LOW-COMPLEX TASK SCHEDULING ALGORITHMS FOR HIERARCHICAL EMBEDDED MANY-CORE ARCHITECTURES AND DYNAMIC APPLICATIONS

Alexandre Guerre and Nicolas Ventroux and Raphaël David
CEA LIST,
Embedded Computing Laboratory
PC 94, Gif-sur-Yvette, F-91191 France
email: name.surname@cea.fr

Alain Mérigot
Institut d'Electronique Fondamentale
Université Paris Sud
Orsay, F-91405 France
email: alain.merigot@u-psud.fr

ABSTRACT

Embedded systems require more and more computational power. Moreover, embedded applications are becoming data-dependent and their execution time depends on their input data. Only a dynamic global scheduling can balance the workload on the computation resources and reach good performances. Thus, a solution to address this problem is to use many-core architectures with a dynamic and centralized control. In this article, we propose new on-line scheduling algorithms adapted to hierarchical many-core embedded systems. The proposed algorithms reduce communications between clusters in order to increase global performance. This paper highlights the good results of a scheduling algorithm named *Static Clustering Dynamic Mapping*. It consists in dividing the application graph off-line and dynamically allocating each part on each cluster.

KEY WORDS

Task scheduling, Embedded system, Many-core architecture, clustering algorithm

1 Introduction

Nowadays, embedded systems need more and more computing power. One solution, widely used to provide more computing power under real-time and energy consumption constraints, consists in using multiprocessor systems on chip (MPSoC) [1]. In this context, according to ITRS [2], a 32% yearly increase in the number of cores will be necessary to keep up with the applications' needs. ITRS assesses that in 2012, the number of cores in a chip will exceed 100 cores. In addition, these many-core architectures must support the execution of dynamic computationally-intensive applications. Algorithms become highly data-dependent and their execution times depend on their input data. Consequently, on a many-core platform, optimal static scheduling cannot exist since all the processing times depend on the given data because static algorithms will use worst case exection to calculate the scheduling. The only optimal solution consists in dynamically allocating tasks according to the availability of computing resources [3]. Global scheduling allows maintaining the system load balanced and supports workload variations that cannot be known off-line. Thus, in the future, many-core architectures have to support dynamic applications and to provide high performance in different application domains.

The many-core architectures that exist todaty cannot meet these specific needs. For example, the TESLA S1070 architecture from Nvidia [4] or the Grape-DR processor from the university of Tokyo [5] have been designed for graphical 3D processing or scientific applications. These processors massively use data parallelism but cannot address over processing with respect to embedded computing constraints. Besides, Am2045 from Ambric [6], or ASAP2 from the university of California [7] and PC 102 from Picochip [8], are respectively designed for image processing and telecommunication. Unfortunately, these architectures use dedicated processors or cannot support dynamic applications since their programming model is based on a static partitioning of tasks. On the contrary, the HL-256 architecture from Plurality [9] is based on a dynamic approach, but only proposes a Single Program Multiple Data (SPMD) parallelism. It cannot exploit multiple concurrent executions of tasks. This limits the number of supported applications and does not allow the execution of multiple applications in parallel.

These many-core architectures can be Uniform Memory Access (UMA) or Non Uniform Memory Access (NUMA) machines. This difference will influence the control strategy. Producing a UMA machine with hundreds of processors involves a heavy interconnection in order to keep high-performance interconnection. NUMA machines can be clustered or not. Programming a nonclustered architecture is a hard job, whereas clustered achitectures are more easily programmable. Also clustered achitectures reduce the communication latency and therefore increase the performance.

In this paper, we propose multiple dynamic global task scheduling algorithms on a clustered architecture, in order to support multiple domain applications and multi-application execution in an embedded environment. To the authors' knowledge, no previous work proposed a global and dynamic control of tasks for clustered many-core architectures. The dynamic task scheduling problem is known to be NP-complete [10]. Here, we compare a few heuristics to find the more efficient one for NUMA homogeneous many-core on-chip.

Many embedded scheduling algorithms already exist for multicore architectures but they are not easily scalable for many-core architectures. In section 2, we introduce the many task scheduling problem on embedded systems in analogy with off-chip many processors scheduling. Then a description of the architecture context and the various new algorithms are presented in section 3. Section 4 explains the different metrics used and the comparison results. Finally, section 5 concludes this study and discusses the perceptives of our work.

