# Volatile STT-RAM Scratchpad Design and Data Allocation for Low Energy

GABRIEL RODRÍGUEZ and JUAN TOURIÑO, Universidade da Coruña
MAHMUT T. KANDEMIR, Pennsylvania State University

On-chip power consumption is one of the fundamental challenges of current technology scaling. Cache memories consume a sizable part of this power, particularly due to leakage energy. STT-RAM is one of several new memory technologies that have been proposed in order to improve power while preserving performance. It features high density and low leakage, but at the expense of write energy and performance. This article explores the use of STT-RAM–based scratchpad memories that trade nonvolatility in exchange for faster and less energetically expensive accesses, making them feasible for on-chip implementation in embedded systems. A novel multiretention scratchpad partitioning is proposed, featuring multiple storage spaces with different retention, energy, and performance characteristics. A customized compiler-based allocation algorithm suitable for use with such a scratchpad organization is described. Our experiments indicate that a multiretention STT-RAM scratchpad can provide energy savings of 53% with respect to an iso-area, hardware-managed SRAM cache.



Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*; B.3.3 [**Memory Structures**]: Performance Analysis and Design Aids; D.3.4 [**Programming Languages**]: Processors—*Memory Management*

General Terms: Design, Performance

Additional Key Words and Phrases: Scratchpad, STT-RAM, Relaxed-retention

**ACM Reference Format:**
Gabriel Rodríguez, Juan Touriño, and Mahmut T. Kandemir. 2014. Volatile STT-RAM scratchpad design and data allocation for low energy. ACM Trans. Architec. Code Optim. 11, 4, Article 38 (December 2014), 26 pages.
DOI: http://dx.doi.org/10.1145/2669556

## 1. INTRODUCTION

In the early 2000s, computer architecture trends switched to multicore scaling as a response to various architectural challenges that severely diminished the gains of further frequency scaling. This approach has enabled processors to take advantage of increasing transistor counts, according to Moore's Law, for the last decade. In order

This work is supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds of the EU (Project TIN2013-42148-P) and by the Galician Government under the Consolidation Program of Competitive Reference Groups (ref. GRC2013-055). Mahmut Kandemir is supported in part by NSF grants 1439021, 0963839, 1409095, 1213052 and grants from Intel and Microsoft. Any opinions, findings, and conclusions or recommendations expressed in this article are the authors', and do not necessarily reflect the views of the NSF.
Authors' addresses: G. Rodríguez and J. Touriño, Department of Electronics and Systems, Universidade da Coruña, Facultade de Informática, Campus de Elviña, 15071 A Coruña, Spain; M. T. Kandemir, Department of Computer Science and Engineering, Pennsylvania State University, 354C IST Building, University Park, PA 16802; email: {gabriel.rodriguez, juan.tourino}@udc.es; kandemir@cse.psu.edu.

DOI: http://dx.doi.org/10.1145/2669556

to keep dynamic power consumption of CMOS transistors constant when increasing their number, supply and threshold voltages are scaled with feature size. However, this exponentially increases subthreshold voltage and, therefore, static power consumption. In current technologies, leakage energy is large enough as to be comparable to dynamic energy. As a result, current processors are rapidly approaching the power wall [Borkar and Chien 2011].

Besides, multicore scaling makes the memory wall problem ever worse. As the number of cores increases, so does the stress on the memory hierarchy. As a result, larger on-chip caches are required to avoid main memory bottleneck. On-chip caches using SRAM memory typically represent up to 45% of the on-chip energy consumption, and more than 50% of on-chip area [Banakar et al. 2002].

Various new memory technologies have been proposed to improve on the weaknesses of SRAM. STT-RAM is one of these nonvolatile technologies, which has also been proposed as a main memory alternative [Kultursay et al. 2013]. It features much better endurance and performance than other magnetic memory technologies. Compared to SRAM, it is up to 4 times denser and has much lower leakage energy. This enables the implementation of very large on-chip memories with near-zero static consumption, which alleviates both main memory stress and power consumption. However, STT-RAM features higher access energy and latencies than SRAM, making it unsuitable for the implementation of on-chip memories. Storage-class STT-RAM preserves data for at least 10 years, which is a much larger time span than the typical on-chip data retention. Smullen et al. [2011a] propose relaxing this nonvolatility as a means to reduce access latency and dynamic energy in order to make the technology competitive for on-chip memories.

Scratchpad memories (SPM) [Banakar et al. 2002; Panda et al. 1997] are fast, software-controlled on-chip memories. Architecturally, their main difference from a conventional hardware-managed cache is that an SPM does not require a tag array or tag comparison logic. As a result, both area and energy per access are reduced when compared to a cache of the same capacity. The application is responsible for efficiently allocating data to the scratchpad, either explicitly or with the aid of the compiler. If it is possible to predict which data will be accessed in an application-custom manner, as is usually the case for applications in the embedded domain, then an SPM can provide significant energy and performance advantages [Yanamandra et al. 2008; Shaffer et al. 2010].

We propose the implementation of on-chip SPMs using STT-RAMs with relaxed volatility as a means to further take advantage of the area and energy characteristics of this technology. Specifically, our main contributions are:

—The proposal of a multiretention, STT-RAM-based SPM architecture and design methodology for embedded systems. More specifically, the SPM is divided into multiple "regions," each with different performance, power, and retention characteristics.
—A novel compiler-based data allocation algorithm customized for the proposed multiregion SPM design. Short-lived data are brought on-chip using fast, low-energy regions. Longer-lived tiles are accommodated to regions with higher retention, capable of fully exploiting their locality.
—An experimental evaluation showing that the proposed design offers potential savings in energy consumption over 60% when compared with different iso-area on-chip memory designs.

The rest of this article is organized as follows. Section 2 discusses related work. Section 3 describes the relevant architectural features of STT-RAM memories, and details the design process for a multiretention SPM built with relaxed volatility STT-RAM. Section 4 covers the allocation algorithm proposed to take advantage of the

proposed memory architecture. Section 5 evaluates the benefits of the multiretention SPM using several benchmarks and on-chip memory organizations. Section 6 presents our conclusions.

## 2. RELATED WORK

Esmaeilzadeh et al. [2011] predicted that, in years to come, an increasing fraction of the chip will be dark silicon, that is, either idle or significantly underclocked. Taylor [2012] has observed a trend to exploit this dark silicon to introduce specialized functional units that exploit the particularities of a computation to achieve power efficiency. Venkatesh et al. [2010] introduce the concept of "conservation cores": specialized processors that focus on reducing energy instead of increasing performance, used for computations that cannot take advantage of hardware acceleration. Hardavellas et al. [2011] propose a server-oriented architecture with specialized cores for different workloads. Kultursay et al. [2012] design an architecture that includes CMOS and TFET cores and an automated runtime scheme to maximize performance under a fixed power budget.

In recent years, many works have focused on power efficiency through the use of new memory technologies. Phase change memory has been proposed as an alternative to DRAM for main memory [Lee et al. 2010; Coburn et al. 2011; Qureshi et al. 2009]. Other works focus on the usage of STT-RAM as a main memory technology [Kultursay et al. 2013]. In order to use STT-RAM to implement on-chip memories, its latency and energy consumption need to be reduced. Guo et al. [2010] propose to implement much of the combinational logic and on-chip storage using scalable RAM blocks, and rearchitecting the pipeline. Rasquinha et al. [2010] work at the microarchitectural level to avoid premature eviction of lines from L1 to L2 and subsequent move back to L1. Smullen et al. [2011a] modify the physical properties of STT-RAM cells to relax their nonvolatility and, in turn, improve latency and energy. These cells are used with a simple refresh policy to build efficient caches. Sun et al. [2011] propose to use different retention levels in different cache regions. L1 caches are implemented using fast, low-retention cells. Lower-level caches are implemented using hybrid designs that include volatile and nonvolatile regions. A migration policy between regions based on write intensity of each cache block is developed. Jog et al. [2012] also trade off nonvolatility for performance, but they focus on optimizing the refresh interval and employ a secondary buffer to minimize the number of writebacks to main memory due to the expiration of a dirty cache line that is still being used. Li et al. [2013b] design an N-refresh scheme and associated coherence protocol for volatile STT-RAM caches. They also propose a compiler-assisted scheme to rearrange the data layout in a way that minimizes active refreshes, which consume extra energy, by strategically timing application writes (passive refreshes) [Li et al. 2013a]. Bathen and Dutt [2012] use a memory manager to virtualize a hybrid address space with both SRAM and MRAM scratchpads, allowing programmers and compilers to specify memory-aware allocation policies at compile time that get enforced during runtime. Hu et al. propose a dynamic programming algorithm to find the optimal allocation of an SRAM/NVM hybrid SPM for both single-core [Hu et al. 2013] and multicore embedded systems [Hu et al. 2014]. Wang et al. [2013] also use a hybrid system with SRAM and STT-RAM regions, alleviating the write latency and energy by using perpendicular Magnetic Tunnel Junctions (MTJs), and allocating most-written data to the SRAM SPM and most-read data to the STT-RAM SPM.

A number of works have focused on how to allocate data to an SPM. Static methods use a single allocation for the entire program, which never changes during runtime. This avoids runtime transfers between main memory and the scratchpad, but does not adapt to the changing working set of the application. Avissar et al. [2002] find

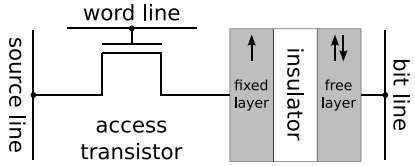


Fig. 1. STT-RAM cell.

the optimal static allocation using integer linear programming. Li et al. [2012] use a graph-coloring approach to allocate a hybrid SPM containing both an SRAM region and a nonvolatile STT-RAM region. Dynamic schemes are able to exploit locality more efficiently. Kandemir et al. [2001] propose a dynamic method that maximizes the reuse of data tiles using both loop and data transformations for affine accesses. Udayakumaran et al. [2006] employ a dynamic allocation method for general applications where data movements between DRAM and scratchpad are under the control of the compiler.

