Center for Embedded Computer Systems
University of California, Irvine

An Embedded Hybrid-Memory-Aware Virtualization Layer for Chip-Multiprocessors

Luis Angel D. Bathen and Nikil D. Dutt

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2620, USA

{lbathen,dutt}@uci.edu

CECS Technical Report #12-03
March 27, 2012

An Embedded Hybrid-Memory-Aware Virtualization Layer for Chip-Multiprocessors

Luis Angel D. Bathen and Nikil D. Dutt
Center for Embedded Computer Systems, University of California, Irvine, Irvine, CA, USA
{lbathen,dutt}@uci.edu

Abstract

Hybrid on-chip memories that combine Non-Volatile Memories (NVMs) with SRAMs promise to mitigate the increasing leakage power of traditional on-chip SRAMs. We present *HaVOC*: a run-time memory manager that virtualizes the hybrid on-chip memory space and supports efficient sharing of distributed ScratchPad Memories (SPMs) and NVMs. *HaVOC* allows programmers and the compiler to partition the application's address space and generate data/code layouts considering virtualized hybrid-address spaces. We define a data volatility metric used by our hybrid-memory-aware compilation flow to generate memory allocation policies that are enforced at run-time by a filter-inspired dynamic memory algorithm. Our experimental results with a set of embedded benchmarks executing simultaneously on a Chip-Multiprocessor with hybrid NVM/SPMs show that *HaVOC* is able to reduce execution time and energy by 60.8% and 74.7% respectively with respect to traditional multi-tasking-based SPM allocation policies.

Categories and Subject Descriptors C.3 [*Special-purpose and Application-based systems*]: Real-time and embedded systems; B.3 [*Design Styles*]: Virtual Memory; D.4 [*Storage Management*]: Distributed memories

General Terms Algorithms, Design, Management, Performance

1. Introduction

The ever increasing complexity of embedded software and adoption of open-environments (e.g., Android OS) is exacerbating the deployment of multi-core platforms with distributed on-chip memories [15, 20]. Traditional memory hierarchies consist of caches, however, it is known that caches may consume up to 50% of the processor's area and power [3]. As a result, ScratchPad Memories (SPMs) are rapidly being adopted and incorporated into multi-core platforms for their high predictability, low area and power consumption. Efficient SPM management can greatly reduce dynamic power consumption [16, 19, 26, 30], and may be a good alternative to caches for applications with high levels of regularity (e.g., Multimedia). As sub-micron technology continues to scale, leakage power will overshadow dynamic power consumption [21]. Since SRAM-based memories consume a large portion of the die, they are a major source of leakage in the system [2], which is a major issue for multi-core platforms.

In order to reduce leakage power in SRAM-based memories, designers have proposed emerging Non-Volatile Memories (NVMs) as alternatives to SRAM for on-chip memories [17, 24, 28]. Typically, NVMs (e.g., PRAM [11]) offer high densities, low leakage power, comparable read latencies and dynamic read power with respect to traditional embedded memories (SRAM/eDRAM). One major drawback across NVMs is the expensive write operation (high latencies and dynamic energy). To mitigate the drawbacks of the write operation in NVMs, designers have made the case for deploying hybrid on-chip memory hierarchies (e.g., SRAM, MRAM, and PRAM) [28], which have shown up to 37% reduction in leakage power [14], and increased IPC as a byproduct of the higher density provided by NVMs [24, 32]. Orthogonal to traditional hybrid on-chip memory subsystems which have been predominately focused on caches, Hu et al. [14] showed the benefits of exploiting hybrid memory subsystems consisting of SPMs and NVMs.

In this paper, we present *HaVOC*, a system-level hardware/software solution to efficiently manage on-chip hybrid memories to support multi-tasking Chip-Multiprocessors. *HaVOC* is tailored to manage hybrid on-chip memory hierarchies consisting distributed ScratchPad Memories (SRAM) and Non-Volatile Memories (e.g., MRAMs). *HaVOC* allows programmers to partition their application's address space into virtualized SRAM address space and virtualized NVM address space through a minimalistic API. Programmers (through annotations) and compilers (through static analysis) can then specify hybrid-memory-aware allocation policies for their data/code structures at compile-time, while *HaVOC* dynamically enforces them and adapts to the underlying memory subsystem. The **novel contributions** of our work are that we:

- Explore distributed shared on-chip hybrid-memories consisting of SPMs and NVMs and virtualize their address spaces to facilitate the management of their physical address spaces
- Introduce the notion of data volatility analysis to drive efficient compilation and policy generation for hybrid on-chip memories
- Present a filter-driven dynamic allocation algorithm that exploits filtering and volatility to find the best memory placement
- Present *HaVOC*, a run-time hybrid-memory manager that enforces compile-time-derived allocation policies through a filter-inspired dynamic allocation algorithm that exploits filtering and volatility to find the best memory placement

To the best of our knowledge, *our work is the first to consider distributed on-chip hybrid SPM/NVM memories and their dynamic management through the use of virtualized address spaces for reduced power consumption and increased performance.* Our experimental results with a set of embedded benchmarks executing simultaneously on a Chip-Multiprocessor with hybrid NVM/SPMs show that *HaVOC* is able to reduce execution time and energy by 60.8% and 74.7% respectively with respect to traditional multi-tasking-based SPM allocation policies.

WXYZ '05 date, City.

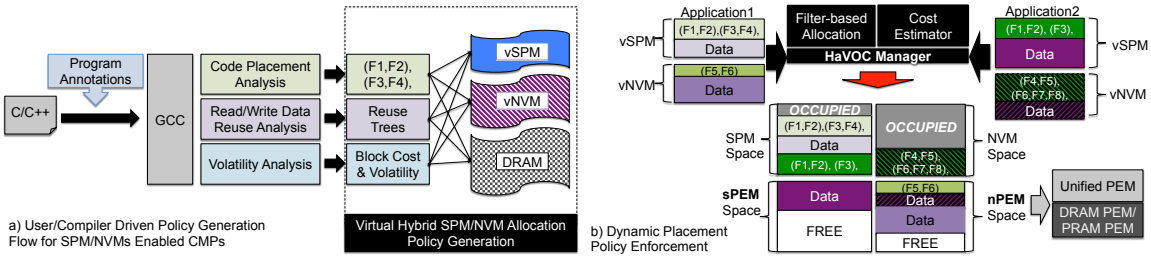

Figure 1. HaVOC-aware Policy Generation (a) and Enforcement (b).

