Improving the Scalability and Capabilities of the Nexus Hardware Task Management System

Tamer Dallou Ben Juurlink
Embedded Systems Architectures Group
Technische Universität Berlin
Einsteinufer 17, D-10587 Berlin
dallou@cs.tu-berlin.de, b.juurlink@tu-berlin.de

Cor Meenderinck
Computer Engineering Laboratory
Delft University of Technology
Mekelweg 4, 2628CD Delft
cor@ce.et.tudelft.nl

***Abstract*—Recently, several programming models have been proposed that try to relieve parallel programming. One of these programming models is StarSs. In StarSs, the programmer has to identify pieces of code that can be executed as tasks, as well as their inputs and outputs. Thereafter, the runtime system (RTS) determines the dependencies between tasks and schedules ready tasks onto worker cores. Previous work has shown that the StarSs RTS may constitute a bottleneck that limits the scalability of the system and proposed a hardware task management system called Nexus to eliminate this bottleneck. Nexus has several limitations, however. For example, the number of inputs and outputs of each task is limited to a fixed constant and Nexus does not support double buffering. In this paper we present Nexus++ that addresses these as well as other limitations. Experimental results show that double buffering increases the utilization from about 60% to almost 100% and that Nexus++ significantly enhances the scalability of applications parallelized using StarSs.**

I. Introduction

Due to the advent of multicore architectures, several parallel programming models have been proposed that aim at relieving parallel programming. Examples include Google's MapReduce [4], Intel's TBB [10], and StarSs [9]. StarSs, like OpenMP [3], enables the programmer to express parallelism by adding pragmas to the code. These pragmas identify pieces of code that can be executed as *tasks*, as well as their inputs and outputs. Based on the inputs and outputs, the runtime system (RTS) can determine the dependencies between tasks and schedule ready tasks onto cores that execute the tasks. The programmer, therefore, does not have to explicitly express dependencies between tasks and the corresponding synchronization. Furthermore, the RTS can also transparently optimize data reuse between tasks and coarsen tasks, thereby relieving the programmer from these burdens.

Previous work [8] has shown, however, that the StarSs RTS, when implemented in software, can be a bottleneck that limits the scalability of applications parallelized using StarSs. Roughly speaking, the RTS cannot compute task dependencies and attend to finished tasks fast enough to keep all *worker cores* that execute the tasks busy. The same work therefore proposed a hardware task management system called *Nexus* to accelerate the RTS. In Nexus, task dependencies are computed using hardware hash tables and a scalable synchronization mechanism with the worker cores is provided. Results show that Nexus improves the scalability of a synthetic application modeled after H.264 decoding by a factor of 4.3 when using 16 worker cores.

Even though Nexus improves the scalability significantly, it has several limitations. For example, since the hash table entries have a fixed size, the number of inputs and outputs of each task is limited to a fixed constant (5 in [8]). Similarly, the number of tasks that can depend on a certain data segment is limited. This limits the applicability of Nexus, i.e., not all StarSs applications can be executed on a multicore system with Nexus. Another limitation is that Nexus does not support double buffering, which allows executing one task while fetching the input data of another task. In [8] double buffering was not needed because the data transfer time was negligible.

In this paper we present Nexus++ that addresses these as well as other limitations. The dependency count constraint is solved by introducing *dummy* tasks and by adding dummy entries to the list of tasks that depend on a certain data segment. Double (in fact arbitrary) buffering is supported by providing a *Task Controller* at each worker core that buffers tasks before they are executed. A SystemC model has been developed to validate and evaluate the design. The preliminary results show that double buffering increases the utilization from about 60% to almost 100% for up to 64 cores. For 128 cores and more, the utilization starts to decrease because the master core that generates tasks and submits them to Nexus++ cannot generate them fast enough to keep all worker cores busy. The results also show that applications that could not be executed by Nexus, such as Gaussian elimination, can be executed efficiently on a multicore system with Nexus++.

Several hardware scheduling units have been proposed in literature. Most of them, however, assume independent tasks and are optimized for a certain application, a certain platform, or both. For example, Carbon [6] assumes independent tasks and uses hardware queues to retrieve tasks with low latency. In StarSs tasks can be dependent and it is the responsibility of the RTS to determine their dependencies.An example of a hardware accelerator targeted at a certain application domain is a hardware task scheduler optimized for H.264 decoding [1]. It requires, however, that the programmer specifies the dependencies between blocks. Etsion et al. [5] also proposed a hardware task management unit for the StarSs RTS, based on the similarity between task dependency checking and the

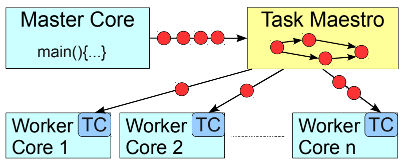

Fig. 1. Nexus++ in a multicore system

instruction scheduler of an out-of-order processor. It was evaluated using high-level simulations, however, and detailed hardware models were not developed.

