Architectural Support for Task Dependence Management with Flexible Software Scheduling

ABSTRACT

The growing complexity of multi-core architectures has motivated a wide range of software mechanisms to improve the orchestration of parallel executions. Task parallelism has become a very attractive approach thanks to its programmability, portability and potential for optimizations. However, with the expected increase in core counts, fine-grained tasking is required to exploit the available parallelism, which increases the overheads introduced by the runtime system.

This work presents Task Dependence Manager (TDM), a hardware/software co-designed mechanism to mitigate runtime system overheads. TDM introduces a hardware unit, denoted Dependendence Management Unit (DMU), and minimal ISA extensions that allow the runtime system to offload costly dependence tracking operations to the DMU and to still perform task scheduling in software. With lower hardware cost, TDM outperforms hardware-based solutions and enhances the flexibility, adaptability and composability of the system. Results show that TDM improves performance by 12.3% and reduces EDP by 20.4% on average with respect to a software runtime system. Compared to a runtime system fully implemented in hardware, TDM achieves an average speedup of 4.2% with 7.3× less area requirements and significant EDP reductions. In addition, five different software schedulers are evaluated with TDM, illustrating the flexibility and performance gains of our approach.

1. INTRODUCTION

The end of Dennard scaling [1] and the subsequent stagnation of the CPU clock frequency has caused a dramatic increase in the core counts of multi-cores [2]. To fully exploit these large core counts in an efficient way, the hardware and the software stack must collaborate to avoid important performance problems such as load imbalance or memory bandwidth exhaustion, while improving energy efficiency.

The growing complexity of multi-cores has brought sophisticated software mechanisms aiming at optimally managing parallel workloads. One of the most extended approaches is task-based programming models, such as OpenMP 4.0 [3], that apply a data-flow execution model to orchestrate the execution of the parallel tasks respecting their control and data dependences. These programming models are a very appealing solution to program complex multi-cores due to their benefits in performance, programmability, cross-platform flexibility, and potential for applying generic optimizations at the runtime system level [4–9].

A key aspect of this execution model is the granularity of the tasks. Fine-grain parallelism exposes large degrees of concurrency to the hardware, which favors load balancing and provides more flexibility to exploit constructive interference on shared resources. However, it can also bring large software overheads due to the runtime system activity, which involves creating the tasks, tracking the dependences between them, and scheduling them to cores. All these actions require synchronizing threads to perform complex operations on internal data structures of the runtime system.

Different solutions have been proposed to support fine-grain parallelism on multi-cores [10–14]. These approaches manage fine-grained tasks completely in hardware, relying on specific execution models to scale to large core counts. However, pure hardware solutions suffer from limited adaptability to changes in the software layers. Tasking support in shared memory programming models is continuously evolving, incorporating new features such as dependence domains or task nesting that are not easy to support at the architecture level. Moreover, implementing a fixed scheduling policy in hardware reduces the adaptability to different application and system characteristics.

Using different task scheduling policies is key to maximize the efficiency of applications and systems [15]. Considering task criticality [16–18] or data locality [13] provides significant benefits in performance and energy in certain contexts. In addition, the adaptability granted by software task schedulers is essential in modern high-performance computing systems with off-chip accelerators and multi-socket configurations, that can further improve performance and energy efficiency but require software intervention for task scheduling and data motion.

This paper presents Task Dependence Manager (TDM), a hardware/software co-designed mechanism that accelerates the most time consuming activities of the runtime system with specialized hardware while allowing flexible task scheduling policies in software. TDM minimally extends the ISA to allow the runtime system to communicate task creation, task dependences and task finalization, and to request ready tasks. At the architecture level, TDM introduces a Dependence Management Unit (DMU) that maintains the information of the in-flight tasks and the dependences between them by means of a set of tables and lists. Tasks ready for execution are exposed to the runtime system, which has the freedom for deploying any software scheduling policy. The main contributions of this paper are:

- A novel hardware/software co-designed mechanism to accelerate task creation and dependence tracking while supporting flexible software schedulers. The hardware design includes novel architectural techniques to minimize conflicts in associative structures and to reduce the hardware cost with respect to previous proposals.
- A detailed evaluation of TDM on a full-system simulator that includes application, runtime system, operating system and architecture layers. On a 32-core processor, TDM achieves a 12.3% average speedup and a 20.4% reduction in energy-delay product (EDP) with respect to a baseline implemented in software.
- A proof of the potential of TDM when combined with five software schedulers that exploit the characteristics of different applications. Thanks to this flexibility, TDM outperforms a runtime fully implemented in hardware by an average 4.2%, improves EDP by an average 6.2% and reduces the area overhead by 7.3×.

This paper is organized as follows. Section 2 introduces the required background in task-based programming models. Section 3 describes TDM, including the interface between the runtime system and the architecture, the hardware support and the operational model. Section 4 explains the experimental methodology, while Section 5 performs a design space exploration of the proposal. Section 6 evaluates TDM using different software schedulers and compares TDM to other proposals. Section 7 describes the related work and Section 8 draws the main conclusions of this work.

2. BACKGROUND AND MOTIVATION

2.1 Task-based Programming Models

Task-based data-flow programming models such as OpenMP 4.0 [3] conceive the execution of a parallel program as a set of *tasks* with dependences among them. Typically, the programmer writes sequential code and adds annotations to define the tasks of the program and to specify which data are used by each task (called *input dependences* or simply *inputs*), and which data are produced by each task (called *output dependences* or simply *outputs*). With this information, the runtime system manages the parallel execution by means of a *Task Dependence Graph (TDG)*, a directed acyclic graph where the nodes are tasks and the edges are dependences between them. Figure 1 shows the task-based implementation of a Cholesky factorization benchmark and its corresponding TDG. The code uses OpenMP 4.0 clauses to specify tasks and dependences (*#pragma omp task depend(in/out/inout)*).

Task-based data-flow programming models implement a decoupled execution model where tasks are created in program order and are executed asynchronously following the synchronization rules defined by the dependences. All threads may execute runtime system activity as well as tasks defined in the application source code. Figure 1 shows the execution timeline of the Cholesky benchmark on an 8-core system. In this experiment, core 1 performs most of the runtime system activities while the other cores mainly execute tasks.

The *master thread* executes the program sequentially and, when it encounters a task creation statement, it enters the *task creation* phase. In this phase the new task is assigned a unique *task descriptor* that stores all the relevant information

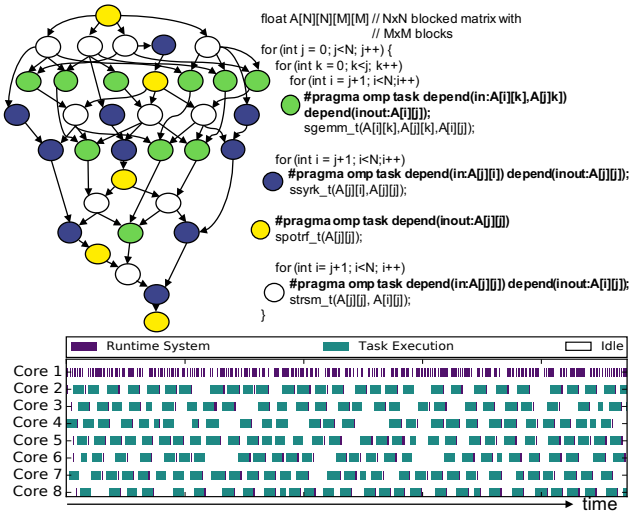

Figure 1: Cholesky task-based annotated code (right), generated task dependence graph (left), and execution timeline on an 8-core system (bottom).

of the task such as its dependences, its number of successors and a pointer to the function to be executed. The address of this task descriptor is used to identify the task. To detect dependences with older tasks, the inputs and outputs of the new task are compared against the inputs and outputs of the older tasks. The new task is marked as a successor of older tasks if a RAW, WAR or WAW dependence is found, and is inserted in the TDG accordingly.

The remaining *worker threads* iterate on the two main phases of the task-based data-flow execution model. The master thread also adopts this behavior when it reaches a global synchronization point.

