Runtime-Guided Management of Scratchpad Memories in Multicore Architectures

Lluc Alvarez*† Miquel Moretó*† Marc Casas* Emilio Castillo*†
Xavier Martorell*† Jesús Labarta*† Eduard Ayguadé*† Mateo Valero*†

*Barcelona Supercomputing Center**
Barcelona, Spain
name.surname@bsc.es

Departament d'Arquitectura de Computadors†
Universitat Politècnica de Catalunya
Barcelona, Spain

***Abstract*—The increasing number of cores and the anticipated level of heterogeneity in upcoming multicore architectures cause important problems in traditional cache hierarchies. A good way to alleviate these problems is to add scratchpad memories alongside the cache hierarchy, forming a hybrid memory hierarchy. This memory organization has the potential to improve performance and to reduce the power consumption and the on-chip network traffic, but exposing such a complex memory model to the programmer has a very negative impact on the programmability of the architecture. Emerging task-based programming models are a promising alternative to program heterogeneous multicore architectures. In these models the runtime system manages the execution of the tasks on the architecture, allowing them to apply many optimizations in a generic way at the runtime system level.**

This paper proposes giving the runtime system the responsibility to manage the scratchpad memories of a hybrid memory hierarchy in multicore processors, transparently to the programmer. In the envisioned system, the runtime system takes advantage of the information found in the task dependences to map the inputs and outputs of a task to the scratchpad memory of the core that is going to execute it. In addition, the paper exploits two mechanisms to overlap the data transfers with computation and a locality-aware scheduler to reduce the data motion. In a 32-core multicore architecture, the hybrid memory hierarchy outperforms cache-only hierarchies by up to 16%, reduces on-chip network traffic by up to 31% and saves up to 22% of the consumed power.

I. Introduction

Cache coherent shared memory has traditionally been the most common memory organization for multicore chips. This memory hierarchy provides important advantages in programmability, but presents significant inefficiencies when the number of cores per chip reaches orders of magnitude beyond 10's [1]. At these levels, complex power-hungry hardware structures and large amounts of traffic in the interconnection network to maintain all the data in a coherent state are required. On the opposite side, scratchpad memories [2] (SPMs) are a well-known alternative to caches in power-constrained domains. SPMs consume less power than caches and they do not generate coherence traffic, but they degrade the programmability of the architecture because the programmer has to explicitly manage the SPMs.

This paper is published in Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT'15), 2015, pp. 379-391. The paper is available at http://dx.doi.org/10.1109/PACT.2015.26.

The trend towards massively parallel multicore chips makes it impossible to keep relying in purely cache-coherent memory hierarchies, due to their power consumption and scalability issues, neither in pure SPMs designs, due to their programmability issues. Instead, computer architecture is exhibiting a trend towards more heterogeneity, clearly shown by proposals like the Cell B. E. [3], GPGPUs [4], or more recently Intel's Knights Landing [5]. These designs have different kinds of cores and hybrid memory hierarchies that combine caches and SPMs. Typically, programming such hybrid designs implies dealing not only with the programmability burdens that SPMs impose, but also with different memory addresses spaces. Such programming hardships seriously limited the Cell B. E. architecture and are seriously hurting the wide-spread usage of GPGPU architectures. More recent proposals are also heterogeneous and massively parallel but provide a single address space to the programmer. For instance, the joint Collaboration of Oak Ridge, Argonne, and Lawrence Livermore (CORAL) leverages the IBM Power Architecture, NVIDIA's Volta GPU, and Mellanox's interconnection technologies to build an extremely parallel, heterogeneous and single-spaced system that combines caches and SPMs in its memory hierarchy. Other proposals like the Intel's Knights Landing also contain hybrid and reconfigurable memory hierarchies, where a high bandwidth 3D stacked DRAM can be configured as a software-managed SPM.

Despite the efforts made by vendors to build systems with single memory address spaces that are parallel and heterogeneous enough to provide performance under an affordable power budget, the programmability issue is still not solved. Exposing deep and hybrid memory hierarchies to the programmer requires adapting current scientific and industrial codes that rely on a cache coherent memory hierarchy and, more importantly, increases the difficulty and the cost of developing new software.

Multicores are usually programmed with thread-based programming models like OpenMP or Pthreads. To handle heterogeneity, OpenMP 4.0 provides support for tasking and dependences, which allows to expose the available parallelism of an application by splitting the code in sequential pieces of work, called tasks, and by specifying the data and control dependences between them. With this information the runtime system manages the parallel execution of the workload following a data-flow scheme, scheduling tasks to cores and taking care of synchronization between tasks. Decoupling the application from the architecture not only eases programmability, but also allows to exploit the available information in the runtime system to drive optimizations in a generic and application-agnostic way [6], [7].

This paper proposes to take advantage of the performance, scalability and power consumption benefits of a hybrid memory hierarchy without adding any programming burden by using well accepted parallel programming models like OpenMP 4.0., exploiting task annotations to manage the SPMs of the hybrid memory hierarchy transparently to the programmer. To do so, the runtime system is in charge of mapping the data specified in the task dependences to the SPMs, so memory accesses to this data are served in a power-efficient way and without generating coherence traffic, while the rest of memory accesses are served by the L1 cache. The proposal exploits two key characteristics of task-based models: first, the inputs and outputs of a task are specified in the source code and, second, the semantics of the programming model ensure that the inputs and outputs of a task are private to that task when it is executed. The main contributions of this paper are:

- A runtime system for task-based programming models that transparently manages the SPMs of a hybrid memory hierarchy that combines caches and SPMs. This runtime exploits the benefits of a locality-aware scheduler that assigns tasks to cores aiming to minimize data movements in the memory hierarchy.
- Two schemes that allow the runtime system to overlap data transfers for the SPMs with the task scheduler or with the execution of the previous task. These schemes hide the communication cost of the data transfers, reaching the performance of an ideal system with zero latency data transfers.
- A complete evaluation of the task-based runtime system managing the hybrid memory hierarchy in a 32-core multicore architecture, demonstrating the benefits of this memory organization when compared to a cache hierarchy. Results show that the hybrid memory hierarchy achieves speedups of up to 16% and reduces power consumption and network traffic by up to 22% and 31%, respectively.

The rest of this paper is organized as follows. Section II introduces the required background in task-based programming models and their suitability for SPMs. Section III explains how SPMs are added in the memory hierarchy to form a hybrid memory hierarchy. Section IV describes the proposed techniques to map task dependences to the SPMs, and Section V evaluates the proposals. Section VI describes the related work, and Section VII concludes this work.

II. Background and Motivation

This section describes the main characteristics of task-based programming models together with the opportunities that they offer for hybrid memory hierarchies.

A. Task-Based Programming Models

Task-based data-flow programming models conceive the execution of a parallel program as a set of tasks with dependences among them. Typically, the programmer adds code annotations to split the serial code in *tasks* and specify what data is used by each task (called *input dependences* or *inputs*) and what data is produced (called *output dependences* or *outputs*). The runtime system is in charge of managing the execution of the tasks, releasing the programmer from the burden of explicitly synchronizing tasks and scheduling them to cores, thus easing programmability.

In order to manage the execution of the tasks the runtime system constructs a *task dependence graph* (TDG), a directed acyclic graph where the nodes are tasks and the edges are dependences between them. Similarly to how an out-of-order processor schedules instructions, the runtime system schedules a task on a core when all its input dependences are ready and, when the execution of the task finishes, its output dependences become ready for the next tasks. This execution model decouples the hardware from the application, so many optimizations can be applied at the runtime system level in a generic and application-agnostic way. For instance, the task scheduler can not only ensure load balancing, but also aim for a power-aware or locality-aware schedule [8]. Another very important characteristic of the task-based paradigm is that the runtime system knows what data is going to be accessed by the tasks that have to be executed, enabling multiple optimizations like data prefetching [9] or efficient data communication between tasks [10], [11]. In this context, this paper is the first one that uses the information available in the runtime system of a task-based programming model to exploit the benefits that SPMs provide in terms of performance, power consumption and network traffic without affecting programmability.

