Algorithms for Scheduling Task-based Applications onto Heterogeneous Many-core Architectures

Michel A. Kinsy
Department of Computer and Information Science
University of Oregon
Email: mkinsy@cs.uoregon.edu

Srinivas Devadas
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Email: devadas@@mit.edu

***Abstract*—In this paper we present an Integer Linear Programming (ILP) formulation and two non-iterative heuristics for scheduling a task-based application onto a heterogeneous many-core architecture. Our ILP formulation is able to handle different application performance targets, e.g., low execution time, low memory miss rate, and different architectural features, e.g., cache sizes. For large size problem where the ILP convergence time may be too long, we propose a simple mapping algorithm which tries to spread tasks onto as many processing units as possible, and a more elaborate heuristic that shows good mapping performance when compared to the ILP formulation. We use two realistic power electronics applications to evaluate our mapping techniques on full RTL many-core systems consisting of eight different types of processor cores.**

I. Introduction

With advances in semiconductor technology multi/many-core microprocessors are being deployed in a broad spectrum of applications: power electronics, geosciences, aerospace, defense, interactive digital media, cloud computing, and bioinformatics. This convergence of application specific processors and multicore systems is entering a new phase where we see two distinct computing architectures emerging: massive parallel homogeneous cores, e.g., Tilera [1] and heterogeneous many-core architectures, e.g., IBM Cell [2]. The Tilera-like homogeneous architectures find most of their application in clusters, clouds and grids computing. They are generally executing applications that are inherently parallel [3]. They are composed of a large number of general-purpose processors like x86, Power, MIPS, or ARM. A homogeneous collection of cores on the die makes the manufacturing and testing process more manageable, and keeps the software support model simple. These architectures have many advantages: they are more programmable and they provide easier import of existing code and compilers onto them. The key disadvantage of massive parallel homogeneous cores architectures is their lack of specialization of processing units to different tasks.

The second paradigm is the heterogeneous many-core systems approach where highly specialized application-specific processor/functional units and general-purpose cores are integrated onto the same chip. They have many advantages, performance and power can be better optimized, specialization of compute units to different tasks promises greater energy/area efficiency. Kumar *et al.* [4] present a heterogeneous architecture which outperforms a comparable-area homogeneous architecture by up to 63%, and with a good task-to-core assignment, they are able to achieve an additional 31% speedup over a naive scheduling policy.

The current trend in system-on-chip (SoC) design is system-level integration of heterogeneous technologies consisting of a large number of processing units such as programmable RISC/CISC cores, memory, DSPs, and accelerator function units/ASIC [5] providing various services. Although the design of heterogeneous many-core systems shares many of the same issues we see in homogeneous general-purpose parallel architectures, there are a few added design and programmability challenges. These functional asymmetric heterogeneous architectures require greater system management. Processing elements need more fine-grained work allocation and load balancing [6], [7]. With the increased adaptation of heterogeneous many-core processors with a number of core types that match the function affinities, we need new techniques for developing, analyzing, and executing software programs on these platforms. For example, task models that characterize the execution time of tasks in heterogeneous hardware environment, task models that partition tasks where different pieces of a task can run on different cores allowing it to improve its performance, and scheduling algorithms and metrics that take into account task dependencies that influence the execution time, runtime control flow and memory access patterns [8].

In this work we present a general framework for decomposing an application into tasks, and propose a set of simple, scalable algorithms for mapping task-based applications onto generic heterogeneous many-core architectures.

II. Related Work

Application scheduling on heterogeneous resources has been shown to be NP-complete in general cases [9], [10], as well as in several restricted cases [11], [12], [13]. Task scheduling on a heterogeneous system can be generally classified into *static* [14] and *dynamic* [15], [16]. With static assignment, task-to-core mapping is done at compile-time offline or once at the beginning of the application, and the schedule is maintained throughout the execution of the application. Compared with dynamic scheduling, task monitoring and migration are used to ensure a certain level of application performance. While dynamic scheduling can potentially achieve better application performance compared to static scheduling, it may become less feasible in large many-core systems. Brandenburg *et al.* [17], for example, consider the issue of scalability of the scheduling algorithms on multiprocessor platforms, particularly in the case of real-time workloads. K. Ramamritham and J. A. Stankovic [18] discuss the four paradigms underlying the scheduling approaches in real-time