- In the *task scheduling* phase the thread selects a task to be executed. The runtime system keeps a pool of ready tasks and selects one of them based on a scheduling algorithm. Different scheduling policies may adapt better to the characteristics of an application, and can provide significant benefits in certain contexts [16–18].
- In the *task execution* phase the thread executes the code of the task that has been just scheduled. After the task is executed, the thread notifies the runtime system that the task has finished. The outputs of this task become available and its successor tasks may become ready if all its dependences are satisfied. In such case, it is added to the pool of ready tasks and will be selected for execution in future scheduling phases.

Apart from executing these phases, threads can experience *idle time*. In parallel regions idle time occurs when a thread enters the task scheduling phase and the pool of ready tasks is empty. This happens if the pace at which tasks are created is lower than the pace at which tasks are executed, or when threads reach a barrier and there are no tasks left to be executed. In addition, idle time happens in sequential parts of the program, where only one thread executes the sequential code and the other threads are waiting.

2.2 Characterizing Runtime System Activity

Performance and scalability of parallel programs is funda-

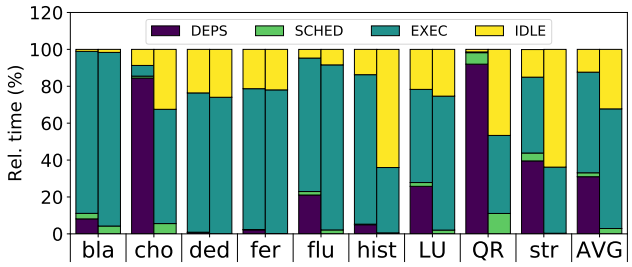

Figure 2: Execution time breakdown of the master and worker threads during the parallel execution. Different states represent dependence management operations during task creation and task finalization (DEPS), scheduling (SCHED), task execution (EXEC), and idle time (IDLE).

mentally limited by the overheads introduced in the form of idle time and runtime system phases to manage tasks and dependences [19]. These two sources of overheads are tightly related to the granularity of the tasks. On the one hand, coarse-grain tasking reduces the overheads of task creation and dependence management, but compromises load balancing and scalability on large-scale multi-cores. On the other hand, fine-grain tasking favors load balancing, but increases the overheads of the runtime system in dependence management and task scheduling phases. In addition, many operations in the runtime system phases need to be serialized to avoid race conditions, potentially becoming a bottleneck as concurrency increases with higher core counts [7].

We characterize the cost of the runtime system phases in 9 representative task-based parallel benchmarks running on a simulated 32-core processor. The optimal task granularity in each experiment has been carefully selected to minimize execution time[1]. Figure 2 shows a complete breakdown of the time spent in the main program phases for the master (left bars) and the worker threads (right bars): task creation and dependence management (DEPS), scheduling (SCHED), task execution (EXEC), and idle time (IDLE).

The master thread spends a significant portion of the time in DEPS for Cholesky, QR and streamcluster (84%, 92% and 40%, respectively). In these cases, illustrated in the timeline of Figure 1 for Cholesky, the bottleneck of the execution is the pace at which tasks are created by the master thread, that limits the amount of available tasks for the worker threads and causes idle time. DEPS has a lower impact in the rest of benchmarks, below 25.8%, but idle time is still relevant in the worker threads due to load imbalance. Most of the time spent in DEPS is devoted to identify the dependences of a task when it is created, which requires comparing the inputs and outputs of the new task against the ones of the older tasks. Thread synchronization overheads are negligible, as they only represent 0.9% of the DEPS time and 2.2% of the SCHED time. Overall, worker threads spend most of the time executing tasks (65% of the time on average) or idle (32% of the time), and the master thread spends a significant amount of time running tasks in the majority of benchmarks, while scheduling time is much less significant.

Introducing hardware support to accelerate the runtime system phases is a good solution to mitigate the overheads of fine-grained tasking. Approaches such as Carbon [10] propose to move the task scheduler to the hardware level, while Task Superscalar [11] offloads all the runtime system activities to the architecture level, including dependence management and task scheduling. An important drawback of these schemes is that the task scheduler is fixed in the architecture, which compromises the flexibility of the system. The system flexibility provided by software runtime systems is of paramount importance in modern computing infrastructures with multiple sockets and off-chip accelerators, since the task scheduler needs to off-load tasks to external components that are only visible to the software and often require software-initiated actions such as data movement between address spaces. In order to maintain these advantages, approaches such as ADM [15] propose architectural support for asynchronous exchanges of short messages between cores that can be used to implement low-overhead thread synchronization primitives within task scheduling phases.

All these solutions drastically reduce runtime system overheads, even in scenarios with extremely fine-grained tasks running on multi-cores with hundreds of cores. However, in scenarios where tasks are not extremely fine-grained[2], the cost of task scheduling is relatively low, less than 11% in all benchmarks in Figure 2, so the benefits of flexible software scheduling can be achieved with minimal performance impact. In contrast, the cost of dependence management operations during task creation is crucial for performance because it determines the idle time in the whole execution, so adding hardware support to perform this operation can effectively reduce the runtime system overheads.

This work addresses the performance bottlenecks of pure software data-flow runtime systems by proposing TDM, a hardware/software co-designed mechanism that performs dependence management operations efficiently in hardware and allows the usage of different task scheduling policies in the runtime system. Thanks to this separation of concerns, TDM is able to mitigate the performance overheads introduced in runtime system phases while providing flexibility to the software layers, so the resulting system is more adaptable, composable, and is able to capitalize on the benefits of different scheduling policies for different applications.

3. TDM DESIGN

TDM is a hardware/software co-designed mechanism to support the runtime system. TDM balances the higher cost and performance of implementing mechanisms in hardware, with the higher flexibility and adaptability of implementing policies in software. At the architecture level TDM introduces a *Dependence Management Unit (DMU)* that keeps a representation of the TDG and allows the runtime system to offload costly dependence tracking operations, while leaving scheduling decisions to the runtime system. As a result, TDM avoids the overheads of software runtime systems and maintains the flexibility of supporting software schedulers.

The runtime system interacts with the DMU in task management phases to communicate task creation, data dependences of the task, and task finalization. With this information, the DMU generates the TDG, tracks dependences

[1]Section 4 describes in detail the experimental setup, and Figure 6 explores the optimal task granularity of each benchmark.

[2]In this paper we use task granularities up to 3 orders of magnitude bigger than other works of the literature.

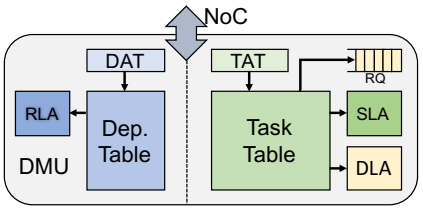

Figure 3: DMU architectural support overview.

between tasks, identifies tasks ready for execution, and exposes them to the runtime system. The runtime system can request ready tasks to the DMU, organize them in software data structures, and schedule them to the cores according to any scheduling policy.

3.1 Runtime System - Architecture Interface

TDM offers an interface to the runtime system so that it can cooperate with the DMU in the management of tasks. The interface between the DMU and the runtime system consists of four new ISA instructions. These instructions are issued by the runtime system in the task creation and task finalization phases to exchange information with the DMU.

- *create_task(task_desc)*: In the task creation phase, the runtime system uses this instruction to inform the DMU that a new task is being created. The DMU receives the task descriptor address of the new task.
- *add_dependence(task_desc, dep_addr, size, direction)*: After creating a task, the runtime system traverses its list of dependences and uses this instruction to inform the DMU of the dependences of the task, sending the task descriptor address, the address of the dependence, the size, and the direction (input or output). The DMU uses this information to track tasks and dependences and to build the TDG to ensure dependences between tasks are fulfilled.
- *finish_task(task_desc)*: When a task finishes its execution, the runtime system uses this instruction to notify it to the architecture. The DMU wakes up the successors of the task and cleans up the information of the task and its dependences from its internal structures.
- *get_ready_task()* $\rightarrow$ *task_desc, #succ*: Just after notifying a task has finished, the runtime system uses this instruction to request to the DMU the successors of the finished tasks that have just become ready. This instruction returns the task descriptor address and its number of successors.

3.2 DMU Hardware Design

The DMU is a centralized module connected to the network-on-chip whose main goal is to keep all the information of the in-flight tasks, track the dependences between them, and expose tasks that become ready to the runtime system. Figure 3 presents its different components. Each task or dependence is internally identified by an ID, which maps to its location in the corresponding table. Tables and list arrays employ direct-access SRAM, addressed by the task or dependence IDs. Two set-associative structures, TAT and DAT, are used to map task descriptor and dependence addresses to internal DMU IDs. The general behavior of each module follows:

- The *Task* and *Dependence Alias Tables* (TAT and DAT) keep a translation of task descriptor addresses or dependence addresses to internal task or dependence IDs.