2 Previous Works

The problem of scheduling and mapping thousands of tasks on many processors is not new [11, 12]. Indeed, high performance computing (HPC) has been facing the same problem for years. In the case of embedded systems, all these processors are on one chip so the two problems have some differences. In comparison to HPC, embedded systems are limited in their power consumption and their area. This will have a strong impact on the task control complexity. Moreover, the scale is different, for instance, a communication between two processors on one chip will be smaller than between two processors on different chips.

In the HPC world, scheduling/mapping techniques are divided into two general categories: off-line and online techniques. As we explained in the introduction offline scheduling does not offer an optimal solution for dynamic applications. Kwok and Ahmad propose in [13] a large list of static scheduling strategies and compare them in [14]. On-line scheduling/mapping proposes dynamic distribution of tasks on processors. So on-line scheduling matches the application dynamism. This paper focuses on on-line scheduling. On-line algorithms can be split into two main categories: list scheduling and cluster scheduling [15].

List scheduling is a similar technique to the one used to solve the problem of on-line bin packing [16]. This technique is divided into two phases. First, we consider a task as ready when all precedent tasks are completed. The first phase is to sort ready tasks depending on criteria like a priority or precedent tasks. In the phase, tasks are put in order to be choosen for the second phase i.e., mapping. Mapping consists in making a selection of ready tasks for each free processor. List scheduling lets two free degrees: the sorting algorithm and the mapping choice. This scheduling is mostly used in embedded systems because it is the simplest to implement and also because no specific mapping is required on a UMA machine [17].

Clustering techniques have also two phases. First, the application is divided into different virtual clusters of tasks. The number and the size of these virtual clusters depend on the clustering technique. To explain, we will detail linear

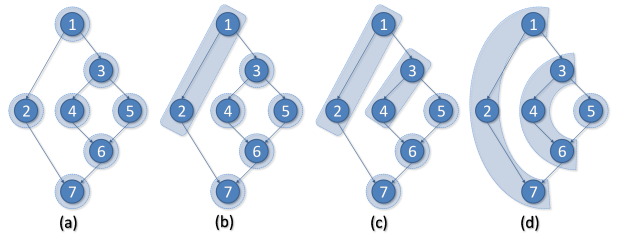

Figure 1. Linear clustering.

clustering. Figure 1-a is the first step of the linear clustering, each task is placed in a different virtual cluster. Figure 1-b and Figure 1-c show intermediate steps. Depending on the clustering criteria, virtual clusters are merged until all tasks have been considered. These criteria are, for example, the size of the data transfer between two tasks or the task graph form. In our case, linear clustering only considers the size of data transfer. Figure 1-d shows final virtual clusters. For the second phase, virtual clusters are mapped on processors. After this phase, the problem is to schedule ready tasks on each processor. Phase 1 and phase 2 can be done dynamically. Clustering scheduling has three free degrees. The added degree is the ability to differently divide the task graph. Clustering algorithms are not usually used in MPSoC even if in [18], Wenzel et al. propose to use linear clustering to reduce power consumption.

All list and clustering schedulings can not always provide an equal computational load on all processors because of the NP completeness of this problem or their choices in their algorithms. The more the number of processors, the more difficult the problem is. So, in different cases, load balancing is used to correctly balance the load on processors [19]. It can dynamically adapt off-line decision, in the case of clustering algorithms. This function is done separately or not from the scheduling/mapping phase. To realise this load balancing, tasks are moved between processors: this is called task migration. Task migration can be performed during the task execution or before its start.

As we focus on embedded systems, genetic algorithms [20] and replication/duplication algorithms [21] will not be considered in this comparison because of their complexity, the difficulty of their on-line implementation and their over utilization of processors. List algorithms and clustering algorithms have different criteria that have been chosen. The next section presents all studied algorithms in order to make this choice.