B. Suitability

Task-based data-flow programming models are specially well suited for SPMs. The specification of the input and output dependences for the tasks provide the runtime system with the information of what data is going to be accessed, which allows to map the tasks input and output dependences to the SPMs. As a consequence, memory accesses to inputs and outputs will always access the SPMs during the execution of tasks. Figure 1 shows the distribution of memory accesses for a set of representative benchmarks [1]. The figure shows, for each benchmark, the percentage of loads and stores that access data specified in task dependences (Dep

[1]The experimental setup is explained in detail in Section V.

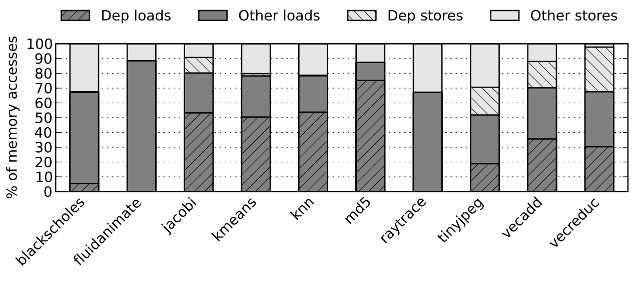

Figure 1. Percentage of memory accesses to tasks dependences (Dep loads and Dep stores) and to other memory locations (Other loads and Other stores)

loads and stores) and the percentage of loads and stores that access other memory locations (Other loads and stores). The benchmarks present a wide range of percentages of memory accesses to task dependences, from 0% in `raytrace` and `fluidanimate` to 76% in `md5`. On average, close to half of the memory accesses are to task inputs and outputs, so a significant amount of memory accesses can be efficiently served by the SPMs. In particular, compared to cache accesses, memory accesses to the SPMs do not suffer performance penalties in the form of cache misses, they consume less power because they do not trigger lookups in the tags of the caches nor in the TLBs, and they do not generate coherence traffic.

The memory model of task-based data-flow programming models is another very important factor for the suitability of SPMs. The memory model of task-based programming models guarantees that, during the execution of a task, its inputs will not be modified by another task and its outputs will not be accessed by another task. This property effectively eliminates the data races to the input and output dependences, so there is no need to maintain coherence for this data during the execution of a task. This allows that the data specified in the task dependences can be safely mapped to the SPMs during the execution of a task without requiring any costly synchronization mechanism in these non-coherent memories.

Additionally, the execution model found in task-based programming models offers the possibility to hide the communication costs of DMA transfers. First, the runtime system can perform scheduling decisions to exploit data locality, aiming to reduce data motion by assigning tasks to a core that already has the dependences mapped to its SPM. Second, when data locality cannot be exploited, the runtime system can trigger the DMA transfers for the task dependences before the task is executed, so the communication is overlapped with other execution phases such as the task scheduling phase or the execution of the previous task.

C. Suitability of Other Programming Models

Besides purely task-based models, other programming models designed for heterogeneous architectures are good

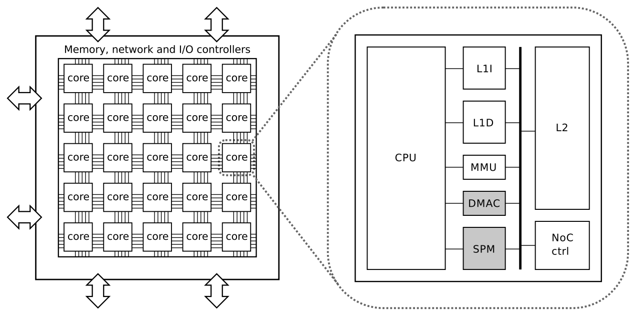

Figure 2. Multicore architecture with the hybrid memory hierarchy. Each core is augmented with a scratchpad memory (SPM) and a DMA controller (DMAC).

candidates to transparently manage hybrid memory hierarchies. Offload programming models for accelerators like OpenACC [12] also use source code annotations and clauses that allow to specify what data has to be copied from the host CPU memory to the accelerator memory, they expose similar memory models in terms of the privateness of the data during the execution of the kernels and they also use a runtime system to orchestrate the data transfers and kernel executions. Thanks to these properties, the code annotations can also be exploited in these models to map data to the SPMs of a hybrid memory hierarchy. Moreover, opportunities to hide the cost of the data transfers are also found in offload models, like in OpenAcc, that supports clauses to allow asynchronous data transfers and to specify at which point the execution should wait for all the asynchronous data transfers to be completed.

Although this paper focuses on how to automatically manage the hybrid memory hierarchy from the runtime system of task-based programming models, the proposed ideas can be easily adapted to other parallel programming models with similar characteristics, so the contributions of this paper are applicable to a wide range of programming models and applications.

III. Baseline Architecture

This section explains the baseline architecture assumed in this paper, a multicore architecture with a hybrid memory hierarchy. The hybrid memory hierarchy consists of extending every core with a SPM and a DMA controller (DMAC), as shown in Figure 2.

Every core is extended with a SPM that is added alongside the L1 D-cache. All the cores can access any SPM by issuing memory instructions to their address spaces. As shown in Figure 3, a range of the virtual address space is reserved for each SPM of the chip, that is direct-mapped to the physical address space of each SPMs. Every core uses eight registers to keep the address mappings for the SPMs, four to store the starting and the final virtual addresses of the local SPM and of the global range of the SPMs, and four to keep the

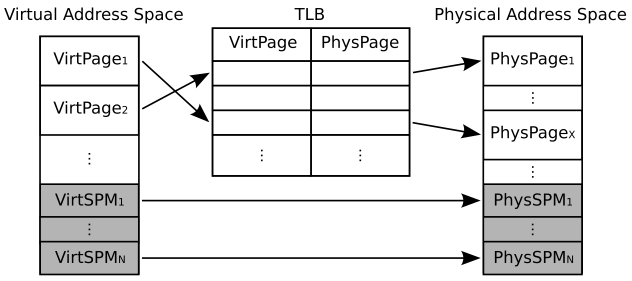

Figure 3. Address space mapping for the SPMs.

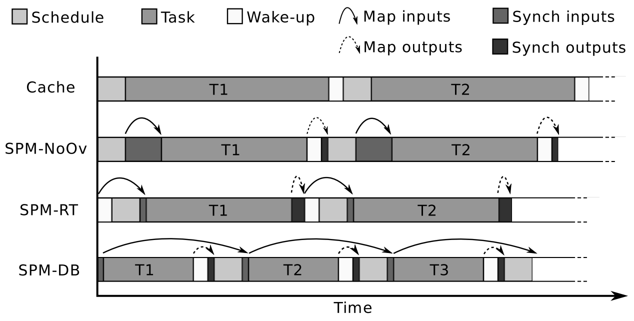

Figure 4. Timeline of a task-based application using cache-only and hybrid memory hierarchies with different overlapping techniques for the DMA transfers: no overlapping (SPM-NoOv), overlapping with runtime activity (SPM-RT), and double buffering with other tasks (SPM-DB).

physical address space of all the SPMs and of the local SPM. These registers are used to identify memory instructions that access the virtual address space of the SPMs and to do the virtual-to-physical address translation, allowing all the cores to access any SPM by issuing loads and stores to their virtual address ranges. When a memory instruction is executed, before any Memory Management Unit (MMU) action takes place, a range check is performed on the virtual address. If the virtual address is in the range reserved for some SPM, the MMU is bypassed and the registers are used to translate the virtual address to a physical address that points to the appropriate SPM.

This way of integrating the SPMs [13], [14], [15], [16] allows to access them without using pagination, so that memory accesses to the SPMs do not need to lookup the TLB, minimizing the power consumption and ensuring deterministic latency. Additionally, the size of SPMs is usually orders of magnitude smaller than the size of the RAM and of the virtual address space of a 64-bit processor, so the address ranges reserved for the SPMs occupy a very minor portion of the virtual and physical address spaces.