The present work builds on the volatile STT-RAM cells proposed by Smullen et al. [2011a] to implement a novel scratchpad memory organization containing different regions with distinct retention, energy, and performance characteristics. An original reuse-guided method that takes into account variable lifetimes is then employed to dynamically allocate data to the scratchpad, exploiting the different characteristics of each particular region. The multiretention scratchpad can be seen as a form of specialization of the on-chip memory, that can provide significant energy gains for codes with well-defined memory access patterns, as is usually the case in embedded applications.

## 3. STT-RAM-BASED SCRATCHPAD DESIGN

As mentioned in Section 1, scratchpad memories can improve energy and performance over caches for embedded applications. As opposed to SRAM, STT-RAM offers the opportunity to include large on-chip SPMs with near-zero leakage. However, these benefits come at the expense of write energy and latency. Consequently, any attempt to replace an SRAM-based SPM with an STT-RAM–based one should address the associated performance degradation and potential power increase. One way of achieving this is to relax the nonvolatility properties of standard STT-RAM cell designs. This section covers the design of an STT-RAM–based, multiretention SPM, first reviewing the design of the STT-RAM memory cell and then covering how to partition the space and select the technological parameters.

In an STT-RAM cell, an access transistor is connected to a memory element, in a design similar to the 1T1C DRAM cell. With STT-RAM, however, the memory element is implemented using an MTJ, which consists of two ferromagnets separated by an insulating layer. One of the ferromagnets has a fixed magnetization, while the other is allowed to change in response to electrical currents flowing through the device. The general scheme is shown in Figure 1. The magnetization of the free layer can be parallel or antiparallel with respect to that of the fixed layer. Its orientation affects the resistance that the MTJ opposes to a current flowing through it. This effect is used to implement the memory behavior of the device. In order to read the stored value, a small voltage is applied between the MTJ terminals. The current flowing through the device is sensed, and the magnetization state is determined as a result.

The orientation of the free layer will not be maintained in time indefinitely. Eventually, a random bit flip will occur. The probability distribution of this event, therefore the expected *retention time* of an MTJ, depends on its *thermal stability* $\Delta$, which itself depends on the physical parameters of the MTJ [Diao et al. 2007]. In particular, $\Delta \propto V$,

the volume of the MTJ. In this work, we use the approximation by Rizzo et al. [2002] to the retention time $r$, shown in Equation (1). We use this single bit retention time to approximate the MTTF of the entire memory array. All designed memories use ECC.

$$r \simeq 1\ ns \times e^{\Delta} \tag{1}$$

Switching the free layer magnetization of an MTJ is an operation that may take anywhere between picoseconds to tenths of nanoseconds, depending on the MTJ build and the intensity of the applied current [Diao et al. 2007]. In this work, we focus on the *precessional switching* operational mode. This is the physical model that characterizes magnetization changes of the free layer when using an operational current density $J$ much higher than the intrinsic switching current density $J_{c0}$, which is dependent on physical parameters of the MTJ. The time required for a magnetization reversal in precessional switching is at or below 3ns, providing quick operation as required by on-chip memories. However, the write current $I_c(\tau)$ necessary to activate a precessional switching process increases as the write pulse length $\tau$ decreases according to Equation (2), where $A$ is the MTJ area, and $C$ and $\gamma$ are fitting constants [Smullen et al. 2011a].

$$I_c(\tau) = A \cdot \left( J_{c0} + \frac{C}{\tau^{\gamma}} \right) \tag{2}$$

$J_{c0}$ is proportional to the thickness of the MTJ. However, it is not dependent on its area. As such, according to Equations (1) and (2), reducing the area of the MTJ reduces both its retention time and the operational current, while not affecting the intrinsic current density. In the following we assume that retention changes are obtained by modifying the MTJ area, while all other physical parameters remain constant. Another approach for the fabrication of relaxed-retention MTJs, as proposed by Sun et al. [2011], is to modify other physical parameters of the MTJ, such as the saturation magnetization, effective anisotropy, or free layer thickness, while leaving the MTJ size constant and equal to the smallest feature size. This method is potentially more energy efficient than simply downsizing the MTJ, as it can reduce $J_{c0}$, and therefore could improve the results obtained by our approach. Note that the MTJ fabrication method is completely orthogonal to our proposal, which is focused on designing and efficiently using multiretention SPMs.

When using a volatile scratchpad, there must be a mechanism in place to ensure that data are not accessed past the decay time of the memory. Besides, it is necessary to write modified data back to main memory once it is close to expiration. Using a refresh mechanism implies a trade-off between static and dynamic energy consumption. Selecting a low retention for the MTJ implies low dynamic consumption, but refresh frequency and therefore static consumption—which is otherwise not a factor in magnetic memories—are increased. If a higher retention is selected, the increase in the MTJ area implies an increase in the write current, and therefore in dynamic consumption. Note that refresh operations impact the memory bandwidth. A writeback-and-invalidate mechanism would require the realization of a memory controller to periodically check the state of each block, which both increases energy consumption and takes up on-chip area.

The approach proposed in this article is to rely on the compiler to allocate data to the scratchpad and use it within its retention frame. To our knowledge, this is one of the first attempts to compiler-guided, retention-aware data placement in STT-RAM–based SPMs. Since the live time of a given piece of data is variable and application-dependent, it would be desirable to have different scratchpad regions with different retention, energy, and performance characteristics.

Previous works [Jog et al. 2012; Liang et al. 2007] have studied the optimal refresh intervals of different levels of cache by analyzing liveness time of cached data. They

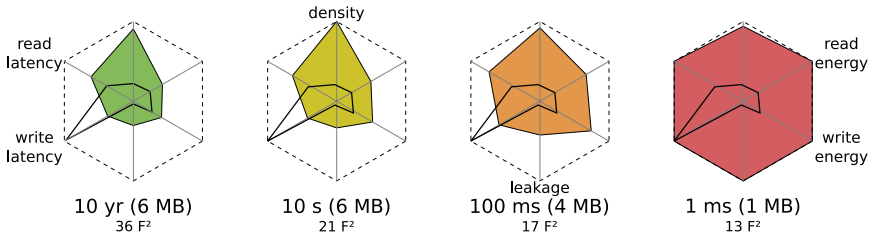


Fig. 2. Characterization of the designed multiretention space. The dashed border marks optimal behavior. The solid black line represents a 4MB ITRS cache. The combined scratchpad regions are iso-area with the cache.

conclude that most data get refreshed after microseconds on an L1 cache. If we analyze the L2 cache, then the majority of the data will be refreshed after a few milliseconds. Other data are alive for variable, larger times. To adapt our scratchpad space to these findings, we create the following "retention regions":

—1 *ms*: to allocate the shorter-lived data.
—100 *ms*: to allocate the data that is alive for a few milliseconds.
—10 *s*: This should accommodate most of the remaining data that goes through the cache, while providing energy gains with respect to a nonvolatile option.
—10 *years*: This is provided as a fall-back region. The compiler will default to it when it cannot guarantee a WCET for a region of code, or when an application-wide allocation is found to be the optimal choice.

We used the STeTSiMS simulation and modeling system [Smullen et al. 2011b] to design the memories. This framework provides performance, energy, and area numbers. It is configured to automatically select optimal architectural parameters (e.g., number of banks) to optimize the cell for different design goals. The baseline cell used is a design by Diao et al. [2007], normalized by Smullen et al. [2011b], with an increased MTJ size of $36F^2$ for a 32nm process to ensure 10-year retention at 350K, typical for a performance microprocessor. The design process for each of the regions of the multiretention scratchpad consisted of multiple iterations of the following steps:

(1) Calculate the required $\Delta_r$ for the desired retention $r$ from Equation (1).
(2) Since $\Delta \propto V$, calculate $A_r$, the area of the MTJ that features the desired retention, as shown in Equation (3).

$$A_r = \frac{A_{10yr} \times \Delta_r}{\Delta_{10yr}} \tag{3}$$

(3) Resize the baseline MTJ to a size that guarantees the desired retention $r$. Analyze the performance/energy characteristics when the write pulse length $\tau$ varies. Select the $\tau$ that minimizes the energy-delay product.

The resulting multiretention scratchpad design is characterized in Figure 2. The solid black line represents a 4MB cache memory following the ITRS roadmap [ITRS 2012] as characterized by CACTI 6.5 [Muralimanohar et al. 2009]. As can be seen in the figure, the resulting sizes for the MTJ cells of the 10s, 100ms and 1ms regions are $21F^2$, $17F^2$, and $13F^2$, respectively. These have been scaled up to provide higher retentions than the nominal ones, approximately 16s, 180ms, and 2ms at 350K. This helps account for process and temperature variations.

After each iteration of the design process, the sizes of the different retention spaces are retuned to make their combination iso-area with the 4MB ITRS cache. The resulting sizes are 6MB for the 10yr and 10s retention regions, 4MB for the 100ms region, and 1MB for the 1ms region. Although it might seem paradoxical due to their better

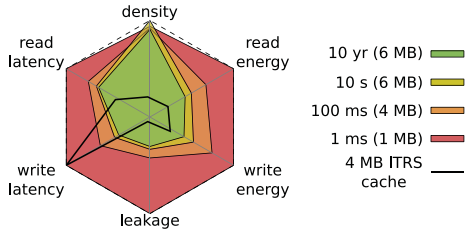


Fig. 3. Integrated view of Figure 2.

energy and latency characteristics, low-retention regions are made smaller than large retention ones for various reasons. First, large-retention regions are more versatile, and can be used to accommodate short-lived data if necessary. As such, higher ratios of low-retention regions to high-retention regions do not improve locality. Second, larger memories are less efficient. If we increase the size of a given region, the latency due to access logic increases, as well as the wire delay, as it will need to be placed farther away from the core. In order to maintain performance, these latency increases need to be offset by reducing $\tau$, which exponentially increases write energy according to Equation (2). If we strive to keep energy constant, then $\tau$ needs to be increased, which directly affects write performance. For example, consider the designed 1ms retention region. Doubling its size to 2MB while keeping performance constant implies a write and read energy overhead of 12% and 37%, respectively. If the design tries to improve write energy, then it will incur 167% slower writes, while still suffering from 32% more energetically costly reads. In both cases, the static energy term is increased by 16%. If the region were to be made 4MB in size, in order to retain performance the writes would be 67% more energetically costly, against a 101% increase for reads and 264% for leakage energy. Considering these issues, it is desirable to build small, low-retention regions that can be efficiently exploited by short-lived data.