This paper is organized as follows. Nexus++ and its novel features are described in Section II. In Section III the simulation environment and the employed benchmarks are described. The experimental results are presented in Section IV, and conclusions are drawn in Section V.

II. Nexus++ Hardware Task Management System

As shown in Figure 1, the multicore system under consideration is assumed to have one master core (MC) that executes the main thread and creates task descriptors (TDs), and several worker cores (WCs) that execute the tasks. A TD contains task-related information such as its function pointer and input/output list. Nexus++ is responsible for the task management responsibilities usually carried out by the software RTS. In an $(n + 1)$-core system (one MC and n WCs), Nexus++ is composed of $n + 1$ hardware modules: one *Task Maestro* (TM), which is mainly responsible for inter-task dependencies resolution, tasks scheduling, and load balancing, and n *Local Task Controllers* (TCs), which are distributed one per worker core, and are mainly responsible for task buffering.

A. System Description

The components of Nexus++ shown in Figure 2 are described through this scenario: when the MC executes the main program, it generates the TDs and sends them to the TM via the *TDsInBuffer*. This buffer is important so that the MC is not blocked while the TM is busy processing an earlier submitted TD. It also enables direct communication between the MC and the TM, in contrast to [8], where the TDs are communicated between the MC and the TM via off-chip memory.

Once the *TDsInBuffer* is written, the TM's *GetTDs_WriteTP* block then gets these TDs, appends a *Dependency Counter* (DC) to each one of them, and stores them in the *Task Pool* (TP). The DC records how many dependencies must be fulfilled before this task can be scheduled to run. After that, the *CheckDeps* block in the TM decides whether the TDs are ready to run or not. The task dependency graph is stored inside the *DependencyInfo* tables. The dependency resolution process is described in Section II-B to emphasize on its capabilities and efficiency. If a task is ready, it will be passed to the *Schedule* block which will schedule it to one of the worker cores, by writing it to the *ready-tasks-list* of that core. Writing a *ready-tasks-list* will trigger the *GetReadyTask* block at the TC's side. The TCs are responsible for communication with the TM, and to enable buffering of tasks, according to the TC's buffering depth (BD). If BD equals m, the TC will buffer $m - 1$ tasks while the worker core is busy executing one task. A BD of 2 implies double buffering, whereas a BD of 1 implies no buffering. The TC's *GetReadyTask* block asks the TM for the ready TD, and the TM's *HandleTC_Requests* block answers the TC's request and sends the requested TD to it. This communication mechanism is necessary in order not to block the TM's work by a TC, since the latter might have no empty slots in its buffer. After that the TC gets the new task's inputs, assigns the task to the associated worker core to execute it, and finally, writes the outputs back and notifies the TM of task completion. The TM's *HandleFinishedTasks* block receives the TC's notification and checks whether there are pending tasks in the TP that are waiting for the finished task. If it finds any pending task that does not depend on other tasks, it passes it to the *Schedule* block, and so on.

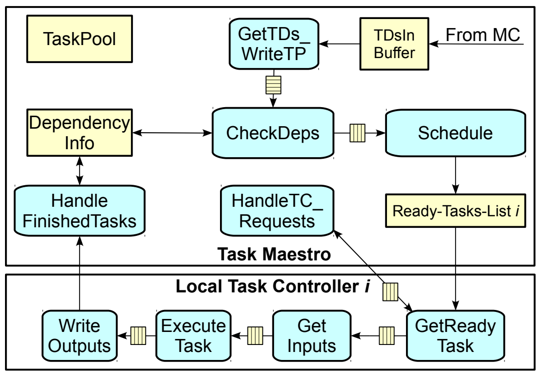

Fig. 2. Nexus++ modules block diagram

Communication and synchronization between the different hardware blocks inside the TM are done using FIFO lists. This is important to pipeline the work of the different blocks. If a FIFO is full, the hardware block writing to it stalls. Furthermore, since tasks are processed in the serial order of execution, a deadlock is not possible.