3 Scheduling algorithms for dynamic applications

As mentioned in the last section, two types of algorithms can be distinguished. In this section, some well-known list algorithms are presented and will serve the comparison, and some new propositions are made. These algorithms aim to decrease the total execution time (makespan) and hence the total communications cost. Before presenting the

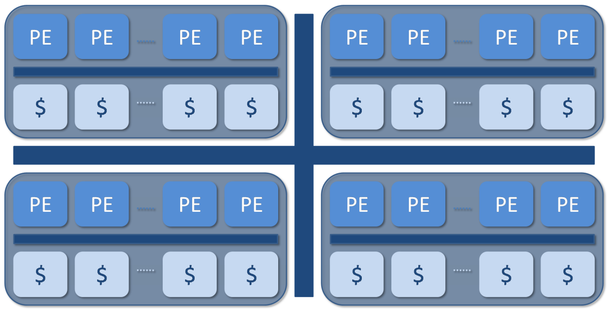

Figure 2. System architecture.

algorithms, it is necessary to specify some terms.

In this paper, we consider that any application can be described by a directed acyclic graph $G = (T, E)$. T represents a finite set of t tasks and E is a finite set of e edges. We consider the edge $e_{i,j}$ as the precedence constraint between the tasks t_i and t_j. In a directed graph, each edge has a unique direction so $e_{i,j} \neq e_{j,i}$. An edge $e_{i,j}$ has a communication cost and $|e_{i,j}|$ corresponds to an image of the data volume $D_{i,j}$. The worst execution time of each task t_i is known as $|t_i|$. In this problem, we suppose that the exact execution time of the task is not known before the end of the task even if the execution time is bounded. A task t_i is executed by one processor without preemption and cannot be distributed on a different one to decrease its execution time (no malleable tasks).

The architecture is composed of a set of processors P distributed in a set of clusters C and presented on figure 2. The architecture is fully connected and communications are concurrent, i.e. each processor p_i can communicate with all processors p_j without contentions. $\forall p_i \in P$, it can execute only one task simultaneously. An application A can be only executed on a set P', $P' \in P$. Communications are made by a dedicated system and all communications are not equal. The machine is a clustered NUMA system where we consider that communications inside the cluster are less expensive than communications outside the cluster. So the system is UMA in a cluster and NUMA outside. $com_{i,j}$ is a communication between processors p_i and p_j and $|com_{i,j}|$ is its cost. It is calculated like below $\forall com_{i,j}\ ,\ p_i \in c_k\ and\ p_j \in c_l$:

$$|com_{i,j}| = \begin{cases} const & \text{if } c_k = c_l \\ k * distance(c_k, c_l) * D_{i,j} & \text{if } c_k \neq c_l \end{cases}$$

where k and $const$ are parameters depending on the network features. Contentions in the network are not taken into account.

Below is a description of each algorithm. The first paragraph describes four well-known algorithms. The other paragraphs detail our proposed algorithms. These algorithms decrease the computation time by reducing the number of intra-cluster communications.

3.1 Existing list scheduling algorithms

The *First-Come-First-Serve* (FCFS) algorithm schedules tasks without sorting them according to their arrival date. The first task in the list is executed on the first available processor.

The *Randomized* (Rand) algorithm consists in randomly selecting a task in the list for each available processor. This scheduling does not take into account any information about the task or the architecture.

The *Priority* algorithm schedules tasks depending on a static priority established off-line. $\forall t_i \in T, p(t_i)$ represents its priority. This list scheduling algorithm orders tasks according to their priority. So when a processor is ready, the task with the highest priority is executed on it.

The *Longest-Task-First* (LTF) and the *Shortest-Task-First* (STF) scheduling algorithms take into account the worst case execution time to order tasks. The LTF sequentially packs the remaining tasks having the longest running time into the schedule. On the contrary, the STF algorithm consists in sorting tasks by their non-decreasing time of execution. The allocation of tasks, for both algorithms, selects the first available processor.

3.2 New list scheduling algorithms

Our new algorithms increase performances by reducing the number of intra-cluster communications, which heavily impacts the total execution time. We propose three new algorithms named *Most-Close-Affinity-Choice* (MCAC), *Most-Global-Affinity-Choice* (MGAC) and *Less-Communication-Cost* (LCC). Within these solutions, the scheduling and the allocation of tasks are merged into a unique function. They schedule tasks according to the clusters that execute their previous tasks. No pre-ordering is done before the allocation.

