uniform memory architecture (NUMA) machines. We have chosen in this paper to study architecture without memory management unit (MMU) as it is the most common case in low-power systems.

1) Scratch Pad Management: A particular case of memory heterogeneity has been studied in the context of Scratch Pad Memories. A Scratch Pad Memory (SPM) plays the same role as a hardware cache, but is managed explicitly in software. SPMs for embedded systems have received a lot of interest [3] because of performance predictability.

In general, the objective of an SPM management policy is to allocate as much code and/or data to the SPM as possible. In its simplest form, this allocation is *static*, i.e. it is fixed at compile-time either via static analysis or by profiling the execution, typically on a simulated platform.

A more flexible approach is to allow for *dynamic* replacement of SPM content. The allocation is still decided at compile time, but the program gets instrumented with additional instructions to perform the required loading and unloading operations [14]. Going even further, some papers delay the decision until *runtime*. For instance Egger et. al. [15] propose to load code fragments on-demand via a software cache approach. Cho et al. [16] use run time information collected using dedicated hardware to chose between pre-computed profiles.

In most instances, the Scatchpad is used for code [15], global variables [17] and/or (parts of) the execution stack [18]. Few papers mentioned hereafter address the issue of heap management.

2) Dynamic Allocation: SPM for heap objects have been studied by Mc Ilroy et al. [19], their proposal concerns the optimization of an allocator for small heaps, but the choice of the destination memory bank for objects is not in their focus. Mück et al. [20] propose to place a subset of the heap objects into a fast memory, but their approach requires from the programmer to adapt the application code to the memory architecture and is unfeasible in the context of legacy code. Dominguez et al. [21] have extended a method for placing code, global data and stack into an SPM to also address heap data, but only on examples making a light usage of dynamic memory allocation.

3) Heterogeneity Management for Scratch Pads: Some researchers have already studied the interaction between NVRAM technology and scratch pad management. Rodriguez et al. [17] study the design of an MRAM-based SPM, exploiting this particular technology trade-off between write energy and retention time to use less energy to write short-lived data. Li et al. [22] considered static allocation for a hybrid SPM containing NVRAM and SRAM, Hu et al. [23] use dynamic replacement on a similar architecture, with a focus on read / write asymmetry and endurance aspects of the NVRAM considered (PCM).

Hence the interest for SPM software management and for their interaction with NVRAM technologies, no work has yet studied the impact of memory heterogeneity on dynamic memory management.

D. Scope of the Paper

Our long-term objective is to provide tools for generalizing dynamic heap allocation in embedded systems with heterogeneous memories such as the architecture presented in [10]. In this paper, we first show that a straightforward adaptation of classical allocation techniques to manage multiple heaps would yield unsatisfactory results. For this purpose, we implement a platform simulator with a configurable memory architecture, as illustrated in Fig. 1. We run several embedded benchmarks on different memory configurations and study the variations on performance. This study focuses on a scenario with only two heaps, backed by two memory banks with different access latencies, i.e. "fast" and "slow" memory. Our experiments show that the placement of each data object in either fast or slow memory has a significant impact on performance.

In order to focus our experiments on heap allocation, we try and minimize the impact of other memory accesses, i.e. to code, global data, and the execution stack. This is achieved by storing these elements in another, separate Scratch Pad bank with no latency penalty. Our justification for doing so is that memory allocation of code, global and stack data has been studied extensively in the literature. In other words, we assume that state-of-the-art techniques like those presented in the previous section work perfectly. Studying the interaction between heap allocation and code/globals/stack allocation is outside the scope of this paper.

In this paper, all memory access latencies are expressed in time units of CPU clock cycles. Because the platform has no CPU cache, all instructions are fetched from the scratchpad. Thus, the clock frequency of the CPU must match the read latency of the fastest bank. Of course all these hypotheses are simplifications, but they allow us to focus on the issue of interest, i.e. the impact of dynamic allocation on the performance of the program.

III. Multi-Heap Dynamic Allocation: Mechanisms and Strategies

In this section, we discuss how the usual `malloc()` and `free()` API should be implemented for multi-heap systems.