2. Motivation

2.1 Comparison of Memory Technologies

Table 1. Memory Technology Comparison [32]

Features	SRAM	eDRAM	MRAM	PRAM
Density	Low	High	High	Very High
Speed	Very Fast	Fast	Fast read Slow write	Slow read Very slow write
Dyn. Power	Low	Medium	Low read High write	Medium read High write
Leak. Power	High	Medium	Low	Low
Non-Volatile	No	No	Yes	Yes

In order to overcome the growing leakage power consumption in SRAM-based memories, designers have proposed emerging Non-Volatile Memories (NVMs) as possible alternatives for on-chip memories [17, 24, 28]. As shown in Table 1 [32], NVMs (MRAM [13], and PRAM [11]) offer high densities, low leakage power, and comparable read latencies and dynamic read power with respect to traditional embedded memories (SRAM/eDRAM). One key drawback across NVMs is the high write latencies and dynamic write power consumption. To mitigate the drawbacks of the write operation in NVMs, designers have made the case for deploying hybrid on-chip memory hierarchies (a combination of SRAM, MRAM, and PRAM) [14, 32], which have shown up to 37% reduction in leakage power [14], and increased IPC as a byproduct of the higher density provided by NVMs [24, 32].

2.2 Programming for SPM/NVM Hybrid Memories

Unlike caches, which have built-in HW policies, SPMs are software controlled memories as their management is completely left to the programmer/compiler. At first glance, we would need to take traditional SPM-management schemes and adapt them to manage hybrid memories. However, SPM-based allocation schemes (e.g., [16, 26, 30]) assume physical access to the memory hierarchy; consequently, the traditional SPM-based programming model would require extensive changes to account for the different characteristics of the NVMs. Thus motivating the need for a simplified address space to minimize changes to the SPM-programming model.

2.3 Multi-tasking Support for SPM/NVM Hybrid Memories

The challenge of programming and managing SPM/NVM-based hybrid memories is aggravated by the adoption of open environments (e.g., Android OS), where users can download applications, install them, and run them on their devices. In these environments, it is possible that many of the running processes will require access to the physical SPMs, therefore, programmers/compilers can no longer assume that their applications are the only ones running on the system. Traditional SPM-sharing approaches [9, 27, 29] would either allocate part of the physical address space to each process (spatial allocation) or time-share the SPM space (temporal allocation). Once the entire SPM space has been allocated, all remaining data is then mapped to off-chip memory. In order to reduce the overheads of sharing the SPM/NVMs, our scheme exploits programmer/compiler-driven policies obtained through static analysis/annotations (Sec. 3.2) and uses the information to efficiently manage the memory resources at run-time (Sec. 3.3).

2.4 Target Platform

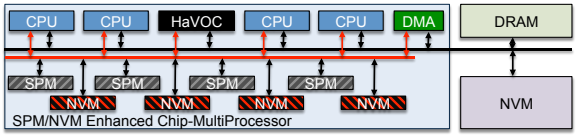

Figure 2. HaVOC Enhanced CMP with Hybrid NVM/SPMs.

Figure 2 shows a high level diagram of our SPM/NVM enhanced CMP, which consists of a number of OpenRISC-like cores, the *HaVOC* manager, a set of distributed SPMs and NVMs (MRAM or PRAM), and a DRAM/PRAM/Flash main memory hierarchy. For this work we focus on bus-based CMPs, so the communication infrastructure used here is an AMBA AHB bus, however, we can also exploit other types of communication fabrics.

2.5 Assumptions

We make the following assumptions: 1) The application can be statically analyzed/profiled so that code/data blocks can be mapped to SPMs [15, 16, 29]. 2) We operate over blocks of data (e.g., 1KB mini-pages). 3) We can map all code/data to on-chip/off-chip memory and do not use caches (e.g., [15]). 4) Part of off-chip memory can be locked in order to support the virtualization of the on-chip SPM/NVM memories.

3. HaVOC Overview

Figure 1 (a) shows our proposed compilation flow, which takes annotated source code, and performs various types of SPM/NVM-aware static analysis techniques (e.g., code placement, data reuse analysis, data volatility analysis); the compiler then uses this information to generate allocation policies assuming the use of virtual SPMs (vSPMs) and virtual NVMs (vNVMs). Figure 1 (b) shows our proposed dynamic policy enforcement mechanism for multi-tasking CMPs. The *HaVOC* manager (black box) takes in the vSPM/vNVM allocation policies provided by each application (Application 1 & 2), and decides how to best utilize the underlying memory resources. The rest of this section will go over each of the different components at a high level. The rest of paper will use data/code block interchangeably as our approach supports placement of both data and code onto the hybrid memories and will refer to SPM/NVM-based hybrid memories as *hybrid* memories.

3.1 Virtual Hybrid-Memory Space

In order to present the compiler/programmer with an abstracted view of the hybrid-memory hierarchy and minimize the complexity of our run-time system we propose the use of virtual SPMs and virtual NVMs. We leverage Bathen's [4] concept of vSPMs, which enables a program to view and manage a set of vSPMs as if they were physical SPMs. In order virtualize SPMs, a small part of main memory (DRAM) called protected evict memory (*PEM*) space was locked and used as extra storage. The run-time system would then prioritize the data mapping to SPM and PEM space based on a utilization metric. In this work we introduce the concept of virtual NVMs (vNVMs), which behave similarly to vSPMs, meaning that the run-time environment transparently allows each application to manage their own set of vNVMs. The main differences (w.r.t [4]) are: 1) Depending on the type of memory hierarchy, vNVMs can use a unified PEM space denoted as *sPEM* or off-chip NVM memory as extra storage space, denoted as *nPEM*. 2) The allocation cost estimation takes into account the volatility of a data (frequency of writes over its lifetime on a given memory), the cost of bringing in the given data, and the cost of evicting other application's data. 3) *HaVOC*'s dynamic policy exploits the notion of volatility and filtering (Sec. 3.5) to efficiently manage the memory real-estate. Management of virtual memories is done through a small set of APIs, which send management commands to the *HaVOC* manager. The *HaVOC* manager then presents each application with intermediate physical addresses (*IPAs*), which point to their virtual SPMs/NVMs. Traditional SPM-based memory management requires the data layout to use physical addresses by pointing to the base register of the SPMs, as a result, the same is expected of SPM/NVM-based memory hierarchies [14]. In our scheme, all policies use virtual SPM and NVM base addresses, so any run-time re-mapping of data will remain transparent to the initial allocation policies as the IPAs will not change.

3.2 Hybrid-memory-aware Policy Generation

The run-time system needs compile-time support in order to make efficient allocation decisions at run-time. In this paper we present various ways by which designers may generate policies (manual through annotations or through static analysis). These policies are then enforced (currently in best effort fashion) by the run-time system in order to prioritize the access to SPM/NVM space for the various applications running on the system. Each policy attempts to map data to virtual SPMs/NVMs, while the *HaVOC* manager dynamically maps the data to physical memories.