systems, namely, static table-driven scheduling, static priority preemptive scheduling, dynamic planning-based scheduling, and dynamic best-effort scheduling. A. Burns [19] also gives a detailed review on task scheduling in hard real-time systems. H. Topcuoglu and M. Wu [20], present two different scheduling algorithms for a bounded number of heterogeneous processors with an objective to simultaneously meet high performance and fast scheduling time. Their algorithms use rankings and priorities to schedule tasks with the objective of minimizing finish times. Arora *et al.* [21] present a non-blocking implementation of a work-stealing algorithm for user-level thread scheduling in shared-memory multiprocessor system. Chaudhuri *et al.* [22] provide in-depth formal analysis of the structure of the constraints, and show how to apply that structure in a well-designed ILP formulation such as the scheduling problem. Yi *et al.* [23] present an integer linear programming formulation for the task mapping and scheduling problem. They use various techniques and architecture characteristics to reduce application execution time. Given an application, Lakshminarayana *et al.* [24] propose an age-based scheduling algorithm that assigns a thread with a larger remaining execution time to a fast core. Shelepov *et al.* [25] propose a heterogeneity-aware signature-supported scheduling algorithm that does not rely on dynamic profiling, where they use thread architectural signatures to do the scheduling.

III. Models of Computations

The problem of finding a schedule for a heterogeneous parallel architecture is complicated by a number of factors: different processing elements, not every processor may be able to execute all processes, the run time of a given process may be different on different processing elements, and communication time between may vary. Before proceeding with discussion of scheduling algorithms, we first give a set of standard definitions.

A. Application Decomposition

In many-core system environments, parallel computing comes naturally. But this parallel computing paradigm also forces the application running to examine the type of parallelism it exhibits. Broadly, an application exhibits some instruction-level parallelism, some data parallelism, and some task parallelism. Task-level parallelism on heterogeneous many-core architectures seems more attractive because it decouples an application expression from the core ISA or the number of cores. This reduces the problem of executing such an application on a given platform to a scheduling problem. For those reasons, we adopt task-based application decomposition and processing unit mapping [14], [20].

B. Task-based application decomposition model

We define an application as a composition of computation tasks (or simply tasks) providing one global compute service. For running such an application in a heterogeneous many-core environment, tasks need to be created, scheduled in time, mapped to the appropriate cores and synchronized.

Definition 1: A task is a computational primitive that operates on a set of inputs $I(i_1, i_2, ..., i_n)$, where i_α is a data or a memory location read by the task. It has a set of outputs, denoted $O(o_1, o_2, ..., o_m)$, where o_α is the data or memory location written in by the task, and an internal working set data w.

Figure 1(a) depicts a generic multi/many-core architecture where each letter represents a particular type of processing unit (core). Cores may vary in frequency, ISA, function units, memory back-end, etc. Figure 1(c) shows an illustrative task graph to be run through the various mapping algorithms. Each task is characterized by its operating data type and runtime (some reference execution time on some idealized machine): ***f*** stands for floating-point operation, ***i*** for integer operation, ***v*** for vector operation, and ***m*** for multiplication/division operation.

Definition 2: There are three elementary tasks: a feedback task, a sequential task, and a parallel task. For generality, they are defined for a given execution time t, but also lend themselves to static application annotation and analysis.

At some time t during application execution, task A exhibits a feedback, denoted $\mathring{A}$,
if $I(i_1, i_2, ..., i_n)^t_A \cap O(o_1, o_2, ..., o_m)^{t-1}_A \neq \phi$.

At some time t during application execution, tasks A and B are sequential, denoted $A \longmapsto B$,
if $O(o_1, o_2, ..., o_m)^t_A \cap I(i_1, i_2, ..., i_n)^t_B \neq \phi$ or $O(o_1, o_2, ..., o_m)^{t-1}_A \cap I(i_1, i_2, ..., i_n)^t_B \neq \phi$.