Instead of enlarging low-retention regions, we could consider lowering the retention of larger ones. Filling with data from main memory, even a fast 1ms retention, 4MB region would take approximately 1ms, that is, 100% of its effective retention time. Adding an advanced refresh scheme to improve usability would increase the overall energy by an additional 5% [Sun et al. 2011]. Increasing the retention time to 100ms would increase the dynamic energy by 8%, on average. Given that the static energy of L2 and lower levels is the main energy component, the nonrefreshed design has potential energy advantages to be exploited.

The final design occupies 99.6% of the area of the 4MB cache, while providing 17MB of memory space. As such, the scratchpad is 4.25 times denser (in bytes per $mm^2$) than the conventional hardware-managed cache. Approximately 10% of this density increase is due to the lack of a tag array in scratchpads, while the remaining 90% is the result of the technological changes in the memory cells. Note that, contrary to what could be expected, the 100ms and 1ms regions are less dense than the 10s region, since smaller MTJ sizes would appear to constitute smaller memory cells. However, there is a second relevant factor on STT-RAM cell size, related to its write energy. As $\tau$ is made lower to increase performance, $Ic(\tau)$ grows exponentially. In order to drive larger write currents, the transistor needs to be made larger, decreasing the density of the memory. As can be seen in Figure 3, which stacks the graphics of Figure 2 for easier comparison, lower-retention regions are designed for increasing performance, which negatively affects density.

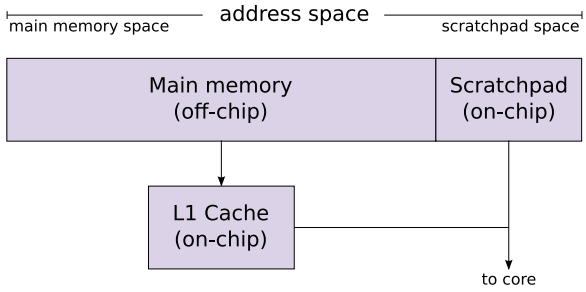


Fig. 4. System architecture and address space partitioning.

## 4. DATA ALLOCATION

Our target data memory architecture consists of three components: a cache memory, a scratchpad memory, and a main memory. The cache and the SPM are located on-chip; the main memory can be assumed to be off-chip DRAM (with a higher access latency). As shown in Figure 4, the address space is divided between off-chip memory and on-chip SPM. It is assumed that there is no direct path to transfer data from main memory to the scratchpad. The CPU will be responsible for transferring data between on-chip and off-chip memory whenever necessary. The SPM is assumed to be shared by all the cores in a processor die, running a single application. The proposed data allocation algorithm can be combined with techniques for the dynamic management of shared SPMs in multiprogrammed environments (e.g., Bathen et al. [2011]). Note that several commercial processors employ a similar data hierarchy model [ARM 2010; Flachs et al. 2005; Lindholm et al. 2008]. Variations include allowing DMA transfers between SPM and main memory, including L2 caches, and having private SPMs for each core.

A fast multiretention scratchpad such as the one proposed in this work leverages data reuse to improve performance. Since some spaces of the SPM are short-lived, the allocation algorithm must necessarily be dynamic. Although a compiler could analyze each location in a program to obtain an optimal solution, the number of possible dynamic scratchpad allocations is exponential [Udayakumaran et al. 2006], and the general problem of optimal data allocation is known to be NP-complete [Avissar et al. 2002]. Since most data reuse takes place inside loop nests, we propose a dynamic algorithm that changes the allocation layout only at loop headers.

The proposed algorithm first analyzes each loop nest in the application individually and selects those data that, if allocated to the scratchpad, would improve execution time, according to a cost model, as given for perfect loop nests in Section 4.1. This model is generalized in Section 4.2. Afterward, optimal allocation points are tuned to take into account internest reuse as shown in Section 4.3. Finally, a greedy algorithm uses the calculated cost model to allocate the scratchpad taking into account capacity restrictions, as described in Section 4.4. A discussion about restrictions and possible improvements follows in Section 4.5.

### 4.1. Allocation for Perfect Loop Nests

Consider a loop nest with $m$ perfectly nested loops and a single access to an $n$-dimensional vector $V$ nested at level $m$:

$$
\begin{array}{l}
\texttt{DO}\ i_1 \ = \ i_1^l,\ i_1^u,\ s_1 \\
\quad \texttt{DO}\ i_2 \ = \ i_2^l,\ i_2^u,\ s_2 \\
\qquad . \\
\qquad . \\
\qquad . \\
\qquad \texttt{DO}\ i_m \ = \ i_m^l,\ i_m^u,\ s_m \\
\qquad\quad \ldots V[f_1(\Omega)][f_2(\Omega)]\ldots[f_n(\Omega)]\ldots,
\end{array}
$$

where $i_j^l, i_j^u, s_j, 1 \leq j \leq m$, denote the lower and upper bounds, and the step of loop $i$, respectively; $\Omega = \{i_1, i_2, \ldots, i_m\}$ is the set of all loop indices; and $f_d(\Omega), 1 \leq d \leq n$, is the set of mapping functions that convert a given point in the iteration space of the nest to a point in the data space of $V$. To simplify notation, we refer to the access $V[f_1(\Omega)]\ldots[f_n(\Omega)]$ as A and to any loop by its index variable $i_j$, $1 \leq j \leq m$. Let $\phi(f_d, \omega_j), 1 \leq d \leq n, 1 \leq j \leq m$, be a boolean function that returns 0 when none of the loop indices in the set $\omega_j = \{i_j, \ldots, i_m\} \subseteq \Omega$ are used by function $f_d$, and 1 otherwise. Conceptually, $\phi(f_d, w_j)$ checks whether $f_d$ is an invariant in the scope of loop $i_j$. Let $|V_d|, 1 \leq d \leq n$, be the size of vector $V$ in dimension $d$. Let us define $\nu_{i_j}(A)$, a function that counts the number of memory accesses emitted by $A$ in loop $i_j$, as:

$$\nu_{i_j}(A) = \prod_{k=j}^{m} \frac{i_k^u - i_k^l + 1}{s_k} \tag{4}$$

The size of the smallest polytope that is guaranteed to contain the data accessed by $A$ under loop $i_j$, denoted by $\mathcal{D}_{i_j}(A)$, can be calculated as:

$$|\mathcal{D}_{i_j}(A)| = \prod_{d=1}^{n} \phi(f_d, \omega_j) \cdot |V_d| \tag{5}$$

Note that this size can be further optimized in the case of affine accesses, as will be discussed in Section 4.5. The reuse per element incurred by access $A$ in loop $i_j$, $R_{i_j}(A)$, can be calculated as:

$$R_{i_j}(A) = \frac{\nu_{i_j}(A)}{|\mathcal{D}_{i_j}(A)|} \tag{6}$$

This calculation of $R_{i_j}(A)$ can be generalized to consider a number $a > 1$ of identical accesses $A$ per iteration of the innermost loop, by multiplying Equation (6) by $a$. Our proposed allocation method calculates the reuse for each access in the nest for each loop $i_1, \ldots, i_m$. Afterward, it tentatively selects an allocation point for each access $A$ as the point immediately before the loop $i_j$ where reuse is maximized. The tentative scratchpad region for allocation is selected by estimating the worst-case execution time of loop $i_j$, $WCET(i_j)$, and choosing the region featuring the smallest retention $r$ such that $r > WCET(i_j)$.

Consider the matrix multiplication code shown in Figure 5 for illustrative purposes, where each loop header has been annotated with the reuse value of the accesses in the internal statement. The optimal allocation site for array $Y$ is just before loop $i$, where each element brought to SPM will be read $N$ times. There are two optimal allocation sites for $X$ and three for $Z$. Considering energy consumption, an allocation in

```
// Ri(Z[*][*]) = 2N^3/N^2 = 2N
// Ri(X[*][*]) = N^3/N^2 = N
// Ri(Y[*][*]) = N^3/N^2 = N
for( i = 0; i < N; i++ )
  // Rj(Z[i][*]) = 2N^2/N = 2N
  // Rj(X[i][*]) = N^2/N = N
  // Rj(Y[*][*]) = N^2/N^2 = 1
  for( j = 0; j < N; j++ )
    // Rk(Z[i][j]) = 2N/1 = 2N
    // Rk(X[i][*]) = N/N = 1
    // Rk(Y[*][j]) = N/N = 1
    for( k = 0; k < N; k++ )
      Z[i][j] += X[i][k] * Y[k][j];
```

Fig. 5. Annotated matrix multiplication code.

an internal loop is more likely to allow the use of a lower retention space and provide power savings. An innermore loop must also have a smaller memory footprint, since the calculated reuse value is the same and $\nu$ is smaller. As such, the obvious choice would be to allocate the tile $X[i][*]$ to SPM before loop $j$, and the scalar $Z[i][j]$ before loop $k$. Each variable could be allocated to a different SPM region depending on the WCET of the loops $i$, $j$, and $k$.