The MCAC minimizes the number of intra-cluster communications with high exchanged data volumes. A weight $D_{k,i}$ is added on each task dependency to represent the exchanged data volume between the task t_k and t_i. Let t_i be the task that is executed on the processor p_j, $d(t_i, p_j)$ the distance with the processor p_j, and $w(t_i, p_j)$ the total data volume that would be transferred between all the previous tasks of t_i and the task t_i executed on the processor p_j. Consider p_j a ready processor of P and $p_j \in c_l$,

$$w(t_i, p_j) = \sum_{t_k}^{previous\_task(t_i)} a(t_k, t_i)$$

with

$$a(t_k, t_i) = \begin{cases} D_{k,i} & \text{if } t_k \text{ was executed in } c_l \\ 0 & \text{if } t_k \text{ was not executed in } c_l \end{cases}$$

and

$$d(t_i, p_j) = \sum_{t_k}^{previous\_task(t_i)} distance(c_l, c_m)$$

with c_m the cluster where t_k was executed. A first choice is made with the highest weight coefficient. If all weight coefficients equal zero, the scheduling chooses the lowest distance coefficient.

The MGAC merges the two coefficients $w(t_i, p_j)$ and $d(t_i, p_j)$ into one named $W(t_i, p_j)$, in order to select in the ready-task list, for a designed available processor, the task that has the lowest global communication cost. The goal of this scheduling algorithm is to minimize the cost of data transfers. Consider p_j a ready processor of P and $p_j \in c_l$,

$$W(t_i, p_j) = \sum_{t_k}^{previous\_task(t_i)} \frac{D_{k,i}}{distance(c_l, c_m)}$$

with c_m the cluster where t_k was executed.

The LCC decreases the total data transfer cost by using an estimation of each communication cost. This scheduling is aware of the network structure and the cost of each data transfer. So this list scheduling uses a function to estimate for each task, the potential communication cost and chooses the task in the waiting list that offers the lowest cost $|com_{i,j}|$.

3.3 Static clustering algorithms

Three new algorithms are based on the same static clustering method. These techniques create off-line different lists for each cluster in the system. Some different steps are necessary. The first step is to divide the application graph in an unbound number of virtual clusters. These virtual clusters are a linear group of successive tasks. This cutting up is a linear clustering as described before with a data transfer criterion. When a divergence occurs, the next chosen task is the task with the most important $|e_{i,j}|$ (which represents the weight of the data transfer). Other tasks are placed in other virtual clusters. This choice maximizes the communication value within the cluster. The second step consists in allocating all virtual clusters and depends on the selected algorithm.

The *Static-Clustering-Static-Mapping* (SCSM) algorithm merges virtual clusters to obtain a number equal to the cluster number. During the execution, a local scheduling chooses tasks in its assigned virtual cluster. Communication costs are constant in a cluster, and the local scheduling is an LTF algorithm because of its good performance for UMA machine [17]. This scheduling is limited because it is a static mapping.

The *Static-Clustering-Static-Mapping-with-Migration* (SCSMM) algorithm adds a load balancing function to the SCSM algorithm. This function is executed when a cluster has too many ready tasks in its waiting task list. The function looks at the list and selects the task with the higher communication cost. Then, it calculates the hypothetical communication costs if this task is migrated ot the other clusters and chooses the cluster where the communication cost is minimum. Finally, this function moves the task to te selected cluster. The migration of the task is done before the begining of its execution. This scheduling moves tasks one by one.

The *Static-Clustering-Dynamic-Mapping* (SCDM) algorithm merges virtual clusters at runtime. Initially, in each cluster, we allocate l virtual clusters, where l is the number of processors in the cluster. During the execution, new virtual clusters are merged within the cluster, when its list has a number of ready tasks inferior to two times its processor number. The merged virtual cluster is selected according to the MCAC criteria. The local scheduling is also done by a LTF algorithm. This dynamic mapping aims to balance the load on all processors.

3.4 Dynamic clustering algorithms

The *Dynamic-Clustering-with-Migration* (DCM) algorithm dynamically constructs a task-list in each cluster and, because of the unoptimized cutting up, a load balancing function is used. This construction is iterative and uses a non-linear clustering technique. For each task, the allocation selects the cluster where its previous tasks are executed. It favors the allocation of a task on this cluster, while the number of ready tasks is inferior to two times its processor number. The SCSMM migration function is used to balance the workload. In the SCSMM and DCM algorithms, load balancing moves tasks which are not already executed on the cluster. This technique provides migration with the minimum cost.