3.2.1 Volatility Analysis

We introduce a new metric, *data volatility*, to facilitate efficient loading of data on the hybrid on-chip memory configurations. Data volatility is defined as the write frequency of a piece of data over its accumulated lifetime. In order to estimate the volatility of a data block we first define a sampling time (ST_i), which can be in cycles, so that the union of all sample times equals the block's lifetime (Eq. 1). Next, we calculate the write frequency for each sample time (Eq. 2). Finally, we estimate the volatility of the data as the variation in its write frequency (Eq. 3). This metric is useful when deciding whether data is worth (cost effective) being mapped onto NVM. Highly volatile data implies that at some point the cost of keeping data in NVM during its entire lifetime might be greater than leaving it in main memory. As a result, when two competing applications request NVM space, the estimated cost function (e.g., energy savings) will be used to prioritize allocation of on-chip space, while volatility can be used as a tie breaker and prediction metric of cost fluctuation. Volatility may also be used to decide the granularity at which designers might do their data partitioning.

$$Data_{lifetime} \leftarrow \bigcup_{i=0}^{n} ST_i \tag{1}$$

$$Write^{i}_{freq.} \leftarrow Writes_i \div ST_i, i = 0 \cdots n \tag{2}$$

$$Data_{volatility} \leftarrow STDEV(Write^{i}_{freq}), i = 0 \cdots n \tag{3}$$

$$C(D_i) = C_{load}(D_i^f) + C_{evict}(D_i^t) + C_{util}(D_i^r, D_i^w) + \Delta_{leak}(D_i^r, D_i^w, D_i^{lifetime}) \tag{4}$$

We define the expected cost metric ($C(D_i)$) for a given data block (D_i) as shown in Eq. 4, which takes into account the cost of transfering data between memory type D_i^f and memory type D_i^t (C_{load}, C_{evict}), the utilization cost (C_{util}), and the extra leakage power consumed by mapping the given data to the preferred memory type. The cost represents *energy or latency*.

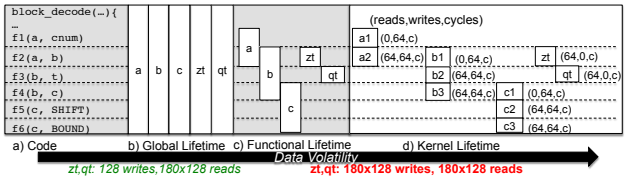

Figure 3. Data volatility across various lifetime granularities.

Figure 3 (a) shows sample code from JPEG [12] and how partitioning its data's lifetime may affect its data's volatility. Figure 3 (b) shows the global life time of the data arrays (*a, b, c, zt, qt*), where the number of accesses to NVM would be (128 rd, 23K wr) for *qt/zt* if we map and keep them in NVM during the entire execution of the program. To accomodate other data structures onto SPM space, arrays *qt/zt*'s lifetime may be split, resulting in finer life-time granularities (Figure 3 (c-d)). Though the read/write ratio of data remains the same (*qt/zt* still have 23K reads to 0 writes), finer granularity lifetimes might result in higher volatility (*qt/zt* now do 23K writes to NVM since they are reloaded every time *block_decode* is executed), making *qt/zt poor* mapping candidates for NVM.

3.2.2 Annotations

Programmers can embedded application-specific insights into source code through annotations [10] in order to guide the compilation process. Since we are working with virtualized address space, programmers can create hybrid-memory-aware policies that define the placement for a given data structure by simply defining the following parameters: *<preferred memory type, reads, writes, lifespan, volatility>*. These annotations are carried through the compilation process and used at run-time by the *HaVOC* manager, which uses the annotated information to allocate the data onto the *preferred* memory type.

3.2.3 Instruction Placement

Instructions/Stack-data are a very good candidates for mapping onto on-chip NVMs since their volatility is quite low (e.g., write once and use many times). In this work, we borrow traditional SPM-based instruction-placement schemes [18] and enhance them to take into account the possibility of mapping the data to NVM memories by introducing data volatility analysis into the flow. Like we discussed in Sec. 3.2.1, the granularity of the code partitioning (e.g, function, basic block, etc.) will affect how volatile the placement will become. As a result, when mapping a piece of code onto vNVM/vSPM, we need to partition our code such that Eq. 5 is met, where $C(D_i)$ represents the cost in Eq. 4. Our goal is to partition

the application such that we can minimize the number of code replacements ([18]) in order to minimize energy and execution time.

$$C(D_i)_{Off-Chip} \ll C(D_i)_{On-Chip} \quad (5)$$

3.2.4 Reuse Tree Analysis

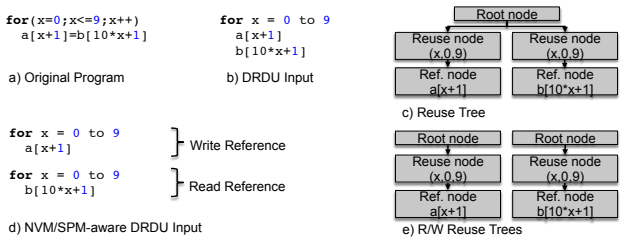

Figure 4. Reuse Tree Examples: a) Original Code, b) Original DRDU Input, c) Original Reuse Tree, d) Hybrid-memory-aware DRDU Input, e) Hybrid-memory-aware Reuse Trees.

We exploit data reuse analysis (DRDU) [16] to guide the mapping of data onto the vNVM/vSPM space. We adapted Issenin's DRDU algorithm to account for differences in NVM/SPM access energy and latency. DRDU takes as input access patterns derived from C source code in affine-expression mode and produces reuse trees (Figure 4 (a-c)), which are then used to decide which data buffers to map to SPMs (typically highly reused data). Our scheme splits the access patterns into read, write, and read/-modify reuse trees (Figure 4 (d)) and feeds them separate to the DRDU algorithm. We then use the reuse trees (Figure 4 (e)) generated to decide what data to map onto vNVM/vSPM space. The idea is to map highly read-reused data with a long access distance onto vNVM to minimize number of fetches from off-chip memory, while highly reused read-data with short lifetimes will be mapped to vSPM preferably. Highly reused-read-modify data with low write-volatility should be mapped to vNVM, while highly reused write-data and read-modify data with high write-volatility should go to vSPM.

3.2.5 Memory Allocation Policy Generation

Policy generation is not limited to the schemes above; they are exemplars of how our virtualization layer can be used to seamlessly manage on-chip hybrid memories. The last step in our flow involves the generation of near-optimal hybrid-memory layouts we define as *allocation policies*. This process takes as input data/code blocks with various pre-computed costs (e.g., Eq. 4) obtained from static analysis (Sections 3.2.1,3.2.4, and 3.2.3) and annotations (Sec. 3.2.2), which are combined and feeds them as input to an enhanced hybrid-memory-aware allocator based on [14].