Depending on whether a variable is read-only, write-only, or read-write in the scope of a loop $i_l$, it is possible to omit the write from or back to main memory before or after the loop, respectively. Note that there is a reuse threshold below which no gains are obtained from transferring a data tile from main memory (MM) to SPM. This threshold can be analytically calculated using Equation (7), where $C$ is the cost function to be optimized (e.g., performance in cycles, energy-per-access,...), and $\Theta$ is a constant with value 1 if $A$ is a read- or write-only access and 2 in the case of a read-write access.

$$R_{i_j}(A) > \frac{\Theta \cdot C(MM)}{C(MM) - C(SPM)} \tag{7}$$

### 4.2. Generalization

Consider the case for two different accesses to the same array, $V[f_1(\Omega)]\dots[f_n(\Omega)]$ and $V[f'_1(\Omega)]\dots[f'_n(\Omega)]$, abbreviated to $A$ and $A'$. If the data spaces accessed by $A$ and $A'$ are copied to the scratchpad independently before loops $i_j$ and $i_{j'}$ consistency issues arise when there are potential overlaps and either of them is a write access. Even if both were read accesses, data would be replicated in the scratchpad. In order to avoid these risks, a single data tile containing the data spaces of both $A$ and $A'$ must be brought into the scratchpad. Without loss of generality, let $i_j$ be outer than $i_{j'}$. The number of accesses to the unified tile is $\nu_{i_j}(A) + \nu_{i_j}(A')$, and its size is upper bounded by $|\mathcal{D}_{i_j}(A)| \cdot |\mathcal{D}_{i_j}(A')|$. The actual size can be calculated as

$$|\mathcal{D}_{i_j}(A \cup A')| = \prod_{d=1}^{n} \left(\phi(f_d, \omega_j) \vee \phi(f'_d, \omega_j)\right) \cdot |V_d|, \tag{8}$$

where $\vee$ is the logical disjunction with its usual definition over the boolean set $\{0, 1\}$. Depending on the access functions, overlaps may be analytically discarded. Note that if $A$ and $A'$ are both read-only accesses, then it is possible to keep two separate allocations even in the case of overlaps. This will be beneficial when the size of the merged tile is bigger than the sum of the original ones.

Considering nonperfectly nested loops, access $A$ may appear any number of times, at any nesting level. Its reuse in a loop $i_j$ can be calculated by iteratively replacing the innermost loop $i_m$ by a placeholder statement that is computed as performing $\nu_{i_m}(A)$ accesses to $A$ until $i_j$ is the innermost loop, then calculating $R_{i_j}(A)$ as before. The case of multiple innermost loops is not problematic, as their respective reuses can be calculated independently. Note that a canonical loop form has been used. Canonicalization passes readily available in compiler frameworks can be used to implement the proposal.

### 4.3. Internest Reuse

Consider a simplified version of the control-flow graph, in which each node corresponds to a loop nest and edges indicate control flow. Each node is annotated with the allocation points in the corresponding nest for each variable in the code, as calculated in previous sections. We call this graph the *allocation graph* of the application. Consider two neighboring nodes representing two nests $i_1$ and $i_1'$, where $i_1$ dominates $i_1'$ in the control-flow graph.[1] Let us assume that two different allocations for overlapping tiles of variable $V$ are performed just before the outermost loop in each nest. In the general case of read-write accesses, a copy from and back to main memory of the appropriate tile in $V$ will be performed before and after, respectively, both $i_1$ and $i_1'$. Merging both allocations may be desirable, since the copies to and from main memory that would be placed in between $i_1$ and $i_1'$ would be omitted, improving locality. However, the merge may be harmful for power consumption, depending on $WCET(i_1)$, $WCET(i_1')$, $WCET(i_1 \cup i_1')$, and the energy characteristics of the different scratchpad regions. A mixed model that takes into account both performance and energy could be used to guide this optimization. In our approach, we always merge allocations if the calculated reuse for the data allocated to SPM is higher, without taking into account the characteristics of the selected SPM regions. The rationale is that two different scratchpad regions have access latencies and energies of the same order, while the energy and performance cost of accessing off-chip memory is assumed to be an order of magnitude higher.

This operation creates a new allocation to replace the old ones. The new allocation point is located before the dominant node, $i_1$. The allocation scope ends after the dominated node, $i_1'$. The size of the allocated tile can be calculated using Equation (8), modified to use the appropriate $\Omega$ for each loop. As with intranest merges, the new number of accesses to calculate reuse is $\nu_{i_1}(A) + \nu_{i_1'}(A')$.

This merge process can be performed iteratively over all the nodes in the graph. When the process ends, a variable $V$ may have a single associated allocation, with all the application in scope. The merges tentatively performed in this step may be undone when capacity restrictions are considered, as will be explained in the next section.

### 4.4. Capacity Restrictions

At this point, optimal locations for copying data tiles to SPM have been selected for all array accesses in the code. These locations have been chosen based on reuse alone, calculated according to Equation (6). Accesses with low reuse, where the benefit of copying the accessed data to the scratchpad does not offset the cost, have been discarded as per Equation (7). However, the maximum capacity of each of the different scratchpad regions has not been considered.

The reuse per element, $R_{i_j}(A)$, has been used in the previous sections as an analytical metric to compare different allocation points for a given access. It is employed at this

[1]In a control-flow graph, a block $D$ dominates a block $N$ if every path from the entry point that reaches $N$ has to go through $D$.

point as a heuristical figure of merit to prioritize some allocations over others. We propose a greedy algorithm that orders the tentative allocations according to their calculated reuse, and selects the data to be allocated to the scratchpad for each loop as follows:

(1) For each loop, select the data space $\mathcal{D}_{i_j}(A)$ with the highest reuse that has not yet been considered for allocation. In the case of a tie, select the one with the highest number of accesses as per Equation (4).
(2) If $\mathcal{D}_{i_j}(A)$ fits the optimal scratchpad region $R_o$ for the required retention time $WCET(i_j)$: allocate it to the beginning of the free memory in that region, and go to step 1.
(3) Otherwise, if there is enough free space in a nonoptimal region with higher retention, $R_h$: fill $R_o$ with the largest subtile of $\mathcal{D}_{i_j}(A)$ that fits, allocate the remaining data to $R_h$, and go to step 1.
(4) Otherwise, if no such $R_h$ exists, discard allocation of $D_{i_j}(A)$. The data accessed by $A$ will be allocated when and if a $D_{i_{j'}}(A)$ is found to be the data space with the highest reuse that has not yet been considered for allocation. Note that $i_{j'}$ will necessarily be innermore than $i_j$, as outer allocations always reference tiles of equal or larger size and their WCET will be longer. Also note that a single unsuccessful allocation may be replaced by several smaller ones, for instance, when data is to be allocated in multiple deeper, nonperfectly nested loops; or when the failed allocation attempt had been originally created to exploit internest reuse as described in Section 4.3.

A pseudocode for the entire allocation process is summarized in Algorithm 1. The pseudocode denotes the allocation of the polytope $\mathcal{D}_{i_j}(A)$ before loop $i_j$ for the scope of nest $N$ as $\mathcal{A}_{i_j,N}(A)$. As an example, consider again the code in Figure 5. Let us assume a memory system with a multiretention SPM identical to the one designed in Section 3, $N = 1024$, and that $X$, $Y$, and $Z$ are variables of type `double`. The allocation algorithm first selects $\mathcal{D}_k(Z[i][j])$ as the data space with the highest reuse of 2048 accesses per element. The size of the accessed data tile is of just 1 double, and the WCET of the loop is well below 1ms. Transfers from memory to the 1ms retention region and vice versa are set up before and after the innermost loop, respectively. Afterward, the allocation algorithm finds that $X[i][k]$ and $Y[k][j]$ have the same reuse of 1,024 accesses per element. Since $\mathcal{D}_i(Y[k][j])$ is accessed $2^{30}$ times, versus $2^{20}$ accesses for $\mathcal{D}_j(X[i][k])$, the algorithm selects the former. This data is accessed during the entire loop nest, and the estimated WCET is slightly under 10s. Consequently, it tries to allocate the entire data space to the 10s region. However, its size is 8MB, while the size of this region is only 6MB. Therefore, it splits the allocation between the 10s region (6MB) and the 10yr one (the remaining 2MB). Afterward, it sets up a transfer from memory to the SPM before loop $i$. No writeback is necessary, as the access is read-only. Finally, for $\mathcal{D}_j(X[i][k])$, the WCET for loop $j$ is determined to be slightly under 10ms. Consequently, the data space is allocated to the 100ms region, and a transfer from memory to SPM is inserted before loop $j$. Again, no writeback is necessary. Assuming that the number of loop nests in the code is smaller than the number of different array accesses $\#A$, which is the most frequent case, the entire algorithm is executed in a polynomial time bounded by $O(\#A^3)$.

### 4.5. Discussion

The proposed algorithm uses known iteration bounds to estimate the benefits of allocating a tile to SPM. In the case of unknown bounds, the compiler can use different complexity metrics or assume either a very large or very small number of iterations, thus favoring or preventing allocations to SPM. The latter approach is used by prefetch-

**ALGORITHM 1:** Pseudocode for the allocation algorithm

**Input**: the allocation graph with no annotations, $G$
// 1. Calculation of reuse and optimal allocation points
**for each** *variable* $V$ **do**
  **for each** *node* $N$ *in* $G$ **do**
    **for each** *loop* $i_j$ *in* $N$ **do**
      **for each** *access* $A$ *to* $V$ *in the scope of* $i_j$ **do**
        calculate $R_{i_j}(A)$;
      **end**
      $i_o \leftarrow i_j / R_{i_j}(A) > R_{i_k}(A), \forall i_k \in N$;
      annotate $N$ with $\mathcal{A}_{i_o,N}(A)$, the optimal allocation for $A$;
    **end**
  **end**
  // Consider intranest reuse
  **for each** *allocation pair* $(\mathcal{A}_{i_j,N}(A), \mathcal{A}_{i_{j'},N}(A')), j \leq j'$ **do**
    **if** *(*$A$ *is write access) OR*
      *(*$A'$ *is write access) OR*
      $(|\mathcal{D}_{i_j}(A)| + |\mathcal{D}_{i_{j'}}(A')| \geq |\mathcal{D}_{i_j}(A \cup A')|)$ **then**
      merge both allocations into $\mathcal{A}_{i_j,N}(A \cup A')$;
    **end**
  **end**
**end**
// 2. Extraction of internest reuse
**for each** *allocation pair* $(\mathcal{A}_{i_1,N}(A), \mathcal{A}_{i'_1,N'}(A')) \,/\, N' \in IDom(N)$ **do**
  **if** $\mathcal{D}_{i_1}(A) \cap \mathcal{D}_{i'_1}(A') \neq \emptyset$ **then**
    merge both allocations into $\mathcal{A}_{i_1 \cup i'_1, N \cup N'}(A \cup A')$;
  **end**
**end**
// 3. Take size restrictions into account
**for each** *node* $N$ *in* $G$ **do**
  select $\mathcal{D}_{i_j}(A) \,/\, R_{i_j}(A)$ *is maximum*;
  **if** *(*$\mathcal{D}_{i_j}(A)$ *fits optimal region* $R_o$*) OR*
    *(*$\mathcal{D}_{i_j}(A)$ *fits* $R_o \cup R_h$*)* **then**
    allocate $\mathcal{D}_{i_j}$ to appropriate address space;
  **else**
    remove allocation $\mathcal{A}_{i_j}(A)$
  **end**
**end**

ing algorithms, in which the impact of unknown bounds has been shown to be minor [Mowry et al. 1992].