4 Performance analysis

In order to choose the best scheduling algorithm, a simulation environment was built to compare proposed algorithms. It reads an application graph and apply a scheduling technique on it. Then, for each algorithm, it calculates the makespan and can easily generate statistics like the load of each processor during the execution or the number of communications and their cost. It takes into account the communication cost between two tasks and also a control overhead cp_i. This control overhead is due to the communication latency between the control and the computation part. Because it is a centralized control, each request from a processor is sequentially processed. Moreover, the scheduling process time is function of the depth of the ready-task list. Consequently, the control overhead depends on the number of ready tasks (nb_ready_tasks).

So when the scheduling occurs, $cp_i = m * nb\_ready\_tasks$ with m, a constant that depends on the implementation of the control. To evaluate the performance of each scheduling technique, three metrics are considered. The acceleration $\lambda(H)$ of the scheduling technique H in comparison with a FIFO scheduling (FCFS) is the first benchmark. If we consider $H(G)$ the makespan of the scheduling technique H applied on a graph G, $\lambda(H)$ is equal to $\frac{FCFS(G)}{H(G)}$. The second metric is the average workload of all processors, and the last one is the volume of communications between clusters.

4.1 Comparison condition

To do these comparisons, application graphs are necessary. To generate these graphs, we used Task Graph For Free (TGFF) [22], which is an open source application that generates graphs based on a parameters file. To provide enough workload for 128 processors, we need massively parallel applications. Their graphs have to be composed of more than 100,000 tasks. Thus, their generation cannot be done by hand. All graphs used in this article are generated by TGFF and we add execution time information and priorities needed by our simulation environment. Execution times and priorities are randomly generated with a uniform distribution.

To explore different solutions of applications, we will consider two different graphs to evaluate these scheduling techniques. The first one is a parallel graph. This graph has long successive task branches. It represents an application with less dependencies and less control constraints. The second one is a highly dependent graph. In this case, the graph represents an application with many dependencies which constrains the execution of each tasks. These two graphs represent a regular and an irregular application.

The simulations use a constant number of processors and these two different graphs. Each result is an average of 10 experimentations with randomly generated execution times and priorities.

4.2 Performance comparison with parallel graphs

We consider an outside communication when a processor reads or writes a memory outside of its cluster. According to figure 3(c), our algorithms divide the number of outside communications for a parallel graph by 10.

Clustered algorithms have a better acceleration than the others because they are more adapted for this kind of graphs. In fact, their virtual clusters fit with the parallel branches of the graph which brings together dependant tasks. Figure 3(a) shows the evolution of the acceleration $\lambda(H)$ depending on the number of clusters with a constant number of processors. Whatever the number of clusters, is the SCDM scheduling algorithm obtains the best result.

Besides, list scheduling algorithms are penalized by their bigger control cost, as depicted on figure 3(e) that represents the average processor load. It can be easily explained, since they use a uniquely centralized control. However, our list scheduling algorithms obtain lower communication costs than clustering algorithms. Indeed, task clustering does not take into account all communications costs between precedence tasks, since the clustering technique linearly selects tasks into the graph. So, list scheduling algorithms are more accurate during the allocation decision than clustering approaches.

We can notice that Dynamic Clustering has really bad results on these graphs because it cannot correctly distribute tasks on clusters. The migration function cannot successfully balance the workload on each cluster. In the case of the SCSMM algorithm, migration does not offer a significant improvement. Finally, the use of task migration is not necessarily the best option.

4.3 Performance comparison with highly dependent graphs

According to figure 3(d), the proposed algorithms do not have the same performance for a highly dependent graph and for a parallel graph, but they still divide by 2 the number of outside communications in the best case. Because, tasks are very constrained, it is difficult to place all tasks on a cluster and this generates many intra-cluster communications.

As depicted on figure 3(b), the best acceleration is obtained with our list scheduling algorithms. Indeed, clustering algorithms are penalized because of their early allocation, while keeping good performances.

4.4 Synthesis

This study shows the good performance of clustering algorithms for parallel graphs and list scheduling for highly dependent graphs. Regardless of the type of graph, the Static Clustering Dynamic Mapping (SCDM) algorithm seems to be a good trade-off for regular and irregular applications. It allows to considerably reduce the number of intra-cluster accesses, while keeping high workload on all processors in order to increase the performance.