A. Heap allocation mechanisms

1) Allocating from a single heap: When dealing with data objects whose size and/or number is unknown at compile time, programs have to resort to dynamic memory allocation. Each time the *application* needs a new *object*, it issues an *allocation request* indicating the required size. The role of the *memory manager* is then to search for a free memory block of sufficient size and mark it as occupied. Symmetrically, each time the application issue a *deallocation request* regarding an object, the memory manager marks the corresponding block as free, and also might decide to merge it with other adjacent free blocks. All of that happens in a fixed region of memory referred to as *the heap*. Because the arrival of requests is unpredictable, it may happen that free space becomes fragmented into too many small blocks, to the point of becoming unusable for allocation. This so-called *heap fragmentation* issue has been studied extensively for decades [24]. Theoretical results show that no "perfect" allocation algorithm can be designed. However, in the standard case of a contiguous heap with uniform latency, some algorithms behave well enough to be used by default in programming language implementations. This is the case of the `dlmalloc` [25] algorithm that we use in this study.

2) Allocating from multiple heaps: Things get more complicated with heterogeneous memories. We can either decide to split the heap across several memory banks, or to maintain one distinct heap for each bank. The first approach has the advantage of preserving the `malloc/free` API but requires to adapt allocation algorithm to architecture specifics. With the second approach, we can just use a state-of-the-art allocation algorithm in each heap.

But still one need to decide in which bank to allocate each object, and as we show, these *placement* decisions have a significant impact on the overall program performances.

We cannot leave these decisions to application programmer as he may not be aware about the hardware architecture specifics and moreover, understanding precisely the dynamic allocation profile of an application increases the development complexity.

We propose to structure the memory manager in two layers: one *dispatcher* and (possibly many) *allocators*. The dispatcher receives all allocation requests and is in charge of deciding which heap to allocate to, according to some *placement strategy*. It then forwards the request to the corresponding allocator.

As dynamic allocation handles variables whose size is unknown before execution, it may happen, because of heap fragmentation, that the chosen allocator fails. In that case, the dispatcher must redirect the allocation to another heap. We refer to this situation as a *fall back*.

B. Placement Strategies

The role of the dispatcher is to choose among the available heaps. In this study, we assume that dynamic allocation happens in only two heaps which we refer to as the *fast heap* and the *slow heap*.

1) The "Fast First" placement strategy: The most straightforward strategy is to always try the fast heap first, and only fall back to the slow heap in case of failure. In our experiments, we use this as our baseline strategy.

However, we cannot implement this strategy naively, as it would yield poor performance. Indeed, after the fast heap has filled up, almost all allocations will have to fall back to the slow heap. Running the full allocation algorithm for every request, just to discover that the heap is full, is a waste of time.

To overcome this problem, we tweak the implementation of `dlmalloc` in order to optimize for the case of allocation failure. With this modification, the cost of repeatedly falling back remains low enough that the performance of the Fast First strategy remains acceptable.

2) Using profiling to devise "Offline strategies": The goal of this study is to evaluate the potential benefits of a clever placement strategy. In other words, we want to know how our

"Fast First" strategy compares to the optimal placement. With this in mind, we design and evaluate *clairvoyant* strategies: by knowing the future in advance, the dispatcher can always take the right decisions.

To predict the future, we leverage the fact our target system is deterministic, both in terms of hardware (single core, in-order CPU) and of software (bare-metal sequential programs). In other words, running the same application twice with the same inputs will produce the same results. Even if we change the memory architecture (number of banks, latencies) and/or the memory manager (dispatcher, allocators), objects will be allocated and deallocated in the exact same order, and accesses to each object will remain the same. This allows us to collect a detailed *profile* for each application, and then take all placement decisions statically, by formulating and solving an optimization problem. Moreover, because our target architecture have no caches the order of the requests have only a marginal impact on performance.

We refer to this approach as using an *offline* placement strategy, because the dispatcher merely follows a pre-computed *placement sequence*. On the contrary we use the term *online* strategy when the placement decisions are taken at runtime.