Inside Nexus++, any task is referred to by the index at which its TD is stored in the TP. This reduces the size and access time of the different tables and FIFO lists. Furthermore all events and notifications are one-bit signals, which ensures low communication overhead between the TM blocks, the TC blocks, and of course between the TM and the TCs.

Both Nexus and Nexus++ provide dependency resolution. However, Nexus can only deal with tasks with a limited number of inputs/outputs, and with a limited number of tasks that depend on those tasks. In addition, Nexus proposed TCs, but did not implement them. Nexus++ solves the above limitations as described next.

B. Dependency Resolution

Dependency resolution is accomplished using two hash tables along with the DC associated with every TD in the TP. The tables are called *Read Only Table* (ROT) and *Read Write Table* (RWT). They store information about task parameters that are read-only and read-write, respectively. Access mode of an input is extracted from the TD. Fields of the ROT and RWT tables are shown in Tables I and II, respectively. To resolve hash collisions, a simple separate chaining hash resolution algorithm $h()$ is used.

TABLE I
READ-ONLY TABLE (ROT)

Memory Address	RC	aWriterWaits
B	2	true

TABLE II
READ-WRITE TABLE (RWT)

Memory Address	Kick-Off-List
A	T_2, T_5, ...
B	T_4, T_6, ...

TABLE III
NEXUS++ VS. NEXUS COMPARISON

	Nexus++	Nexus
Number of tables	2	3
No. Kick-Off-Lists per Mem. Addr	1	2
Access all tables per check	not always	always

To decide that a task T_2 depends on T_1, currently we assume that the base address of an input of T_2 matches the base address of an output of T_1, etc. Write-after-write hazards are handled as follows: if T_2 needs to write memory address A, and A is already in the RWT, i.e. A is the output of another task, say T_1, then T_2 is registered on the Kick-Off-List of RWT[$h(A)$] and the DC of T_2 in the TP is incremented. There is no need to check the ROT since A was found in the RWT. When T_1 finishes, the *HandleFinishedTasks* block is triggered, it accesses the RWT and checks the Kick-Off-List of A, it reads T_2, looks up TP[T_2] and decrements its DC. If the DC is zero, then T_2 can be scheduled to be run. Since T_2 will write A, no more tasks are read from the Kick-Off-List of A. In the previous example, assuming that T_2 is a reader of A shows how the read-after-write (RAW) hazards are handled.

Similarly, if a task T_3 needs to read-only memory address B, and B is already in the ROT, then the readers counter (RC) of B in the ROT is incremented, and the DC of T_3 is not altered, so if it is zero, T_3 can be scheduled to run. On the other hand, if a task T_4 needs to write B, then the TM sets the flag *aWriterWaits* of ROT[$h(B)$] to true, creates an entry for B in the RWT and registers T_4 to the Kick-Off-List of RWT[$h(B)$], and finally increments the DC of TP[T_4]. Any task that comes after T_4 that wishes to read or write B registers itself to the Kick-Off-List of RWT[$h(B)$] and its DC is incremented. Using the *aWriterWaits* flag resolves the write-after-read (WAR) hazards.

Dependency resolution in Nexus++ is more efficient than that in Nexus [8], since we use fewer and simpler tables and Kick-Off-Lists. Furthermore, when determining the dependencies of a new task or after the completion of a task in Nexus++, the ROT is accessed only if the memory address does not exist in the RWT. In Nexus, on the other hand, all tables are accessed for all kinds of scenarios. Hence, Nexus++ is simpler and performs fewer computations to resolve dependencies. A brief comparison between the two is shown in Table III.

C. Dummy Tasks and Entries

In a task descriptor (TD), a task has a limited number of inputs/outputs, so applications with tasks that have more inputs/outputs can not be executed directly on a system with Nexus. In addition, not all tasks necessarily have a number of inputs/outputs equal to the TD's limit, which could yield a poor memory utilization. We solve this problem by introducing dummy tasks. These dummy tasks will not be really executed, they just take the form of a task by having their own TD in the TP, only to store inputs/outputs that did not fit in the parent's input/output list. Figure 3 shows a scenario to demonstrate the need for dummy tasks. If T_1 has 8 outputs, and a TD can only store 4 of them, then dummy tasks are created having their inputs/outputs as those that did not fit in the parent's (T_1) TD. The same principle can be deployed in the RWT table shown in Table II, where the Kick-Off-List has a limited size, restricting the number of tasks that might depend on a single task's output. As a solution, we add dummy entries to the RWT to extend the Kick-Off-List of a certain RWT entry.