The case of an affine access $A$ can be further optimized by calculating the optimal polytope that contains the data in $\mathcal{D}_{i_j}(A)$, instead of allocating the cartesian product of entire dimensions. This enables further optimizations, such as retention-based loop tiling, in which a loop $i_j$ could be tiled attending to $WCET(i_j)$ to optimize energy consumption.

The optimal scratchpad region for allocation could be speculatively selected in the case of read-only accesses. Consider a tile $\mathcal{D}_{i_j}(A)$ to be allocated to a scratchpad region $R$ with retention time $r > WCET(i_j)$. If a region $R'$ exists, with $r'$ only marginally below $WCET(i_j)$, it may be advantageous to allocate $\mathcal{D}_{i_j}(A)$ to $R'$. Data decay can be detected during runtime and the remaining accesses performed through off-chip main memory.

The proposed model does not take into account the different characteristics of each scratchpad region, or the different characteristics for read and write accesses, when calculating the optimal allocation places. These differences are found to be almost irrelevant compared to the much bigger ones with off-chip accesses. As such, the potential benefits of more complex models are severely limited, as any change in the allocation that diminishes reuse is going to increase the number of off-chip accesses. Also, the model does not consider the access patterns of each data tile and cache line size. Cache-friendly accesses could be penalized depending on their stride and reuse distance. Furthermore, constant $\Theta$ in Equation (7) would also be dependent on the line size and access stride.

For simplicity, the case of scalar variables has not been considered. However, they can be incorporated into the proposal without significant changes. Also this proposal does not consider interprocedural reuse, although the fundamental ideas could be applied, particularly the ones related to exploiting internest reuse. Conditional statements have not been considered. In this case, the optimal allocation point for each data space cannot be calculated analytically. Reuse weighted by the chance of executing each conditional branch could be used as a heuristic metric.

## 5. EXPERIMENTAL EVALUATION

Experiments were performed using the gem5 simulator [Binkert et al. 2011], a tool used for architectural modeling. The framework was used in "syscall emulation" mode, which simulates only the CPU and memory system and emulates system-level services. The memory system was modified to include a configurable multiretention scratchpad, and instructions to manage the different address spaces were added to the ISA. The *detailed* processor model was used, simulating a pipelined, out-of-order CPU, at a 2Ghz frequency. L1 access latency is 1 cycle and ITRS L2 SRAM latency is 3 cycles. Latencies for simulated SPMs and STT-RAM caches vary between 1–3 cycles for reads and 3–9 cycles for writes, depending on the memory size and selected volatility. Note that these numbers are purely the times for accessing the data array for each memory (including tag array accesses on caches). On top of these latencies the simulator adds any delay caused by cache contention and coherence, depending on the execution context for each access. All simulated memories are assumed to have a single read/write port. A MESI coherence protocol is used for configurations with shared L2. Wire latency has not been taken into account for any of the simulated memories, as it is completely dependent on the floorplan of each specific architecture.

Our test applications are computational kernels extracted from the SPEC CPU2006 benchmarks [Henning 2006] (`bwaves`, `cactusADM`, `leslie3d`, and `hmmer`), the Mantevo benchmarks [Heroux et al. 2009] (`HPCCG` and `miniMD`), the Mediabench suite [Lee et al. 1997] (`gsm` and `adpcm`), and the PARSEC benchmarks [Bienia 2011] (`canneal` and `streamcluster`). Each kernel was configured to simulate at least $10^{10}$ cycles. All the execution setups used in this section include a 64KB SRAM L1 cache. The differences between setups are: 1) whether they have an L2 cache, and its characteristics; and 2) whether they have an SPM space, and its characteristics.

### 5.1. Characterization of the Multiretention Scratchpad

This section is devoted to the analysis of the effect of dividing the scratchpad into regions with different retentions and characteristics. To this end, the multiretention scratchpad designed in Section 3 is compared with an STT-RAM scratchpad with a single, nonvolatile region. This baseline system features a 16MB SPM with 10yr retention time. Data are allocated to this scratchpad using the strategy proposed by Udayakumaran et al. [2006]. This method modifies the scratchpad allocation at program points where locality behavior changes, using profiling and an analytical cost model to guide

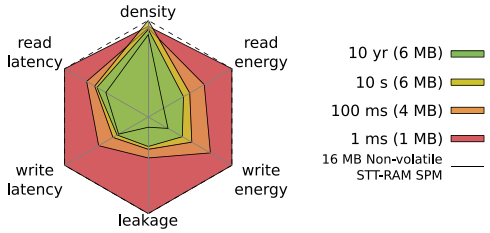


Fig. 6. Characterization of the multiretention space against the baseline. The dashed border marks optimal behavior.

the allocation process. Figure 6 graphically compares the characteristics of both memories using the configurations further detailed in Table I (see entries "16MB NVM SPM" and "Multiret. SPM").

Figure 7 analyzes how SPM accesses are distributed among the different retention regions. Accesses are categorized into reads and writes, and normalized to the number of baseline accesses. The SPEC kernels have large nested loops with heterogeneous working sets that take advantage of all the retention spaces. In contrast, the Mantevo and Mediabench codes are smaller, with fast loops that do not employ the higher retention regions. The PARSEC benchmarks have the smallest loops, and the allocation scheme has trouble exploiting locality in these object-oriented codes. Note that the addition of the normalized number of accesses is not always 1.0. This is due to the fragmentation in the scratchpad. For `cactusADM`, some variable allocations cannot be performed at the optimal reuse point, as there is not enough combined free space in regions with the appropriate retention (see steps 2 and 3 in Section 4.4). The compiler is forced to allocate data in an innermore loop, which reduces the size of the accessed data tile and the necessary retention. However, it also reduces reuse. Since the computational kernel will require exactly the same number of accesses and reuse is reduced, according to Equation (6) the size of the data brought to the scratchpad must increase, which implies a potentially larger number of loads and/or writebacks to main memory, depending on whether the affected variable is read-only, write-only, or read-write.

Regarding execution times, the impact of faster accesses in the multiretention SPM is limited. The number of cycles required for the simulation of the kernels is within the 1% range in all cases. Off-chip memory accesses, which are an order of magnitude slower, dominate the execution times. This fact makes the differences in static energy almost constant for all applications, well below 1%. For this reason, static energy is not taken into consideration for the remainder of this section.

Figure 8 shows the dynamic power consumption for each scratchpad region. The impact of fragmentation is clearly visible in `cactusADM`, where an energy overhead due to main memory accesses appears. Since it is not possible to allocate all data tiles at their optimal reuse point as in the baseline, the number of accesses to main memory increases. The energy consumed by off-chip accesses has to be taken into account. Even in the presence of fragmentation issues, the multiretention approach still offers significant gains. Energy savings range between 22% for `cactusADM` and 80% for `streamcluster`. Total savings, calculated using a workload composed of all the benchmark executions, are 63%.

The effect of fragmentation was studied using different baseline sizes of 4MB, 8MB, and 32MB and comparing their results with multiretention SPMs built using the same design principles previously exposed. In all cases, some applications present fragmentation issues, which are translated into an overhead in the energy spent for off-chip memory access. This overhead is not large enough to offset the benefits of relaxing the nonvolatility, and the trends exposed in this section remain unchanged.

Table I. Summary of the Configurations Used

| Name | Description | Latency (cycles) Read | Latency (cycles) Write | Energy Read (pJ) | Energy Write (pJ) | Leakage (mW) | Area ($mm^2$) |
|---|---|---|---|---|---|---|---|
| SRAM cache | · 4 MB ITRS L2 cache | 3 | | 712.51 | | 1625.1 | 15.97 |
| 16MB NVM SPM | · 10yr: 16MB | 3 | 9 | 665.62 | 749.43 | 532.04 | 16.57 |
| STT-RAM cache | · 12MB STT-RAM L2 cache | 3 | 8 | 585.67 | 652.27 | 575.56 | 16.09 |
| Cache hybrid | · 8MB STT-RAM L2 cache | 3 | 8 | 479.89 | 521.53 | 368.54 | 9.65 |
| | · 4.25MB multiretention SPM: | | | | | | |
| | · 1ms: 256KB | 1 | 3 | 51.95 | 114.36 | 14.95 | 0.30 |
| | · 100ms: 1MB | 2 | 3 | 134.94 | 219.15 | 54.90 | 1.02 |
| | · 10s: 1.5MB | 2 | 3 | 195.98 | 295.83 | 49.51 | 1.83 |
| | · 10yr: 1.5MB | 2 | 6 | 223.38 | 323.87 | 79.04 | 2.26 |
| | | | | | | 566.94 | 15.06 |
| SPM hybrid | 4MB STT-RAM L2 cache | 2 | 6 | 371.25 | 464.51 | 188.96 | 5.58 |
| | 8.5MB multiretention SPM: | | | | | | |
| | · 1ms: 512KB | 1 | 3 | 77.15 | 131.63 | 18.90 | 0.38 |
| | · 100ms: 2MB | 2 | 3 | 212.85 | 312.99 | 85.36 | 1.95 |
| | · 10s: 3MB | 2 | 3 | 318.96 | 484.47 | 127.24 | 3.39 |
| | · 10yr: 3MB | 2 | 6 | 320.07 | 424.63 | 165.20 | 4.65 |
| | | | | | | 585.66 | 15.95 |
| Multiret. SPM | 17MB multiretention SPM: | | | | | | |
| | · 1ms: 1MB | 2 | 3 | 121.98 | 176.24 | 56.14 | 0.94 |
| | · 100ms: 4MB | 2 | 4 | 254.41 | 258.02 | 132.81 | 3.84 |
| | · 10s: 6MB | 2 | 7 | 348.36 | 368.25 | 168.39 | 5.28 |
| | · 10yr: 6MB | 2 | 8 | 410.29 | 470.48 | 184.60 | 5.84 |
| | | | | | | 541.94 | 15.90 |