3.3 HaVoC Manager

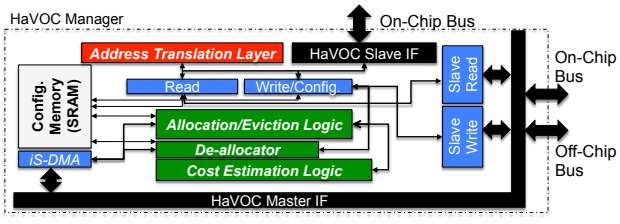

Figure 5. *HaVOC* Manager

The *HaVOC* manager may be implemented in software as an extended memory manager embedded within a hypervisor/OS or as a hardware module (e.g., [4, 9]). The software implementation is quite flexible and does not require modifying existing platforms. The hardware version requires additional hardware and the necessary run-time support, but the run-time overheads will be much lower than the software version. In this work, we present a proof-of-concept embedded hardware implementation (Figure 2). Figure 5 shows a block diagram of the *HaVOC* manager. It consists of a memory-mapped slave interface, which is used by the system's masters (e.g., CPUs) and handles the read/write/configuration requests. The address translation layer module converts IPAs to physical addresses (PAs) in one cycle [4, 7]). The manager consists of 1KB to 256KB of configuration memory used to keep block metadata information (e.g., volatility, # accesses, etc.). The allocation/eviction logic uses the cost estimation (e.g., efficiency) logic to prioritize access to on-chip storage. Finally, the internal DMA (iS-DMA) allows the manager to asynchronously transfer data between on-chip and off-chip memory. In order to use the *HaVOC* manager, the compiler generates two things: 1) the creation of the required virtual SPM/NVMs through the use of our APIs (creates configuration packets) and 2) The use of IPAs instead of PAs during the data/code layout stage (e.g., memory maps using purely virtual addresses for SPMs/NVMs). Any read/write to a given IPA is translated and routed to the right PA. Any write to *HaVOC* configuration memory space is then used to mange the virtualized address space. The goal is to allow each application to manage its virtual on-chip memories as if it had full access to the on-chip real-estate.

3.4 *HaVOC*'s Intermediate Physical Address

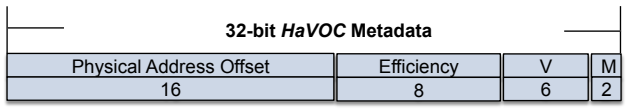

Figure 6. *HaVOC*'s Virtual Memory Metadata

Figure 6 shows the metadata needed to keep track of the allocated blocks. Note that these fields are tunable parameters and we leave the exploration for future work. The *Physical Address Offset* is used to complement the byte offset in the original IPA (Figure 7) when translating IPA to PA addresses. The $Efficiency$ and the volatility (V) fields are used during the allocation/eviction phases as metrics to guide the allocator's decisions. The M field represents the preferred memory type (e.g., SPM, NVM, DRAM).

Figure 7 shows a high level view of the layout of one application during its execution and the address translation an IPA undergoes. Each application requiring access to a virtual SPM or virtual NVM must do so through an IPA, which means creating the desired # of vSPMs and vNVMs for use (see Sec. 3.8). For the sake of demonstration, we set the on-chip SRAM to 4KB, so to address a vSPM's block metadata we only need 12bits. The IPA is translated and complemented by the *Physical Address Offset* fetched from the block metadata (Figure 7 (a)). These 16bits combined with the 10bit *Byte Offset* make out the physical address of the on-chip SPM/NVM memory (Figure 7 (b)). Though we can virtualize 2^{26} Bytes of on-chip storage, we are bounded by the total metadata stored in configuration memory. Figure 7 (c) shows the initial mapping of *Application 1*, and Figure 7 (d) shows an updated mapping after *Application 2* has launched. Notice that the IPA remains the same for both, as *HaVOC* updates its allocation table and its block's metadata. Re-mapping blocks of data/code requires the *HaVOC* manager to launch an interrupt which tells the CPU running *Application 1* that it needs to wait for the re-mapping of its

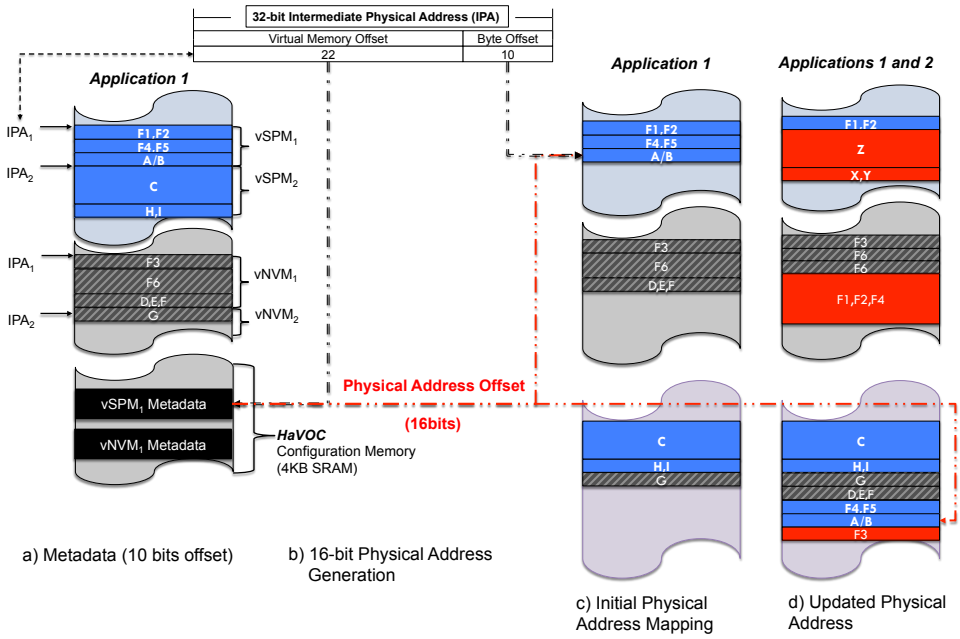

Figure 7. *HaVOC*'s Intermediate Physical Address

data. It is possible to avoid interrupting execution as *HaVOC* has an internal DMA module so it may asynchronously evict blocks. In the event that there is a request for a piece of data belonging to a block being evicted, the *HaVOC* manager waits until re-mapping is done since read/write/configuration requests to *HaVOC* are blocking requests (e.g., *HaVOC* behaves like an arbiter).