The DMACs are in charge of transferring data between the SPMs and the global memory (GM, which includes caches and main memory). They support three operations: (1) *dma-get* transfers data from the GM to a SPM, (2) *dma-put* transfers data from a SPM to the GM and (3) *dma-synch* waits for the completion of certain DMA transfers. Every DMAC exposes a set of memory-mapped I/O registers to the software so it can explicitly trigger the DMA operations. When a DMA transfer is triggered by the software, the DMAC forms a DMA command and stores it in a queue. The DMAC splits the DMA command in bus requests of the same size as the cache lines and issues them one by one to the memory subsystem. The bus requests issued by the DMACs are integrated in the cache coherence protocol of the GM. The bus requests generated by a *dma-get* look for the data in the caches and read the value from there if it exists, otherwise they read it from the main memory. The bus requests of a *dma-put* copy the data from the SPM to the main memory and invalidate the corresponding cache line in the whole cache hierarchy.

IV. Transparent Management of SPMs in Task-Based Runtime Systems

The goal of the runtime system is to transparently manage the SPMs of the hybrid memory hierarchy. This section describes what data structures are added in the runtime system and how they are operated to map task dependences to the SPMs. In addition, it is explained how the runtime can perform optimizations such as overlapping of DMA transfers with computation and locality-aware scheduling.

A. Mapping Data Dependences to the SPMs

The typical behaviour of a thread in a task-based program is an iterative process that consists of requesting a task to the scheduler, executing the task and waking up its dependent tasks. This behaviour with a cache-only memory hierarhcy is shown in Figure 4 in the timeline labeled as `Cache`.

In the scheduling phase the thread running on a core requests a new task to the scheduler. The scheduler selects a task from the ready queue based on a certain policy[2], removes the task from the ready queue and passes its associated task descriptor to the requesting thread. The task descriptor includes information about the task such as a pointer to the function that encapsulates the code or the addresses of the dependences, that are passed to the function as parameters when the task is executed. When the task finishes, the scheduler wakes up its dependent tasks. The scheduler locates in the TDG the node that represents the task that has just finished and, for every out-going edge representing an output dependence, marks as ready the in-going edge of the neighbour node, which represents an input dependence of a dependent task. When an input dependence of a task is marked as ready the scheduler checks if all the other input dependences of the task are also ready, so it can be woken up. In the `Cache` behaviour the scheduler wakes up ready tasks by inserting them in the ready queue.

[2]The default policy is First-In First-Out (FIFO), but Section IV-C presents other policies aware of data locality.

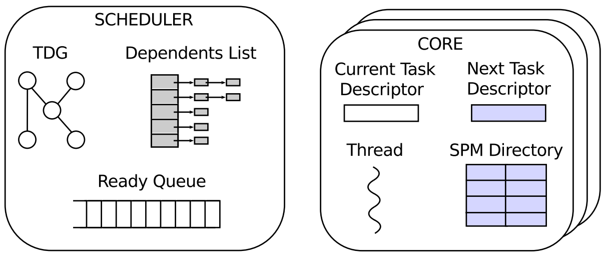

Figure 5. Extensions in the runtime system (shaded in gray) to support hybrid memory hierarchies.

For the hybrid memory hierarchy, four phases are added to this execution model to map the task dependences to the SPM of the core. The phases are *map inputs* and *synchronize inputs* before the execution of a task and *map outputs* and *synchronize outputs* after the execution of a task. In addition, several data structures are added in the runtime system to operate in these phases. Figure 5 shows the extensions in the runtime system, where added data structures are shaded in gray. Apart from the described ready queue and TDG, the scheduler requires a *Dependents List* to perform data locality-aware schedulings. Section IV-C further describes this extension. Next, each core abstraction in the runtime system has an associated thread that is pinned to a physical hardware thread, and a task descriptor of the currently executing task. A new per-core data structure, the *SPM directory*, is added to the runtime system to manage the mapping of inputs and output to the SPM. The SPM directory keeps, for every dependence mapped to the SPM of the core, the base address of the copy of the data in the SPM. Finally, a *Next Task Descriptor* is required to perform double buffering of DMA transfers with task execution, as described in Section IV-B.

Figure 4 shows the behaviour of a task-based workload on the hybrid memory hierarchy with different policies for the DMA transfers. In the timeline `SPM-NoOv` DMA transfers are not overlapped. In the map inputs phase, once a task has been scheduled on a core, the task dependences are mapped to the SPM of the core. First, for each entry in the SPM directory of the core, it is checked if the mapping matches any dependence of the task. If there is no match the SPM directory entry is erased and the space in the SPM is freed while, if a match is found, the task dependence is marked as already mapped. Then, for every task dependence that is not already mapped to the SPM, the necessary space is allocated for it in the SPM, the new mapping is recorded in the SPM directory and a DMA transfer is issued to copy the data to the SPM. Note that the data for an output dependence is also brought to the SPM because, if only some parts of the chunk of data are modified, the write-back at the end of the task execution will update the copy of the data in GM with wrong values. Once all the dependences are mapped to the SPMs the pointers that are passed to the task for the inputs and outputs are changed, substituting the original pointers in the task descriptor for the pointers to the data in the SPM.

In the synchronize inputs phase, just before the task starts executing, the thread waits for the DMA transfers of the task dependences to finish. When these DMA transfers have finished the thread jumps to the code of the task to start its execution. During the execution of the task the new pointers to the SPM mappings ensure that memory operations to the inputs and outputs access the address space of the SPM, so that this memory serves the accesses.

At the end of the task execution the map outputs phase takes place. In this phase the thread consults the SPM directory and, for each output dependence of the task, a DMA transfer is triggered to write back the results to the GM. Note that, even in the case that the output dependence is going to be reused as input by the following task executed on the core, the DMA transfer to write back the data to the GM is still done because other tasks that also reuse the output dependence as input may be executed on other cores.

While the data of the output dependences is written back, the scheduler wakes up the tasks that depend on these dependences using the TDG. The main difference with the already explained behavior is that new ready tasks are kept apart from the ready queue until the write-back DMA transfers finish. Finally, in the synchronize outputs phase, the thread synchronizes with the write-back DMA transfers for the output dependences of the task that has just been executed. When the DMA transfers have finished, the scheduler finally inserts the tasks that were woken up in the previous phase in the ready queue. Then the thread repeats the whole process to execute the next task.

B. Overlapping DMA Transfers with Computation

The DMA transfers triggered by the runtime system to manage the SPMs may impose high overheads in the synchronization phases if they are not overlapped with any computation phase. This section explains two mechanisms to reduce the impact of the communication cost of the data transfers for the SPMs. The solutions consist of overlapping the DMA transfers with the scheduling phase, denoted `SPM-RT`, and double buffering with the execution of the previous task, denoted `SPM-DB`.

For both approaches the runtime system needs to assign two tasks per core instead of one. For this purpose a new element is added in the core, the *Next Task Descriptor*, which keeps the task that is going to be executed by the core after the *Current Task Descriptor*.

The first approach consists of overlapping the DMA transfers with the task wakeup and scheduling phases. This behaviour is shown in Figure 4 in the timeline labeled as `SPM-RT`, which shows how the phases for the execution of two tasks T1 (the current task) and T2 (the next task) are interleaved. Task T1 starts by copying its input dependences to the SPM of the core in the map inputs phase. While the data for T1 is being transferred using DMA transfers the wakeup phase of the previous task T0 takes place, which marks as ready the tasks that depend on T0. Then the thread requests a task to be executed after T1 to the scheduler, which assigns task T2 to the core. This task T2 is kept in the next task descriptor field of the core abstraction in the runtime system. Note that, since the map inputs phase of T1 has already happened, the mappings for T1 are already present in the SPM directory of the core when the next task T2 is requested, so the locality-aware scheduler explained in the next section takes into account the data mapped by T1 although it has not been yet executed. Once T2 has been scheduled as the next task on the core, the synchronization with the inputs of T1 takes place and the task is executed normally. Just after the task T1 finishes its execution, its outputs are written back to GM in the map outputs phase and the thread waits for the write-back to finish in the synchronize outputs phase. At this point the runtime system triggers the DMA transfers of the map inputs phase of the next task T2, so they are overlapped with the wake-up phase of T1 and the scheduling phase of the task that is going to be executed after T2.