Continued

Table I. Continued

| Name | Description | Latency (cycles) Read | Latency (cycles) Write | Energy Read (pJ) | Energy Write (pJ) | Leakage (mW) | Area ($mm^2$) |
|---|---|---|---|---|---|---|---|
| No 10s SPM | 19MB multiretention SPM: | | | | | | |
| | · 1ms: 2MB | 2 | 3 | 172.18 | 191.08 | 57.84 | 1.25 |
| | · 100ms: 7MB | 2 | 6 | 347.86 | 356.07 | 175.18 | 5.25 |
| | · 10yr: 10MB | 3 | 9 | 525.25 | 597.30 | 309.89 | 9.77 |
| | | | | | | 542.91 | 16.27 |
| No 100ms SPM | 17 MB multiretention SPM: | | | | | | |
| | · 1ms: 2MB | 2 | 3 | 172.18 | 191.08 | 57.84 | 1.25 |
| | · 10s: 8MB | 2 | 7 | 392.19 | 393.51 | 184.94 | 6.10 |
| | · 10yr: 7MB | 3 | 8 | 465.34 | 557.84 | 257.33 | 8.77 |
| | | | | | | 500.11 | 16.12 |
| No 1ms SPM | 19MB multiretention SPM: | | | | | | |
| | · 100ms: 7MB | 2 | 6 | 347.86 | 356.07 | 175.18 | 5.25 |
| | · 10s: 6MB | 2 | 7 | 348.36 | 368.25 | 168.39 | 5.28 |
| | · 10yr: 6MB | 2 | 8 | 410.29 | 470.48 | 184.60 | 5.84 |
| | | | | | | 528.17 | 16.37 |

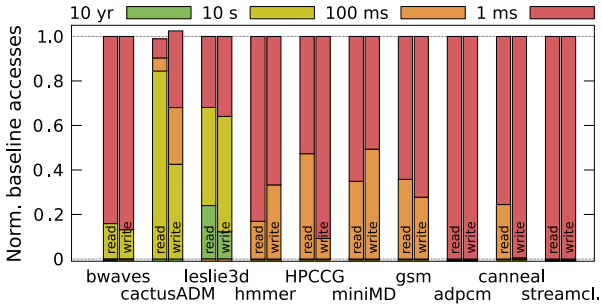


Fig. 7. Distribution of accesses to each region in the scratchpad.

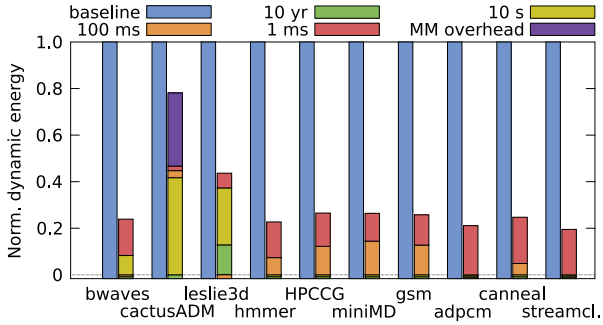


Fig. 8. Dynamic energy consumption per scratchpad region.

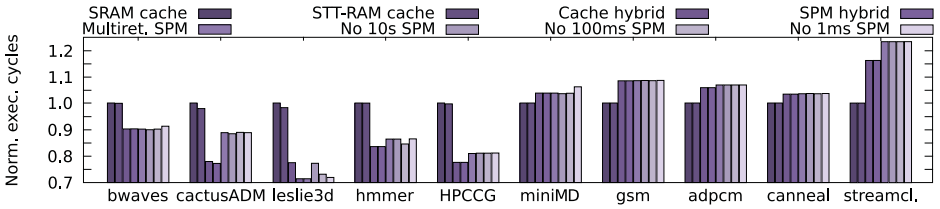


Fig. 9. Normalized execution cycles.

### 5.2. Comparison with Other Configurations

This section compares the proposed multiretention SPM with several different on-chip memory configurations. Our baseline is a 4MB ITRS cache featuring SRAM cells. All other configurations are approximately iso-area with this baseline. The second configuration switches the SRAM L2 cache for an STT-RAM L2 cache. Due to its higher density, this cache accommodates 12MB of memory. Two hybrid configurations are tested, including both an STT-RAM L2 cache and an SPM. The difference between the two are the sizes of each memory: one has an 8MB cache and a 4MB multiretention SPM, while the other features a 4MB cache and an 8MB multiretention SPM. The third set of memories are included to test the effects of reducing the number of regions in the multiretention design by removing the 10s, 100ms, and 1ms retention spaces. The nonvolatile region is not removed since it would require a mechanism to deal with data decay. Table I details the configurations used.

Figure 9 shows the normalized execution cycles for each application and the different on-chip memory configurations. The use of scratchpad memories improves execution cycles for those benchmarks with a working set bigger than the on-chip memory. In these situations, the analysis of the code achieves better performance than the LRU behavior of the caches. This happens for the SPEC benchmarks, as well as for the

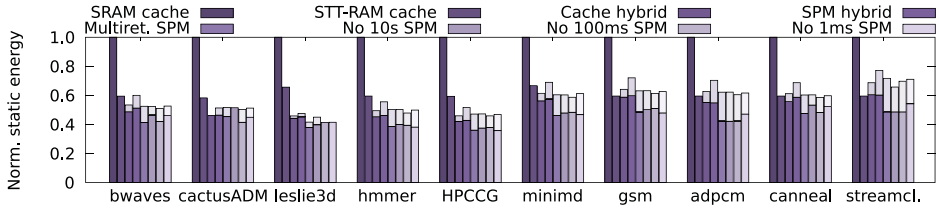


Fig. 10. Normalized static energy consumption of the memory hierarchy. Faded out bars represent savings if unused scratchpad regions are turned off.

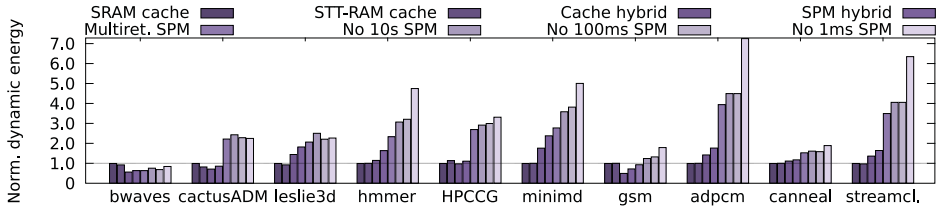


Fig. 11. Normalized dynamic energy consumption of the memory hierarchy.

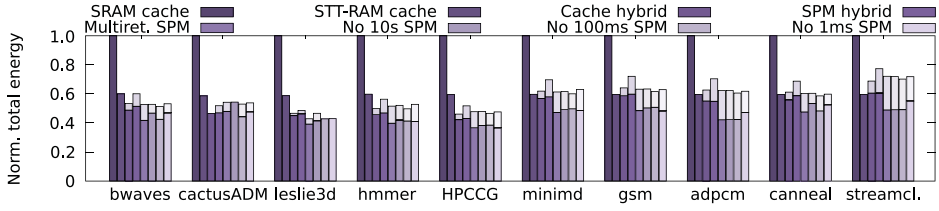


Fig. 12. Normalized total energy consumption of the memory hierarchy. Faded out bars represent savings if unused scratchpad regions are turned off.

Mantevo benchmarks. For Mediabench and PARSEC, the data allocated to SPM fits the on-chip memory of the baseline configuration. In this situation, the explicit copy of data from main memory to SPM implies an irrecoverable overhead. Furthermore, when the kernel is very small and reuses data, the intraprocedural allocation algorithm flushes and reallocates data each time the kernel is called, while in the cache configurations data is still available on-chip to be readily used. The difference between the pure SPM configurations is negligible, only 0.1% on average.

Static consumption, shown in Figure 10, is directly derived from the leakage energy of each configuration and the execution cycles. It is possible to turn off the SPM banks that are not used, as they are statically determined by the compiler. This effect is represented in the figure by the faded out bars of the relevant configurations. A similar, more complex technique can be applied to cache memories [Flautner et al. 2002]. STT-RAM configurations have lower static consumption, independently of the run times, due to their inherent characteristics. Also, pure SPMs tend to have lower leakage energy than configurations that include caches as no tag arrays are present. Configurations without a given retention region usually consume more than the full multiretention one when unused banks are turned off. This largely depends on the powered on-chip area, since the difference in execution times between these configurations is negligible. The allocation algorithm could be improved to take advantage of this by accounting for static power savings if allocations to low-retention regions are moved to higher-retention ones with enough free space.

Figure 11 details dynamic consumption. Configurations including caches are less

power hungry in this case. The main reason for this effect is that, in order to improve locality, much data that would be accessed through L1 is allocated to the SPM. While the STT-RAM SPM has better access energy characteristics than the SRAM L2, accessing some regions is an order of magnitude more costly than accessing L1. This directly translates into a higher dynamic consumption. The effect is much more noticeable for the configuration that does not include a 1ms retention scratchpad, as it is the only region with access energy comparable to an L1 cache. This overhead could be alleviated by taking into account the access patterns of each data tile during allocation, as discussed in Section 4.5.

