J Supercomput (2015) 71:3567–3592
DOI 10.1007/s11227-015-1449-1

Design space exploration of hardware task superscalar architecture

Fahimeh Yazdanpanah[1] · Mohammad Alaei[1]

Published online: 10 June 2015

Abstract For current high performance computing systems, exploiting concurrency is a serious and important challenge. Recently, several dynamic software task management mechanisms have been proposed. In particular, task-based dataflow programming models which benefit from dataflow principles to improve task-level parallelism and overcome the limitations of static task management systems. However, these programming models rely on software-based dependency analysis, which are performed inherently slowly; and this limits their scalability specially when there is fine-grained task granularity and a large amount of tasks. Moreover, task scheduling in software introduces overheads, and so becomes increasingly inefficient with the number of cores. In contrast, a hardware scheduling solution, like Task SuperScalar (TSS), can achieve greater values of speed-up because a hardware task scheduler requires fewer cycles than the software version to dispatch a task. TSS combines the effectiveness of Out-of-Order processors together with the task abstraction. It has been implemented in software with limited parallelism and high memory consumption due to the nature of the software implementation. Hardware Task Superscalar (HTSS) is proposed to solve these drawbacks. HTSS is designed to be integrated in a future high performance computer with the ability to exploit fine-grained task parallelism. In this article, a deep latency and design space exploration of HTSS is described. For design space exploration, we have designed a full cycle-accurate simulator of HTSS, called SimTSS. The simulator has been tuned based on latency exploration of HTSS components resulted from VHDL description of each component. As the result of this exploration, we have found the number of components and memory capacity of HTSS for HPC systems.

✉ Fahimeh Yazdanpanah
yazdanpanah@uk.ac.ir

Mohammad Alaei
m_alaei@uk.ac.ir

[1] Computer Engineering Department, Faculty of Engineering, Shahid Bahonar University of Kerman, Kerman, Iran

Keywords Task scheduling · Task parallelism · Task superscalar · OmpSs

1 Introduction

Motivated by challenges of increasing frequency of the traditional microprocessors based on a single processing unit (i.e., physical limit of transistor size, power consumption, and heat dissipation [15,16]), around 2003 the architectures have been shifted to microprocessors with multiple processing elements known as cores. Nowadays, there are two kinds of architectures: multi-core and many-core architectures. Multi-core architectures integrate few cores into a single microprocessor, preserving the execution speed of sequential programs. Many-core architectures use a large number of cores oriented to execution throughput of parallel programs.

Regardless of the chosen model, exploiting concurrency consists of breaking a problem into discrete parts (that can be called *tasks* when they are composed of several instructions), and managing and coordinating them to ensure correct execution, simultaneously or interleaved in one or more processing units. Although the simple definition, exploiting concurrency is a difficult and important challenge for current high-performance systems. Recently, there has been a growing interest in developing runtime task scheduling techniques due to their flexibility and high- performance capability. Different dynamic software task management systems, such as task-based dataflow programming models [3,4,13,28,29], benefit from dataflow principles to improve task-level parallelism and overcome the limitations of static task management systems. These models implicitly perform computation scheduling and data dependency analysis, thereby the programmer does not need to explicitly manage parallelism. In addition, these models use tasks instead of instructions as a basic work unit.

OmpSs is a general-purpose dataflow task-based programming model that simplifies parallel programmers' life. It benefits from dynamic data dependency analysis, dataflow scheduling, and out-of-order executing. OmpSs has been implemented in software through Mercurium compiler and Nanos++ runtime system. Although the software implementation is optimized, it introduces some more overhead in task execution. This limits the efficiency for small tasks which are generated close to each other [34]. This is a considerable problem that increases with the number of available cores. In contrast, a tiled hardware task scheduler would be more efficient for small tasks and would provide larger task throughput.

TSS [9,10] is a hybrid dataflow/von-Neumann architecture [35] that supports OmpSs programming model as a hardware task scheduler. This architecture takes benefits of the effectiveness of Out-of-Order processors applied to the task- level parallelism of the program. It implements in hardware the task dependency management and task scheduling functionalities of Nanos++ runtime system, thus results in reducing the per-task overhead; allowing the efficient exploitation of parallelism at a finer granularity. In this sense, the TSS processor combines dataflow execution of tasks with control flow execution within the tasks. The idea behind this behavior is that TSS uncovers task-level parallelism among tasks generated by a sequential thread similar to ILP pipelines uncover parallelism in a sequential instruction stream.

The initial design of the TSS architecture had only been simulated in software with limited parallelism and high memory consumption due to the nature of the software implementation. Although that approach demonstrated the validity of the idea, a real working proof of concept was beyond the initial study.

This work wants to achieve a realistic Hardware Task Superscalar (HTSS) design, and provides the prerequisites of the real implementation with all details that arise in real-world applications. For this, TSS architecture has been re-designed to be synthesizable in hardware. For the final version of HTSS, we have applied several improvements to our previous HTSS proposals [37,37]. Evaluation of one version of hardware implementation of Task Superscalar (called Picos), and also comparison of Picos with Nanos++ runtime system has been presented in [34]. The main components of HTSS are more or less the same as Picos, but there are considerable differences between these two architectures.

The contributions of this article are (1) creating a simulator based on the hardware implementation of TSS; (2) configuring the simulator with the latency cycles obtained from simulating of VHDL description of HTSS components; and (3) performing a rapid design space exploration of HTSS using real benchmark applications. The main goal of this article is performing design space exploration of HTSS design in order to determine the number of components and capacity of their embedded memory for our next plan which is to synthesis the whole HTSS described in VHDL, and implement and map it on an FPGA.

The remainder of this document is organized as follows: Sect. 2 explains the operational flow of our final proposal for HTSS architecture and also the latency exploration of HTSS design as well as the challenges of hardware prototyping. Section 3 describes our proposed cycle-accurate simulator called SimTSS which is designed for hardware design space exploration of HTSS. In Sect. 4, the methodology, experimental framework and the benchmark applications are explained. Section 5 presents the design space exploration of HTSS. The obtained results are presented in Sect. 6 for different HTSS configurations to determine the best HTSS configuration with the minimum number of components and the minimum memory capacity that provides maximum performance. The related work is presented in Sect. 7. Finally, this article concludes in Sect. 8, by presenting some insights on the results.

2 HTSS overview

For designing the final HTSS, the operational flow of the original TSS architecture was modified in order to fit the hardware constrains and also to solve some stalls and simplify some time-consuming operations. More improvements were also applied to the base HTSS design in order to reduce packet communications and packet processing cycles. Additionally, the hardware design was improved by reducing the latency of entering a new task into the pipeline system.

Figure 1 shows the components of TSS and also their equivalents in HTSS. As the figure illustrates, TSS includes one GW (gateway), several TRSs (Task Reservation Stations), several OVTs (Object Versioning Tables), and several ORTs (Object Renaming Tables), and several arbiters and FIFOS, and also one TS (Task Scheduler).

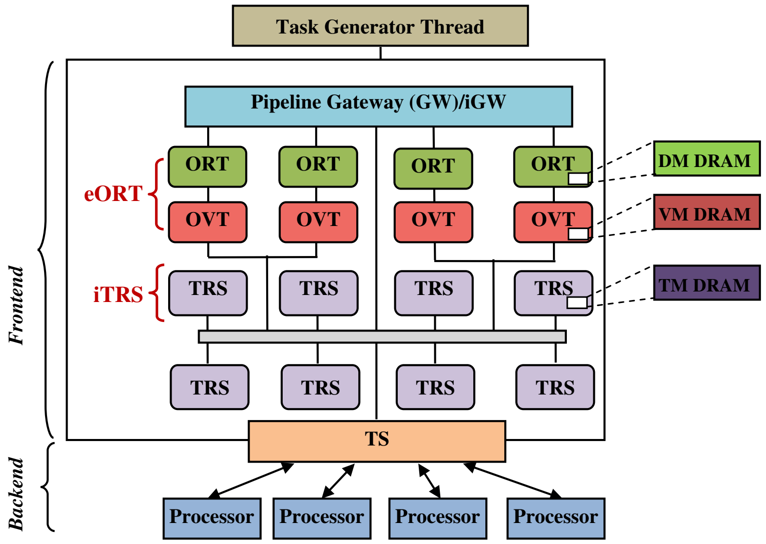

Fig. 1 TSS components and their equivalents in HTSS

HTSS includes one iGW (improved GW), several iTRSs (improved TRS), several eORTs (extended ORTs), several FIFOs, and arbiters and one TS.

iGW is responsible for controlling the flow of tasks into the pipeline and distributing tasks to the different iTRS modules, sending parameters to their assigned eORT modules, and stalling the task generator thread whenever the pipeline fills. iTRSs store the in-flight task information and track the readiness of task operands. Inter-iTRSs communication is used to register consumers with producers, and notify consumers when data is ready. In-flight tasks include (1) the tasks waiting for their parameters to be ready; (2) ready tasks waiting to be sent for execution; (3) the tasks are being executed; and finally, (4) the tasks which are finished but still not retired. In HTSS, the functionality of OVT and ORT modules of TSS are merged into a new module referred to as eORT. eORTs map parameters that access the same memory object, and thereby detect task dependencies. Storing information of producer or consumer(s) of tasks parameters allows the system to maintain the dependence chain with realistic memory sizes. To keep this chain, eORTs track live operand versions, which are created whenever a new data producer is decoded. Effectively, eORT manages data anti- and output-dependencies by chaining different output operands and unblocking them in order by sending a ready message when the previous version is released.