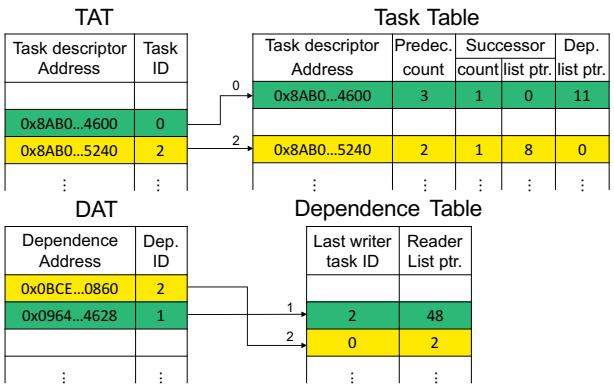

Figure 4: Overview of TAT, DAT, Task and Dependence Table. Two active elements are presented in each table.

- The *Task Table* and the *Dependence Table* track all the information of the in-flight tasks and dependences.
- The *List Arrays* (*Successor, Dependence* and *Reader*) contain lists of elements associated to in-flight tasks or dependences. The *successor* and *reader* lists store task IDs, while the *dependence* list stores dependence IDs.
- The *Ready Queue* (RQ) is a FIFO queue that contains task IDs ready to be executed.

3.2.1 Task and Dependence Identifier Renaming

The alias tables are depicted in Figure 4. Both TAT and DAT modules consist of a directory that maps task descriptor and dependence addresses to task and dependence IDs, respectively, and an additional queue of free IDs. Both modules are implemented using set-associative memories.

Selecting the correct bits to index the DAT is crucial to avoid conflicts. In task-parallel programs it is common that different tasks access different blocks of the same data structure, so the lower bits of the addresses of different dependences share the same values. For example, if tasks access different 4KB blocks of a vector, the lower 12 bits of all the dependences are equal. To avoid conflicts in the DAT, the size of the dependence is used to select the index bits, which start at the $log_2 size$ lower bit of the dependence address.

The alias tables allow the rest of DMU modules to work with internal IDs, which offers two important advantages. First, the Task and Dependence Tables employ direct access, indexed with the internal task and dependence IDs, avoiding costly associative lookups of 64-bit task descriptor and dependence addresses. Therefore, using TAT and DAT a single associative lookup is required per DMU instruction, followed by many subsequent direct accesses to the Task and Dependence Tables, as explained in Section 3.3. Second, the storage requirements of the list arrays can be reduced significantly, as the size of the internal IDs is much smaller than the 64-bit identifiers used in the runtime system. Our experiments in Section 5.1 show that DAT and TAT with 2048 entries suffice for any application, so 11-bit IDs can be used and the size of the list arrays is reduced by a factor of 5.8$\times$.

3.2.2 Task and Dependence Tracking

The Task and Dependence Tables are used to keep the information of the tasks and the dependences. The Task Table is an SRAM indexed by the Task ID. Figure 4 shows each

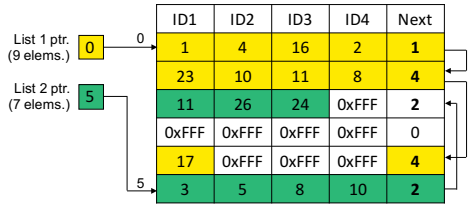

Figure 5: Overview of a generic list array.

Data: taskID, depID, dir
Insert depID in dependence list of taskID;
if *lastWriterID of depID is valid* **then**
  Insert taskID in successor list of lastWriterID;
  Increment #succ of lastWriterID;
  Increment #pred of taskID;
end
if *dir is In* **then**
  Insert taskID in reader list of depID;
end
if *dir is Out* **then**
  for *readerID in reader list of depID* **do**
    Insert taskID in successor list of readerID;
    Increment #succ of readerID;
    Increment #pred of taskID;
  end
  Flush reader list of depID;
  Set lastWriterID of depID to taskID and mark valid;
end

Algorithm 1: Algorithm for *add_dependence* instruction.

entry of the Task Table contains the relevant information of a task: its descriptor address, the number of successors and predecessors, and pointers to the lists of successors and dependences. The Dependence Table follows the same scheme to track dependences, storing the task ID of the last task that writes the dependence and a pointer to the list of readers.

The lists of successors, dependences and readers are implemented in three list array structures. As shown in Figure 5, each list array is an SRAM that can store multiple lists. To accommodate a variable number of elements in each list we use a storage layout inspired by UNIX filesystem inodes. The maximum number of elements in each entry is fixed by design (4 in the example), but the list can continue in other unused entries. Every entry contains a *Next* control field that points to the entry in the list array where the list continues. The *Next* field is set to the current entry number if the list finishes in this entry. Invalid elements are set to all ones.

The Successor List Array uses this organization to store the lists of successors of each in-flight task, identified by their task IDs. Task IDs are also stored in the lists of the Readers List Array, which track the reader tasks of all the in-flight dependences. The Dependence List Array keeps the lists of dependences of the in-flight tasks, so dependence IDs are stored in the lists. Note that OpenMP 4.0 uses the input/output dependences provided by the programmer to build the TDG when tasks are created in program order. The DMU preserves this model by decoupling the dependences, that are tracked in the dependence and readers lists, from the edges of the TDG, that are tracked in the successors lists.

3.3 Operational Model

The runtime system triggers DMU operations using the ISA instructions in the task creation and finalization phases.

Data: taskID
for *succID in successor list of taskID* **do**
  Decrement #pred of succID;
  if *#pred of succID = 0* **then**
    Insert succID in the Ready Queue;
  end
end
for *depID in dependence list of taskID* **do**
  Remove taskID from reader list of depID;
  if *lastWriterID of depID = taskID* **then**
    Mark lastWriterID of depID as invalid;
  end
  if *lastWriterID of depID is invalid && reader list of depID is empty* **then**
    Free reader list of depID;
    Free depID entry in DepTable and DAT;
  end
end
Free successor list of taskID;
Free dependence list of taskID;
Free taskID entry in TaskTable and TAT;

Algorithm 2: Algorithm for *finish_task* instruction.

3.3.1 Task Creation

The runtime system uses the *create_task* instruction to communicate the task descriptor address to the DMU. Then, for every dependence, it uses *add_dependence* to inform the DMU of the dependences of the task.

When the *create_task* instruction is executed, the DMU uses the TAT to generate a task ID. The Task Table is indexed with the task ID and the entry is initialized by setting the task descriptor address, setting to 0 the number of successors and predecessors, and reserving a new list of successors and a new list of dependences in the Successor and Dependence List Arrays. If some structure of the DMU has no entries available the instruction blocks until an entry is freed.

After the task is created, for every *add_dependence* instruction an entry is allocated in DAT and Dependence Table. The DMU uses TAT to obtain the task ID and DAT to obtain the dependence ID. Then, the DMU behaves as described in Algorithm 1. First the dependence is inserted in the list of dependences of the task and the task ID is inserted in the successor list of the last writer of the dependence. Then, if the dependence is an input, the task ID is inserted in the readers list of the dependence. Otherwise, if the dependence is an output, all the readers of the dependence insert the task in their successor lists,the reader list is flushed, and the task becomes the last writer of the dependence.

3.3.2 Task Finalization

When a task finishes, the runtime system uses the *finish_task* instruction to communicate the task descriptor address to the DMU, and this carries out the steps described in Algorithm 2. In the first loop the DMU wakes up the successor tasks. This is done by traversing the successor list of the task and decrementing the number of predecessors of each successor. If the number of predecessors becomes zero, the successor task is moved to the Ready Queue. In the second loop the task is removed from the reader list and the last writer field of each of its dependences. Finally the DMU frees the entries allocated for the task in the Task Table, the TAT, and the Successor and Dependence List Arrays.