5 Conclusions

This paper presented few algorithms for hierachical embedded many-core architectures and dynamic applications. Only global scheduling approaches can take into account the application dynamism. But, this is a NP complete problem. We compared different dynamic scheduling policies applied to highly dependent and parallel graphs. These proposed algorithms are devoted to clustered architectures and reduce the number of communications between clusters to increase the performance of the architecture. We have shown in this paper that the Static Clustering Dynamic Mapping (SCDM) algorithm offers the best trade-off thanks to a hierarchical control and a dynamic task allocation policy. For the moment, tests have been done on synthetic application graphs. Work to come will focus on adding other tests coming from real applications.

References

[1] W. Wolf, A. Jerraya, and G. Martin, “Multiprocessor System-on-Chip (MPSoC) Technology,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2008.

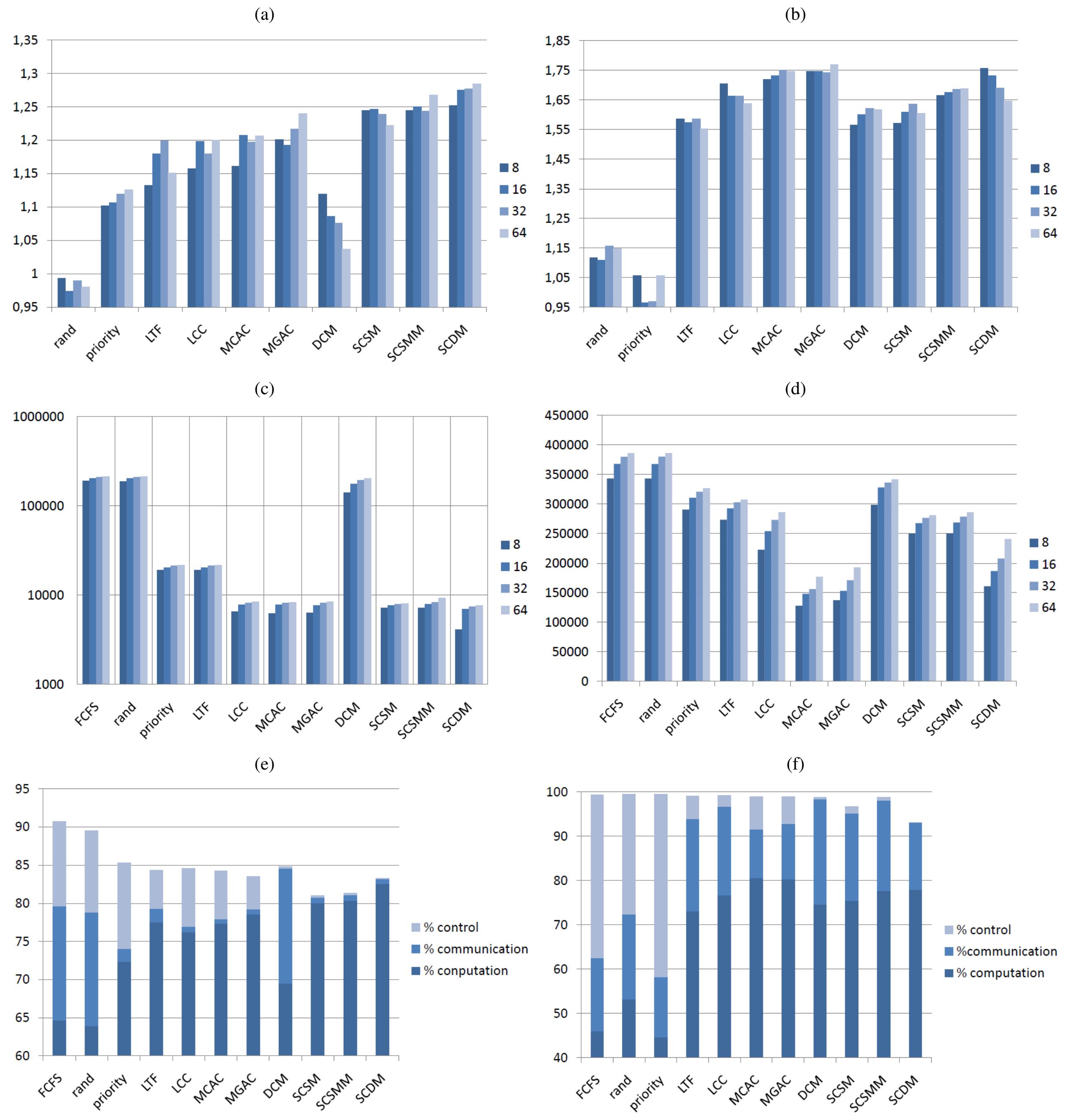

Figure 3. Results on (a), (c) and (e) are for a parallel graph. (b), (d) and (f) are for a highly dependent graph. For (a), (b), (c) and (d), each scheduling is tested with a 8, 16, 32 and 64 clusters. (a) Acceleration $\lambda(H)$ for 128 processors on parallel graph. (b) Acceleration $\lambda(H)$ for 128 processors on highly dependent graph. (c) Total number of outside communication for 128 processors on parallel graph. (d) Total number of outside communication for 128 processors on highly dependent graph. (e) Average processor load for 128 processors in 32 clusters on parallel graph. (f) Average processor load for 128 processors in 32 clusters on highly dependent graph.

[2] Semiconductor Industry Association, "International Technology Roadmap for Semiconductors," 2005.

[3] M. Bertogna, M. Cirinei, and G. Lipari, "Schedulability analysis of global scheduling algorithms on multiprocessor platforms," *IEEE Transactions onParallel and Distributed Systems*, vol. 20, no. 4, pp. 553–566, 2009.

[4] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[5] J. Makino, "The Grape Project," *Computing in Science & Engineering*, vol. 8, no. 1, pp. 30–40, 2006.

[6] T. R. Halfhill, "Ambrics New Parallel Processor," *Mi-*

croprocessor Report, 2006.

[7] D. Truong, W. Cheng, T. Mohsenin, Z. Yu, A. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, A. Tran, Z. Xiao, E. Work, J. Webb, P. Mejia, and B. Baas, “A 167-Processor Computational Platform in 65 nm CMOS,” *IEEE Journal of Solid-State Circuits*, vol. 44, no. 4, pp. 1130–1144, 2009.