3) The "ILP" placement strategy: The first offline strategy that we study consists in formulating the placement problem as an Integer Linear Program and feeding it to an ILP solver. The optimization problem is very close to a knapsack problem: for each object, we know its size, lifetime, and number of read/writes accesses, so we can deduce the cost of placing it in either heap, taking into account possible latency asymmetry. We also introduce constraints to model the bank capacity: at every instant, the total size of the objects in one heap should be smaller than the bank size.

The utility of this strategy is to approximate an optimal placement. However, it is only an approximation, as we will discuss in Sec. III-B5.

4) The "Density" placement strategy: This strategy is based on a simple, greedy algorithm. The idea is to go through all objects in the profile and rank them according to some metric. Then, only the "most relevant" objects are considered for placement in the fast heap, until the heap is full. To assess relevance, we propose a so-called *access density* metric which we define as the number of accesses to a heap object b divided by its space-time contention with other objects:

$$Density(b) = \frac{Access\_count(b)}{Size(b) \times Overlap(b)}$$

Beside the access count to the object and the size it occupies in memory, we consider $Overlap(b)$, the number of objects allocated during its lifetime, allowing to take into account the number of times the associated memory block will be occupied when allocating other objects.

The utility of this strategy is to investigate which statistics are relevant for object placement, and help us to design online strategies in the future.

5) Offline Strategies and Heap Fragmentation: In both strategies described above, we have to make an important simplification to make the problem tractable: we assume that the total size of objects allocated in one heap can reach up to the total capacity of the heap. However, because of the fragmentation problem, this is typically not the case in practice. Ignoring fragmentation while solving the offline placement problem is thus an overly optimistic assumption. This is a problem because an overly optimistic placement sequence will cause the allocator to fail repeatedly (at a cost, cf Sec. III-B1) and also many objects will end up in the "wrong" bank.

It would not be practical to include every implementation detail of the allocator in the model, so we cannot calculate fragmentation in advance. This means that our offline strategies are not implementing the *optimal* placement. Instead, they each yield an *under-approximation* of the optimum. To improve this approximation, we run the solvers using a smaller heap capacity which ensures that the allocations will succeed at run time despite heap fragmentation.

6) Finding an Upper Bound on Achievable Performance: On the other hand, we also want an *over-approximation*. For that we use the ILP solver with the real heap capacity, and then we apply the result on a "relaxed" architecture where all memory banks have their usual characteristics but with no capacity limit. In this conditions, the allocator never fails, and the overly optimistic placement sequence can be applied exactly. This "pseudo-strategy" guarantees a better (or equal) performance that would be achievable on the real system.

As our experimental results will show, in practice both approximations are quite close to each other, which allows us to talk about "the optimum" and use it to evaluate concrete strategies.

IV. Experiments

A. Experimental Setup

Our simulation environment is built using SystemC/TLM libraries and based on MIPS32 Instruction Set Simulator from the SoCLib project [26]. We consider an SPM with multiple memory banks including a *fast memory* and *slow memory* which will contain heap data. As was discussed in Sec. II-D, everything else (code, static variables and the stack) is assumed to be stored in a SRAM-like memory with no latency penalty. This allows our experiments to highlight the performance impact of heap management. The architecture is illustrated in Fig. 1.

Our experiments consists in running several benchmarks on the simulator of this architecture, with varying parameters such as the proportion of heap in fast memory bank or the memory bank latencies. The read and write latencies for the considered technologies (see fig. 3) are given in cycles, and as stated in Section II-D, in the presence of memory heterogeneity, the fastest memory on the platform yields read latency of a single cycle.

1) Application benchmarks: We choose several embedded programs to serve as application benchmarks. Each of them

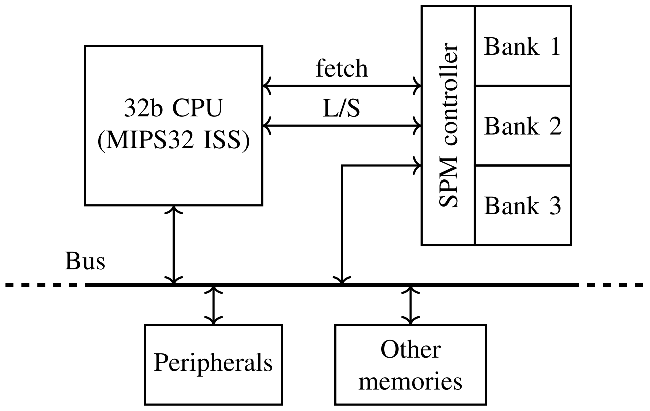

Fig. 1. Overview of our simulated platform.