3.5 *HaVOC*'s Dynamic Policy Enforcement

Table 2. Filter Inequalities and Preferred Memory Type

Filter	Pref.	Inequalities
F1	sram	$E(D_i^{spm}) > E(D_i^{dram}) \bigwedge E(D_i^{nvm}) < E(D_i^{dram}) \bigwedge V > T_{vol}$
F2	nvm	$E(D_i^{nvm}) > E(D_i^{dram}) \bigwedge V < T_{vol}$
F3	either	$E(D_i^{spm}) > E(D_i^{dram}) \bigwedge E(D_i^{nvm}) > E(D_i^{dram})$
F4	dram	$E(D_i^{spm}) < E(D_i^{dram}) \bigwedge E(D_i^{nvm}) < E(D_i^{dram})$

Algorithm 1 *FilterDynamic* Allocation Algorithm

```
Require: req{size, cost, volatility}
pref_mem ← filter(req)
if allocatable(req, pref_mem) then
    return ipa ← update_alloc_table(req)
end if
min_set ← sort_Hi2LowEff(alloc_table, size)
if E(min_set) < C_evict(min_set) + E(req) then
    evict(min_set)
    return ipa ← update_alloc_table(req)
else
    return ipa ← mm_malloc(req)
end if
```

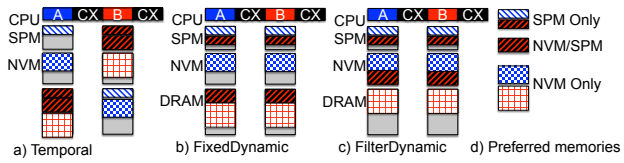

Figure 8. Dynamic Hybrid-memory Allocation Policies

In order to support dynamic allocation of hybrid-memory space we define *three* block-based allocation policies: 1) *Temporal* allocation, which combines temporal SPM-allocation ([29]) and hybrid-memory allocation ([14]), and adheres to the initial layout obtained through static analysis (Sec. 3.2); however, the application's SPM and NVM contents must be swapped on a context-switch to avoid conflicts with other tasks (Fig. 8 (a)). 2) *FixedDynamic* allocation, which combines dynamic-spatial SPM-allocation ([9]) and hybrid-memory allocation [14], and maps the data block to the preferred memory type (adhering to the initial layout) as long as there is space, otherwise, data is mapped to DRAM (Fig. 8 (b)). 3) *FilterDynamic* allocation (Alg. 1), which exploits the concept of filtering and volatility to find the best placement. Each request is filtered according to a set of inequalities (shown in Table 2) which determine the preferred memory type (Fig. 8 (d)). The volatility of the data block (V) and its mapping efficiency ($E(D_i) = C(D_i)/|D_i|$) are used to determine what memory type would minimize the block's energy (or access latency). For instance, data with low volatility and high energy efficiency could potentially benefit more from being mapped to NVM than SRAM (e.g., filter *F2* in Table 2). If there is enough preferred memory space (e.g., SPM or NVM), the dynamic allocator adheres to Eq. 4 prior to loading the data. If there is not enough space, then the allocator follows Alg. 1 and sorts the allocated blocks from highest to lowest efficiency (e.g., energy per bit). It then compares the cost of evicting the least important blocks (MIN_{Set}) with the cost of dedicating the space to the new block. If the efficiency of bringing the new block offsets the eviction cost and efficiency of the data already mapped to the preferred memory type ($E(MIN_{Set})$), then *HaVOC* evicts the MIN_{Set} blocks and updates the allocation table with the new block ($|new\ block| \leq |MIN_{Set}|$). In the event the preferred memory type is either NVM or SPM (filter *F3* in Table 2), the allocator evicts the $min(MIN_{Set}^{spm}, MIN_{Set}^{nvm})$. At the end, *HaVOC* allocates either on-chip space or off-chip space (unified PEM space (*sPEM*)), resulting in the allocation shown in Fig. 8 (c), where a data block originally intended for SPM is mapped to NVM.

3.6 Dynamic Allocation Decision Flow

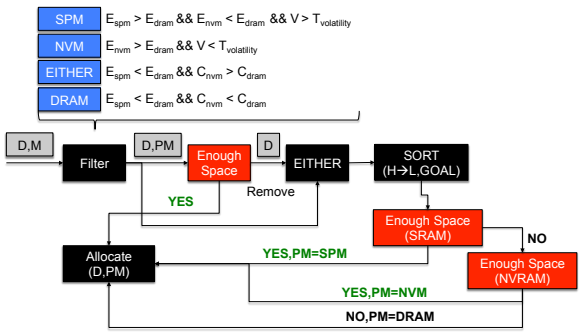

Figure 9. Application Execution

Figure 9 shows the decision flow followed by our *FilterDynamic* allocation policy. Each allocation request for data/code block (D) is filtered by applying a set of inequalities to derive the preferred memory type (PM) for the given block. This results in a tuple (D, PM), which is forwarded to the allocation engine. The allocator checks if there is enough space in the preferred memory (PM), if so, allocate the block in the preferred memory (D, PM). If there is not enough space or the preferred memory type is either SPM or NVM, then we sort the allocated blocks from highest to lowest efficiency generating a set of least useful allocated blocks (MIN_{Set}^{mem}) as long as the sum of the blocks is less than or equal to the requested block size. Since we are using block-based allocation, it is possible to optimize this step. For instance, it the block unit is 1KB and each request block is 1KB, then the (MIN_{Set}^{mem}) is composed of a single 1KB block. The backend implementation is open for exploration so that we may find what the optimal/best block size should be as the block size affects how much required on-chip configuration memory we may need to store the block's metadata. The next step may be done in parallel (if either memory is preferred), or synchronous (if there is not enough space in the preferred memory). In the case either memory is acceptable, then the allocator evicts the $min(MIN_{Set}^{spm}, MIN_{Set}^{nvm})$, and allocates the space to the new block (D). Another optimization to bypass the sorting step, it is possible to maintain a small linked list of blocks, sorted from lowest to highest efficiency, so that rather than sorting on every transaction, we keep a sorted list, and compute the MIN_{Set}^{mem} using the head of the list, and on eviction we can update the list. Computing MIN_{Set}^{mem} would then be $O(1)$, and updating the list $O(Blocks)$. This of-course comes at the cost of extra storage to keep the list, therefore increasing the size of the configuration memory.

3.7 Application Layout

Figure 10 shows a snapshot of the resulting allocation policy (Figure 10 (c)) for the JPEG [12] benchmark. First, we start with the source code (Figure 10 (a)) which is statically analyzed. Second, the application's memory map is shown in Figure 10 (b). Note that the source code itself is executed 180 times, and if loaded once, its volatility is very low, if loaded every time it gets executed, then its volatility increases. Figure 10 (c) shows the block information consisting of the virtual start/end addresses, number of accesses, volatility, and the *preferred* memory type (e.g., NVM).

Figure 11 shows a snapshot of a subset of blocks for the three dynamic allocation policies discussed in this paper (*Oracle, FixedDynamic, FilterDynamic*). The bold-red lines mark difference in mappings between the various policies. These allocations are a combination of the concurrent execution of the ADPCM and AES benchmarks [12]. As we can observe, the *FixedDynamic* policy runs out of SPM space, and all data mapped to SRAM or NVM is mapped to off-chip memory (DRAM). The *Oracle and FilterDynamic* policies have similar resulting mappings, however, there are some difference as in this case; *Buffer_0x10<25>* is mapped to NVM by *Oracle* and to SRAM by *FilterDynamic*. The *Buffer_0x10<25>* block appears at a time when evicting the (MIN_{Set}^{nvm}) is more expensive than the benefits of mapping *Buffer_0x10<25>* to NVM, however, the *FilterDynamic* policy realizes that SRAM space is more cost-effective (at the time), and maps the block to SRAM.