Memory components are the most important components of HTSS. Similar to TSS, in HTSS, there are three kinds of memory modules: task memory (TM), dependence memory (DM), and version memory (VM). The interconnection network is also an important component since it can easily limit the scalability of the design. To overcome the potential limitation problem, the network includes arbiters and FIFOs that decouple the processing of every component in the system. This kind of network configuration allows the system to scale easily while preventing the stalls that may happen during processing a sequence of tasks and their parameters.

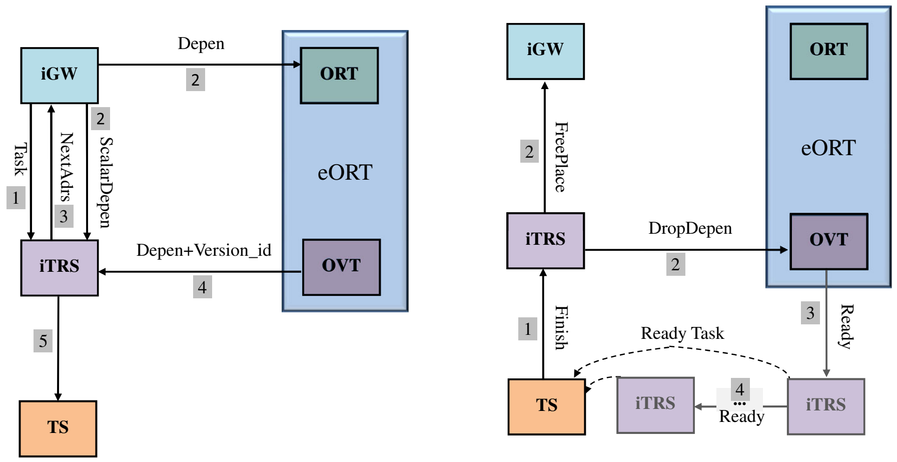

Fig. 2 Operational flow of HTSS, when a task arrives to the pipeline (*left*), when a task finishes (*right*)

Another important implementation consideration is minimizing the cycles required for processing the system packets in order to increase the overall system throughput. The main functionality of each component is performed by a finite state machine (FSM). The FSMs are designed to have a minimum number of states; each state takes only one cycle to be completed. For accessing the memories, one state initializes the control signals of the memory and the following state performs the memory accessing.

Figure 2 illustrates the HTSS components and the order of packet passing between the components for processing an arriving task and also for a finished task. When a new task arrives to the pipeline, iGW sends it to the selected place of one of iTRSs and sends its parameters to eORT to be analyzed for data dependency. iTRS allocates a space for the task and if it has more space for next tasks, it sends a new address to iGW. When eORT gets a parameter, it updates its entry and creates a new version; or updates an existing version; or does both depending on the direction of the parameter and whether the parameter appears for the first time or not. After that, eORT sends the parameter with the address of the version to iTRS. By this information, the producer/consumer chain for each parameter is formed. When all of input parameters of a task become available, the task is ready to be executed, so it is sent to TS.

When a task is finished, the responsible iTRS starts to release the task and its parameters from TM. For each parameter, it notifies eORT. eORT updates the entries and counters of the version. If there is no user for the version, the version will be deleted. Meanwhile, for each of the output parameter the OVT sends a ready message to the last consumer of the parameter. The consumer TRS while updating its entry, passes the ready message to the next consumer and the next consumer passes to the next one until the first consumer gets the ready message.

Algorithms 1 and 2, respectively, describe the operational flow of HTSS when a new task arrives and when a task is finished. As the algorithms show, HTSS operational flow is similar to what we have described in our previous works [36,37] with some improvements applied to the procedure of communication between iTRSs and iGW. The improvements cause a task and its parameters are issue faster to the pipeline.

```
iGW gets meta-data of a task and its parameters;
iGW selects a free iTRS based on round robin algorithm and free slot availability;
iGW sends the task with a slot address to the allocated iTRS;
iGW starts to send the parameters of the task;
if #parameters = 0 then
    iTRS sends the task to execute;
else
    for each of the parameters do
        if parameter is an scalar then
            iGW directly sends the parameter to iTRS;
            iTRS saves it in the TM;
        else
            iGW sends each non-scalar parameter to eORT for data dependency analysis;
            eORT saves the parameter in the DM;
            if parameter is an input then
                if first time then
                    eORT creates a version for the parameters in the VM;
                else
                    eORT updates the current version of the parameter in the VM;
                end
            else
                eORT creates a version for the parameter in the VM;
                if NOT first time then
                    eORT updates the previous version of the parameter in the VM;
                end
            end
            eORT sends the parameter to iTRS;
            iTRS saves it in the TM;
        end
    end
    if all the parameters are ready then
        iTRS sends the task to execute;
    end
end
```

Algorithm 1: HTSS algorithm for processing a new task

Based on these algorithms, we have described each component of HTSS with VHDL in order to perform a detailed latency analysis of each component for processing different kinds of packets. To verify the functionality of each module and to obtain the latencies, the hardware design has been simulated using ModelSim 6.6d of Altera using several bit traces. Those traces represent input packets that test the main and corner working conditions of the system. The correctness of the output packets generated by the modules and also the modifications to their related memories have been tested. Then, we use these latencies to tune the cycle-accurate simulator for design space exploration of HTSS in a real implementation (Fig. 3).

Table 1 presents the latency (in cycles) required by each FSM to process the received packets as well as description of the packets and the responsible unit (i.e., the unit that process the packet). Each module has been tested with all possible different types of input packets. Processing of each packet, depending on the type of packet, may have accesses to memories and/or produce another packet. Therefore, processing of a packet may have different latency from others depending on the carried information

```
iTRS releases all parameters and then the task from its memory;
iTRS notifies a DropDepen packet to eORT for each parameter;
if parameter is output then
    eORT notifies readiness of the parameter to iTRS which is the top element of the consumer stack;
    Last consumer of the processed version will notify next producer (next version);
end
if #users of the version = 0 then
    if the version is the last one then
        if all other version deleted then
            eORT deletes the last version and eORT entry;
        end
    else
        eORT deletes the version;
        if all other version deleted then
            eORT deletes the last version and eORT entry;
        end
    end
end
iTRS sends a message to iGW to inform it the freed slot address;
```

Algorithm 2: HTSS algorithm for processing a finished task

and the internal state of the system. In addition to the latencies shown in Table 1, each module uses four extra cycles in order to check the input packets, select the one with the highest priority and initialize its FSM.

3 Cycle-accurate simulator of HTSS

For design space exploration of HTSS, we have designed a cycle- accurate simulator, called SimTSS, based on the functionality of each HTSS components. Figure 3 illustrates the high-level structure of SimTSS which is a tiled pipelined architecture consisting of one iGW, *n* iTRSs, *n* eORTs, one TS, and a network configuration including several arbiters and several FIFOs. SimTSS is a configurable architecture that accepts several parameters to simulate a specific HTSS design. These parameters include the number of iTRSs, the number of eORTs, the number of entries in every memory, and some configuration parameters of DM that has a set-associative structure.

Similar to HTSS, components of SimTSS communicate to each other using packets for message passing communication. Each component uses at least one FSM for processing input packets and producing output packets, as well as accessing to the memory units. The components of SimTSS are interconnected by several FIFOs and arbiters that are scaled in accordance with the rest of the modules. Note that the network configuration decouples the work in the modules reducing stalls in the system.

The amount of parallelism that the HTSS pipeline can uncover depends on the capacity of the memories used for storing task information, their parameters, and the versions of these parameters. On the other hand, the performance of the HTSS architecture depends on both the capacity of the memories and the number of components.

The TM is embedded into iTRS module while both DM and VM are placed in eORT module. As it was mentioned, iTRSs keep the information of all alive tasks

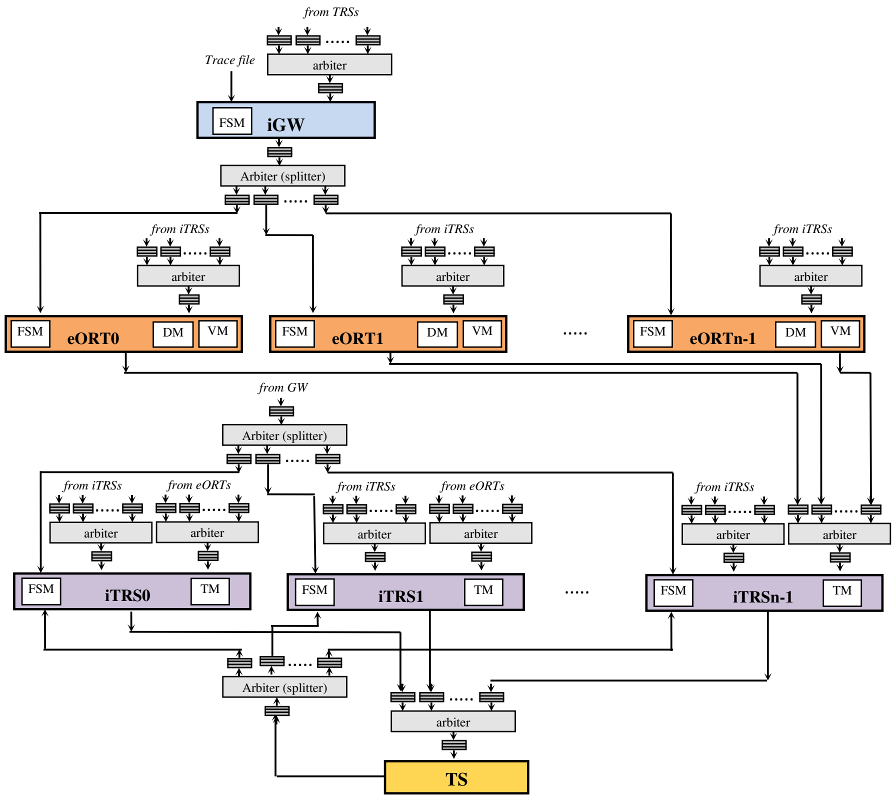

Fig. 3 High level description of SimTSS

(in-flight tasks) in the pipeline. However, it should be emphasized that TMs size does not exclusively determine the effective number of in-flight tasks, as this number might be limited by the number of entries of VM and DM. eORTs maintain an entry for each parameter used by in-flight tasks in the DM, and the corresponding version(s) of that objects in the VM. As such, the number of entries they can store affects the number of in-flight tasks.