App.	Object Count	Maximum Heap Size
json	1638	38 kB
dijkstra	14980	10 kB
jpg2000	10257	1665 kB
h263	53821	1232 kB

(a)

Arch.	Fast	Slow
A0	100%	0%
A1	75%	25%
A2	50%	50%
A3	25%	75%
A4	10%	90%
A5	5%	95%
A6	0%	100%

(b)

Fig. 2. (a) Applications and (b) Memory Architectures

makes significant use of dynamic allocation: Fig. 2.(a) shows the total number of allocation requests (i.e. object count) as well as the maximum heap size for each program. The **json** application is based on an embedded JSON parsing and serialization library [27]. The **dijkstra** program is taken from the Mibench suite [28]. It computes shortest paths through a graph represented by an adjacency matrix. The **h263** application is based on a video compression algorithm. It comes from the Mediabench 2 video benchmark suite [29]. Finally the **jpg2000** application is an image compression algorithm, also from Mediabench 2 video. We consider here only application making a significant use of dynamic memory allocation.

2) Memory architectures: In our experiments, we vary the distribution of the heap between the *fast* and *slow* memory banks. Because each benchmark has a different maximum heap size, we set the total size of both banks to this maximum. Then, each *memory architecture* is expressed as a percentage of fast vs. slow memory, as illustrated in Fig. 2.(b). Our reference execution time is obtained on architecture A6, i.e. with the whole heap stored in slow memory. The best execution time is always obtained on architecture A0, i.e. with the whole heap stored in fast memory. However for architectures A1–A5, each placement strategy yields different performance gains.

3) Experiments Sets: As illustrated in figure 3, we consider two pairs of memory technologies for the fast and slow memory banks. In our first experiment set, SRAM is used to compensate for the access times of a generic, symmetric NVRAM with a 10 cycles latency both for reading and writing. The second experiment set is a more realistic scenario where

Experiment Set	Fast (R/W)	Slow (R/W)
(1) SRAM + NVRAM	1/1	10/10
(2) fast MRAM + dense MRAM	1/3	2/30

Fig. 3. Read and Write latencies for heap memory banks (in CPU cycles).

we consider two actual MRAM types from the industry [30] with different flavors: “fast MRAM” favors low access times and “dense MRAM” achieves high density at the cost of slower access.

4) Strategies for Dynamic Allocation: Each experiment set consists in evaluating, given the considered memory technologies, each application on each architecture (A0 to A6, see fig. 2). The architectures A0 and A6 are evaluated with single heap, along with our baseline (Fast First strategy) for architectures A1 to A5. The ILP and Density offline solutions are evaluated on architectures A1 to A5 and we also provide an upper bound for ILP strategy, as explained in section III.

B. Results and Discussion

Our experimental results are shown on Fig. 4 for both set of experiments referenced in Fig. 3. While the overall trends are similar in both columns, the vertical scale is typically smaller on the right. This is because, even for heap objects, programs tend to read much more than they write. As such, the 10-cycle read latency of the slow heap in experiment set 1 is actually a lot worse than the 30-cycle write latency. Some programs however (json, jpg2000) have more balanced access profiles and show more similar speedups.

In all graphs, the blue curve is the performance obtained by our baseline “Fast First” placement strategy. Most often, adding fast memory to the system results in a shorter execution time. However, for some applications (json, dijkstra) the performance curve is not strictly increasing. We even observe a slight slowdown when we replace 5% of the heap with fast memory (A1), compared to having *only* slow memory (A0). This is caused by the multi-heap strategy overhead. In other words, the Fast First strategy is sometimes unable to exploit additional, faster memory.