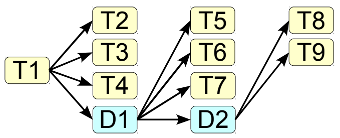

Fig. 3. Dummy Tasks/Entries added to the TP/RWT

Figure 3 can be interpreted as follows: assuming that eight tasks (T_2 - T_9) depend on a single output O of T_1 and given that the size of the Kick-Off-List of the RWT entry RWT[$h(O)$] is 4, then the eight tasks cannot be added to the Kick-Off-List. A stall is not a solution because the RWT might still have empty slots. Furthermore, other tasks cannot be added to the system since they might depend on an output of one of the extra tasks that did not fit in the Kick-Off-List of T_1. Another non-optimal solution is to increase the size of the Kick-Off-List, since it would still limit, although larger, the Kick-Off-List size. It also would waste memory since not all outputs will be inputs for a number of tasks that is equal to the Kick-Off-List size. Furthermore, implementing the worst-case scenario is not scalable. Our proposed solution is, therefore, to insert some dummy entries in the RWT, namely D_1 and D_2, resulting in a chain of Kick-Off-Lists.

In Figure 3, having only one dummy entry per Kick-Off-List is efficient, since when reading a Kick-Off-List, the TM might read only one entry (in case of a writer task), or more. Reading only one entry from the Kick-Off-List chain is simply reading the first entry of the first Kick-Off-List in the chain. On the other hand, allowing more than one dummy entry per Kick-Off-List will result in multiple lookups in order to reach the leaf Kick-Off-List, which is an increase in the overhead.

The compiler could also add dummy tasks when it discovers that a task has more inputs/outputs than the maximum. However, the master core then would have to generate and submit more TDs, and our results indicate that eventually the master core forms the bottleneck. Furthermore, the compiler can not add dummy entries since it depends on runtime information which is not available to the compiler. For these reasons we have decided that the TM adds dummy tasks and entries.

III. EXPERIMENTAL SETUP

A. Benchmarks

To evaluate Nexus++ we used several benchmarks. First, we used a trace of parallel H.264 decoder decoding one

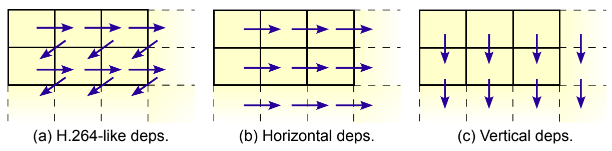

Fig. 4. Dependency patterns: (a) Ramp effect, (b, c) Fixed # of parallel tasks

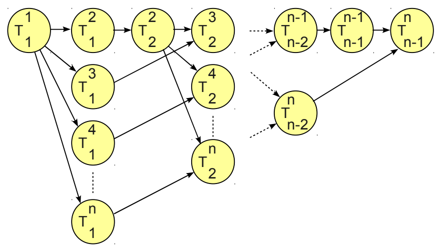

Fig. 5. Dependency pattern for the Gaussian elimination benchmark.

full HD frame, consisting of 8160 tasks in total. A trace consists of tasks execution times and the time they have spent reading/writing their inputs/outputs from/to memory. On average a task spends $7.5\mu s$ on accessing off-chip memory and $11.8\mu s$ on execution [2]. The benchmark processes a matrix of 120×68 macroblocks and the dependency pattern is shown in Figure 4(a) [11]. Tasks are generated from left to right and from top to bottom. Initially there is only one task ready for execution, but this number increases until halfway execution, after which it decreases again. This ramp effect influences the amount of parallelism in the benchmark and thus its scalability.

To evaluate Nexus++ for a range of dependency patterns, we created two additional synthetic benchmarks derived from the H.264 benchmark. The dependency patterns of these benchmarks are shown in Figure 4(b) and (c). We also used an additional benchmark without dependencies, which is not shown in the figure. In contrast to dependency pattern (a), the dependency patterns depicted in 4(b) and (c) do not suffer from the ramping effect. In (b), however, the dependency pattern has the same direction as the order in which tasks are generated. As a consequence, the amount of effective available parallelism could be reduced by the speed of the addition process or the size of the TP (and other tables) , since when the tables are full, tasks of the first row have to be executed to make room for other tasks.

To validate the dummy tasks/entries approach, the task graph of Gaussian elimination with partial pivoting [12] is used. In this benchmark, the number of tasks that depend on certain outputs depends on the size of the input matrix as depicted in the dependency pattern of Figure 5, assuming an $n \times n$ matrix.