Table 2 lists the input parameters of SimTSS. In the simulator, it is possible to select any number of iTRSs and eORTs, as well as any number of entries for the different types of memories (i.e., TM entries, DM entries, and VM entries). In addition, since the DM has been designed as a set-associative memory, it has two more parameters to be determined: the number of ways and the access mode. Access mode determines whether DM uses the bit-based mapping (standard) hash function or an improved hash function for distributing parameters in DM. As it will be shown later on, the hash system is a critical point in the system as it significantly reduces system stalls caused by several parameters with the same index trying to be placed in the same DM set (DM memory conflicts).

The configurable simulator allows to deeply evaluate the number of components and memory entries of HTSS suitable for different systems. Therefore, it can be found

Table 1 Latencies of processing the packets

Packet	Description	Processing latency	Responsible unit
ContIssue	Notifies the GW that an space in the iTRS memory is available.	2 cycles	iGW
DataReady	Notifies another task(s) that a parameter is ready.	2 cycles	iTRS
DepeneORT	Includes a non-scalar parameter for data dependency analysis.	−2 cycles if the (input or output) dependence appears for the first time −3 cycles if the dependence is input and does not appear for the first time −4 cycles if the dependence is output and does not appear for the first time	eORT
DropDepen	Is sent for informing the releasing of the parameter.	2 cycles	eORT
DropVersion	Is for getting the permission of releasing a version.	−2 cycles for deleting the last version −3 cycles for deleting the last version and the previous one	eORT
Execute	Includes the meta-data of a ready task for executing.	1 cycle (for loading meta-data of a task) +2 cycles for loading every dependence fromTM +1 cycle for sending the task to the TS	iTRS
Finish	Notifies TRSs that execution of the task has been finished	3 cycles +2 additional cycles for loading every dependence from TM	iTRS
Issue	Includes meta-data of a task.	−4 cycles for tasks without any dependence −3 cycles for tasks with at least one dependence +2 additional cycles for every dependence	iTRS

that an HTSS configuration with the minimum number of components and memory entries provides maximum performance for a given amount of processing resources.

Figure 4 shows the workflow of SimTSS usage. Based on that, SimTSS gets an input configuration file which initializes its parameters (i.e., number of workers, number of iTRSs, number of eORTs, and number of memory entries and parameters), and selects an input trace. The input trace includes data and meta-data of tasks including task identifiers, execution cycles, and number of parameters, direction of parameters, and issue time of each task that are obtained by instrumentation source code of the applications. When the execution of a trace is completed, SimTSS reports finished time of each task, total number of required cycles, and some statistics that have been used in the design process.

For generating the traces, the cycle counters of the target processor have been used in order to find out the cycles required for executing each task, the cycle in which the task was issued, and the total number of cycles for completing all tasks of an application. In

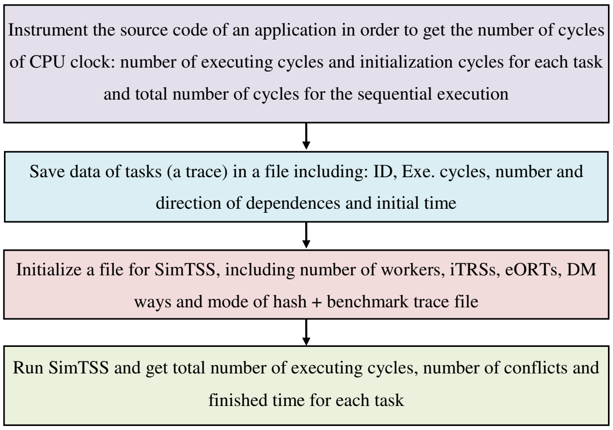
Instrument the source code of an application in order to get the number of cycles of CPU clock: number of executing cycles and initialization cycles for each task and total number of cycles for the sequential execution
Save data of tasks (a trace) in a file including: ID, Exe. cycles, number and direction of dependences and initial time
Initialize a file for SimTSS, including number of workers, iTRSs, eORTs, DM ways and mode of hash + benchmark trace file
Run SimTSS and get total number of executing cycles, number of conflicts and finished time for each task

Fig. 4 Workflow of SimTSS usage

Table 2 Parameters of SimTSS

SimTSS Parameters	Description
# iTRSs	Number of iTRSs that can be any number between 1 to n
# eORTs	Number of eORTs that can be any number, power of two, between 1 to n
TM entries	Number of in-flight tasks per iTRS
VM entries	Number of versions that the system can keep per eORT
DM entries	Number of dependencies that the system can keep per eORT
# Ways	Number of ways in each set of the DM (Associativity of DM)
DM access mode	Selecting standard hash or the improved SimTSS hash
# Workers	Number of available workers

fact, the total number of cycles of executing tasks is equal to the sequential execution of the application. It is noticeable that the number of cycles used for instrumenting instructions is not accounted in the total number of cycles.

4 Methodology and experimental frameworks for design space exploration

4.1 Methodology

By modifying the parameters of SimTSS, different HTSS configurations have been examined to find the best configuration for HTSS that provides the highest performance for a given number of processors (workers) with employing minimum hardware resources. Moreover, the number of iTRSs and eORTs has been explored for each benchmark considering different number of workers. The distributed design of the pipeline facilitates speeding up the overall decode rate, by overlapping the different works of managing task dependencies and decoding tasks. Specifically, replicating eORTs enables multiple dependencies to be recorded in parallel, whereas iTRS replication reduces the per-TRS load and distributes iTRS loads, and thereby increases the overall processing rate of inter-iTRS communication.

At the start point of design space exploration (*Phase I*), it is assumed that there are unlimited resources available for prototyping HTSS. Based on this assumption, a design with a lot of number of components and a lot of number of memory entries is used. With this configuration (referred as BigConf), the minimum number of workers that provides maximum performance is determined for all the benchmarks. Then, using the minimum number of workers, the minimum number of entries for each memory and the minimum number of modules that allow to obtain this maximum performance is determined. At the end of this phase, a design configuration which provides high performance is obtained, called HPCConf. Using the results of *Phase I* (BigConf and HPCConf), in *Phase II*, an HTSS with limited number of workers (i.e., 32 workers) is explored. In *Phase III*, different aspects of HTSS are studied, compared to the *Parallelism* and *ZeroHTSS* systems.

In the *Parallelism* system, an ideal design with any number of components and memory entries, that provides the highest possible speed- up, is used. The goal of this system is to show the performance of the system compared to different HTSS configurations for a given application. The performance provided by the *Parallelism* system does not depend on the number of workers or the number of HTSS components and memory entries. It only depends on input trace and, in fact, the critical path of an application limits the *Parallelism* system to provide infinite performance. Indeed, $Parallelism = T_1/T_\infty$ where T_1 is the sequential time and T_∞ is the time of the critical path in the parallel strategy supposing infinite resources. On the other hand, the *ZeroHTSS* system is an HTSS with unlimited number of resources, where packets are processed in each FSM of the component immediately (in zero cycles). Those two ideal approximations, the *Parallelism* and the *ZeroHTSS* systems, will be used to highlight the strength of the HTSS system and provide some insights for future improvements.

4.2 Benchmarks

In order to evaluate the capabilities of the HTSS, a group of real applications has been selected. All the applications can be obtained from the BSC Application Repository (BAR) [5]. The applications have been annotated with OmpSs pragmas to determine the tasks and their parameters. For every task, one pragma specifies the parameters and their directions (i.e., input, output, and in-out). With this information, the source code of the selected applications has been instrumented to generate the traces used as input data to SimTSS. Here, the selected applications are shortly described:

Cholesky factorization The `Cholesky` factorization decomposes a symmetric, positive definite matrix $A = LL'$, with L lower triangular. This distribution includes three different variants of the `Cholesky` decomposition. We have selected the `llchol` variant which is the left-looking variant, as implemented by the routine DPOTRF() in LAPACK.

LU factorization The `LU` factorization decomposes an m×n matrix (*m* should be larger or equal *n*) $A = L \times U$, with *L* unit lower triangular (m×n) and *U* upper triangular (n×n).