On the contrary, offline strategies generally perform very well on A5. For instance jpg2000 and h263 experience speedups which are almost equivalent to replacing all memory with fast memory. This means that even a small amount of fast memory is enough to get significant gains, provided that we have a clever placement strategy for heap objects.

In this regard, the Fast First strategy performs quite poorly. In most cases, Fast First would require about 50% of fast memory just to reach the same speedup than offline strategies achieve with 5%. This means that, for dynamic allocation in heterogeneous memory systems, a good online strategy should be designed.

We designed the Density strategy with the objective of exploring which criteria are relevant for object placement. Indeed, while the ILP strategy yields near-optimal results, it does not help us understand which objects are the most relevant. In most cases Density performs almost as well as ILP, which is promising: we could imagine an online strategy

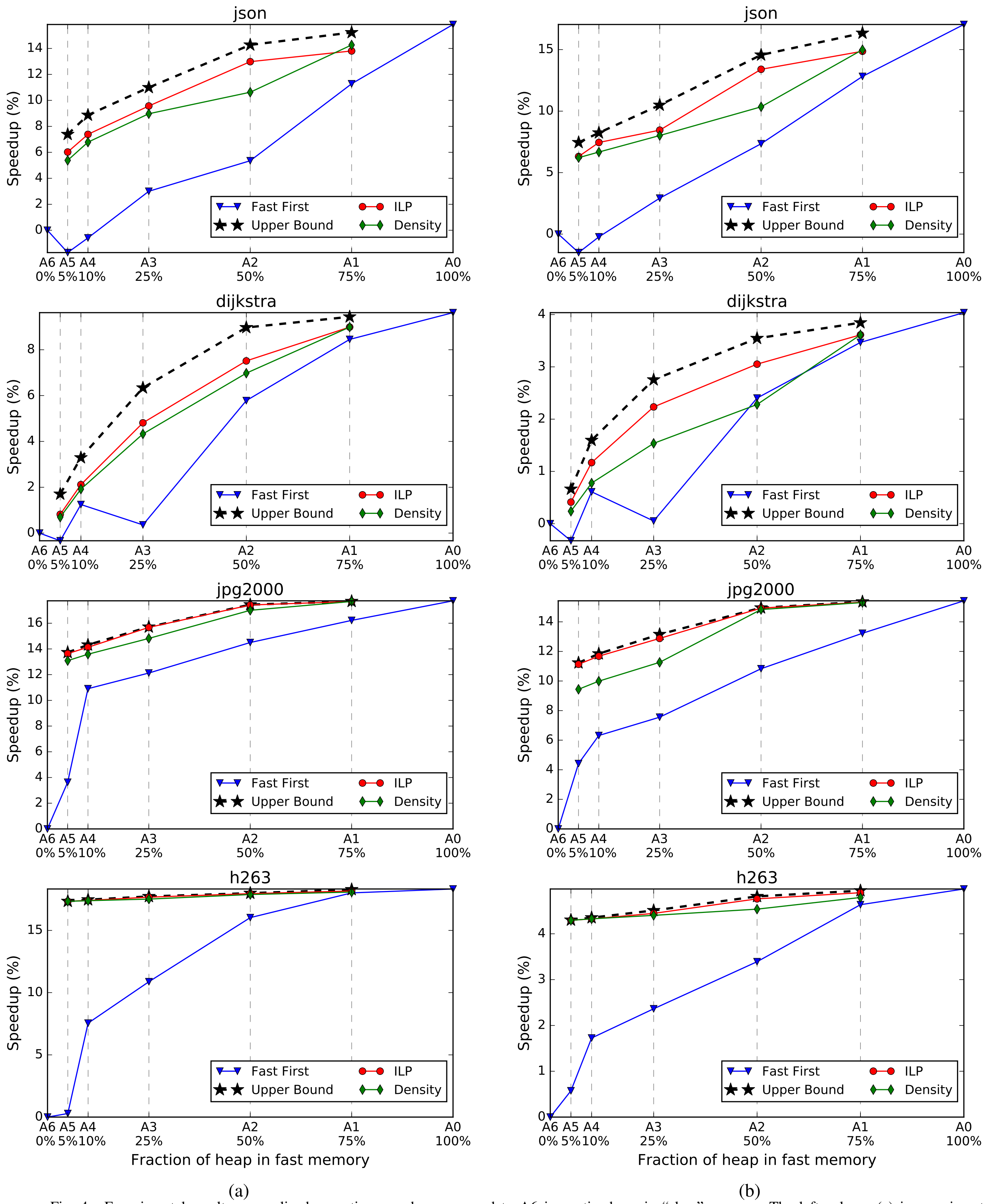

Fig. 4. Experimental results: normalized execution speedup compared to A6 i.e. entire heap in "slow" memory. The left column (a) is experiment set 1 (SRAM + NVRAM) and the right column is experiment set 2 (fast MRAM + slow MRAM).

that would measure and then use access density to help with object placement.

However in some cases offline strategies fail to reach the Upper Bound. As was discussed in Sec. III-B5, this is largely due to the inability of the optimization problem to account for heap fragmentation (json, dijkstra). But it also reveals interesting properties of our benchmarks. For instance in dijkstra and jpg2000, the green and red lines are a lot further apart in experiment set 2, i.e. with asymmetric R/W latencies, than they are in experiment set 1. This shows a limitation of our Density strategy: sometimes it would be worth distinguishing between read and write accesses to each object. But even on asymmetric memory, the Density strategy still seems quite promising.

V. Conclusion and Future Work

We show in this paper that dynamic memory allocation can leverage memory heterogeneity to improve application performance in low power systems. And besides, our results highlight the importance of the software placement strategy to achieve these improvements at low memory costs. Indeed our experiments show that the straightforward "fast first" strategy performs badly with small amounts of fast memory.