B. Simulation Environment

Nexus++ was simulated using SystemC. It is designed to match modern real systems. First of all, the clock cycle time is 400 *ps*, which equals a clock frequency of 2.5 *GHz*. The TM tables and the FIFO lists are on-chip storage and therefore their access times are relatively fast. The hash table access time equals the on-chip access time multiplied by the number of lookups required per access. As the benchmarks are trace-based, task execution is modeled by a wait.

TABLE IV
SYSTEM PARAMETERS

System Parameter	Value
Clock Frequency	2.5 GHz
On Chip Access Time	6 cycles
No. Parameters per TD	8
TD size	96 Byte
TP size	384 KB (4K TDs)
Kick-Off-List Size	8
RWT entry size	64 Byte
RWT size	512 KB (8K entries)
ROT entry size	24 Byte
ROT size	192 KB/(8K entries)

The sizes of the TM tables were empirically determined. We observed that for the current benchmarks the task pool should be able to contain 4k task descriptors. Assuming 8 parameters per task descriptor yields a task pool size of 384 KB. The sizes of the other tables and lists, as indicated in Table IV, were derived in a similar way. These parameters are fully configurable.

The access time for the 192 KB, 384 KB, and 512 KB on-chip memories was determined using Cacti 5.3 [7], for 45nm eDRAM technology, and were found to be 6 cycles for each of them. The latency of preparation and submission of task descriptors by the master core was estimated. These times were measured in Nexus in detail. As Nexus++ avoids off-chip communication in this part, we had to compensate for this. As a result, the task preparation was set to 100 cycles while the task submission has a latency of 60 cycles. The latter depends on the size of the input/output list of a task, and is therefore not a fixed number.

IV. Evaluation

Nexus++ was tested under different conditions, varying the number of worker cores, the buffering depth, and with different dependency patterns.

Figure 6 depicts the utilization for the benchmarks with independent tasks and for different number of cores and buffering depths. A core's utilization equals its computation time divided by (computation time + communication time with the TM and memory), and we report the average utilization. The baseline experiments were performed without using any of Nexus++ enhancements, i.e., no buffering of task descriptors between the MC and the TM, no buffering of tasks at the TC side, and no dummy tasks or entries. For the other experiments, these features are enabled.

Figure 6 shows that the utilization increases from 60% to 98% when enabling double buffering (*BD = 2*) for up to 64 cores. This baseline results differs (up to 60% utilization) from those shown in [8], because the average task computation and communication times are different. In our system, based on [2], we assume these values to be 11.81 μsec and 7.5 μsec, respectively, while in [8] they are assumed to be 19 μsec and 2 μsec, respectively. Furthermore, in [8], on-chip access time equals 2 cycles which is one third of that in Nexus++.

The utilization does not increase when increasing the buffer depth. On the contrary, the utilization slightly decreases as is most visible in the 128-core experiment. Since the TM stalls when it fills all buffers of the worker cores, increasing the

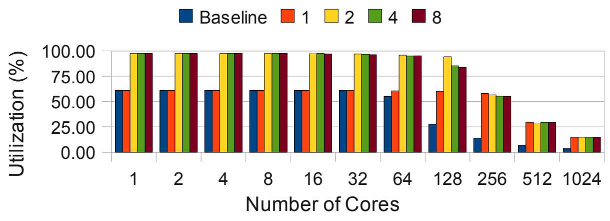

Fig. 6. Utilization for different number of cores with different buffer depths running independent tasks.

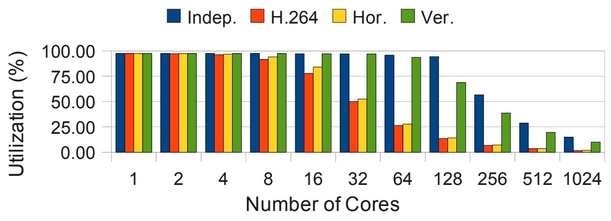

Fig. 7. Utilization of different number of cores running tasks with dependencies shown in Figure 4.

buffer depth increases the number of accesses to the different TM tables and FIFOs, mainly the TP, the RWT, and the ROT. This increases the number of collisions when accessing the different hash tables and, hence, increasing the search time. It can therefore be concluded that the experiment with double buffering (*BD = 2*) is sufficient and most efficient.

In Figure 6, the only difference between the baseline experiment and the experiment with (*BD = 1*) is that TD buffering is disabled in the baseline experiment. The effect of this can be seen when the number of worker cores is larger than 128, which demonstrates how TD buffering between the MC and the TM improves the scalability to larger numbers of cores.