SparseLU decomposition This application as the above one, performs an `LU` decomposition, but over a square sparse matrix. The matrix is allocated by blocks of contiguous memory.

Heat diffusion This is an implementation of an iterative solver for `Heat` distribution. There are three user-selectable algorithms: *Jacobi*, *Gauss-Seidel*, and *Red-Black*. For this work, the *Gauss-Seidel* method has been selected.

4.2.1 Traces descriptions

Table 3 presents the characteristics of the benchmark applications. It includes the input configuration, number of tasks, and average of task sizes as well as maximum and minimum task size for each benchmark. In addition, it presents average number of parameters, average tasks distance, and sequential execution cycles.

Input configuration of the applications in Table 3 states the parameters that have been chosen for running each application. It is important to mention that the configuration has been selected for stressing HTSS system. Note that those are not the best configurations to solve the problem in a sequential system, but generate several small tasks issued as fast as possible. In other words, the configuration parameters of the benchmarks were chosen trying to obtain executions that generate several short-distant, fine-grained tasks. The execution time obtained for the given problem size was not a concern in this point as the experiments try to measure the ability of the hardware to manage tasks. The size of a task is measured as the number of cycles that are used for executing it. This number directly depends on the amount of computation that should be carried out for each task. Task distance is the number of cycles

Table 3 Information of benchmark traces

Application	Cholesky	SparseLU	Heat	LU
Input configuration	100,2	64,8	256,32	256,1
Number of tasks	22100	11472	1025	32896
Average task size (in cycle)	778	9835	1116	1970
Max task size (in cycle)	84704	147148	3124	229924
Min task size (in cycle)	416	3344	24	488
Average number of parameters	2.88	2.90	4.996	2
Average task distance (in cycle)	31.06	139.06	39.63	24.78
Sequential execution cycles	19942850	114419887	1184928	65601061

between the arrivals of two successive tasks. In the table, the average of tasks distance is presented. Sequential execution cycles states the number of cycles required in the sequential execution of each application.

Here, the different input configurations are described. Those configurations have been selected in order to have large number of small tasks that are issued close to each other. For `Cholesky`, 100 is the size of the matrix, and 2 is the size of the block. Therefore the number of blocks is equal to 50. For `SparseLU`, the first number is the number of blocks in each matrix dimension. In Table 3, the matrix has 64×64 blocks. The second number of this application is the block size in each block dimension (8×8 blocks in the table). For `LU`, the first number is the number of rows and columns of the matrix (dimensions of the matrix) and second one is the number of columns in a panel. In the case of `Heat`, the first number determines the matrix size and the second one is the number of blocks in each matrix dimension.

The number of tasks and average of task sizes as well as maximum and minimum task size for each benchmark are presented. We compute average tasks distance as *((initial time of last task—initial time of first task)/number of tasks)*. As average task distance shows, compared to the average task sizes, the tasks are really close to each other, especially for `Cholesky`, `LU`, and `Heat` applications. On the other hand, as `SparseLU` operates on a sparse matrix, it has a greater number of cycles between two successive tasks.

Finally, the sequential execution cycles number is a baseline number for comparing results of SimTSS with. For SimTSS analysis, speed-up results are presented. Those values of speed-up are obtained by comparing the sequential baseline execution of an application to the simulation of the input trace of that application, with the same application parameters (i.e., speed-up = sequential execution cycles/SimTSS execution cycles). Note that each task execution in the simulator lasts the same number of cycles than the sequential task execution.

5 Design space exploration of HTSS

In this section, exploring the design space of the HTSS using the software simulator (SimTSS) is presented. The main goal is to explore which amount of resources (i.e., number of iTRSs, number of eORTs, capacity of TM, VM, and DM, and also suitable structure of DM) is necessary for exploiting emerging many-core designs. For this, the results obtained from running the benchmarks on SimTSS are presented. Then, the number are evaluated in order to find a suitable configuration of the HTSS prototype for different performance configurations.

5.1 HTSS for high-performance computing systems

For determining the minimum number of workers, a very big configuration of the HTSS system with a lot of resources is assumed. This configuration is called BigConf that has 32 iTRSs, 32 eORTs, 16K capacity for all memory modules, and DM with 16 ways. For each configuration parameter, a large number is selected in such a way that even if the value is halved the time results of the system remain the same. Using

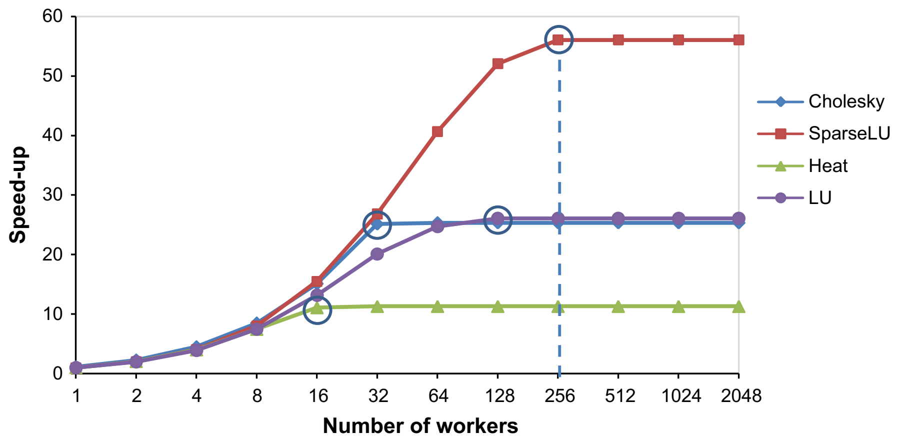

Fig. 5 Summary of speed-up of an HTSS with a lot of resources

BigConf, the number of workers that provide maximum speed-up can be determined. To do this, first, the minimum number of workers that provides the highest speed-up for each benchmark is found, and then the maximum number across all the applications is selected.

Figure 5 presents the speed-up (*Y*-axis) obtained executing the traces of the four benchmark applications on SimTSS with a lot of resources for different number of workers (*X*-axis). As the figure shows, the speed-up increases by increasing the number of workers. Of course, the diagrams become flatten when we reach a certain number of workers; this number is 64 for `Cholesky`, 256 for `SparseLU`, 32 for `Heat`, and 128 for `LU`. Therefore, 256 workers (processors) are enough for an HTSS system suitable for task scheduling of HPC applications.

`SparseLU` provides the highest speed-up compared to the other selected applications, because it has more than ten thousands big tasks which are more or less far from each other. Moreover, sequential execution of `SparseLU` takes a large number of cycles. But, since `SparseLU` works on sparse matrices, it has the highest potential of parallel execution on many-core systems with high performance among other selected applications.

`Heat` has few numbers of tasks, with five parameters in average for each task. The tasks are small and close to each other. Every 255 consumers in `Heat` are related to one producer, this matter causes restrictions for parallel execution of this application. In addition, the serial section of `Heat` that limits parallel execution is considerable. Hence, we have got the least speed-up for Heat in all of our experiences for these four benchmark applications.

Compared to the other selected applications, `LU` has a lot of small tasks. Parallel execution time of `LU` is almost one third of its sequential execution; but, because of data dependency of small tasks having small average task distance, its speed-up diagram is flattening when more than 128 cores are available.

`Cholesly` has the smallest task size, and its tasks are much closed to each other. In this application, there are several nested loops. On the other hand, data dependency

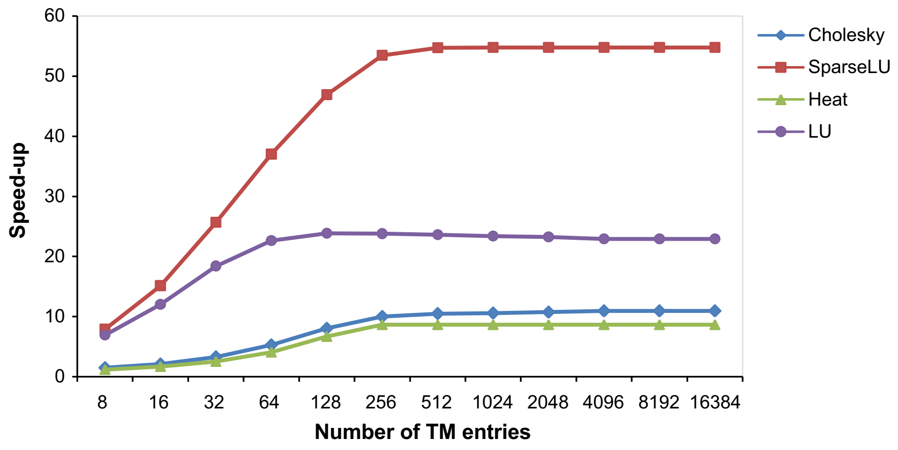

Fig. 6 Speed-up obtained as a function of the number of TM entries

of consecutive tasks and also existence different kinds of tasks with different sizes are usual in this application. Therefore, a lot of restrictions in parallel extraction are yielded although there are unlimited resources available.