Using application profiling is an interesting lead to elaborate efficient online strategies. However these strategies would remain specific to one application. We are currently working on the proposal of efficient online placement strategies for multi-heap dynamic allocation.

A solution to avoid the profiling phase is to consider hardware counters integration, granting to the software information required for an adaptive efficient online strategy. In this perspective, the good results shown by our metric of access density are promising.

Funding: Région Auvergne-Rhône-Alpes, Insa-Spie IoT Chair

References

[1] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy, "Operating system implications of fast, cheap, non-volatile memory," in *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, 2011.

[2] H. Volos, S. Panneerselvam, S. Nalli, and M. M. Swift, "Storage-class memory needs flexible interfaces," in *Proceedings of the 4th Asia-Pacific Workshop on Systems*. New York, NY, USA: ACM, 2013, p. 11.

[3] S. Alam and N. Horspool, "A survey: Software-managed on-chip memories," *Computing and Informatics*, vol. 34, no. 5, 2015.

[4] H. Jayakumar, K. Lee, W. S. Lee, A. Raha, Y. Kim, and V. Raghunathan, "Powering the internet of things," in *IEEE/ACM International Symposium on Low Power Electronics and Design*. IEEE, 2014.

[5] A. S. Adila, A. Husam, and G. Husi, "Towards the self-powered Internet of Things (IoT) by energy harvesting," in *International Symposium on Small-scale Intelligent Manufacturing Systems*, 2018.

[6] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini, "Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems," *IEEE Embedded Systems Letters*, vol. 7, no. 1, 2015.

[7] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac, "Peripheral State Persistence For Transiently Powered Systems," INRIA, Research Report RR-9018, Feb. 2017.

[8] B. Ransford, J. Sorber, and K. Fu, "Mementos: System support for long-running computation on rfid-scale devices," *SIGARCH Comput. Archit. News*, 2011.

[9] H. Jayakumar, A. Raha, and V. Raghunathan, "Quickrecall: A low overhead hw/sw approach for enabling computations across power cycles in transiently powered computers," in *International Conference on VLSI Design and International Conference on Embedded Systems*, 2014.

[10] Layer, Javerliac, Bernard-Granger, Decloedt, Becker, Jabeur, Claireux, Dieny, Prenat, Pendina, Di, and et al., "Reducing System Power Consumption Using Check-Pointing on Nonvolatile Embedded Magnetic Random Access Memories," *ACM Journal on Emerging Technologies in Computing Systems*, 2016.