The second approach is a double buffering technique that overlaps the DMA transfers for a task with the execution of the previous task. The timeline labeled as `SPM-DB` in Figure 4 shows this behaviour for the execution of two tasks, the current task T1 and the next task T2. The timeline shows that the succession of phases is the same as in the `SPM-NoOV` behaviour, with the only difference that the map input phases are not for the task that is about to be executed but for the following one. The timeline starts with the map inputs phase of the next task T2. While the dependences for T2 are being transferred to the SPM, the thread waits for the inputs of the current task T1 (its map inputs phase is not shown because it happened before the execution of the previous task), executes the task, copies the outputs, calls the scheduler to wake up its dependent tasks and to schedule a new task T3 and synchronizes with the output DMA transfers. Then, this process is repeated for T2, which gets executed while the inputs of T3 are transferred, and for the subsequent tasks.

C. Locality-Aware Scheduling

The task scheduler is a fundamental part of a task-based runtime system. As explained in the previous section, when a thread wants to execute a task it first requests a new task to the scheduler, which selects one of the available ready tasks according to a certain policy. The locality-aware scheduler selects tasks for execution aiming to minimize the amount of data that has to be moved in the memory hierarchy. This scheduler can be used to minimize the number of DMA transfers for the SPMs of the hybrid memory hierarchy and also to improve data locality in traditional cache hierarchies.

The locality-aware task scheduler uses an additional data structure, the *dependents list*, as shown in Figure 5. The dependents list tracks, for a given dependence, what are the ready tasks that depend on it. This data structure is used by the locality-aware scheduler to quickly identify tasks in the ready queue that depend on a given dependence, avoiding a traversal of the ready queue.

The dependents list is updated every time a task is inserted or erased from the ready queue. When the scheduler inserts a task in the ready queue it checks, for all the dependences of the task, if they are present in the dependents list. If the dependence is found the task is inserted in the list associated to that entry, otherwise a new entry for the dependence is created along with an empty list, in which the task is then inserted. When the scheduler assigns a task to a core it removes the task from the ready queue and traverses, for each dependence of the task, its associated dependents list to remove the task from the list.

When a core requests a new task to the scheduler this selects from the ready queue the task that already has more data mapped to the SPM of the core. In order to do this the SPM directory of the core is traversed and, for every dependence already mapped to the SPM, its dependents list is accessed to obtain a list of ready tasks that reuse the dependence as an input. The scheduler selects from this list the ready task that has more data mapped to the SPM to be executed on the core. If the data present in the SPM is not a dependence of any ready task the scheduler selects the task at the head of the ready queue.

D. Discussion

The ideas proposed in this paper allow the runtime system to map task dependences to the SPMs of the hybrid memory hierarchy. As a first approach, in this paper it is assumed that the size of the task dependences is always smaller than the available space in the SPMs for them. Under this assumption it is the programmer who has to ensure that the data for the task dependences fits in the SPMs so, when the application is divided in tasks, this restriction has to be taken into account. To allow programmers to taskify their codes without this restriction several solutions can be applied at the runtime system level. A straightforward solution would be to discard mapping the data for the dependences that do not fit in the SPM, so they are served by the cache hierarchy. Since this solution may end up underutilizing the SPMs, some other approaches could be studied, such as performing automatic task coarsening in the runtime system to fuse or split tasks according to the available space in the SPM, or including a lightweight user-level pagination mechanism for the SPMs so parts of the input and output dependences are mapped and unmapped on demand.

The two proposed overlapping techniques have different trade-offs. On the one hand, the execution of a task is usually longer than only the wake-up and scheduling phases, so the

Table I
MAIN SIMULATOR PARAMETERS.

Cores	32 cores, Out-of-order, 6 instructions wide, 2GHz
Pipeline front end	13 cycles. Branch predictor 4K selector, 4K G-share, 4K Bimodal. 4-way BTB 4K entries. RAS 32 entries
Execution	ROB 160 entries. IQ 64 entries. LQ/SQ 64/48 entries. 3 INT ALU, 3 FP ALU, 3 Ld/St units. 256/256 INT/FP RegFile. Full bypass.
L1 I-cache	2 cycles, 32 KB, 4-way, pseudoLRU
L1 D-cache	2 cycles, 32 KB, 4-way, pseudoLRU, stride prefetcher
L2 cache	Shared unified NUCA sliced 256KB/core 15 cycles, 16-way, pseudoLRU
Cache coherence	Real MOESI with blocking states, 64B line size distributed 4-way cache directory 64K entries
NoC	Mesh, link 1 cycle, router 1 cycle
SPM	2 cycles, 32 KB, 64B blocks
DMAC	DMA command queue 32 entries, in-order Bus request queue 512 entries, in-order

double buffering with the previous task has more time to overlap the DMA transfers. On the other hand, doing double buffer with the previous task imposes that the available space in the SPM has to be shared by two tasks. Due to this restriction, and depending on how the application is split in tasks, more tasks may be needed to perform the same amount of work, which can incur in higher runtime system overheads [17], [18], [19].

Finally, task-based programming models themselves have some limitations. Data structures with pointers and indirections are hard to handle by task programming models, specially in those where dependences are statically declared using pragmas. In addition, in shared memory multicores it is not strictly necessary to specify all the data produced and consumed by the tasks as dependences, so programmers some times only specify the minimum amount of dependences that ensure the execution is correct, or introduce additional variables to synchronize tasks manually. This kind of bad programming practices can also cause an underutilization of the SPMs in some cases.

V. EVALUATION

This section evaluates the hybrid memory hierarchy with the runtime system techniques to manage the SPMs.

A. Experimental Setup

Gem5 [20] has been used to evaluate the proposal. The architecture is simulated in full system mode, using the cycle-accurate detailed out-of-order core model with a x86 ISA and the detailed memory hierarchy model (Ruby). McPAT [21] has been used to evaluate the power consumption, using a process technology of 22nm and the default clock gating scheme. The SPMs and the DMACs for the hybrid memory hierarchy are added in both simulators. Table I shows the main parameters of the simulated architecture. For fairness, the L1 data cache of the cache-only hierarchy is augmented to 64KB without affecting access latency, matching the 32KB L1 data cache plus the 32KB SPM of the hybrid memory hierarchy.

The simulated system is a Gentoo Linux with a kernel 2.6.28-4. The runtime system for the task-based programming model is Nanos++ [22] version 0.7a, which natively supports the OpenMP 4.0 [23] task constructs. The runtime system has been extended to manage the SPMs with the policies explained in Section IV.

Several representative HPC kernels together with parallel benchmarks from the PARSEC suite [24] have been used in the evaluation. As shown in Figure 1, the evaluated benchmarks have a wide range of percentages of memory accesses to inputs and outputs, from 0% to 76%. The benchmarks are a Jacobi method (`jacobi`), a k-means clustering algorithm (`kmeans`), a k-nearest neighbors algorithm (`knn`), an MD5 hashing algorithm (`md5`), an image raytracing and rotating application (`raytrace`), a decoding of JPEG images with fixed encoding of 2x2 MCU size and YUV color (`tinyjpeg`), and a one-dimensional vector addition and reduction (`vecadd` and `vecreduc`, respectively). From the PARSEC benchmark suite [24], `blackscholes` calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation (PDE) and `fluidanimate` uses the Smoothed Particle Hydrodynamics (SPH) method to simulate an incompressible fluid for interactive animation purposes. Simlarge input sets are used for the PARSEC benchmarks.