3.3.3 Implementing Task Schedulers in Software

After the finalization of a task the runtime system requests ready tasks to the DMU by issuing *get_ready_task* instructions in a loop. For every *get_ready_task* instruction the DMU consults the Ready Queue. If it is empty, a null pointer is returned. Otherwise, the task ID at the head of the queue is retrieved and used to index the Task Table to get the task descriptor address and the number of successors that are returned to the runtime system. Then the runtime system adds the returned task descriptor address to a pool of ready tasks and stores the number of successors in the task descriptor.

The pool of ready tasks can be used by the runtime system to implement any scheduling policy. The scheduling algorithms can traverse the pool of ready tasks in any order, move ready tasks to different data structures, or perform any action required by each particular implementation. By allowing the usage of different task schedulers, TDM provides flexibility, adaptability and composability to the system.

3.4 Additional Considerations

The size of the hardware structures of the DMU limit the number of in-flight tasks and dependences. To preserve correctness, the proposed ISA instructions have barrier semantics, so they cannot be re-ordered in the CPUs and younger instructions cannot be executed before the TDM instructions commit. The DMU processes the instructions sequentially and, if there is no space available in some of the structures, the instruction is blocked until some in-flight task finishes.

TDM manages tasks and dependences inside parallel regions and relies on the runtime system to handle barriers and other global synchronization points. To do so, the master thread executes the code sequentially and creates the tasks while the worker threads request tasks and execute them. The runtime system tracks how many tasks have been created by the master thread and how many have been executed. When the master thread reaches the barrier it adopts the behavior of a worker thread, and when all the tasks have been executed it resumes the sequential execution of the program.

The proposed design of TDM can be easily extended to support context switches and multiprogrammed workloads. A simple and effective solution is to tag TAT and DAT with the operating system process ID, so different processes can use TDM concurrently and the structures of the DMU do not need to be saved and restored at context switch.

The centralized design of the DMU is not a limiting factor for scalability in our setup. The DMU executes several instructions per task that, all together, take 10s to 100s ns, while the average task duration in our experiments is 4771 μs. Given that the task duration is 5 orders of magnitude larger than the latency of the DMU instructions per task, The DMU should be able to scale up to thousands of concurrent tasks before becoming a bottleneck.

4. EXPERIMENTAL FRAMEWORK

4.1 Full-System Simulation Infrastructure

We employ gem5 [20] to simulate an ARM full-system environment that models the application, the runtime system, the operating system and the architecture in detail. We simulate a 32-core processor using the detailed out-of-order CPU and memory models of gem5, extended with the proposed architectural support for TDM. Table 1 summarizes the main simulation parameters, including the selected sizes of the TAT, DAT, Task Table, Dependence Table and the successor (SLA), dependence (DLA) and reader (RLA) list arrays. Section 5.1 performs a detailed design space exploration to justify the selected sizes. Note that the list arrays contain 8 elements per entry, and the latency of accessing an entry is 1 cycle. As explained in Section 3, TDM operations may require multiple accesses to the corresponding hardware structures, so they require multiple cycles to finalize. These latencies are modeled in detail in our simulator.

Table 1: Configuration of the gem5 full-system simulations.

Chip details	
Cores	32 Out-of-order cores, single threaded, 2.0GHz
Core details	
Fetch, issue, commit bandwidth	4 instr/cycle
Branch predictor	Tournament: 2K local pred., 8K global and choice pred. 4-way BTB 4K entries, RAS 16 entries
Issue queue	Unified 64 entries
Reorder buffer	128 entries
Register file	256 int, 256 FP
Functional units	INT: 4 ALU (1 cycle), 2 mult (3 c.), 2 div (20 c.) FP: 2 ALU (2 cycle), 2 mult (4 c.), 2 div (12 c.) 2 Ld/St unit (1 cycle)
Instr L1 cache	32KB, 2-way, 64B/line (2 cycles hit)
Data L1 cache	32KB, 2-way, 64B/line (2 cycles hit)
Shared L2 cache	4MB 16-way, 64B/line
Instruction TLB	256 entries fully-associative (1 cycle hit)
Data TLB	256 entries fully-associative (1 cycle hit)
DMU structures	
TAT DAT	2048 entries, 1 cycle per access, 8-way associative 2048 entries, 1 cycle per access, 8-way associative
Task Table, Dependence Table	2048 entries, 1 cycle per access 2048 entries, 1 cycle per access
SLA, DLA, RLA	1024 entries, 1 cycle per access, 8 elements/entry

The simulated system is a Ubuntu 14.04 with a kernel 4.3. We use the Nanos++ 0.10a [21] runtime system, which supports OpenMP 4.0 [3]. The runtime system is extended to communicate with the DMU using the instructions described in Section 3. The ISA is extended to support the new instructions and their execution is modeled in the architecture.

We also extend gem5 to model a hardware FIFO queue for task scheduling. This hardware structure is not required in TDM, but we make use of it to model Carbon [10]. Combining this hardware queue and the DMU we also model Task Superscalar [11], which tracks data dependences and schedules tasks in hardware. In the modeled Task Superscalar pipeline, renaming of data dependences is not performed as the evaluated benchmarks do not benefit from this feature.

Power consumption is evaluated with McPAT [22] using a process technology of 22 nm, a voltage of 0.6V, and the default clock gating scheme. We incorporate the changes suggested by Xi *et al.* [23] to improve the accuracy of the models. The hardware structures of the DMU are modeled using CACTI 6.0 [24], adding the appropriate counters in gem5 to measure the extra power introduced by the DMU.

4.2 Benchmarks and Task Granularity

To test TDM we use five benchmarks from PARSECSs [25], a task-based OpenMP 4.0 implementation of the PARSEC suite [26], together with four benchmarks from the high performance computing domain: Cholesky, Histogram, LU and QR. These benchmarks are representative algorithms and use different parallelization strategies: Blackscholes and Stream-

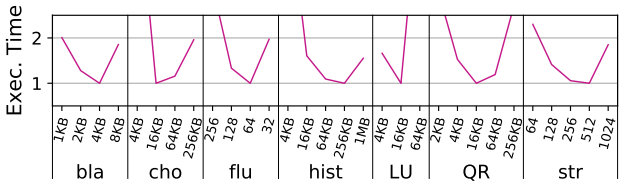

Figure 6: Execution time for different task granularities. The X axis shows the size of the blocks processed by each task in Blackscholes, Cholesky, Histogram, LU, and QR; the number of partitions of the 3D volume in Fluidanimate; and the number of points processed by each task in Streamcluster.

Table 2: Benchmark characteristics. Number of tasks and average task duration with the optimal task granularity for the software runtime system and for TDM.

	Software		TDM	
	# tasks	Duration (μs)	# tasks	Duration (μs)
Blackscholes	3,300	1,770	6,500	823
Cholesky	5,984	183	5,984	183
Dedup	244	27,748	244	27,748
Ferret	1,536	7,667	1,536	7,667
Fluidanimate	2,560	1,804	2,560	1,804
Histogram	512	3,824	512	3,824
LU	1,512	424	1,512	424
QR	1,496	997	11,440	96
Streamcluster	42,115	376	42,115	376
Average	**6,584**	**4,976**	**8,056**	**4,771**

cluster use fork-join parallelism, Fluidanimate is a 3D stencil, and Dedup and Ferret use pipeline parallelism. Regarding the other four benchmarks, Cholesky performs a Cholesky decomposition of a matrix, Histogram computes a cumulative histogram for all pixels of an image [27], LU does a LU decomposition of a matrix, and QR calculates a QR factorization of a matrix. Tiling is applied in these algorithms so that tasks process 2D blocks of the matrices.

The benchmarks are compiled with Mercurium 1.99 source-to-source compiler [28] with gcc 4.6.4 as backend compiler. *Simlarge* input sets are used for the PARSEC benchmarks, Cholesky decomposes a dense 2048×2048 matrix, histogram processes a 4096×4096 image and generates a histogram with 10 bins, LU decomposes a sparse 2048×2048 matrix, and QR a dense 1024×1024 matrix.

In all benchmarks we ensure that parallel regions scale well to 32 cores using performance analysis tools to visualize the parallel executions. The optimal task granularity is carefully selected to minimize load imbalance and execution time in the baseline software approach. Figure 6 shows the execution time with different task granularities growing along the X axis (i.e., smaller to bigger from left to right). Execution time is normalized to the optimal task granularity. In Dedup and Ferret the task granularity cannot be changed without modifying the application, as each task processes a pipeline stage. In general, shorter task duration increases parallelism, but leads to higher runtime system overheads.