[11] K. Ma, Y. Zheng, S. Li, K. Swaminathan, X. Li, Y. Liu, J. Sampson, Y. Xie, and V. Narayanan, "Architecture exploration for ambient energy harvesting nonvolatile processors," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, IEEE. Washington, DC, USA: IEEE, 2015, pp. 526–537.

[12] Z. Fan, J. Chen, and J. Wang, "Ferroelectric hfo 2 -based materials for next-generation ferroelectric memories," *Journal of Advanced Dielectrics*, vol. 06, no. 02, p. 1630003, Jun 2016.

[13] S. Yu and P. Y. Chen, "Emerging memory technologies: Recent trends and prospects," *IEEE Solid-State Circuits Magazine*, vol. 8, no. 2, pp. 43–56, Spring 2016.

[14] J. Hu, C. J. Xue, Q. Zhuge, W.-C. Tseng, and E. H.-M. Sha, "Towards energy efficient hybrid on-chip scratch pad memory with non-volatile memory," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. Washington, DC, USA: IEEE, 2011.

[15] B. Egger, S. Kim, C. Jang, J. Lee, S. L. Min, and H. Shin, "Scratchpad Memory Management Techniques for Code in Embedded Systems without an MMU," *IEEE Transactions on Computers*, vol. 59, no. 8, p. 1047–1062, Aug 2010.

[16] D. Cho, S. Pasricha, I. Issenin, N. D. Dutt, M. Ahn, and Y. Paek, "Adaptive scratch pad memory management for dynamic behavior of multimedia applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 4, pp. 554–567, 2009.

[17] G. Rodríguez, J. Touriño, and M. T. Kandemir, "Volatile STT-RAM Scratchpad Design and Data Allocation for Low Energy," *ACM Transactions on Architecture and Code Optimization*, vol. 11, no. 4, 2014.

[18] A. Shrivastava, A. Kannan, and J. Lee, "A software-only solution to use scratch pads for stack data," *IEEE Transactions on computer-aided design of integrated circuits and systems*, vol. 28, no. 11, 2009.

[19] R. McIlroy, P. Dickman, and J. Sventek, "Efficient dynamic heap allocation of scratch-pad memory," in *Proceedings of the 7th international symposium on Memory management*, 2008.

[20] T. R. Mück and A. A. Fröhlich, "Run-time scratch-pad memory management for embedded systems," in *37th Annual Conference on IEEE Industrial Electronics Society*.

[21] A. Dominguez, S. Udayakumaran, and R. Barua, "Heap data allocation to scratch-pad memory in embedded systems," *Journal of Embedded Computing*, vol. 1, no. 4, pp. 521–540, 2005.

[22] Q. Li, Y. Zhao, J. Hu, C. J. Xue, E. Sha, and Y. He, "Mgc: Multiple graph-coloring for non-volatile memory based hybrid scratchpad memory," in *16th Workshop IEEE Interaction between Compilers and Computer Architectures*, 2012.

[23] J. Hu, C. J. Xue, Q. Zhuge, W.-C. Tseng, and E. H.-M. Sha, "Data allocation optimization for hybrid scratch pad memory with SRAM and nonvolatile memory," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 6, p. 1094–1102, Jun 2013.

[24] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," in *Memory Management*. New York, NY, USA: Springer, 1995, pp. 1–116.

[25] D. Lea, "A memory allocator," http://g.oswego.edu/dl/html/malloc.html, 2012.

[26] SOCLIB, "Projet SOCLIB: Plate-forme de modélisation et de simulation de systèmes integrés sur puce. Technical report, CNRS, 2003."

[27] K. Gabis, "Parson: Lightweight JSON library written in C," https://github.com/kgabis/parson, 2012.

[28] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization*, 2001.

[29] J. E. Fritts, F. W. Steiling, and J. A. Tucek, "Mediabench II video: expediting the next generation of video systems research," *Microprocessors and Microsystems*, vol. 33, 2009.

[30] G. Foundries, "Embedded memory: emram, eflash, sip," https://www.globalfoundries.com/sites/default/files/product-briefs/pb-emem.pdf, 2018.