B. Performance Evaluation

Figure 6 shows the normalized execution time of the hybrid memory hierarchy with respect to a cache-only memory hierarchy on a 32-core multicore. The execution time of the three proposed data transfer strategies (SPM-NoOv, SPM-RT and SPM-DB) are evaluated and, moreover, an ideal configuration (SPM-Ideal) where DMA transfers occur instantaneously is also shown for comparison purposes. All results are normalized against the cache configuration, so values below 1 represent reduction in execution time. This figure further distinguishes how the execution time is distributed between phases: execution of tasks (`Task`), synchronization with DMA transfers (`Sync`), which includes the synchronize inputs and outputs phases, and runtime (`Runtime`), that includes the wake-up, scheduling and map inputs and outputs phases.

It can be observed that the task execution phases are accelerated in all benchmarks except `fluidanimate`, `raytrace` and `tinyjpeg`, achieving an speedup of up to 22% (`md5`). This happens because task dependences are served by the SPMs in the hybrid memory hierarchy, so performance penalties due to cache misses are minimized. When the cache hierarchy presents close to 100% hit ratio in the L1 D-cache (`tinyjpeg`) no performance improvements are observed in the execution of tasks. In benchmarks that do not map data to the SPMs (`fluidanimate` and

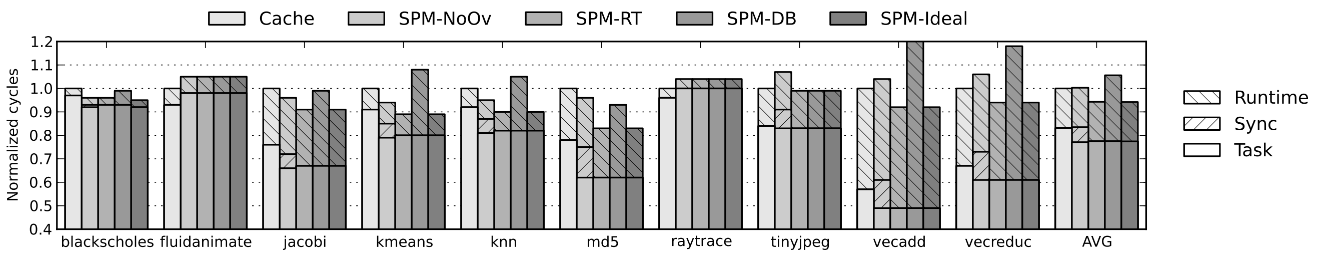

Figure 6. Reduction of execution time with respect to a cache-only memory hierarchy for different data transfer strategies: no overlapping (SPM-NoOv), overlapping with runtime activity (SPM-RT), double buffering with other tasks (SPM-DB) and ideal (SPM-Ideal)

`raytrace`) the performance in the task execution phases decreases because of the augmented L1 D-cache in the cache-only baseline. These performance improvements in the task execution phases allow the hybrid memory hierarchy to achieve up to 5% speedup if DMA transfers are not overlapped. In this SPM-NoOv approach, when big amounts of data are mapped to the SPMs, the synchronization time adds overheads of up to 11% (`tinyjpeg`), limiting the performance of the hybrid memory hierarchy. On average, the cache-only and the hybrid memory hierarchy offer the same perfomance if DMA transfers are not overlapped. SPM-RT shows that, by overlapping DMA transfers with runtime activity, speedups of up to 16% (`md5`) are obtained, resulting in an average speedup of 6% in all benchmarks. In the SMP-RT approach the time spent in the synchronization phases becomes negligible in all cases, so the performance is very close to the one of the ideal configuration. For SPM-DB, that uses double buffering to overlap the DMA transfers with the execution of the previous task, the synchronization time also becomes negligible but the number of executed tasks increases together with the runtime overhead in some cases. As a consequence, the time spent in runtime phases increases significantly in some benchmarks (`jacobi`, `kmeans`, `vecadd` and `vecreduc`) and negates the performance benefits of using the SPMs. In other benchmarks (`knn` and `md5`) the double buffering does not cause a big increase of the runtime overhead, resulting in speedups of 11% and 6%, respectively.

C. Power Consumption Evaluation

Figure 7 shows the reduction in power consumption of the hybrid memory hierarchy with respect to the cache-only memory hierarchy. All results are normalized to the power consumption of the cache-only hierarchy, so values below 1 represent a reduction in power consumption. All the data mapping techniques (SPM-NoOv, SPM-RT and SPM-DB) present similar results, so only one bar is shown for the hybrid memory hierarchy. The figure also shows how the power consumption is distributed among different components: cores (CPU), L1 caches, L2 caches, prefetchers, MSHRs and cache directories (Cache), the SPMs and DMA controllers (SPMs + DMACs), and the network-on-chip and

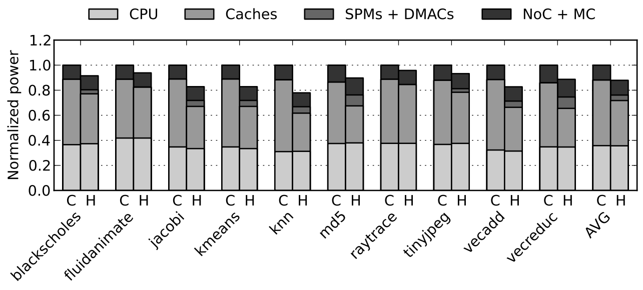

Figure 7. Reduction of power consumption of the hybrid memory hierarchy (H) with respect to a cache-only memory hierarchy (C)

the memory controller (NoC + MC). Results show that the power consumed by the CPUs is nearly the same in both systems. It can be observed a reduction of the power consumed in the caches, with savings of up to 47% in benchmarks that map a significant portion of accesses to the SPMs (`jacobi`, `kmeans`, `knn` and `md5`). The big power savings in these components happen because, in the hybrid memory hierarchy, many memory accesses are served by the SPMs instead of the cache hierarchy. SPMs are able to serve these memory accesses in a much more power-efficient way, contributing with less than 10% of the total power consumed for all benchmarks. In all cases, the overall power consumption in the components of the memory hierarchy (Caches and SPM+DMACs) is lower on the hybrid memory hierarchy than in the cache-based hierarchy, resulting in an average reduction in power consumption of 13%.

The speedup in Energy Delay Product (EDP) of the hybrid memory hierarchy with respect to the cache-only hierarchy is shown in Figure 8. Speedups in EDP are achieved in almost all configurations, with average improvements of 14%, 29% and 5% for SPM-NoOv, SPM-RT and SPM-DB, respectively. These improvements are particularly significant in the benchmarks that map more data to the SPMs, achieving up to 65% improvement in EDP in `knn` with the SPM-RT configuration. Some benchmarks present slowdowns in EDP caused by the performance overheads in the runtime system in the SPM-DB configuration and by the synchronization time spent in SPM-NoOV configurations.