Once the minimum number of workers is determined, we have to reduce the amount of resources which has been allocated to BigConf while maintaining the performance. Considering 256 workers and only one iTRS, we have changed number of TM entries in order to find the minimum TM entries that provides highest performance. All the other parameters are the same as in the BigConf configuration. The results are shown in Fig. 6 (*Y*-axis shows the speed-up over the sequential execution, and *X*-axis is the number of TM entries). For the selected traces, 1024 TM entries (the same as number of in-flight tasks) seem enough when the system has only one iTRS. But comparing to values of speed-up observed in Fig. 5, we have obtained less values of speed-up, particularly for `Cholesky` benchmark. This happens because of having only one iTRS module in the system. Regardless of the TM size, the time that iTRS uses to process the tasks may become the bottleneck of the system. That can be solved by increasing the number of modules of HTSS, hiding the latency of iTRS processes, and distributing 1024 TM entries over them.

Figure 7 shows how changing the number of iTRSs and its memory size influences the number of cycles for the applications. As it can be seen, the optimum design point is to have eight iTRS modules with the capacity to store 128 tasks each (or even eight iTRSs with 512 TM entries), for the `Cholesky` benchmark. However, that configuration is only ideal for the specific case of `Cholesky` and presents serious drawbacks from the hardware resources point of view: eight iTRS modules represent a large interconnection network and, furthermore, 4K in-flight tasks demand roughly 240KBytes of memory storage for the tasks and more space in the other memories that should be scaled accordingly. Considering both the results of Figs. 6 and 7 for all the benchmarks and the hardware resources requirements of an eight-TRS configuration, the selected prototype has been limited to four iTRSs with a 256-entry TM each,

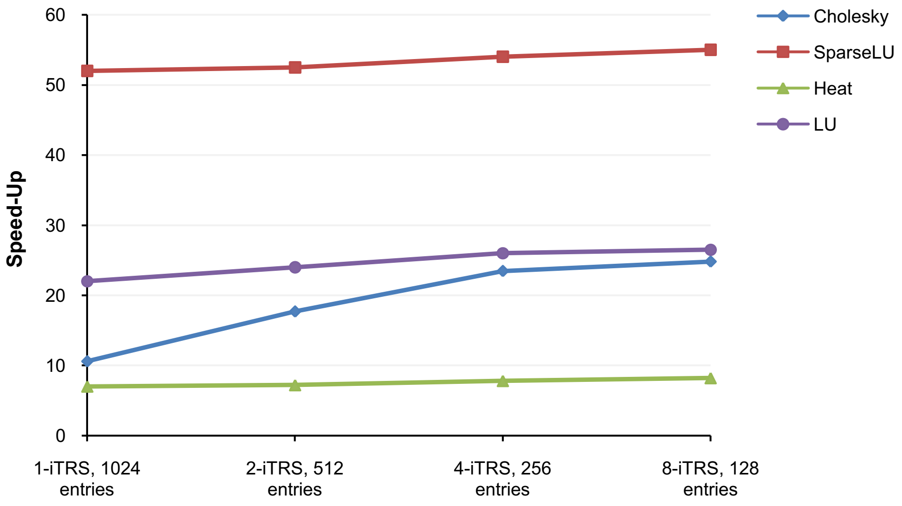

Fig. 7 Speed-up as a function of the number of iTRSs and TM entries

reducing the interconnection network and memory requirements, while guaranteeing high speed-up.

After selecting iTRS modules configuration, the next step is to select a good eORT modules configuration. This work is more difficult as the eORT module is more complex with two different memories and its effect on the system performance is not so obvious. As explained before, the eORT modules keep track of the dependency chain and to do so, they have to store both all the dependencies (parameters) of all the tasks in the DM and all the versions of those dependencies (the different values that the dependency can have due to the different in-flight tasks that produce this value) in the VM.

Like TM, VM is used in an indexed way, so, any value can be stored in any entry with direct memory access. Figure 8 shows the speed-up obtained for each benchmark when the number of VM entries is varied and we have only one eORT module. As it can be seen in the figure, the `Cholesky` application is, as in the case of the TM, the most demanding one, needing 4096 entries in the VM to achieve the peak performance.

Considering that number of VM entries, we show in Fig. 9 how changing the number of eORT modules affects the speed-up when the total number of VM entries is maintained constant and the DM is kept at its large value. In particular, it can be observed that four eORTs with 1024 VM entries each (for a total of 4096 versions) achieves the upper limit of the performance.

DM has been designed as a set-associative memory and is more complex than TM and VM, so, it is the key element in eORT module. When a parameter enters the system, it should efficiently find if the parameter is new or not, and update its metadata correspondingly. However, the DM is not a cache and when a block in the DM is full, the system cannot replace and flush the existing entry. Instead, it should stall and wait until the parameter that uses the same entry is no longer alive. The reason is that the addresses of parameters are unrelated because of the data alignment in the

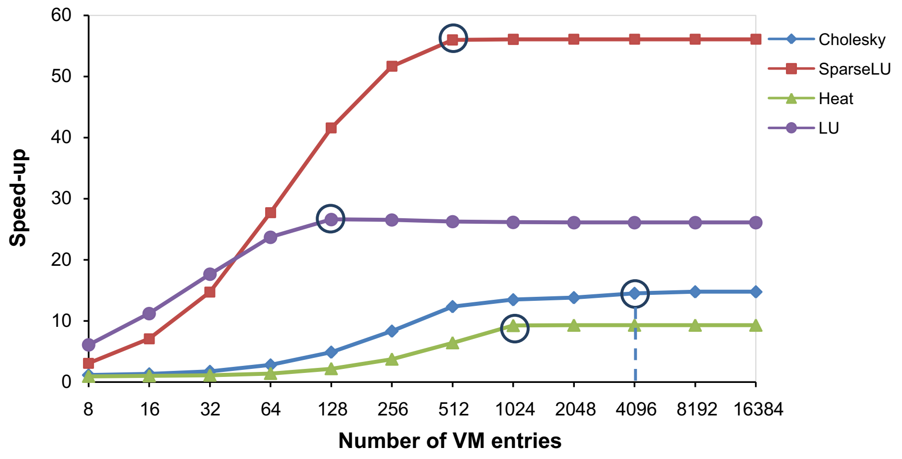

Fig. 8 Speed-up obtained as a function of the number of VM entries

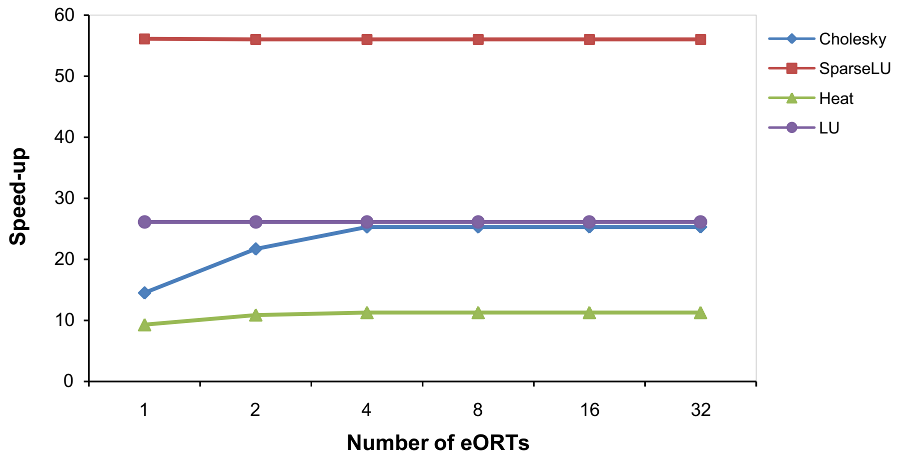

Fig. 9 Speed-up obtained as a function of the number of eORT modules

application tasks. Therefore, when storing them in the corresponding DM, several consecutive parameters may be stored in the same entry of DM. As the DM cannot discard entries, until the previous parameter is deleted from the pipeline when all the tasks that use it finishes the requester parameter(s) and its tasks have to wait. This may cause large stalls if DM is implemented as a direct-mapped memory or if the hash function does not appropriately randomize the addresses of the parameters. For this reason, an associative memory, and a more complex hash function (Pearson-like hash [27]) than the usual (address less significant bits—LSB hash in Fig. 10) is used to select the set.