Table 2 summarizes the number of tasks and their average duration for each benchmark. The number of tasks ranges from 244 (Dedup) to 42,115 (Streamcluster), and the average duration between 96μs (QR) and 27ms (Dedup). The optimal task granularity is used for the corresponding approach (software or TDM) in all the experiments of the evaluation.

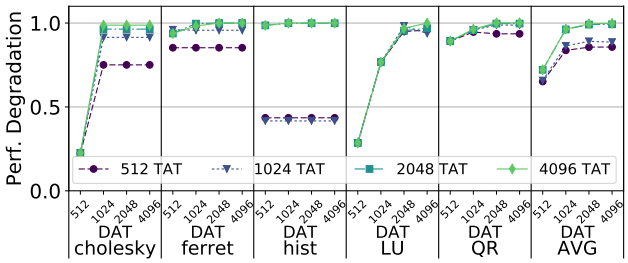

Figure 7: Average performance with different sizes of the TAT and DAT. Results are normalized to an ideal DMU with unlimited entries and equal latency.

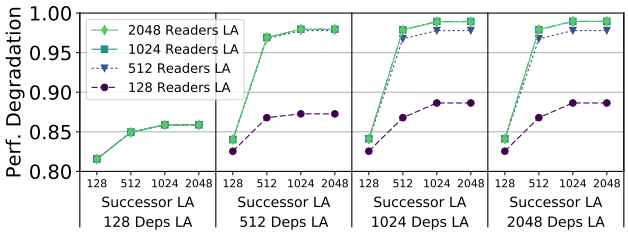

Figure 8: Average performance with different sizes of the list array (LA) structures. Results are normalized to an ideal DMU with unlimited entries and equal latency.

5. DESIGN SPACE EXPLORATION

5.1 TAT, DAT and List Arrays

Next, we perform a design space exploration to determine the required size of the DMU hardware structures. We first study the sizing of TAT and DAT. In this experiment we consider an unlimited number of entries in the three list arrays. We consider a DMU implementation with N TAT entries and M DAT entries. The size of the TAT determines the size of the Task Table, and the size of the DAT determines the size of the Dependence Table. Figure 7 shows the performance obtained when N and M vary between 512 and 4096. Performance is normalized to an ideal design with an infinite number of entries in all DMU structures and same latency.

Figure 7 shows results for 5 benchmarks. The rest of benchmarks already achieve maximum performance with 512 entries in DAT and TAT. The geometric mean considers all the benchmarks. The figure shows LU and QR are sensitive to the DAT size, achieving maximum performance with 2048 entries. The other three benchmarks are more sensitive to the TAT size. The most demanding benchmark is Histogram, as its tasks have a significant amount of dependences between them and the distance between independent tasks is high. Thus, it requires at least 2048 TAT entries to achieve maximum performance. On average, with 2048 entries in both DAT and TAT, the DMU only suffers a 0.91% performance degradation with respect to the ideal case with infinite entries and same latency. We have also explored the associativities of TAT and DAT, results show that 8-way associative structures minimize conflicts and offer the best performance.

After selecting the size of TAT, DAT, Task Table and Dependence Table, we perform an exploration for the successor, dependence and reader list arrays. Figure 8 shows the average performance when these structures vary from 128 to 2048 entries, normalized to an ideal design with an infinite number of entries in all DMU structures and same latency.

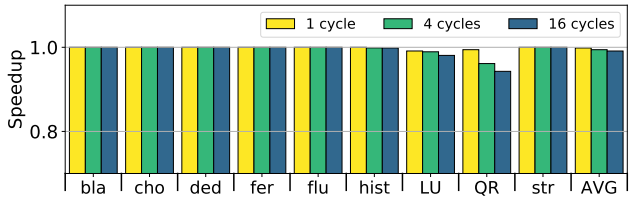

Figure 9: Performance degradation when varying the access time of all DMU structures from 1 to 16 cycles. Results are normalized to DMU structures with zero latency.

Table 3: DMU storage (KB) and area (mm^2) requirements.

	Storage	Area		Storage	Area
Task Table	23.00	0.026	SLA	12.25	0.019
Dep Table	5.25	0.013	DLA	12.25	0.019
TAT	18.75	0.031	RLA	12.25	0.019
DAT	18.75	0.031	ReadyQ	2.75	0.012
Total	**105.25 KB**	**0.17 mm^2**			

These results clearly show that a design with 128 entries in any of the list arrays leads to suboptimal performance. In contrast, with 1024 entries in all the list arrays, performance already saturates. On average, with 1024 entries in all list arrays, the DMU only suffers a 1.1% performance degradation with respect to the ideal case with an infinite number of entries and same latency. Doubling the size of all list arrays leads to an average 1.0% performance degradation, but requires a significant increase in area. For this reason, we size all list arrays in the DMU with 1024 entries.

5.2 DMU Access Latency

As explained in Section 3, the algorithms that implement TDM instructions require accessing different hardware structures. Also, the lists stored in the list arrays may spread over multiple entries, which requires multiple accesses to traverse the complete lists. Consequently, DMU operations require multiple cycles to finalize. Next, we evaluate the performance of the DMU when varying the latencies of its internal hardware structures. In these experiments we use the sizes of the DMU structures determined in the previous section.

Figure 9 shows the performance degradation of all benchmarks when increasing the access time of all DMU structures from 1 to 16 cycles. Most benchmarks do not suffer any performance degradation due to higher latencies, as with the optimal task granularity DMU operations happen infrequently. Only LU and QR are slightly affected by this parameter. On average, performance degrades only 0.2% with a 1-cycle access time and 0.9% with a 16-cycle access time.

5.3 DMU Area and Power Overhead

Table 3 shows the storage and area requirements of the DMU for the sizes selected in Section 5.1. Storage values consider the number of bits of the task and dependence IDs, which depend on the size of the tables they point to. The structures are modeled in CACTI 6.0 [24] to obtain the area values with a process technology of 22nm.

The components of the DMU have a negligible effect on the power consumption, less than 0.01% of the total power. The low power requirements of the DMU combined with the small sizes of the hardware structures allow to design the DMU with a 1-cycle access time to each data structure.

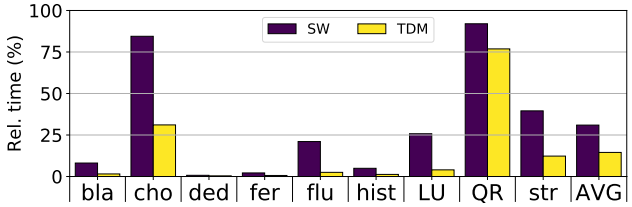

Figure 10: Percentage of time spent in task creation with a pure software approach (SW) and with TDM.

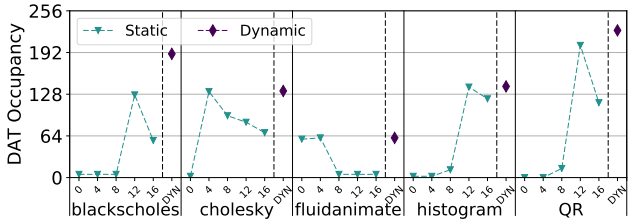

Figure 11: Occupancy of sets in DAT with static index bit selection and with dynamic index bit selection.

As a conclusion of this design space exploration, we select a design with a DAT and TAT of 2048 entries and all the list arrays of 1024 entries. The storage and area requirements for this configuration, 105.25KB and 0.17mm^2, are very affordable with current design technology. The rest of this paper makes use of this configuration in all the experiments.

5.4 Runtime Overhead Reduction

Next, we measure the impact of TDM in the task creation time. Figure 10 shows the average time spent by the master creating tasks and managing their dependences, which corresponds to the DEPS category in Figure 2. Task creation time is not completely eliminated with TDM because of the latency of the DMU structures and because some operations are still performed in the runtime system, such as creating task descriptors, issuing TDM instructions, etc. All benchmarks benefit from the hardware support provided by the DMU, being Blackscholes the one with the maximum reduction in task creation time (5.2× reduction). On average, task creation is reduced from from 31.0% to 14.5% of the total CPU time, proving the effectiveness of TDM. This reduction of task creation time has a big impact on the idle time, that is reduced from 32% to 22% on average, and translates into overall application speedups as will be shown in Section 6.