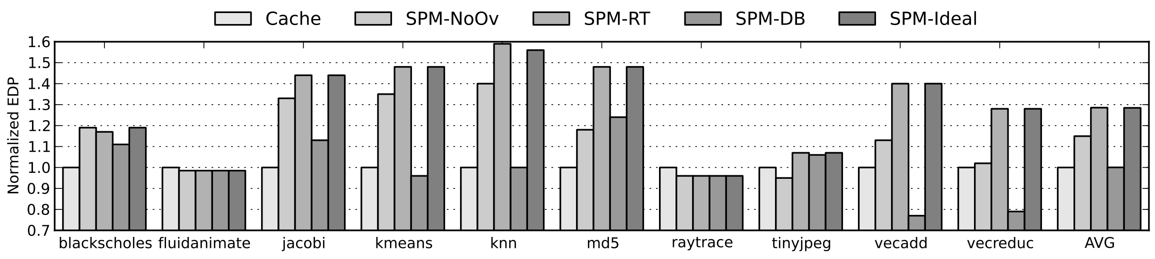

Figure 8. Speedup in EDP with respect to a cache-only memory hierarchy for different data transfer strategies: no overlapping (SPM-NoOv), overlapping with runtime activity (SPM-RT), double buffering with other tasks (SPM-DB) and ideal (SPM-Ideal)

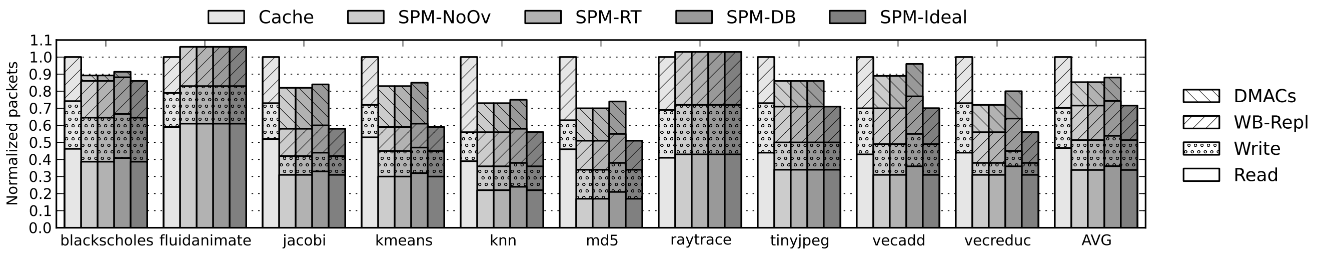

Figure 9. Reduction of NoC traffic with respect to a cache-only memory hierarchy for different data transfer strategies: no overlapping (SPM-NoOv), overlapping with runtime activity (SPM-RT), double buffering with other tasks (SPM-DB) and ideal (SPM-Ideal)

D. NoC Traffic Evaluation

Another important benefit of hybrid memory hierarchies is the reduction of the interconnection Network on-Chip (NoC) traffic. Figure 9 presents the reduction in NoC traffic with respect to the cache-only hierarchy. Each bar shows the percentage of traffic originated by different actions: cache reads and writes (which include packets for data requests, prefetch requests, data and acknowledgements), write-back and replacement of cache lines (Wb-Repl, which includes packets for write-back requests, replacements, invalidations, data and acknowledgements), and DMA transfers (which include packets for DMA requests, data and acknowledgements). Results show that the hybrid memory hierarchy, for all configurations, reduces the Noc traffic related to cache reads, writes and WB-Repl significantly. This reduction is directly proportional to the percentage of mapped accesses to the SPMs, reaching a maximum of 62% reduction of read traffic in `md5` as most of the loads access task dependences (as shown in Figure 1). Similarly, NoC traffic originated by cache writes is reduced if output dependences are mapped to the SPMs, achieving savings of up to 39% in `jacobi` in this category, although the average reduction is 18% as the portion of writes mapped to the SPMs is smaller than in the case of loads. The reduced activity in the caches also reduces cache misses, replacements and invalidations, so the traffic in the WB-Repl group is reduced between 17% (`blackschoes`) and 59% (`md5`). In the hybrid memory hierarchy all this NoC traffic is saved thanks to the introduction of SPMs, that needs DMA transfers to move the data. The NoC traffic generated by DMA transfers to move the task dependences contributes with less than 30% of the original traffic in all cases, and never overweights the traffic saved in the other categories. Consequently, an average reduction in NoC traffic of 15% is obtained with the hybrid memory hierarchy.

E. Mitigating the Effects of Fine-Grained Tasks

It has been shown that the runtime system overheads can degrade performance when fine-grained tasks are required. This is an important factor for the hybrid memory hierarchy, as the size of the SPMs determines the task granularity.

One way to alleviate the runtime system overheads is to increase the size of the SPMs. Figure 10 shows the average reduction in execution time of all the benchmarks with different SPMs sizes for the proposed data transfer strategies, and each bar also shows the time distribution among program phases. Four SPM sizes are studied: 32, 64, 128 and 256 KB with access times of 2, 3, 4 and 6 cycles, respectively. The ROB is augmented to 192 entries in the experiments with 256 KB SPMs to tolerate the latency.

Results show that, for all the data transfer strategies, the average execution time of all the benchmarks decreases as the size of the SPMs increase. It can be observed that the size of the SPMs has a big impact in the runtime phases, that represent more than 15% of the total execution time with 32 KB SPMs and is reduced to less than 10% with 256 KB SPMs. This happens because bigger SPMs allow to use coarser grain tasks in 6 of the 10 benchmarks, so less tasks are needed to perform the computation and the runtime overhead is lower. In the rest of benchmarks the task

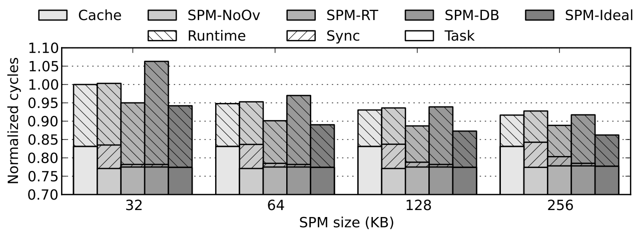

Figure 10. Average reduction of execution time with respect to a cache-only memory hierarchy for different SPM sizes and data transfer strategies: no overlapping (SPM-NoOv), double buffering with other tasks (SPM-DB) and ideal (SPM-Ideal)

granularity is fixed by the way the benchmark is decomposed in tasks, so having bigger SPM sizes does not decrease the runtime overhead. Another effect of augmenting the size of the SPMs is that the synchronization time increases in the SPM-RT approach for some benchmarks due to the reduced length of the runtime phases and the bigger DMA transfers. This causes that, for SPM-RT, the percentage of time spent in synchronization phases goes from less than 1% in all benchmarks with SPMs of 32 KB to an average 3% with SPMs of 256 KB, reaching up to 8% in `md5`. It can also be observed that the size of the SPMs has a negligible effect on the execution time of the tasks, as the additional latency of bigger SPMs can be hidden with the execution of other instructions. All together, increasing the size of the SPMs from 32 KB to 256 KB provides average execution time reductions of more than 7% in all cases.

Another solution to mitigate the runtime system overheads is to add hardware support for the runtime system [17], [18], [19]. These solutions report speedups of 2 to 3 orders of magnitude for the runtime system phases, eliminating the overheads caused by fine-grained tasks.

Figure 11 shows an estimation of the performance of the hybrid memory hierarchy when combined with this hardware support. To estimate the performance the execution time of the runtime system phases is accelerated by a factor of 100x. The SPM-RT configuration is not considered in the study because the runtime phases are too short to hide the cost of the DMA transfers. It can be observed that the results for SPM-NoOv are very similar to the ones presented in Figure 6, as the impact in performance of the hardware runtime system is equal for both the baseline cache-only memory hierarchy and the SPM-NoOv configuration. In contrast, the hardware runtime system completely eliminates the runtime overhead introduced by the bigger amount of tasks in the SPM-DB approach, and provides close to ideal performance because DMA transfers are completely overlapped with the execution of the tasks. On average, an speedup of 8% is achieved against a cache-only memory hierarchy, reaching up to 22% for `md5`. These estimated results indicate that SPM-DB is the appropriate solution for future multicores with hardware support for the runtime system.

VI. Related Work

A. Data-Flow and Task-Based Models

Task scheduling was first studied in the context of tiled architectures like Raw [25] and stream architectures like Imagine [26] or Merrimac [27]. These architectures used compiler techniques [28], [29] to schedule tasks to cores, transfer data and overlap communication with computation. The drawback of these systems was that their programming languages were either too complex for programmers or either relatively simple but required of complex compiler heuristics to ensure load balancing and real-time requirements.