Figure 10 shows the effect of the Pearson-like hash function on the speed-up obtained as a function of the number of DM entries comparing to LSB-hash function. As it can be seen, the Pearson-like hash function has better speed-up for all the

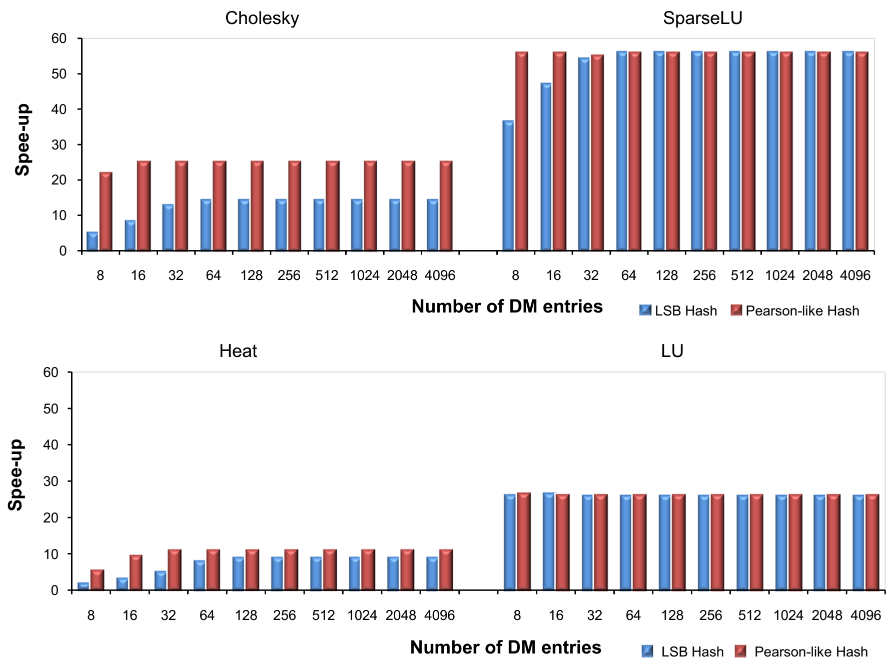

Fig. 10 Speed-up obtained as a function of the DM entries with LSB and Pearson-like hash

cases or, from another point of view, allows the system to obtain the same results with a smaller number of DM entries. While the LSB-hash function only uses the less significant bits of parameter addresses to distribute them between eORTs and in the DM memory, Pearson-like hash function improves the distributions. The rest of the results presented in this section uses the improved hash. Figure 10 also shows that 32 entries (with 16-way set associative each) for DM is enough to obtain the upper limit performance for the studied benchmarks.

Selecting memory associativity is also a key for the performance. The ideal would be having a full associative DM, but this is not possible in a real environment. In SimTSS, we can configure DM with different number of ways. Therefore, the effect of having different associativities with the Pearson-like hash has been studied and it is shown in Fig. 11. This figure shows that using a larger number of ways results in only a little higher speed-up maintaining the total amount of memory, but with much more hardware resources and a more complex DM structure. Hence, a four-way structure seems fine to provide good performance results with reasonable amount of hardware usage. It is noticeable that the DM structure is the main difference between HTSS and Picos. HTSS will utilize less logic than Picos [34] because of its smaller and simpler set-associative memory unit.

In conclusion, the proposed configuration for an HTSS design is composed by ten modules: one GW, four iTRSs, four eORTs, and one TS unit. Each iTRS has a 256-entry TM. Each eORT has 2 memory units: the VM is an indexed array of 512 entries while the DM is a four-way set-associative memory with 128 entries. This configuration is

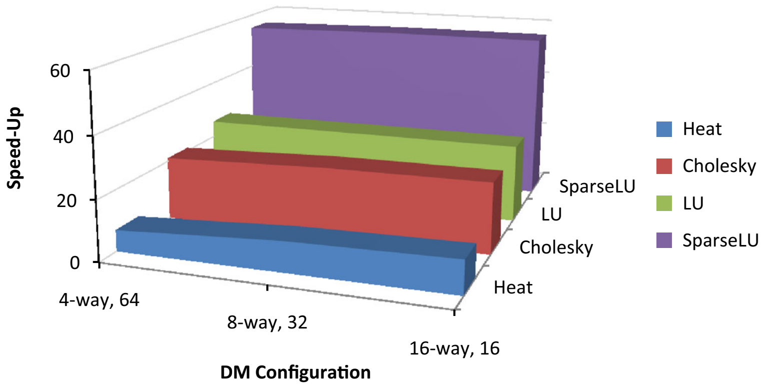

Fig. 11 Speed-up obtained as a function of the associativity of the DM

an HPC configuration of HTSS (called HPCConf) that provides maximum speed-up while utilizing a minimum amount of resources for up to 256 workers.

It should be mentioned that both HTSS and Picos have improved gateway (iGW) and improved TRSs (iTRSs), so they act very similar in values of speed-ups obtained from the similar experiences in design space exploration except for DM parameters. In HTSS, DM is a four-way set-associative memory, while Picos has an eight-way set-associative DM, hence, HTSS DM utilizes almost half of memory modules of dependence memory of Picos. In addition, the control unit of DM in HTSS is simpler than that of Picos. This is the main difference between Picos and HTSS architectures.

In addition, we have improved network configuration of HTSS by reducing its utilized hardware resources, so it has less logic in communication network than Picos. Albeit, this improvement has not considerably affected the network level (i.e., number of FIFOs and arbiters in critical path between the main components in both system is the same), therefore there is no reduction in communication latency and obtained speed-up in HTSS compared to Picos.

5.2 HTSS design with limited workers

In this section, HTSS is analyzed with a more limited number of workers to evaluate the results obtained from the previous section, and find a suitable configuration for smaller systems like the ones that can be found nowadays. For this, it is assumed that the selected design has only 32 available workers and an HTSS with four iTRSs and four eORTs. The goal is to repeat the study of the effect of different memory sizes on performance but with limited resources in order to find out the minimum number of memory entries for an HTSS configuration for current parallel systems.

Figure 12 shows the effect of different number of TM entries on performance when there are 32 workers, four iTRSs, and four eORTs. In particular, it can be seen that 256 entries are enough for TM of each iTRS when it is used in conjunction with a 64-element four-way DMs with Pearson-like hash in each eORT. Figures 13 and 14 show the same study varying the number of VM and DM entries, respectively. Both figures

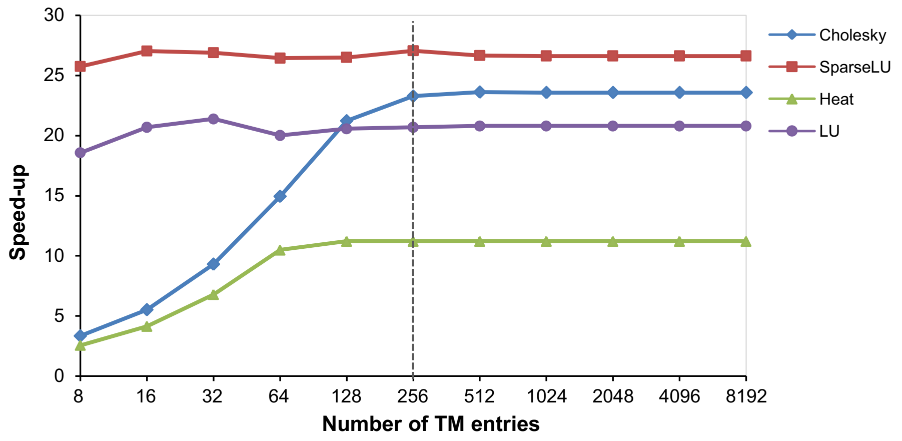

Fig. 12 Effect of different number of TM entries on the performance of an HTSS with 32 workers

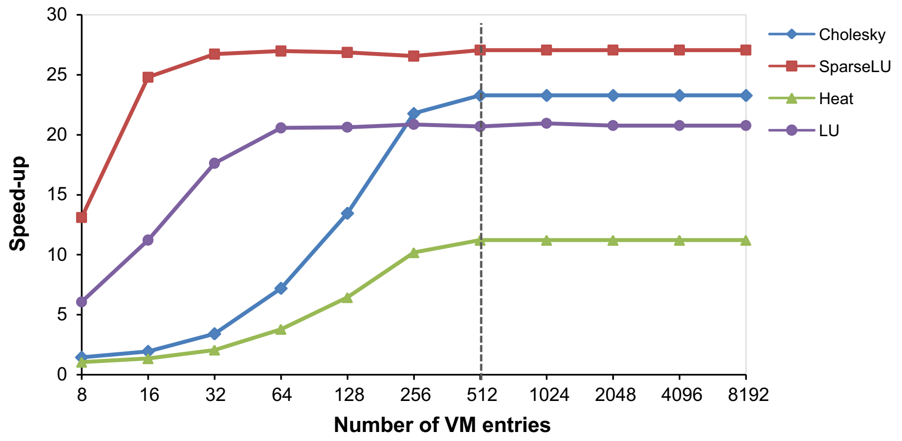

Fig. 13 Effect of different number of VM entries on the performance of an HTSS with 32 workers

show eORT memories effect on the performance of HTSS with only 32 processors. Results in Fig. 13 indicate that 512 entries should be selected, as a good size versus performance trade-off for VM entries. This is also the same number of VM entries that was found in the previous study in Sect. 5.1. In the case of Fig. 14, it can be seen that a 64-element four-way DM with Pearson-like hash is enough for each eORT, also as found in previous analysis in Sect. 5.1. As a result, the best configuration for using HTSS with a current system (with up to 32 cores) is four iTRSs with 256 entries for TM, and four eORTs with 512 VM entries and 128 DM entries for four-way DMs with improved hash. This architecture is able to store up to 1024 in-flight tasks in iTRSs, 2048 versions, and 2048 parameters in eORTs. Note that the total sizes of the memories depend on the word-size of each memory.