3.8 Execution Timing Diagram

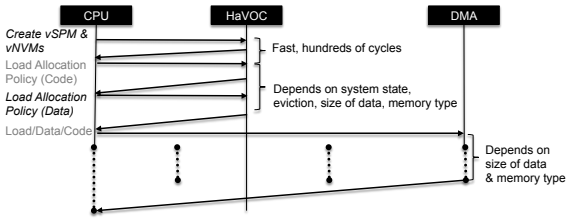

Figure 12. Application Execution

Figure 12 shows a timing diagram for the loading/execution of an application. First, prior to executing the application, we create the vSPMs/vNVMs. Second, we load the application's allocation policies, which determine how to allocate the memory space. Finally, once the space is created, the application will continue to execute as if it was talking to physical SPMs/NVMs.

4. Related Work

Most work in hybrid memories has focused on replacing/complementing main memory (DRAM) or caches (SRAM) with a combination of various NVMs to reduce leakage power and increase throughput (a by product of higher NVM density). Joo et al. [17] proposed PCM as an alternative to SRAM for on-chip caches. Sun et al. [28] introduced MRAM into the cache hierarchy of a NUCA-based 3D stacked multi-core platform. Mishra et al. [24] followed up by introducing STT-RAM as an alternative MRAM memory and hid the overheads in access latencies by customizing how accesses are prioritized by the the interconnect network. Wu et al. [32] presented a hybrid cache architecture consisting of SRAM (fast L1/L2 accesses), eDRAM/MRAM (slow L2 accesses), and PRAM (L3). Hu et al. [14] were the first to explore a hybrid SPM/NVM memory hierarchy and proposed a compile-time allocation algorithm that exploits the benefits of both NVMs and SPMs for a single application. Hybrid main memory has also been studied [33]. Mogul et al. [25] and Wongchaowart et al. [31] attempted to reduce write overheads in main memory by exploiting page migration and block content signatures. Ferreira et al. [8] introduced a memory management module for hybrid DRAM/PCM main memories. Static analysis has been explored to efficiently map application data on off-chip hybrid main memories [22, 23].

HaVOC is different from approaches that address hybrid cache/-main memories in that we primarily focus on hybrid SPM/NVM-based hierarchies, however, our scheme can be complemented by both existing schemes that leverage the benefits of both on-chip SRAMs and NVMs as well as hardware/system-level solutions that address hybrid off-chip memories (e.g., DRAM/eDRAM and PCM). Our work is different from [14] in that we consider: 1) shared distributed on-chip SPMs and 2) dynamic support for multi-tasking systems, however, our scheme can benefit from compile-time analysis schemes that provide our runtime system with allocation hints (e.g., [14, 22, 23]). Like [4], we use part of off-chip

a) Source Code

```
void IZigzagMatrix ()
{
  int i;
  for (i = 0; i < DCTSIZE2; i++)
  {
     omatrix[i] = imatrix[zigzag_index[i]];
  }
}
```

b) Memory Map

```
.text      0x004001f0    0x6130 /tmp/cc7mgwib1.o
...
0x00402b28            IZigzagMatrix
...
```

c) Buffer Information and Mapping

Buffer Name	Start Addr	End Addr	Reads	Writes	Volatility	Mapping
Buffer_0x40<10>	4204544	4205567	398708	0	0	NVM
Buffer_0x40<11>	4205568	4206591	595805	0	0	NVM
Buffer_0x40<12>	4206592	4207615	440754	0	0	NVM
Buffer_0x40<13>	4207616	4208639	313128	0	0	NVM

Figure 10. Layout Information After Static Analysis

ORACLE		reads	writes	efficiency	FIXED		reads	writes	efficiency	FILTER		reads	writes	efficiency
Buffer_0x40<9>	NVM	34012	0	33	Buffer_0x40<9>	NVM	34012	0	33	Buffer_0x40<9>	NVM	34012	0	33
Buffer_0x7f<31>	DRAM	7	0	0	Buffer_0x7f<31>	DRAM	7	0	0	Buffer_0x7f<31>	DRAM	7	0	0
Buffer_0x7f<32>	SRAM	139055	88045	135	Buffer_0x7f<32>	DRAM	139055	88045	135	Buffer_0x7f<32>	SRAM	139055	88045	135
Buffer_0x7f<34>	DRAM	84	82	0	Buffer_0x7f<34>	DRAM	84	82	0	Buffer_0x7f<34>	DRAM	84	82	0
Buffer_0x10<19>	DRAM	230	120	0	Buffer_0x10<19>	DRAM	230	120	0	Buffer_0x10<19>	DRAM	230	120	0
Buffer_0x10<20>	DRAM	62	36	0	Buffer_0x10<20>	DRAM	62	36	0	Buffer_0x10<20>	DRAM	62	36	0
Buffer_0x10<21>	NVM	15254	416	14	Buffer_0x10<21>	DRAM	15254	416	14	Buffer_0x10<21>	SRAM	15254	416	14
Buffer_0x10<22>	SRAM	18680	512	18	Buffer_0x10<22>	DRAM	18680	512	18	Buffer_0x10<22>	SRAM	18680	512	18
Buffer_0x10<23>	SRAM	18722	512	18	Buffer_0x10<23>	DRAM	18722	512	18	Buffer_0x10<23>	SRAM	18722	512	18
Buffer_0x10<24>	SRAM	18892	512	18	Buffer_0x10<24>	DRAM	18892	512	18	Buffer_0x10<24>	SRAM	18892	512	18
Buffer_0x10<25>	NVM	24496	154	23	Buffer_0x10<25>	DRAM	24496	154	23	Buffer_0x10<25>	SRAM	24496	154	23

Figure 11. Resulting Map

memory to virtualize on-chip memories, however, our approach differs in that our primary focus is the efficient management of hybrid on-chip memories, and as a result, *HaVOC*'s programming model and run-time environment account for the different physical characteristics of SRAMs, DRAMs and NVMs (MRAM and PRAM). Moreover, we believe that we can complement our static-analysis/allocation policy generation with other SPM-management techniques [9, 19, 27, 29, 30].