5.5 Index Bit Selection for DAT

We show the importance of selecting the appropriate bits of the dependence addresses to index the DAT. As described in Section 3.2.1, when different blocks of the same data structure are specified as dependences, many dependence addresses have the same values in the lower bits, causing conflicts if these bits are selected to index the DAT. To avoid this problem, the DMU uses the size of the dependence to select the bits of the dependence addresses to index the DAT.

Figure 11 shows the average number of occupied sets in the DAT for the six benchmarks that are sensitive to this situation. The X axis shows 5 numerical values that correspond to different options to statically select the index bits (e.g., 4 means the index bits start at the 4th lower bit of the dependence address), and the proposed dynamic mechanism

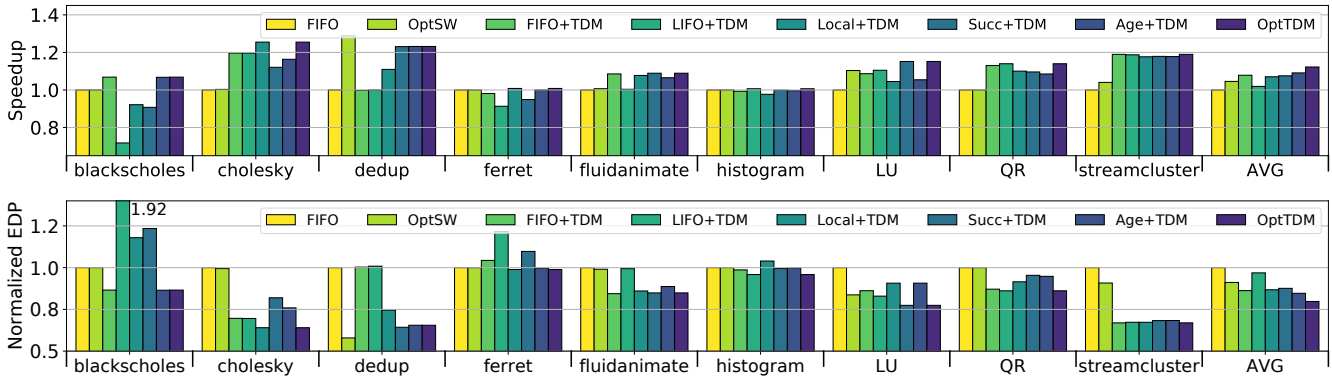

Figure 12: Speedup (top) and EDP reduction (bottom) with FIFO, LIFO, Locality-aware and Criticality-aware schedulers using software runtime system and TDM. Results are normalized to the software runtime system with a FIFO scheduler.

(DYN) that uses the size of the dependence. Results show that each fixed value drastically changes the occupancy of the DAT, from 1% to 88%. More importantly, every benchmark requires selecting different index bits. This happens because the benchmarks use different block sizes, so the number of lower bits that are equal in the dependence addresses changes in every benchmark. By using the size of the dependences provided by the runtime system to dynamically select the index bits, the DMU avoids conflicts in the DAT and maximizes its occupancy in all benchmarks.

6. FLEXIBLE SCHEDULING WITH TDM

This section illustrates the synergy of TDM with different software schedulers that exploit the characteristics of the applications to improve performance and power consumption.

Five schedulers are used in the experiments: *First-In First-Out (FIFO)* schedules tasks in the same order as they become ready. *Last-In First-Out (LIFO)* schedules first the last task that has become ready. *Locality* scheduler exploits data locality and assigns tasks to cores aiming to minimize data movements. When a task finishes executing on a core and some of its successor tasks is ready, a successor is executed on the core. If no successors are ready the first task in the ready queue is scheduled. *Successor* scheduler counts the number of successors of a task. If this number is above a threshold it is placed in a high priority ready queue, otherwise it is placed in a low priority ready queue. Threads first check the high priority ready queue and, if it is empty, they look for tasks in the low priority ready queue. *Age* scheduler sorts tasks in the ready queue by their creation time, so older tasks have higher priority than younger ones.

These schedulers can be used with TDM without any modification. The runtime system communicates with the DMU at task creation and finalization phases, and requests all the tasks that have become ready after the finalization of a task. The schedulers organize the ready tasks in software data structures and ready queues that implement the different policies. TDM reduces task creation overheads and, consequently, all schedulers benefit from this architectural support.

6.1 Performance Evaluation

We evaluate the performance of the different software schedulers when they are deployed by an entirely software-based runtime system and when they are combined with TDM. For each application we select the best scheduler with and without TDM, denoted OptTDM and OptSW, respectively. Figure 12 shows the speedups of OptSW, FIFO+TDM, LIFO+TDM, Locality+TDM, Successor+TDM, Age+TDM and OptTDM policies over a FIFO scheduler without hardware support. The geometric mean of the speedups is also reported (AVG).

In general, FIFO and LIFO schedulers show similar performance except for Blackscholes, which is parallelized with 64 independent chains of dependent tasks. With FIFO, all independent chains progress at the same pace, while with LIFO, 32 chains (one per core) progress much faster than the others, leading to a significant load imbalance and 29.3% performance degradation. A similar situation happens with Locality+TDM and Successor+TDM, although performance only degrades 7.8% and 9.2%, respectively.

TDM significantly reduces the task dependence management overheads in Cholesky, as reported in Figure 10. The locality scheduler further improves performance, as this is a memory intensive application that reads blocks of a dense matrix from memory. Thus, Cholesky is very sensitive to data locality and achieves 4.2% higher IPC than FIFO+TDM.

Schedulers based on priorities (Successor and Age) achieve important improvements in benchmarks with a clear critical path in the TDG. Dedup has many compute-intensive tasks and each one of them is followed by a long I/O-intensive task. I/O tasks cannot be executed in parallel, which is enforced by means of control dependencies between them, so overlapping I/O with compute tasks maximizes parallelism. Successor+TDM achieves this overlap, as I/O and compute tasks have the same priority (all tasks have 1 successor), and yields a 23.2% performance improvement. FIFO prioritizes compute tasks because they become ready before their I/O counterparts, so it fails in overlapping I/O and computation. However, the successor scheduler harms performance in Cholesky, as it delays the execution of tasks that process the borders of the matrix, limiting the available parallelism.

Overall, OptSW performs worse than TDM combined with any scheduler, while the best scheduler (Age+TDM) achieves an average 9.1% performance speedup. It is important to note that the best performance is achieved with FIFO+TDM, LIFO+TDM, Locality+TDM, Successor+TDM, and Age+TDM for 2, 2, 2, 2, and 1 different benchmarks, respectively.

When the best scheduler per application is used, average 4.5% and 12.2% performance improvements are obtained with OptSW and Opt+TDM, respectively. The benefits of TDM are demonstrated by two facts: first, TDM provides enhanced results for all the schedulers and, second, TDM exposes the scheduler policy to the software, which yields large performance benefits due to the flexibility it provides.

6.2 Energy Efficiency

This section evaluates the energy efficiency of TDM combined with different schedulers. The bottom chart of Figure 12 shows the energy delay product (EDP) of FIFO, LIFO, Locality, Successor and Age schedulers when combined with TDM. This figure considers the extra power introduced by the hardware structures of the DMU. Results are normalized to a pure software runtime system with a FIFO scheduler, and a geometric mean (AVG) of all the results is shown.

Figure 12 shows that TDM provides significant EDP reductions in seven benchmarks, and minimal reductions are obtained in Ferret and Histogram. On average, EDP is reduced up to 8.9% with the best software solution (OptSW), while EDP is reduced between 3.1% and 15.4% when combining different schedulers with TDM. Combining TDM with the best scheduler per application (OptTDM) yields the best results, achieving average reductions in EDP of 20.3%.

In terms of power consumption, the DMU consumes a negligible fraction of the total power, less than 0.01%. All benchmarks consume very similar power with the considered schedulers on a software runtime system and when they combine the schedulers with TDM (less than 1.0% difference). In addition, average power results show less than 1.0% variation between different schedulers. Since the average power consumption does not significantly change, the improvements in total energy to solution follow the same trends as Figure 12.