The results show that the same hardware is necessary to manage 256 or 32 workers. Although this result seems counter intuitive, it is due to the fact that memory sizes

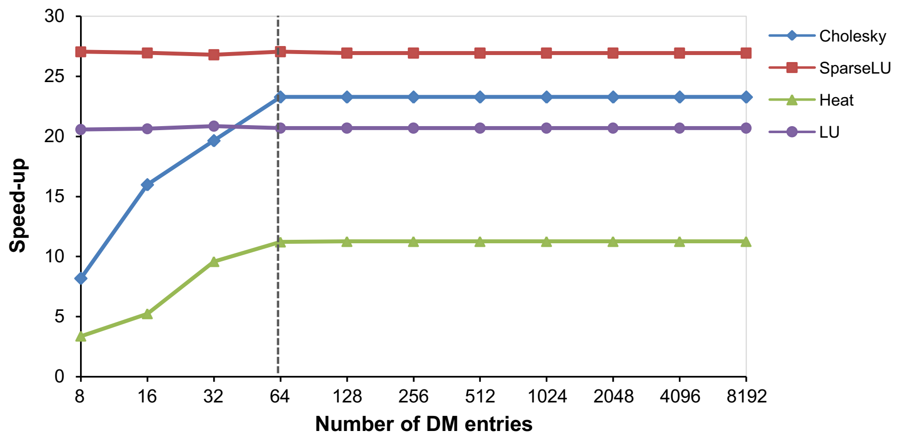

Fig. 14 Effect of different number of DM entries on the performance of an HTSS with 32 workers

are necessary to discover enough parallelism in the applications while the number of modules is necessary to keep pace with the task issue rate. Both factors depend mainly on the applications, not on the available number of workers. However, it can be seen in Figs. 12, 13, and 14 that with 32 workers the system gets a maximum speed-up of about 27×, while for 256 workers, a speed-up up to 56x can be obtained.

6 Results of the design space exploration

In this section, the selected different configurations of HTSS: HPCConf, MinConf, and BigConf are evaluated. MinConf is an HTSS with minimum resources (i.e., one iTRS and one eORT) for small many-core systems with eight workers which is similar to MinConf of Picos [34]. In this article, we ignore the explanation of analysis on MinConf.

To obtain a good idea of how well the proposed final systems behave, they have been evaluated with a different number of processors comparing the obtained values of speed-up to two control configurations: *Parallelism* and *ZeroHTSS* systems (see Sect. 4). For each of the benchmarks, Fig. 15 shows the maximum speed-up that can be obtained with the chosen parallelization strategy of each benchmark. Figure 15 also shows the results that would be obtained with an HTSS that uses zero cycles to process any packet (*ZeroHTSS*), the results obtained from BigConf, the HTSS for high performance computing systems (HPCConf), and the minimum configuration of HTSS (MinConf).

As it can be observed in Fig. 15, HPCConf performance is almost the same as *ZeroHTSS* system for all the benchmarks. Only for `Cholesky` a small slow-down can be appreciated as a trade-off of downsizing the resources. Also for systems with a small number of processors (up to eight), MinConf is able to keep pace and so, it would be enough and affordable to be implemented in embedded systems.

Comparing the results shown in Fig. 15 to the *Parallelism* configuration, it can be seen that the implementable HTSS can extract all the possible parallelism for three of

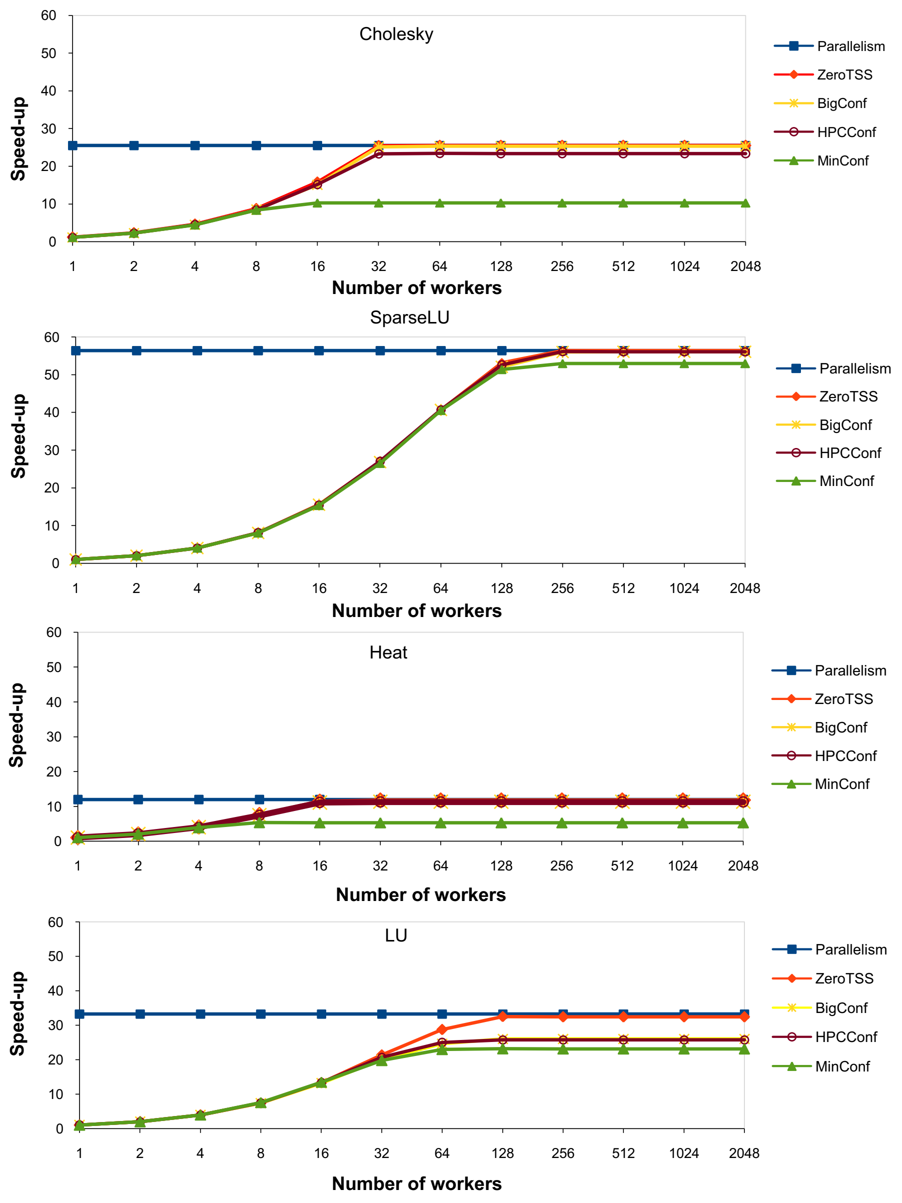

Fig. 15 Speed-ups obtained for different number of workers of the *Parallelism*, *ZeroTSS*, BigConf, HPC-Conf, and MinConf configurations

the four benchmarks once a certain number of workers is reached. The only exception is for the LU application which can obtain the maximum speed-up with the *ZeroHTSS* implementation, but not with HPCConf or even with the BigConf. The difference in performance here is due to the large dependency chains of consumers (255 for each producer) that the LU application creates. Awakening 255 consumers means creating a sequential chain of 255 packets between iTRSs and, consequently, when the last consumer is awakened several cycles have been wasted. To improve this, it can be proposed a system that simply creates a new version of a parameter when several consumers are detected. This new version awakens at the same time as the original one and splits the chain of packets into two different and parallel chains. However, this improvement has not been implemented as with more realistic task sizes this behavior will disappear by the longer task execution times.

7 Related work

Different dynamic software task management systems such as runtime systems have been proposed to overcome the problems of static task management systems. In particular, task-based dataflow programming models automate data dependency and solve the synchronization problem of static task management systems. Main examples of this class are StarSs family [3,4,28] (including OmpSs [6,8]), OoOJava [12,13], JADE [18,29–31], and OpenMP 4.0 [25]. These models try to support dynamic task creation and scheduling with a simple programming model [4]. However, their flexibility comes at the cost of a rather laborious task management that should be done at runtime [2]. The cost of that potentially huge task management affects the scalability and performance of such systems, and limits their functionality to applications with a lot of tasks.

The main purpose of a hardware task scheduler is accelerating the task management. The parallel program will continue calling the programming model software runtime, but this will subministrate the task dependency information to the hardware task scheduler. The hardware scheduler will gradually and efficiently create the task dependency graph while preparing the ready tasks for execution on the cores.

Some hardware support solutions for task scheduling [7,11,17,19,22,26,33] have been proposed to speed-up the task management but most of them only schedule independent tasks, leaving it to the programmer to deliver tasks at the appropriate time. Several research studies evaluated hardware task queues [11,17]. In these studies, task submission to a particular core is accelerated by hardware task queues, replacing software data structures and leveraging the corresponding synchronization overhead. However, in most of the cases, inter-task synchronization is still performed in software. Saez et al. [32] describe a hardware scheduler accelerator that combines scheduling of soft and hard real-time jobs on an uni-processor. In contrast, Al-Kadi and Terechoko [1] propose a video scheduler that tackles the task scheduling problem for a multi-core, involving complex task-to-core mapping. Furthermore, their task scheduler can create task dependency graphs. Other architectures, such as NVIDIA Tesla [19], are also known to provide hardware acceleration for task scheduling of independent tasks. Sjalander et al. [33] propose a programmable task management unit (TMU) accelerates task creation and synchronization in hardware. TMU runs a look-ahead program preparing tasks that will become ready in a short while. However, for fine-grain tasks executing for a few tens of cycles, it is needed a faster task scheduling. In order to achieve lower overhead, in this work it is designed as a dedicated hardware task scheduler to efficiently manage the task creation, scheduling, mapping, and synchronization of the tasks.