5. Experimental Results

Table 3. Configurations

Config.	Applications	CPUs	vSPM/vNVM Space	SPM/NVM Space
C1	adpcm,aes	1	32/128 KB	16/64 KB
C2	adpcm,aes,blowfish,gsm	1	64/256 KB	16/64 KB
C3	C2 & h263,jpeg,motion,sha	1	128/512 KB	16/64 KB
C4	same as C2	2	64/256 KB	32128 KB
C5	same as C3	2	128/512 KB	32/128 KB
C6	same as C3	4	128/512 KB	64/256 KB

5.1 Experimental Setup and Goals

Our goal is to show that *HaVOC* is able maximize energy savings and increase application throughput in a multi-tasking environment under various scenarios. First, we generate two sets of hybrid-memory aware allocation policies (Sec. 3.2.5), one set of policies attempts to minimize execution time (Sec. 5.2) and the other attempts to minimize energy (Sec. 5.3). These policies are generated at compile-time and enforced at run-time by a set of dynamic allocation policies (Sec. 3.5) under various system configurations (Table 3). Next, we show the effects of the allocation policy's block-size on execution time (Sec. 5.4). We built a trace-driven simulator that models a light-weight RTOS, with a round-robin scheduler and context-switching enabled (time slot = 50K cycles). Our simulator is able to model CMPs consisting of an AMBA AHB bus, OpenRISC-like in-order cores, distributed SPMs and NVMs, and the *HaVOC* manager (Fig. 2). We bypassed the cache and mapped all data to either SPM, NVM, or main memory (see Sec. 3.2.5). We obtained traces from the CHStone [12] embedded benchmarks by using SimpleScalar [1]. We model on-chip SPMs (SRAMs), MRAMs and PRAMs by interfacing our simulator with NVSim [6] and set leakage power as the optimization goal. To virtualize SPMs/NVMs we use the unified PEM space model discussed in Sec.3.1 (sPEM). The *HaVOC* manager consists of 4KB low power configuration memory (SRAM).

5.2 Enforcing Performance Optimized Policies

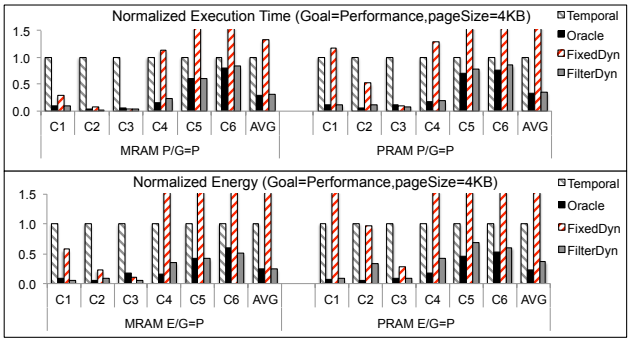

Figure 13. Normalized Execution Time and Energy Comparison for Performance Optimized Policies

For this experiment we generated allocation policies that minimized execution time for each application. We then executed each application on top of our simulated RTOS/CMP. Table 3 shows each configuration (C1-6), which has a set of applications running concurrently over a # of CPUs, and a pre-defined hybrid-memory physical space. To show the benefit of our approach we implemented four policies: The three described in Sec. 3.5 (*Temporal, FixedDynamic, FilterDynamic*), and a policy we call *Oracle* (black bar in Fig. 13), which is a near-optimal policy because on every block-allocation request, it feeds the entire memory map to the same policy generator the compiler uses to generate policies statically (see Sec. 3.2.5). The idea is to show that our *FilterDynamic* policy (backward-slashed bars in Fig. 13) achieves around the same quality allocation solutions as the more complex *Oracle* policy. Fig. 13 shows the the normalized execution time and energy for each of the different configurations (*C1-6*, Goal=Min Execution Time denoted as *G=P*) using 4KB blocks and different memory types with respect to the *Temporal* policy. The *FixedDynamic* policy (forward-slashed bars in Fig. 13) suffers the greatest impact on energy and execution time as memory space increases (*C4-6*) since it adheres to the decisions made at compile-time and does not efficiently allocates memory blocks at run-time. In general, we see that the *FilterDynamic* policy performs almost as good (in terms of energy and execution time) as the *Oracle* policy (within 8.45% execution time). Compared with the *Temporal* policy, *HaVOC*'s *FilterDynamic* policy is able to reduce execution time and energy by an average 75.42% and 62.88% respectively when the initial application policies have been optimized for execution time minimization.

5.3 Enforcing Energy Optimized Policies

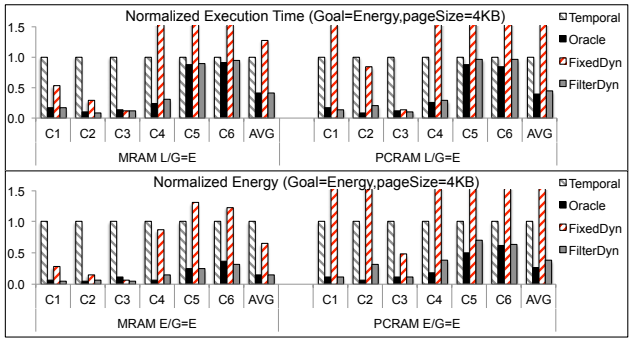

Figure 14. Normalized Execution Time and Energy Comparison for Energy Optimized Policies

Fig. 14 shows the the normalized execution time and energy for each of the different configurations and memory types (Goal=Min Energy denoted as *G=E*) with respect to the *Temporal* policy. Like it was in the case of *G=P*, both the *Temporal and FixedDynamic* policies are unable to efficiently manage the on-chip real-estate. The *FilterDynamic and Oracle* policies are able to greatly reduce execution time and energy, the *FilterDynamic* is within 3.54% of the execution time achieved by the The *Oracle* policy. Compared with the *Temporal* policy, *HaVOC*'s *FilterDynamic* policy is able to reduce execution time and energy by an average 85.58% and 61.94% respectively when the initial application policies have been optimized for energy minimization. The goal of this experiment was to show that regardless of the initial optimization goal, *HaVOC*'s *FilterDynamic* policy is able to achieve as good results as the *Oracle* policy.

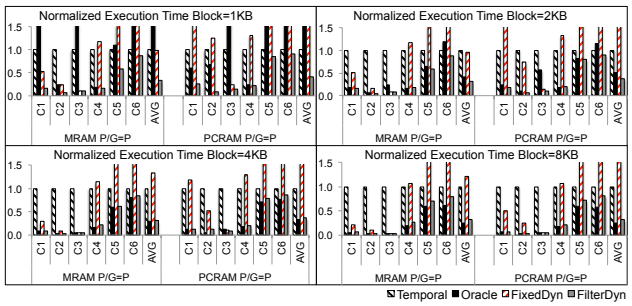

Figure 15. Effects of Varying Block Size on Policy Enforcement and Allocation

5.4 Block Size Effect on Allocation Policies

So far we have seen that *Oracle* policy seems as a feasible dynamic allocation solution, which would potentially enhance *HaVOC*'s virtualization engine. However, as we mentioned before, the *Oracle* policy is very complex as it runs in $O(Blks * spm_{size} * nvm_{size})$. The *FixedDynamic* policy on the other extreme runs in $O(Blks)$, however, its efficiency may be even worse than the *Temporal* policy. *HaVOC*'s *FilterDynamic* policy on the other hand, keeps a semi-sorted list of data blocks, as a result it can be $O(Blks)$ best case or $O(Blks\ log\ Blks)$ worst case for sorting, and the final filtering decision runs in $O(Blks_{spm} + Blks_{nvm} + Blks_{dram})$, which results in $O(Blks\ log\ Blks) + O(Blks_{spm} + Blks_{nvm} + Blks_{dram})$. Thus, the complexity and execution time of the *Oracle* policy will increase orders of magnitude as the number of data/code blocks to allocate increases (*Blks*) or as the available resources increases (spm_{size}, nvm_{size}). This is validated in Fig. 15, where we have varying block size (as a result, number of blocks to allocate increases) and increase in available resources (*C3-C6*). As we can see, for block size = 1KB, where the number of blocks to allocate is in the hundreds, the allocation time of the *Oracle* is orders of magnitude greater than the *FilterDynamic* policy. For block size = 2KB, we see that as resources increase, the *Oracle* allocation time once again prevents it from being a feasible solution. On average, we observe that *HaVOC*'s *FilterDynamic* is capable of achieving as good solutions as the *Oracle* policy (within 10% margin) with much lower complexity. On average, across all test scenarios (varying page sizes, different optimization goals, and different configurations) we see that the *FilterDynamic* is able to reduce execution time and energy by 60.8% and 74.7% respectively.