Several task-based programming models have emerged in the last years. OpenMP 3.0 [30] supports basic tasking constructs, that are extended with data dependences in OpenMP 4.0 [23]. Cilk [31] is a fork-join model enhanced with work-stealing primitives to improve load balancing. OmpSs [22] is a data-flow programming model that extends OpenMP 4.0 with additional features like task priorities or special tasking constructs. The Codelets programming model [32] breaks applications into tasks with dependences but, unlike in OpenMP 4.0, the programmer needs to explicitly specify the particular codelet each dependence is associated with. Intel TBB [33] is a C++ template library that implements a task-based execution model where the programmer splits the serial code into tasks with data dependences. In Legion [34] programs are decomposed in tasks that access data partitions manually specified by the programmer, while in Sequoia [35] tasks have their private address space and it is the programmer who organizes them hierarchically. Charm++ [36] is a C++ based asynchronous message driven programming model, while the Habanero [37] project proposes a programming model, a compiler and a runtime system for asynchronous task-based parallelism.

The ideas proposed in this paper apply to runtime-managed task-based programming models that specify data dependences between tasks, either using the real addresses of the data or some abstraction from which the runtime system can extract the addresses, like in OpenMP 4.0, OmpSs, Codelets, Charm++ and Habanero. In task-based programming models that do not specify data dependences, like Cilk or Intel TBB, the runtime does not have the information of what data is going to be accessed, so transparently managing SPMs with the proposed ideas is unfeasible.

B. SPM Management in Hybrid Memory Hierarchies

Several forms of hybrid memory hierarchies have been proposed in the past. Although the architecture details are similar, the solutions manage the SPMs in different ways.

Static mapping schemes allocate data in the SPMs at the beginning of the execution and the contents of the SPMs do not change during the computation. This model is used in the embedded domain, where the compiler identifies data for the SPMs [38], and in NVIDIA GPUs [4], where the

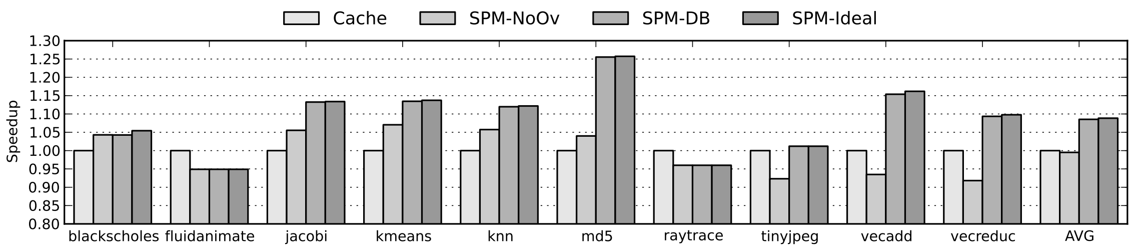

Figure 11. Speedup of the hybrid memory hierarchy with a hardware runtime system with respect to a cache-only memory hierarchy for different data transfer strategies: no overlapping (SPM-NoOv), double buffering with other tasks (SPM-DB) and ideal (SPM-Ideal)

programmer uses keywords provided by CUDA to declare what data is allocated in the SPMs.

Some works propose hybrid memory hierarchies where the data is dynamically mapped to the SPMs. In these approaches the data accessed in a loop is moved to the SPMs in a blocking fashion.Virtual Local Stores [16] partition parts of the cache as SPMs and the programmer writes code to map data to the SPMs. Bertran et al. [15] add a SPM alongside the L1 cache of a single core processor and give the compiler the responsibility to generate code to manage the SPM, and Alvarez et al. [13], [14] propose a hardware/software coherence protocol that allows the compiler to generate code to manage the SPMs even in the presence of unknown memory aliases. In [39] the programmer maps data to the SPMs statically or dynamically and a memory manager decides what SPMs addresses are used for the mappings.

Compared to these works, this paper uses the information found in task-based programming models to manage the SPMs transparently to the programmer and without any hardware support. In addition, this paper shows that optimizations to overlap DMA transfers with computation and to improve data locality can be performed at the runtime system level without imposing further programmability difficulties.

Task-based programming models have also been proposed for architectures that combine caches and SPMs with different approaches to the one used in this paper. Architectures such as SARC [40] or Runnemede [41] are designed to be programmed with these programming models, and OmpSs [42] is supported on the Cell processor [3].

C. Runtime-Aware Architectures

A new trend in computer architecture is to rethink the design of multicores being aware of the runtime system that manages the available architectural resources [6], [7].

For task-based programming models, some works propose to add hardware support to accelerate functionalities of the runtime system such as the construction of the TDG [17] or the scheduling decisions [18], [19], minimizing the runtime system overhead for fine-grained tasks. Some proposals use the information of the task dependences to optimize the memory hierarchy. The runtime system can do software-guided prefetching [9] to the desired level of the cache hierarchy, and also lock and flush cache lines to improve the efficiency of the technique. Data communication in producer-consumer relationships can also be efficiently done by the runtime system with the adequate hardware support [10], and simplified coherence protocols guided by the runtime system can be used to reduce coherence traffic [11].

Other programming models also offer the possibility to optimize parts of the architecture. DeNovo [43] exploits the data-race-freedom of disciplined programming models to eliminate the transient states of the cache coherence protocols, but requires additional hardware support for synchronization primitives [44]. Totoni et al. [45] propose a runtime-guided mechanism to switch off cache banks using formal language theory to detect application phases.

VII. Conclusions

This paper proposes to manage the SPMs of multicore processors with hybrid memory hierarchies in the runtime system of task-based programming models, transparently to the programmer. Task-based data-flow programming models are very well suited for SPMs, since the task dependences specify what data is going to be accessed during the execution of the tasks, the programming model ensures that no data races will happen on the data dependences during the execution of the tasks, and the DMA transfers can be overlapped with different phases of the execution model. These properties allow the runtime system to exploit the information of what data is going to be accessed by the tasks, mapping the task dependences to the SPM of the core where each task is going to be executed and applying optimizations like locality-aware scheduling and overlapping of DMA transfers with computation.

Results show that the hybrid memory hierarchy outperforms cache-only hierarchies by up to 16% when DMA transfers are overlapped with the task scheduler, it consumes up to 22% less power, and reduces NoC traffic by up to 31%. When DMA transfers are not overlapped with useful work the performance benefits reach up to 5%, and double buffering with the previous task increases the runtime system overheads so it is better suited for architectures with hardware runtime systems or SPMs of hundreds of kilobytes.

Acknowledgment

This work has been supported by the Spanish Government (grant SEV-2011-00067 of the Severo Ochoa Program), by the Spanish Ministry of Science and Innovation (contract TIN2012-34557), by Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272), and by the RoMoL ERC Advanced Grant (GA 321253). Miquel Moretó has been partially supported by the Ministry of Economy and Competitiveness under Juan de la Cierva postdoctoral fellowship number JCI-2012-15047, Marc Casas is supported by the Secretary for Universities and Research of the Ministry of Economy and Knowledge of the Government of Catalonia and the Cofund programme of the Marie Curie Actions of the 7th R&D Framework Programme of the European Union (Contract 2013 BP_B 00243), and Emilio Castillo is partially supported by the Spanish Ministry of Education, Culture and Sports under grant FPU2012/2254.

References

[1] J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis, "Comparing Memory Systems for Chip Multiprocessors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. ACM, 2007, pp. 358–368.

[2] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad Memory: Design Alternative for Cache On-chip Memory in Embedded Systems," in *Proceedings of the 10th International Symposium on Hardware/Software Codesign*, ser. CODES '02. ACM, 2002, pp. 73–78.

[3] J. Kahle, "The Cell Processor Architecture," in *Proceedings of the 38th Annual International Symposium on Microarchitecture*, ser. MICRO 38. IEEE Computer Society, 2005, p. 3.

[4] P. N. Glaskowsky, "NVIDIA's Fermi: The First Complete GPU Computing Architecture," 2009.