Figure 12 summarizes the total power consumption of the memory hierarchy. In our experimental setup, the static power consumption clearly dominates, as predicted by ITRS below 65nm technology, and the leakage advantage of the scratchpad memories is translated to the total consumption. However, note that these tests have been conducted using a single execution thread, and the actual access frequency to the on-chip memories is far from the design maximum. As the number of cores increases, the static power is expected to increase slightly due to the additional L1 caches, while the dynamic power would increase significantly, as will be seen in Section 5.5. Total energy savings for the full multiretention SPM configuration with respect to the SRAM cache range from 51% for `streamcluster` to 63% for `HPCCG`. Savings calculated using all benchmark executions as a single workload are of 53.4%.

### 5.3. Optimality Analysis

The optimality of the results can be studied from two different points of view: optimality of the allocation algorithm and optimality of the designed architecture.

The footprint of some applications is larger than the space available in the better-fitting retention region. The allocation algorithm deals with these situations by either allocating part of the data in a higher-retention region, if available, or selecting a different allocation point for the data (i.e., an innermore loop), therefore sacrificing reuse. A way to measure the optimality of the allocation algorithm is to compare the results to those obtained by an ideal scratchpad architecture with infinite allocation space per region that preserves the energy and latency characteristics of the multiretention SPM.

However, some applications face the exact opposite situation: the available scratchpad space is larger than their memory footprint. These applications are paying a price in terms of both static and dynamic energy for having large memories beyond their requirements. Ideally, the designer could build an ad-hoc system to fit the application requirements exactly. These memories would present energy and latency differences with respect to the multiretention SPM, depending on the ad-hoc sizes of each retention region.

Figures 13 and 14 show the optimal static and dynamic energy consumptions, respectively, obtained according to both optimality definitions, and normalized to the original energy consumption using the multiretention scratchpad designed in Section 3. The original static energy is obtained by turning off the scratchpad regions that are unused during the entire execution, as discussed in Section 5.2. As can be seen, assuming infinite scratchpad regions only makes a difference in two of the kernels, `cactusADM` and `leslie3d`, as their total footprints are larger than the available space in the desired retention regions. With infinite-sized memories, it is possible to perfectly exploit reuse, improving runtimes and therefore static energy. Furthermore, as less off-chip accesses are issued, dynamic energy improves. When considered together, the configuration employing infinite scratchpad regions consumes on average 99% of the baseline

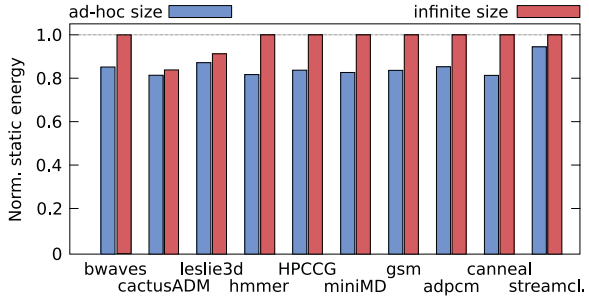


Fig. 13. Static energy of optimal configurations, normalized to static energy of the multiret. SPM.

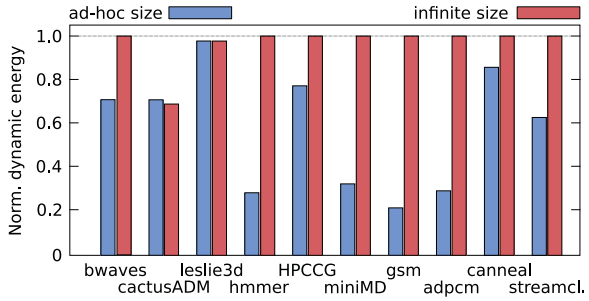


Fig. 14. Dynamic energy of optimal configurations, normalized to dynamic energy of the multiret. SPM.

energy when using all benchmark executions as a single workload, with a minimum consumption of 83% of the total energy for `cactusADM`.

Executing applications with ad-hoc SPMs has several advantages. Smaller memories occupy less area and consume less energy, both static and dynamic. Since many of the applications did not fill the space available in the multiretention SPM, significant improvements can be achieved. Additionally, smaller memories can be made faster, presenting further improvements in static energy. However, in applications for which some region needs to be made larger than the originally available, some energy components will increase. This is the case for the dynamic energy of `leslie3d`: the 10s retention region needs to be made larger, consequently increasing its dynamic energy. This increase is not offset by the decrease in the dynamic energy of other regions that were only partially used. As a result, the total dynamic energy term is slightly increased. However, this increase is more than offset by the improvement in static energy, and ultimately `leslie3d` consumes approximately 88% of the baseline energy. The configuration using ad-hoc scratchpad regions consumes on average 89% of the baseline energy when using all benchmark executions as a single workload, with a minimum consumption of 81% of the baseline energy for `canneal`.

### 5.4. Dynamic Behavior

Figure 15 shows the dynamic behavior of the `leslie3d` kernel using the cumulative access distribution of memory accesses over time. This kernel is divided into six different allocation scopes, each corresponding to a loop nest in the code (separated by vertical dotted lines in Figure 15). During the first one, the application writes an array with no reuse, reading a different array with reuse over the fastest changing dimension. The selected allocation brings the read-only data space to the 1ms retention region. The scope is executed 48% faster than using a hardware-managed SRAM cache. The

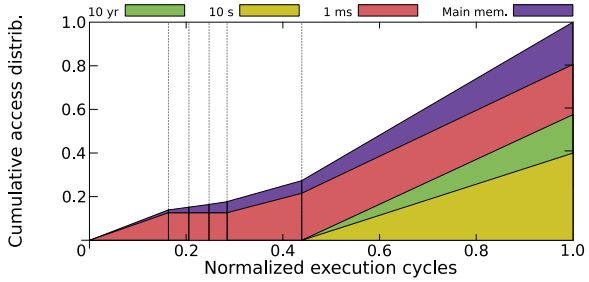


Fig. 15. Cumulative access distribution for `leslie3d`.

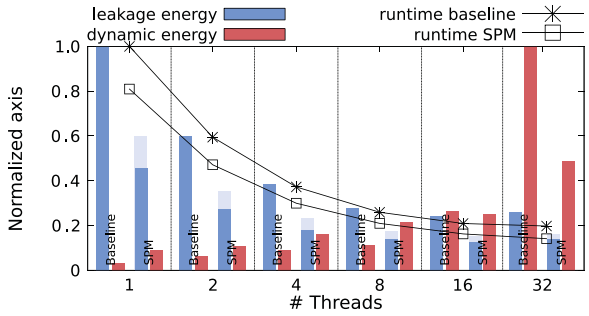


Fig. 16. Evolution of power consumption and execution cycles of `HPCCG` increasing the execution threads.

second, third, and fourth scopes are very similar loops that initialize small arrays. The scratchpad is not used during their execution, as there is no reuse to exploit. The difference in execution times with respect to the SRAM cache configuration is negligible. The fifth scope initializes two small arrays reading from four different arrays. As in the first scope, small tiles of these arrays are selected for allocation to the 1ms region to exploit reuse over the fastest changing dimension. This section is 9% slower than using caches, as the scratchpad version has no locality advantage and includes explicit memory transfers. The last scope is the largest and the one that involves the majority of the accesses and reuse in the kernel. Only the 100ms region remains unused in this scope, which is executed 25% faster in the SPM version.

This analysis shows that potential improvements from turning off scratchpad banks are higher than suggested by Figures 10 and 12, where only regions that are not used during the entire execution are turned off. Figure 15 shows that, for `leslie3d`, it would be possible to turn off the 10yr and 10s regions of the scratchpad for 45% of the execution cycles, and 12% for the 1ms region. This would bring additional power savings of 5% against the 4MB L2 cache, for a grand total improvement of 65%.

### 5.5. Analysis of Parallel Applications

The Mantevo codes, parallelized using OpenMP, are used to study the behavior of our proposed multiretention SPM with parallel workloads. Figures 16 and 17 show the energy consumption and execution cycles of `HPCCG` and `miniMD`, respectively, as the number of threads used for execution increases. Each thread is executed on its own core. Each core has a private L1. The SPM is shared among all the cores. The baseline system uses a 4MB ITRS L2 cache, while the scratchpad system is the full multiretention design.

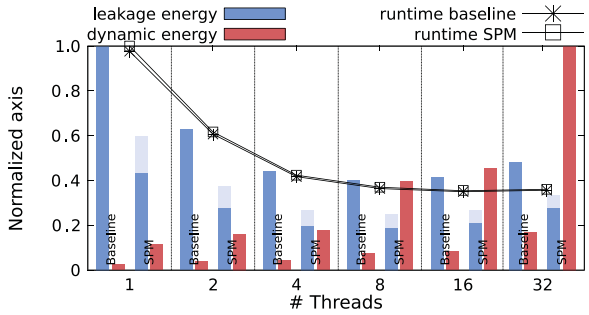


Fig. 17. Evolution of power consumption and execution cycles of `miniMD` increasing the execution threads.

As expected, the execution time decreases with the number of cores. So does the leakage energy, as the caches and scratchpad memories have to be on for a smaller amount of time. However, each additional core includes its L1 cache, with an associated static energy cost. At some point, the diminishing speedup is not enough to achieve net gains, and the static consumption is larger as new cores are added, even if execution time decreases slightly. This effect is present in both the cache and the scratchpad configurations. As can be seen, the static gains offered by the multiretention STT-RAM SPM are retained in multithreaded configurations.

As seen in Section 5.2, the SPM configurations have an overhead on dynamic energy consumption due to the bypass of the L1 cache, with lower access energy. This is clearly visible in `miniMD`, where the dynamic energy growth rate is approximately the same for both the cache and the SPM configurations, but differences are much more significant for 32 cores. However, this bypass also avoids the false sharing that is present in `HPCCG`. As a result, dynamic consumption increase is not as steep when using SPMs as is the case with caches.

As for total power, the increase in the number of threads negatively affects the energy advantages offered by the multiretention scratchpad. This has two main causes: on the one hand, static power consumption savings become smaller as the execution time decreases; on the other, dynamic power consumption increases as a result of the growth in the number of memory accesses due to cache coherency issues. Total energy savings for executions with 32 threads are 48% for `HPCCG`, and 42% for `miniMD`.