6.3 Comparison with Other Proposals

This section compares TDM with two alternative hardware support proposals for the runtime system. Carbon [10] implements the task scheduler at the hardware level and task dependence management is done in software by the runtime system so, conceptually, it is the opposite to TDM. Carbon provides ISA instructions that allow threads to add and request ready tasks, and the hardware support consists of a set of distributed hardware queues to keep ready tasks and a fixed FIFO scheduling policy with work stealing. In contrast, Task Superscalar [11] offloads all the runtime system activities to the architecture, including task dependence management and task scheduling with a fixed FIFO policy. It uses an interface similar to Carbon, and its hardware support consists of a gateway, a ready queue, and distributed tables to track tasks and dependences. As explained in Section 2.2, thread synchronizations overheads are negligible in our experiments, so proposals that accelerate thread synchronization such as ADM [15] are not included in the study.

The top chart of Figure 13 presents the speedup of Carbon, Task Superscalar and TDM over a software runtime system with a FIFO scheduler. TDM makes use of the best scheduling policy per benchmark found in the previous section, and the geometric mean of the results is also presented.

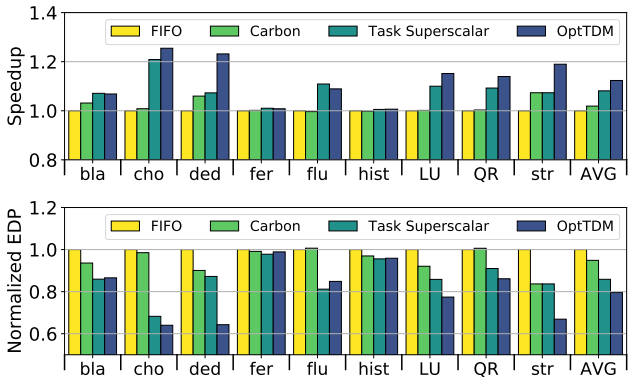

Figure 13: Speedup (top) and EDP reduction (bottom) of Carbon, Task Superscalar and TDM over a software runtime system with FIFO scheduler.

Carbon improves performance in Blackscholes, Dedup and Streamcluster, reaching speedups of up to 7.3%. In the rest of benchmarks its impact is negligible because, as shown in Figure 2, the time spent in scheduling phases is very low, while tracking task dependences is much more costly. As a result, Carbon obtains a modest average speedup of 1.9%.

Task Superscalar performs both task scheduling and dependence management in hardware. This approach provides significant speedups in several benchmarks, reaching an average 8.1% speedup. TDM achieves similar reductions in runtime system overheads and further improves performance by allowing flexible software schedulers, achieving an average speedup of 12.3% and clearly qualifying as the best option. The advantage of TDM is particularly significant in cases where using the appropriate scheduling policy is fundamental to increase the parallelism, as in Dedup, where TDM improves performance by 23.1% while Carbon and Task Superscalar just reach 5.9% and 7.2%, respectively.

The bottom chart of Figure 13 shows the Energy Delay Product (EDP) of Carbon, Task Superscalar and TDM normalized to the baseline software solution with a FIFO scheduler. Results considers the extra power consumption added by the hardware structures of TDM, Carbon and Task Superscalar. Important EDP reductions are obtained in seven of the benchmarks, while more modest EDP reductions are obtained in the remaining two benchmarks. On average, TDM reduces EDP by 20.4% while Carbon and Task Superscalar only achieve reductions of 5.1% and 14.1%, respectively.

Regarding hardware complexity, TDM lays between Carbon (simple hardware queues) and Task Superscalar. Table 3 shows that the DMU requires 105.25KB for the selected configuration. For the same configuration in terms of number of in-flight tasks and dependences, Task Superscalar requires 769KB: a 1KB Gateway, a 256KB TRS (2048 entries$\times$128B), a 256KB ORT (2048 entries$\times$128B), and a 256KB Ready Queue (2048 entries$\times$128B). The cost of the OVT is not taken into account, as the DMU does not perform dependence renaming. In addition, Task Superscalar requires more power-hungry CAM look-ups than the DMU. All together, the DMU requires 7.3$\times$ lower hardware complexity than Task Superscalar.

Finally, an alternative solution is to add an extra core devoted to the runtime system. We observe that the performance of a 33-core system with a pure software runtime system improves marginally, 0.8% on average. In the 32-core baseline task creation is already executed by one thread running on a core, so the extra core just adds one more worker thread and has no impact on dependence tracking overheads.

7. RELATED WORK

TDM accelerates the dependence management operations of task-based data-flow programming models such as OpenMP 4.0 [3], OmpSs [21], Codelets [29], Charm++ [30], Habanero [31], or StarPU [32]. Other task-based models like Cilk [33] or TBB [34] do not use data-flow annotations and require the programmer to manually synchronize tasks. This harms programmability but saves the overheads of discovering and managing dependences in the runtime system.

Data-flow architectures like Monsoon [35], *T [36], or EARTH [37] included hardware support for dependence management and communication between instructions or threads. These architectures were programmed with specific-purpose programming models where the compiler statically generated the TDG and established the dependences between producers and consumers [38, 39], and scheduling was either done statically at compile time using graph partitioning techniques or dynamically in hardware using a fixed FIFO queue. TDM targets modern data-flow programming models that require a runtime system to generate the TDG, track dependences and schedule tasks, which generate overheads that were not encountered in data-flow architectures.

Similar to Carbon [10] and Task Superscalar [11], other architectures use hardware task schedulers. These approaches rely on the programmer or on the semantics of different programming models to establish dependences between tasks, so they do not offer hardware support for dependence management in task-based data-flow programming models. GPUs use hardware schedulers for the kernels, that can be synchronized with CUDA streams [40] or with queues and barrier packets in HSA [41]. In Pangaea [42] the CPU schedules tasks on the GPU, and both communicate via user-level interrupts. Swarm [12] relies on speculative task execution and conflict detection to preserve dependences. Swarm requires hardware support for speculation instead of for dependence management and uses either a FIFO or a spatial scheduler fixed in the architecture [13]. Fractal [14] extends Swarm to allow nested parallelism by means of task domains, that can be ordered or unordered to avoid over-serialization.

Like ADM [15], some works propose to add architectural support for thread synchronization primitives, reducing the overheads caused by concurrency. These solutions allow to implement different scheduling policies in software with reduced hardware complexity, but they do not accelerate all the operations of the dependence management and task scheduling phases, so they are less effective in mitigating runtime system overheads. CAF [43] provides hardware support to optimize core-to-core queue-based communications, adding a specialized accelerator that supports various queue management functionalities. QOLB [44] proposes an implementation for lock primitives based on distributed queues where the waiting cores spin locally, preventing unnecessary network traffic. Active Memory Operations [45] extend the memory controllers of distributed shared-memory systems so that synchronization and heavy write sharing operations can be executed in the node where the data resides.

Another way to mitigate the task creation bottleneck is parallelizing it with nested parallelism. Although most parallel programming models support nesting, the practical usage of this paradigm requires a hierarchical decomposition of the algorithm and does not allow to specify dependencies across different nesting levels. In addition, most accelerators have null or limited support for nesting. Due to all these restrictions, it is much more appropriate to alleviate the task creation bottleneck via the TDM hardware support instead of transferring this responsibility to the software stack.

In general, works that propose hardware support for task scheduling show promising results for hundreds of cores running workloads with extremely fine-grained tasks. For TDM we consider a 32-core architecture and we select the best task granularity for each benchmark. In this scenario the task size is orders of magnitude bigger than the ones used in other works of the literature, so the trade-offs and the sources of overheads change significantly. This paper shows that, in our setup, the main bottleneck is dependence management, while the overheads of task scheduling and thread synchronization are very low. In a scenario with finer-grained parallelism and higher core counts, TDM would still be able to mitigate dependence management overheads, so task scheduling or thread synchronization could become the main bottleneck. TDM is compatible with many proposals that accelerate task scheduling and, in particular, TDM would combine nicely with the proposals that accelerate thread synchronization primitives to reduce task scheduling overheads while maintaining the advantages of flexible task scheduling.

8. CONCLUSIONS

Task-based programming models are very appealing for large-scale multi-cores. A key aspect of task-parallel programs is the task granularity, which determines the potential to exploit the available parallelism and to ensure load balancing, but also dictates the runtime system overheads.