Dynamic scheduling for system-on-chip (SoC) with dynamically reconfigurable architectures is interesting for the emerging range of applications with dynamic behavior such as the work of Noguera and Badia [23,24]. Kalra and Lyseeky [14] addressed the relationship between the several hardware task scheduling algorithms and their impact on the number of reconfigurations required to execute.

TSS architecture [9,10] has been designed as a hardware support for the OmpSs programming model [8] for scheduling all dependent and independent tasks. Unlike Noguera's work, the task dependency graph is dynamically created and maintained using runtime data flow information, therefore increasing the range of applications that can be parallelized. The TSS architecture provides coarse-grain parallelism management through a dynamic dataflow execution model. In addition, it supports imperative programming on large-scale CMPs without any fundamental changes to the microarchitecture. Based on TSS, we have designed different versions of hardware Task Superscalar [36,37]. Evaluation of hardware design of Task Superscalar compared to Nanos++ runtime system has been presented in [34].

Nexus++ [20,21] is another hardware task management system designed based on StarSs that is implemented in a basic SystemC simulator. Both designs leverage the work of dynamically scheduling tasks with a real-time data dependence analysis while maintaining the programmability, generality, and easiness of use of the programming model.

8 Conclusions

The goal of this article was to propose a hardware design of the TSS architecture, a hybrid dataflow task scheduler proposed to accelerate the execution of applications annotated with the OmpSs programming model. To this end, once the potential benefits of a hardware task manager have been discussed, the operational flow of final HTSS architecture has been presented. This architecture is a new and more realistic hardware implementation of TSS that significantly reduces processing time and hardware resource usage over the initial proposal. Components of the HTSS have been coded in VHDL in order to obtain real-time constraints of such system for latency exploration. Based on the VHDL implementation of HTSS modules, a cycle-accurate software simulator has been developed in order to further improve HTSS design by detecting and correcting the system bottlenecks and perform a full design space exploration of a real and full hardware prototype.

As a result of this article, a realistic and implementable hardware prototype of an HTSS consisting of 10 modules (one gateway, four dependency chain trackers, four task reservation stations, and one scheduler) has been proposed. The proposed design has demonstrated its ability to deal with systems that manage up to 256 processors with a set of real benchmarks. The results show that the final HTSS system can obtain values of speed-up closer to the maximum of the parallelization strategy of the analyzed applications (up to 100× in the tests) using a reasonable amount of memory (less than 100 KB).

References

1. Al-Kadi G, Terechko AS (2009) A hardware task scheduler for embedded video processing. In: Proceedings of the international conference on high performance and embedded architectures and compilers (HiPEAC), pp 140–152
2. Badia RM (2011) Top down programming methodology and tools with StarSs, enabling scalable programming paradigms: extended abstract. In: Proceedings of the workshop on scalable algorithms for large-scale systems (ScalA), pp 19–20
3. Bellens P, Perez JM, Cabarcas F, Ramirez A, Badia RM, Labarta J (2009) CellSs: scheduling techniques to better exploit memory hierarchy. Sci Program 17(1–2):77–95
4. Bellens P, Perez J, Badia R, Labarta J (2006) CellSs: a programming model for the cell BE architecture. In: Proceedings of the supercomputing (SC). ACM, New York
5. Bsc application repository, bar (2014). In: Barcelona Supercomputing Center (BSC). https://pm.bsc.es/projects/bar. Accessed 06 Feb 2014
6. Bueno J, Martinell L, Duran A, Farreras M, Martorell X, Badia RM, Ayguade E, Labarta J (2011) Productive cluster programming with OmpSs. In: Proceedings of the International conference on parallel processing (Euro-Par), pp 555–566
7. Castrillon J, Zhang D, Kempf T, Vanthournout B, Leupers R, Ascheid G (2009) Task management in MPSoCs: an ASIP approach. In: Proceedings of the international conference on computer-aided design (ICCAD), pp 587–594
8. Duran A, Ayguade E, Badia RM, Labarta J, Martinell L, Martorell X, Planas J (2011) Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel Process Lett 21(2):173–193
9. Etsion Y, Cabarcas F, Rico A, Ramirez A, Badia RM, Ayguade E, Labarta J, Valero M (2010) Task superscalar: an out-of-order task pipeline. In: Proceedings of the international symposium on microarchitecture (MICRO), pp 89–100
10. Etsion Y, Ramirez A, Badia RM, Ayguade E, Labarta J, Valero M (2010) Task superscalar: using processors as functional units. In: Proceedings of the hot topics in parallelism (HOTPAR)
11. Hoogerbrugge J, Terechko A (2011) A multithreaded multicore system for embedded media processing. Trans High-Perform Embedded Archit Compil (THEA) 3(2):154–173 (2011)
12. Jenista JC, Eom YH, Demsky B (2010) OoOJava: an out-of-order approach to parallel programming. In: Proceedings of the USENIX conference on hot topic in parallelism (HotPar), pp 11–11
13. Jenista JC, Eom YH, Demsky BC (2011) OoOJava: software out-of-order execution. In: Proceedings of the ACM symposium on principles and practice of parallel programming (PPoPP), pp 57–68
14. Kalra R, Lysecky R (2010) Configuration locking and schedulability estimation for reduced reconfiguration overheads of reconfigurable systems. IEEE Trans Very Large Scale Integr Sys 18(4):671–674
15. Kish LB (2002) End of Moore's law: thermal (noise) death of integration in micro and nano electronics. Phys Lett A 305:144–149
16. Kish LB (2004) Moore's law and the energy requirement of computing versus performance. IEE Proc Circuits Dev Syst 151(2):190–194
17. Kumar S, Hughes CJ, Nguyen A (2007) Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In: Proceedings of the international symposium on computer architecture (ISCA), pp 162–173
18. Lam MS, Rinard MC (1991) Coarse-grain parallel programming in Jade. In: Proceedings of the ACM symposium on principles and practice of parallel programming (PPoPP). ACM, New York, pp 94–105
19. Lindholm E, Nickolls J, Oberman S, Montrym J (2008) NVIDIA Tesla: a unified graphics and computing architecture. IEEE Micro 28(2):39–55
20. Meenderinck C, Juurlink B (2010) A case for hardware task management support for the StarSs programming model. In: Proceedings of the conference on digital system design (DSD), pp 347–354
21. Meenderinck C, Juurlink B (2011) Nexus: hardware support for task-based programming. In: Proceedings of the conference on digital system design (DSD), pp 442–445
22. Nacul AC, Regazzoni F, Lajolo M (2007) Hardware scheduling support in SMP architectures. In: Proceedings of the conference on design, automation and test in Europe (DATE), pp 642–647
23. Noguera J, Badia RM (2003) System-level power-performance trade-offs in task scheduling for dynamically reconfigurable architectures. In: Proceedings of the international conference on compilers, architectures and synthesis for embedded systems (CASES), pp 73–83
24. Noguera J, Badia RM (2004) Multitasking on reconfigurable architectures: microarchitecture support and dynamic scheduling. ACM Trans Embedded Comput Syst 3(2):385–406

25. Openmp application program interface, version 4.0 (2013). www.openmp.org/. Accessed 06 Feb 2014
26. Park S (2008) A hardware operating system kernel for multi processors. IEICE Electron Express 5(9):296–302
27. Pearson PK (1990) Fast hashing of variable-length text strings. Commun ACM 33(6):677–680
28. Perez, Badia RM, Labarta J (2008) A dependency-aware task-based programming environment for multi-core architectures. In: Proceedings of the international conference on cluster computing (CC), pp 142–151
29. Rinard MC, Lam MS (1998) The design, implementation, and evaluation of Jade. ACM Trans Program Lang Syst (TPLS) 20(3):483–545
30. Rinard MC, Scales DJ, Lam MS (1992) Heterogeneous parallel programming in Jade. In: Proceedings of the conference on supercomputing, pp 245–256
31. Rinard MC, Scales DJ, Lam MS (1993) Jade: a high-level, machine-independent language for parallel programming. Computer 26(6):28–38
32. Saez S, Vila J, Crespo A, Garcia A (1999) A hardware scheduler for complex real time system. In: Proceedings of the IEEE international symposium industrial electronics (ISIE). IEEE, pp 43–48
33. Sjalander M, Terechko A, Duranton M (2008) A look-ahead task management unit for embedded multi-core architectures. In: Proceedings of the conference on digital system design (DSD), pp 149–157
34. Yazdanpanah F, Alvarez C, Jimenez-Gonalez D, Badia RM, Valero M (2015) Picos: a hardware runtime architecture support for ompss. Future Gener Comput Syst
35. Yazdanpanah F, Jimenez-Gonzalez D, Alvarez-Martinez C, Etsion Y (2013) Hybrid dataflow/von-Neumann architectures. IEEE Trans Parallel Distrib Syst (TPDS) 25(6):1489–1509
36. Yazdanpanah F, Jimenez-Gonzalez D, Alvarez-Martinez C, Etsion Y, Badia RM (2013) Analysis of the task superscalar architecture hardware design. In: Proceedings of the international conference on computational science (ICCS)
37. Yazdanpanah F, Jimenez-Gonzalez D, Alvarez-Martinez C, Etsion Y, Badia RM (2013) FPGA-based prototype of the task superscalar architecture. In: Proceedings of the 7th HiPEAC workshop of reconfigurable computing (WRC)