[5] R. Hazra, "Accelerating Insights in the Technical Computing Transformation," in *ISC keynote*, 2014.

[6] M. Valero, M. Moretó, M. Casas, E. Ayguadé, and J. Labarta, "Runtime-Aware Architectures: A First Approach," *International Journal on Supercomputing Frontiers and Innovations*, vol. 1, no. 1, pp. 29–44, Jun. 2014.

[7] M. Casas, M. Moretó, L. Alvarez, E. Castillo, D. Chasapis, T. Hayes *et al.*, "Runtime-Aware Architectures," in *Euro-Par 2015: Parallel Processing*. Springer Berlin Heidelberg, 2015, pp. 16–27.

[8] J. Bueno, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta, "Implementing OmpSs Support for Regions of Data in Architectures with Multiple Address Spaces," in *Proceedings of the 27th International Conference on Supercomputing*, ser. ICS '13. ACM, 2013, pp. 359–368.

[9] V. Papaefstathiou, M. G. Katevenis, D. S. Nikolopoulos, and D. Pnevmatikatos, "Prefetching and Cache Management Using Task Lifetimes," in *Proceedings of the 27th International Conference on Supercomputing*, ser. ICS '13. ACM, 2013, pp. 325–334.

[10] M. Manivannan, A. Negi, and P. Stenström, "Efficient Forwarding of Producer-Consumer Data in Task-Based Programs," in *Proceedings of the 42nd International Conference on Parallel Processing*, ser. ICPP '13. IEEE Computer Society, 2013, pp. 517–522.

[11] M. Manivannan and P. Stenstrom, "Runtime-Guided Cache Coherence Optimizations in Multi-core Architectures," in *Proceedings of the 28th International Parallel and Distributed Processing Symposium*, ser. IPDPS '14. IEEE Computer Society, 2014, pp. 625–636.

[12] "The OpenACC Application Program Interface. Version 1.0," 2011.

[13] L. Alvarez, L. Vilanova, M. Gonzàlez, X. Martorell, N. Navarro, and E. Ayguadé, "Hardware-software Coherence Protocol for the Coexistence of Caches and Local Memories," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. IEEE Computer Society, 2012, pp. 89:1–89:11.

[14] L. Alvarez, L. Vilanova, M. Moretó, M. Casas, M. Gonzàlez, X. Martorell *et al.*, "Coherence Protocol for Transparent Management of Scratchpad Memories in Shared Memory Manycore Architectures," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. ACM, 2015, pp. 720–732.

[15] R. Bertran, M. Gonzàlez, X. Martorell, N. Navarro, and E. Ayguadé, "Local Memory Design Space Exploration for High-Performance Computing," *IEEE Computer Journal*, vol. 54, no. 5, pp. 786–799, May 2011.

[16] H. Cook, K. Asanovic, and D. A. Patterson, "Virtual Local Stores: Enabling Software-Managed Memory Hierarchies in Mainstream Computing Environments," University of California at Berkeley, Tech. Rep. UCB/EECS-2009-131, 2009.

[17] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguadé *et al.*, "Task Superscalar: An Out-of-Order Task Pipeline," in *Proceedings of the 43rd Annual International Symposium on Microarchitecture*, ser. MICRO '43. IEEE Computer Society, 2010, pp. 89–100.

[18] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural Support for Fine-grained Parallelism on Chip Multiprocessors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. ACM, 2007, pp. 162–173.

[19] D. Sanchez, R. M. Yoo, and C. Kozyrakis, "Flexible Architectural Support for Fine-grain Scheduling," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '10. ACM, 2010, pp. 311–322.

[20] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu *et al.*, "The Gem5 Simulator," *SIGARCH Computer Architure News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[21] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *Proceedings of the 42nd Annual International Symposium on Microarchitecture*, ser. MICRO 42. ACM, 2009, pp. 469–480.

[22] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell *et al.*, "OmpSs: A Proposal for Programming Heterogeneous Multi-core Architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[23] "OpenMP Application Program Interface. Version 4.0," 2013.

[24] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. ACM, 2008, pp. 72–81.

[25] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee *et al.*, "Baring It All to Software: Raw Machines," *IEEE Computer Journal*, vol. 30, no. 9, pp. 86–93, Sep. 1997.

[26] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens *et al.*, "Imagine: Media Processing with Streams," *IEEE Micro*, vol. 21, no. 2, pp. 35–46, Mar. 2001.

[27] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju *et al.*, "Merrimac: Supercomputing with Streams," in *Proceedings of the 2003 Conference on Supercomputing*, ser. SC '03. ACM, 2003, pp. 35–42.

[28] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal, "Maps: A Compiler-managed Memory System for Raw Machines," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ser. ISCA '99. IEEE Computer Society, 1999, pp. 4–15.

[29] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *Proceedings of the 11th International Conference on Compiler Construction*, ser. CC '02. Springer Berlin Heidelberg, 2002, pp. 179–196.

[30] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli *et al.*, "The Design of OpenMP Tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 404–418, Mar. 2009.

[31] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," in *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '95. ACM, 1995, pp. 207–216.

[32] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, "Using a "Codelet" Program Execution Model for Exascale Machines: Position Paper," in *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, ser. EXADAPT '11. ACM, 2011, pp. 64–69.

[33] J. Reinders, *Intel Threading Building Blocks*. O'Reilly Media, 2007.

[34] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing Locality and Independence with Logical Regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. IEEE Computer Society, 2012, pp. 66:1–66:11.

[35] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park *et al.*, "Sequoia: Programming the Memory Hierarchy," in *Proceedings of the 2006 Conference on Supercomputing*, ser. SC '06. ACM, 2006, pp. 83:1–83:11.

[36] L. V. Kale and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of the 8th Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '93. ACM, 1993, pp. 91–108.

[37] J. Shirako, J. M. Zhao, V. K. Nandivada, and V. N. Sarkar, "Chunking Parallel Loops in the Presence of Synchronization," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09. ACM, 2009, pp. 181–192.

[38] O. Zendra, E. Jul, and M. Cebulla, "Survey of Scratch-Pad Memory Management Techniques for low-power and -energy," in *Proceedings of the 2nd ECOOP Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ser. ICOOOLPS '07. Springer Berlin Heidelberg, 2007, pp. 31–38.

[39] P. Francesco, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. M. Mendias, "An Integrated Hardware/Software Approach for Run-time Scratchpad Management," in *Proceedings of the 41st Annual Design Automation Conference*, ser. DAC '04. ACM, 2004, pp. 238–243.

[40] A. Ramirez, F. Cabarcas, B. Juurlink, M. Alvarez Mesa, F. Sanchez, A. Azevedo *et al.*, "The SARC Architecture," *IEEE Micro*, vol. 30, no. 5, pp. 16–29, Sep. 2010.

[41] N. P. Carter, A. Agrawal, S. Borkar, R. Cledat, H. David, D. Dunning *et al.*, "Runnemede: An Architecture for Ubiquitous High-Performance Computing," in *Proceedings of the 19th International Symposium on High Performance Computer Architecture*, ser. HPCA '13. IEEE Computer Society, 2013, pp. 198–209.

[42] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: A Programming Model for the Cell BE Architecture," in *Proceedings of the 2006 Conference on Supercomputing*, ser. SC '06. ACM, 2006.

[43] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve *et al.*, "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism," in *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '11. IEEE Computer Society, 2011, pp. 155–166.

[44] H. Sung, R. Komuravelli, and S. V. Adve, "DeNovoND: Efficient Hardware Support for Disciplined Non-determinism," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. ACM, 2013, pp. 13–26.

[45] E. Totoni, J. Torrellas, and L. V. Kale, "Using an Adaptive HPC Runtime System to Reconfigure the Cache Hierarchy," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. IEEE Computer Society, 2014, pp. 1047–1058.