6. Comparison Between Oracle and FilterDynamic Run-times

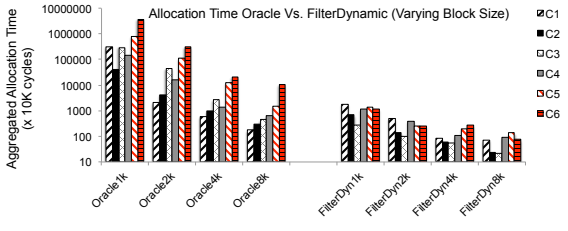

Figure 16. Oracle and FilterDynamic Run Time Comparison

Figure 16 shows a more detailed comparison between the aggregate run time (in cycles) between the *Oracle* and *FilterDynamic*

policies. As we can see, the configuration with the highest resources ($C6$) causes the *Oracle* policy to spend orders of magnitude more cycles deciding where to allocate blocks than the *Filter-Dynamic* policy. Another factor that affects the *Oracle* policy's decision time is the number of buffers to allocate, which is the highest when the block size is 1KB.

7. Conclusion

We presented *HaVOC*, a hybrid-memory aware virtualization layer for dynamic memory management of applications executing on CMP platforms with hybrid on-chip NVM/SPMs. We introduced the notion of data volatility analysis and proposed a dynamic filter-based memory allocation scheme to efficiently mange the hybrid on-chip memory space. Our experimental results for embedded benchmarks executing on a hybrid-memory-enhanced CMP show that our dynamic filter-based allocator greatly minimizes execution time and energy by 60.8% and 74.7% respectively with respect to traditional multi-tasking-aware SPM-allocation policies. Future work includes: 1) Integrating *HaVOC*'s concepts in a hypervisor to support full on-chip hybrid-memory virtualization. 2) Integration, evaluation, and enhancement of other SPM-based allocation policies for hybrid-memories. 3) Add off-chip NVM memories to support the virtualization of on-chip NVMs (*nPEM*).

References

[1] T. Austin et al. Simplescalar: an infrastructure for computer system modeling. *Computer*, 35(2), Feb. 2002.

[2] N. Azizi et al. Low-leakage asymmetric-cell sram. *TVLSI*, Vol. 11, aug. 2003.

[3] R. Banakar et al. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES '02*, 2002.

[4] L. Bathen et al. SPMVisor: dynamic scratchpad memory virtualization for secure, low power, and high performance distributed on-chip memories. In *CODES+ISSS '11*, 2011.

[6] X. Dong et al. Pcramsim: system-level performance, energy, and area modeling for phase-change ram. In *ICCAD '09*, 2009.

[7] B. Egger et al. Dynamic scratchpad memory management for code in portable systems with an mmu. *ACM TECS*, 7, January 2008.

[8] A. Ferreira et al. Using pcm in next-generation embedded space applications. In *RTAS '10*, april 2010.

[9] P. Francesco et al. An integrated hardware/software approach for runtime scratchpad management. In *DAC '04*, 2004.

[10] S. Guyer et al. An annotation language for optimizing software libraries. In *DSL '99*, 1999.

[11] S. Hanzawa et al. A 512kb embedded pcm with 416kb/s write throughput at 100 ua cell write current. In *ISSCC '07*, feb. 2007.

[12] Y. Hara et al. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *ISCAS '08*, May 2008.

[13] M. Hosomi et al. A novel nonvolatile memory with spin torque transfer magnetization switching: spin-ram. In *IEDM '05*, dec. 2005.

[14] J. Hu et al. Towards energy efficient hybrid on-chip scratch pad memory with non-volatile memory. In *DATE '11*, march 2011.

[15] IBM. The cell project. *http://www.research.ibm.com/cell/*, 2005.

[16] I. Issenin et al. Multiprocessor system-on-chip data reuse analysis for exploring customized memory hierarchies. In *DAC '06*, 2006.

[17] Y. Joo et al. Energy- and endurance-aware design of phase change memory caches. In *DATE '10*, march 2010.

[18] S. Jung et al. Dynamic code mapping for limited local memory systems. In *ASAP '10*, july 2010.

[19] M. Kandemir et al. Dynamic management of scratch-pad memory space. In *DAC '01.*, 2001.

[20] S. Kaneko et al. A 600mhz single-chip multiprocessor with 4.8gb/s internal shared pipelined bus and 512kb internal memory. In *SSCC '03*, 2003.

[21] N. Kim et al. Leakage current: Moore's law meets static power. *Computer*, 36(12), dec. 2003.

[22] K. Lee et al. Application specific non-volatile primary memory for embedded systems. In *CODES+ISSS '08*, 2008.

[23] T. Liu et al. Power-aware variable partitioning for dsps with hybrid pram and dram main memory. In *DAC '11*, june 2011.

[24] A. Mishra et al. Architecting on-chip interconnects for stacked 3d stt-ram caches in cmps. In *ISCA '11*, 2011.

[25] J. Mogul et al. Operating system support for nvm+dram hybrid main memory. In *HotOS'09*, 2009.

[26] P. Panda et al. Efficient utilization of scratch-pad memory in embedded processor applications. In *EDTC '97*, 1997.

[27] V. Suhendra et al. Scratchpad allocation for concurrent embedded software. In *CODES+ISSS '08*, 2008.

[28] G. Sun et al. A novel architecture of the 3D stacked MRAM L2 cache for CMPs. In *HPCA '09*, feb. 2009.

[29] H. Takase et al. Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems. In *DATE '10*, 2010.

[30] M. Verma et al. Data partitioning for maximal scratchpad usage. In *ASP-DAC '03*, 2003.

[31] B. Wongchaowart et al. A content-aware block placement algorithm for reducing pram storage bit writes. In *MSST '10*, 2010.

[32] X. Wu et al. Hybrid cache architecture with disparate memory technologies. In *ISCA '09*, 2009.

[33] P. Zhou et al. A durable and energy efficient main memory using phase change memory technology. In *ISCA '09*, 2009.