[8] A. Duller, G. Panesar, and D. Towner, “Parallel Processing – the picoChip way!” *Communicating Processing Architectures*, pp. 125–138, 2003.

[9] N. Bayer and R. Ginosar, “High Flow-Rate Synchronizer/Scheduler Apparatus And Method For Multiprocessors,” *United States Patent*, no. 5,202,987, 1993.

[10] M. R. Garey and D. S. Jonhson, *Computers and Intractability - A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[11] O. Sinnen, *Task Scheduling for Parallel Systems*. John Wiley, 2007.

[12] A. Gerasoulis and T. Yang, “On the granularity and clustering of directed acyclic task graphs,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 6, pp. 686–701, 1993.

[13] Y.-K. Kwok and I. Ahmad, “Static scheduling algorithms for allocating directed task graphs to multiprocessors,” *ACM Computer Survey*, vol. 31, no. 4, pp. 406–471, 1999.

[14] ——, “Benchmarking the Task Graph Scheduling Algorithms,” in *Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*. IEEE Computer Society, 1998, p. 531.

[15] J. Sgall, “On-Line Scheduling - A Survey,” 1997.

[16] C. C. Lee and D. T. Lee, “A simple on-line bin-packing algorithm,” *Journal ACM*, vol. 32, no. 3, pp. 562–572, 1985.

[17] N. Ventroux, F. Blanc, and D. Lavenier, “A Low Complex Scheduling Algorithm for Multi-processor System-on-Chip,” in *Proceedings of Parallel and Distributed Computing and Networks*, 2005, pp. 540–545.

[18] E. Wenzel Briao, D. Barcelos, and F. Wagner, “Dynamic Task Allocation Strategies in MPSoC for Soft Real-time Applications,” in *Proceedings of Design, Automation and Test in Europe*, 2008, pp. 1386–1389.

[19] Y. Azar, A. Broder, and A. Karlin, “On-line load balancing,” in *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, 1992, pp. 218–225.

[20] A. Zomaya, C. Ward, and B. Macey, “Genetic scheduling for parallel processor systems: comparative studies and performance issues,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 8, pp. 795–812, 1999.

[21] N. Auluck and D. Agrawal, “A scalable task duplication based algorithm for improving the schedulability of real-time heterogeneous multiprocessor systems,” in *Proceedings of the International Conference on Parallel Processing Workshops*, 2003, pp. 89–96.

[22] R. Dick, D. Rhodes, and W. Wolf, “TGFF: task graphs for free,” in *Proceedings of the Sixth International Workshop on Hardware/Software Codesign*, 1998.