Figure 7 shows the utilization for the benchmarks illustrated in Figure 4. As before, we simulate 8160 tasks with execution and communication times obtained from a parallel H.264 decoder. Furthermore, since the previous experiments has shown that double buffering is most efficient, we simulate a buffering depth of 2. Limited application scalability explains why the utilization decreases faster for the H.264 benchmark compared to the utilizations shown in Figure 6. More interesting is the difference between the benchmarks with horizontal and vertical dependencies illustrated in Figures 4(b) and 4(c), respectively. Although the TP is larger than a single row, the processing of non-ready tasks before reaching the next ready task limits the scalability of this benchmark to at most 32 cores, whereas the benchmark illustrated in Figure 4(c) scales well to 128 cores.

Finally, Gaussian elimination with partial pivoting for a matrix size of 100×100 was tested using Nexus++, and it ran successfully on different number of cores with Kick-Off-Lists of size 8. For such a matrix size to run without dummy tasks, the Kick-Off-Lists should be of size 99, which means 12.5 times the Kick-Off-Lists size. Furthermore, this number increases with the matrix size.

V. Conclusion

We have presented Nexus++, a hardware management accelerator for the StarSs runtime system. Compared to previous work Nexus++ makes three main contributions. First, it overcomes the limitation of Nexus that only a fixed, limited number of tasks can depend on a certain task by introducing dummy tasks and dummy entries in the kick-off lists. Second, it support double buffering by providing a task controller in each worker core. Third, it implements task dependency resolution more efficiently, since fewer hash table lookups are required to determine if tasks depend on each other.

Experimental results obtained using a SystemC model show that double buffering increases the utilization from about 60% to almost 100% for a benchmark modeled after H.264 decoding, while triple and higher buffering slightly decrease the utilization. Furthermore, double buffering also increases the scalability of the system. Eventually, for large (256 cores and more) systems, the utilization starts to decrease, mainly because the application does not exhibit sufficient task-level parallelism, and/or because the master core cannot generate tasks fast enough to keep all worker cores busy. We have also shown that a benchmark modeled after Gaussian elimination, where the number of tasks that depend on a certain task is not constant, ran successfully and efficiently.

Although Nexus++ targets StarSs applications, parts of it can be reused for other programming models. For example, it contains hardware queues that can be used for low-latency retrieval of independent tasks. Future work will focus on how to make Nexus++ more versatile.

Acknowledgment

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the ENCORE Project (www.encore-project.eu), grant agreement n° 248647.

References

[1] G. Al-Kadi and A. S. Terechko. A Hardware Task Scheduler for Embedded Video Processing. In *Proc. 4th Int. Conf. on High Performance Embedded Architectures and Compilers*, 2009.
[2] C. C. Chi, B. Juurlink, and C. Meenderinck. Evaluation of Parallel H.264 Decoding Strategies for the Cell Broadband Engine. In *Proc. 24th ACM Int. Conf. on Supercomputing*, 2010.
[3] L. Dagum and R. Menon. OpenMP: an Industry Standard API for Shared-Memory Programming. *IEEE Computational Sci. Eng.*, 1998.
[4] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. 6th Symp. on Operating Systems Design & Implementation*, 2004.
[5] Y. Etsion, A. Ramirez, and R. M. B. Jesuslabarta. Cores as Functional Units: A Task-Based, Out-of-Order, Dataflow Pipeline. In *Proc. Int. Summer School on Advanced Computer Architecture and Compilation for Embedded Systems*, 2009.
[6] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. In *Proc. 34th Annual Int. Symp. on Computer Architecture*, 2007.
[7] H. Laboratories. Cacti 5.3. http://www.hpl.hp.com/research/cacti/.
[8] C. Meenderinck and B. Juurlink. A Case for Hardware Task Management Support for the StarSS Programming Model. In *Proc. 13th Euromicro Conf. on Digital System Design: Architectures, Methods and Tools*, 2010. Sp. Session on Multicore Systems: Des. and Apps.
[9] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical Task-Based Programming With StarSs. *Int. J. High Perf. Comp. Appl.*, 2009.
[10] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., 1st edition, 2007.
[11] E. B. van der Tol, E. G. Jaspers, and R. H. Gelderblom. Mapping of H.264 Decoding on a Multiprocessor Architecture. In *Proc. SPIE Conf. on Image and Video Communications and Processing*, 2003.
[12] M. Veldhorst. Gaussian Elimination with Partial Pivoting on an MIMD Computer. *Journal of Parallel and Distributed Computing*, 1989.