This paper proposes TDM, a hardware/software co-designed mechanism to accelerate task dependence management operations while allowing flexible task scheduling in software. Unlike previous works with schedulers fixed in the architecture, the separation of concerns in TDM provides high degrees of flexibility, adaptability and composability, which are key in modern computing infrastructures with multiple sockets and off-chip accelerators, and also allows to capitalize on the benefits that different scheduling policies provide for certain applications and environments. In addition, the architectural support of TDM includes novel techniques that maximize efficiency, such as renaming IDs to reduce the storage requirements or leveraging the size of the dependences to avoid conflicts in the hardware structures when the lower bits of the dependence addresses are equal.

As a result, TDM outperforms software runtime systems by an average 12.3% while reducing EDP by 20.4%. Compared to pure hardware solutions, TDM achieves an average speedup of 4.2% with 7.3× lower hardware complexity.

9. REFERENCES

[1] R. H. Dennard, F. H. Gaensslen, H. nien Yu, V. L. Rideout, E. Bassous, Andre, and R. Leblanc, "Design of ion-implanted MOSFETs with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, pp. 256–268, Oct. 1974.

[2] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-ISA heterogeneous multi-core architectures for multithreaded workload performance," in *ISCA*, pp. 64–75, 2004.

[3] "OpenMP Application Program Interface. Version 4.0. July 2013."

[4] M. Manivannan, V. Papaefstathiou, M. Pericas, and P. Stenstrom, "Radar: Runtime-assisted dead region management for last-level caches," in *HPCA*, pp. 644–656, 2016.

[5] A. Pan and V. S. Pai, "Runtime-driven shared last-level cache management for task-parallel programs," in *SC*, pp. 11:1–11:12, 2015.

[6] L. Alvarez, M. Moreto, M. Casas, E. Castillo, X. Martorell, J. Labarta, E. Ayguade, and M. Valero, "Runtime-guided management of scratchpad memories in multicore architectures," in *PACT*, pp. 379–391, 2015.

[7] E. Castillo, M. Moretó, M. Casas, L. Alvarez, E. Vallejo, K. Chronaki, R. M. Badia, J. L. Bosque, R. Beivide, E. Ayguadé, J. Labarta, and M. Valero, "CATA: criticality aware task acceleration for multicore processors," in *IPDPS*, pp. 413–422, 2016.

[8] M. Valero, M. Moreto, M. Casas, E. Ayguade, and J. Labarta, "Runtime-aware architectures: A first approach," *International Journal on Supercomputing Frontiers and Innovations*, vol. 1, pp. 29–44, June 2014.

[9] M. Casas, M. Moreto, L. Alvarez, E. Castillo, D. Chasapis, T. Hayes, L. Jaulmes, O. Palomar, O. Unsal, A. Cristal, *et al.*, "Runtime-aware architectures," in *Proceedings of the 21st International Conference on Parallel and Distributed Computing*, Euro-Par 2015, (Berlin, Heidelberg, Germany), pp. 16–27, Springer-Verlag, 2015.

[10] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural support for fine-grained parallelism on chip multiprocessors," in *ISCA*, pp. 162–173, 2007.

[11] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero, "Task superscalar: An out-of-order task pipeline," in *MICRO*, pp. 89–100, 2010.

[12] M. C. Jeffrey, S. Subramanian, C. Yan, J. S. Emer, and D. Sanchez, "A scalable architecture for ordered parallelism," in *MICRO*, pp. 228–241, 2015.

[13] M. C. Jeffrey, S. Subramanian, M. Abeydeera, J. S. Emer, and D. Sanchez, "Data-centric execution of speculative parallel programs," in *MICRO*, pp. 1–13, 2016.

[14] S. Subramanian, M. C. Jeffrey, M. Abeydeera, H. R. Lee, V. A. Ying, J. Emer, and D. Sanchez, "Fractal: An execution model for fine-grain nested speculative parallelism," in *ISCA*, pp. 587–599, 2017.

[15] D. Sanchez, R. M. Yoo, and C. Kozyrakis, "Flexible architectural support for fine-grain scheduling," in *ASPLOS*, pp. 311–322, 2010.

[16] K. Chronaki, A. Rico, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero, "Criticality-aware dynamic task scheduling for heterogeneous architectures," in *ICS*, pp. 329–338, 2015.

[17] H. Topcuouglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, pp. 260–274, Mar. 2002.

[18] M. Hakem and F. Butelle, "Dynamic critical path scheduling parallel programs onto multiprocessors," in *IPDPS*, 2005.

[19] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, pp. 33–38, July 2008.

[20] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Architect. News*, vol. 39, pp. 1–7, Aug. 2011.

[21] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 2, pp. 173–193, 2011.

[22] 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 *MICRO*, pp. 469–480, 2009.

[23] S. Xi, H. Jacobson, P. Bose, G.-Y. Wei, and D. Brooks, "Quantifying sources of error in McPAT and potential impacts on architectural studies," in *HPCA*, pp. 577–589, 2015.

[24] N. Muralimanohar and R. Balasubramonian, "CACTI 6.0: A tool to understand large caches," 2009.

[25] D. Chasapis, M. Casas, M. Moretó, R. Vidal, E. Ayguadé, J. Labarta, and M. Valero, "PARSECSs: Evaluating the impact of task parallelism in the parsec benchmark suite," *ACM Trans. Archit. Code Optim.*, vol. 12, pp. 41:1–41:22, Dec. 2015.

[26] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *PACT*, pp. 72–81, 2008.

[27] F. Porikli, "Integral histogram: A fast way to extract histograms in Cartesian spaces," in *Conference on Computer Vision and Pattern Recognition Workshops (CVPR)*, pp. 829–836, 2005.

[28] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta, "Nanos mercurium: a research compiler for OpenMP," in *6th European Workshop on OpenMP*, pp. 103–109, 2004.

[29] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, "Using a "Codelet" program execution model for exascale machines," in *EXADAPT*, pp. 64–69, 2011.

[30] L. V. Kale and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on C++," in *OOPSLA*, pp. 91–108, 1993.

[31] J. Shirako, J. M. Zhao, V. K. Nandivada, and V. N. Sarkar, "Chunking parallel loops in the presence of synchronization," in *ICS*, pp. 181–192, 2009.

[32] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," in *Euro-Par*, pp. 863–874, 2009.

[33] 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 *PPoPP*, pp. 207–216, 1995.

[34] J. Reinders, *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly Media, 2007.

[35] G. M. Papadopoulos and D. E. Culler, "Monsoon: An explicit token-store architecture," in *ISCA*, pp. 82–91, 1990.

[36] R. S. Nikhil, G. M. Papadopoulos, and Arvind, "*T: A Multithreaded Massively Parallel Architecture," in *ISCA*, pp. 156–167, 1992.

[37] H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, X. Tang, G. R. Gao, P. Cupryk, N. Elmasri, L. J. Hendren, A. Jimenez, S. Krishnan, A. Marquez, S. Merali, S. S. Nemawarkar, P. Panangaden, X. Xue, and Y. Zhu, "A design study of the EARTH multiprocessor," in *PACT*, pp. 59–68, 1995.

[38] K. Arvind and R. S. Nikhil, "Executing a program on the mit tagged-token dataflow architecture," *IEEE Transactions on Computers*, vol. 39, no. 3, pp. 300–318, 1990.

[39] R. S. Nikhil, "The programming language id and its compilation for parallel machines," *International Journal of High Speed Computing*, vol. 5, no. 2, pp. 171–223, 1993.

[40] "CUDA C Programming Guide. Version 8.0. June 2017."

[41] "HSA Platform System Architecture Specification. Version 1.0. January 2015."

[42] H. Wong, A. Bracy, E. Schuchman, T. M. Aamodt, J. D. Collins, P. H. Wang, G. Chinya, A. K. Groen, H. Jiang, and H. Wang, "Pangaea: A tightly-coupled ia32 heterogeneous chip multiprocessor," in *PACT*, pp. 52–61, 2008.

[43] Y. Wang, R. Wang, A. Herdrich, J. Tsai, and Y. Solihin, "CAF: Core to core communication acceleration framework," in *PACT*, pp. 351–362, 2016.

[44] A. Kägi, D. Burger, and J. R. Goodman, "Efficient synchronization: Let them eat QOLB," in *ISCA*, pp. 170–180, 1997.

[45] Z. Fang, L. Zhang, J. B. Carter, A. Ibrahim, and M. A. Parker, "Active memory operations," in *ICS*, pp. 232–241, 2007.