## 6. CONCLUDING REMARKS

This work has explored the implementation of scratchpad memories using volatile STT-RAM. By relaxing the nonvolatility of this technology, its latency and dynamic energy characteristics are improved. Together with its very high density and near-zero leakage energy, this makes it a serious alternative for implementing on-chip memories. Scratchpads further capitalize on the latency and energy advantages, by removing the tag array present in cache memories. The designed memories feature different retention regions, which can be used to closely match the lifespan of allocated data and optimize performance and energy. This type of technology could be used not necessarily as a built-in replacement for hardware-managed caches, but as a form of specialization of the on-chip memory hierarchy to exploit the predicted increasing percentage of dark silicon.

This article has presented the steps for designing an STT-RAM-based multiretention scratchpad and a customized compiler-based data allocation algorithm. The experimental evaluation shows that the proposed system has significant benefits for embedded

applications. It provides savings of 63% in dynamic power consumption compared to a nonvolatile STT-RAM-based scratchpad; and executes up to 28.5% faster, saving 53% of the energy consumed by a system featuring an iso-area hardware-managed SRAM cache. Experimentation suggests that these savings can be further improved by more aggressive optimizations to turn off unused scratchpad banks during idle periods. The use of scratchpad memories can also improve the dynamic power consumption of multithreaded workloads by reducing false sharing.

## ACKNOWLEDGMENTS

The authors would like to thank Prof. Sudhanva Gurumurthi and the University of Virginia for providing access to the STeTSiMS simulation and modeling system, as well as the anonymous reviewers for their comments and suggestions toward the improvement of this work.

## REFERENCES

ARM. 2010. *Cortex-R5 Technical Reference Manual*. Technical Report DDI-0460D. ARM Limited, Cambridge, UK.

O. Avissar, R. Barua, and D. Stewart. 2002. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Trans. Embed. Comput. Syst.* 1, 1, 6–26.

R. Banakar, S. Steinke, L. Bo-Sik, M. Balakrishnan, and P. Marwedel. 2002. Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign*. 73–78.

L. A. D. Bathen and N. Dutt. 2012. 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*. 447–452.

L. A. D. Bathen, N. D. Dutt, D. Shin, and S.-S. Lim. 2011. SPMVisor: Dynamic scratchpad memory virtualization for secure, low power, and high performance distributed on-chip memories. In *Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis*. 79–88.

C. Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Department of Computer Science, Princeton University, Princeton, NJ.

N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, et al. 2011. The gem5 simulator. *ACM Comput. Arch. News* 39, 2, 1–7.

S. Borkar and A. A. Chien. 2011. The future of microprocessors. *Commun. ACM* 54, 5, 67–77.

J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. 2011. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. 105–118.

Z. Diao, Z. Li, S. Wang, Y. Ding, A. Panchula, E. Chen, et al. 2007. Spin-transfer torque switching in magnetic tunnel junctions and spin-transfer torque random access memory. *J. Phys. Condens. Matter* 19, 165209.

H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. 2011. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th International Symposium on Computer Architecture*. 365–376.

B. Flachs, S. Asano, S. H. Dhong, P. Hotstee, G. Gervais, R. Kim, et al. 2005. A streaming processing unit for a CELL processor. In *Proceedings of the IEEE International Solid-State Circuits Conference*. 134–135.

K. Flautner, N. S. Kim, S. M. Martin, D. Blaauw, and T. N. Mudge. 2002. Drowsy caches: Simple techniques for reducing leakage power. In *Proceedings of the 29th International Symposium on Computer Architecture*. 148–157.

X. Guo, E. Ipek, and T. Soyata. 2010. Resistive computation: Avoiding the power wall with low-leakage, STT-MRAM based computing. In *Proceedings of the 37th International Symposium on Computer Architecture*. 371–382.

N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. 2011. Toward dark silicon servers. *IEEE Micro* 31, 4, 6–15.

J. L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM Comput. Arch. News* 34, 4, 1–17.

M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, et al. 2009. *Improving Performance via Mini-applications*. Technical Report SAND2009-5574. Sandia National Laboratories, Albuquerque, NM.

J. Hu, C. J. Xue, Q. Zhuge, W.-C. Tseng, and E. H.-M. Sha. 2013. Data allocation optimization for hybrid scratch pad memory with SRAM and nonvolatile memory. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* 21, 6, 1094–1102.

J. Hu, Q. Zhuge, C. J. Xue, W.-C. Tseng, and E. H.-M. Sha. 2014. Management and optimization for nonvolatile memory-based hybrid scratchpad memory on multicore embedded processors. *ACM Trans. Embed. Comput. Syst.* 13, 4, 79.

ITRS. 2012. International Technology Roadmap for Semiconductors. Retrieved from http://www.itrs.net/Links/2012ITRS/Home2012.htm.

A. Jog, A. K. Mishra, C. Xu, Y. Xie, V. Narayanan, R. Iyer, and C. R. Das. 2012. Cache revive: Architecting volatile STT-RAM caches for enhanced performance in CMPs. In *Proceedings of the 49th Annual Design Automation Conference*. 243–252.

E. Kultursay, K. Swaminathan, V. Saripalli, V. Narayanan, M. T. Kandemir, and S. Datta. 2012. Performance enhancement under power constraints using heterogeneous CMOS-TFET multicores. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*. 245–254.

E. Kultursay, M. T. Kandemir, A. Sivasubramaniam, and O. Mutlu. 2013. Evaluating STT-RAM as an energy-efficient main memory alternative. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software*. 256–267.

B. C. Lee, P. Zhou, J. Yang, Y. Zhang, B. Zhao, E. Ipek, et al. 2010. Phase-change technology and the future of main memory. *IEEE Micro* 30, 1, 131–141.

C. Lee, M. Potkonjak, and W. H. Mangione-Smith. 1997. MediaBench: A tool for evaluating and synthesizing multimedia and communication systems. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*. 330–335.

J. Li, L. Shi, Q. Li, C. J. Xue, Y. Chen, and Y. Xu. 2013b. Cache coherence enabled adaptive refresh for volatile STT-RAM. In *Proceedings of Design, Automation and Test in Europe*. 1247–1250.

Q. Li, J. Li, L. Shi, C. J. Xue, Y. Chen, and Y. He. 2013a. Compiler-assisted refresh minimization for volatile STT-RAM cache. In *Proceedings of the 18th Asia and South Pacific Design Automation Conference*. 273–278.

Q. Li, Y. Zhao, J. Hu, C. J. Xue, E. Sha, and Y. He. 2012. MGC: Multiple graph-coloring for non-volatile memory based hybrid scratchpad memory. In *Proceedings of the 16th Workshop on Interaction between Compilers and Computer Architectures*. 17–24.

X. Liang, R. Canal, G.-Y. Wei, and D. Brooks. 2007. Process variation tolerant 3T1D-based cache architectures. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. 15–26.

E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. 2008. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro* 28, 2, 39–55.

T. C. Mowry, M. S. Lam, and A. Gupta. 1992. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems.* 62–73.

N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. 2009. *CACTI 6.0: A Tool to Model Large Caches*. Technical Report HPL-2009-85. HP Laboratories, Palo Alto, CA.

P. R. Panda, N. D. Dutt, and A. Nicolau. 1997. Efficient utilization of scratch-pad memory in embedded processor applications. In *Proceedings of the European Design and Test Conference*. 7–11.

M. K. Qureshi, V. Srinivasan, and J. A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th International Symposium on Computer Architecture*. 24–33.

M. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh. 2001. Dynamic management of scratch-pad memory space. In *Proceedings of the 38th Design Automation Conference*. 690–695.

M. Rasquinha, D. Choudhary, S. Chatterjee, S. Mukhopadhyay, and S. Yalamanchili. 2010. An energy efficient cache design using Spin Torque Transfer (STT) RAM. In *Proceedings of the International Symposium on Low Power Electronics and Design*. 389–394.

N. D. Rizzo, M. DeHerrera, J. Janesky, B. Engel, J. Slaughter, and S. Tehrani. 2002. Thermally activated magnetization reversal in submicron magnetic tunnel junctions for magnetoresistive random access memory. *Appl. Phys. Lett.* 80, 13, 2335–2337.

A. Shaffer, B. Einfalt, and P. Raghavan. 2010. PFFTC: An improved fast Fourier transform for the IBM cell broadband engine. In *Proceedings of the International Conference on Computational Science*. 1045–1054.

C. W. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. R. Stan. 2011a. Relaxing non-volatility for fast and energy-efficient STT-RAM caches. In *Proceedings of the 17th International Conference on High-Performance Computer Architecture*. 50–61.

C. W. Smullen, A. Nigam, S. Gurumurthi, and M. R. Stan. 2011b. The STeTSiMS STT-RAM simulation and modeling system. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. 318–325.

Z. Sun, X. Bi, H. H. Li, W.-F. Wong, Z.-L. Ong, X. Zhu, and W. Wu. 2011. Multi retention level STT-RAM cache designs with a dynamic refresh scheme. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. 329–338.

M. B. Taylor. 2012. Is dark silicon useful? Harnessing the four horsemen and the coming dark silicon apocalypse. In *Proceedings of the 49th Annual Design Automation Conference*. 1131–1136.

S. Udayakumaran, A. Dominguez, and R. Barua. 2006. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst.* 5, 2, 472–511.

G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, et al. 2010. Conservation cores: Reducing the energy of mature computations. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*. 205–218.

P. Wang, G. Sun, T. Wang, Y. Xie, and J. Cong. 2013. Designing scratchpad memory architecture with emerging STT-RAM memory technologies. In *Proceedings of the IEEE International Symposium on Circuits and Systems*. 1244–1247.

A. Yanamandra, B. Cover, P. Raghavan, M. J. Irwin, and M. T. Kandemir. 2008. Evaluating the role of scratchpad memories in chip multiprocessors for sparse matrix computations. In *Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing*. 1–10.

Received April 2014; revised July 2014; accepted September 2014