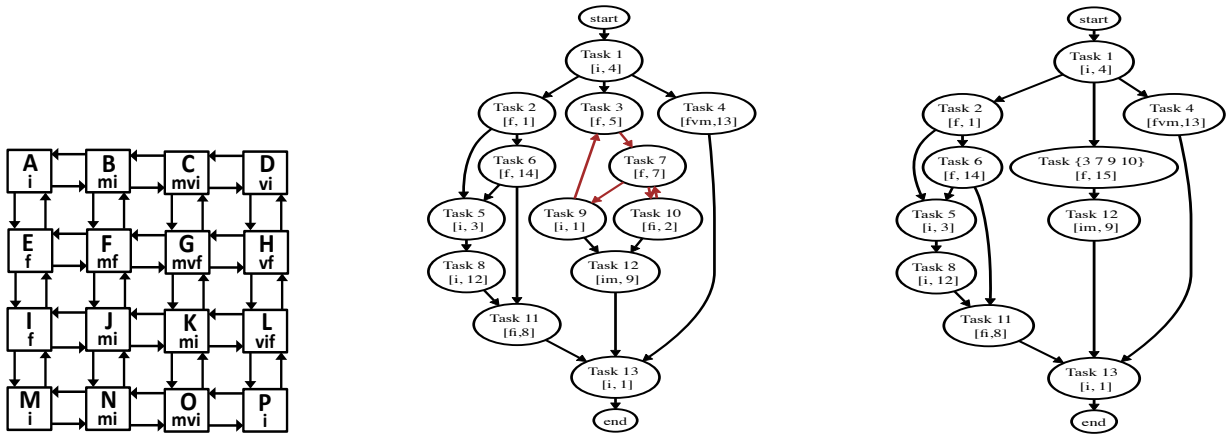
At all time t during application execution, tasks A and B are parallel, denoted $A \parallel B$,
if $O(o_1, o_2, ..., o_m)^t_A \cap I(i_1, i_2, ..., i_n)^t_B = \phi$ and $O(o_1, o_2, ..., o_p)^t_B \cap I(i_1, i_2, ..., i_q)^t_A = \phi$.

For mapping and routing purposes, one can fuse together any two tasks A and B, if and only if $O(o_1, o_2, ..., o_m)^t_A = I(i_1, i_2, ..., i_n)^t_B$ and all other tasks C $O(o_1, o_2, ..., o_m)^t_A \cap I(i_1, i_2, ..., i_n)^t_C = \phi$ and $O(o_1, o_2, ..., o_m)^{t-1}_A \cap I(i_1, i_2, ..., i_n)^t_C = \phi$. In other words, the intermediate A to B state is not observable by any other task. Task fusion is useful in forcing a set of tasks to be assigned to the same processing, particularly when tasks share a large amount of state, and communication costs between processing units are prohibitive or expensive for those tasks. It also allows us to reduce cyclic task-graphs (Figure 1(b)) to acyclic ones (Figure 1(c)).

C. Scheduling Algorithm: Taxonomy

For understanding and completeness, we list some basic scheduling terminologies [26]:

- A processor has a processing power $\wp$.
- Processor allocation: on which processor a task should execute.
- Task priority: when, and in what order with respect to other tasks, should each task execute.
- Fixed task priority: Each task has a single priority that remains the same through all phases of the application.
- Dynamic priority: a task may have different priorities at different execution points.



(a) A generic 2D-mesh heterogeneous multi-cores.

(b) A generic task-based decomposition of an application.

(c) The acyclic representation of the application task graph.

Fig. 1. Example of a generic heterogeneous multi-cores and an application task-based decomposition.

- Preemptive: tasks can be preempted by a higher priority task at any time.
- Non-preemptive: once a task starts executing, it will not be preempted and will therefore execute until completion.
- Cooperative: tasks may only be preempted at defined scheduling points within their execution.
- No migration: Each task is allocated to a processor and no migration is permitted
- Task-level migration: threads of a task may execute on different processors; however each thread can only execute on a single processor.

In our algorithm, many of these characteristics or classifications are not examined, simply because prior work may have done it, or this particular instance of the problem does not warrant such examination, or it is obvious how to extend the algorithm to support such characteristic.

IV. Formulation of task-based scheduling

Our framework provides a set of algorithms for mapping an application onto a heterogeneous many-core architecture, starting from simple ones to more complex ones, depending on the level of available system characterization and application analysis data.

A. Definitions

In our framework we define an application A to be $A = \{T_1, T_2, ..., T_k\}$ where task $T_i = (w_i, d_i, r_i, c_i)$, in addition to the characteristics described above, has the following properties: w_i (working set of task T_i), d_i (deadline of task T_i), r_i (input rate of task T_i), and c_i (instruction count in task T_i). While we only enumerate these properties, it is not difficult to see that other properties can be incorporated into the framework. We define a processing element $p_i = (\wp_i, l_{min_i}, l_{max_i})$, where $\wp_i = f(IPC, EUs, CSs)$, L_{min} represents the memory latency on a cache hit, and L_{max} the maximum miss latency. $\wp_i$ is a function of the IPC (instructions-per-cycle), EUs (execution units), and CSs (cache sizes). The EU factor helps specify which processing unit has what execution functional unit, e.g., floating-point unit, multiplication unit.

Definition 3: An task admissible schedule (TAS) for an application A is a set of tuples that associates to each task T a set of processing elements such that the data dependencies and timing constraints between tasks are respected.

Definition 4: A task T_i is schedulable on a processing element p_j, denoted $T_i \triangleright p_j$, if its worst-case execution time p_j is less than or equal to its deadline. In this work, equations 7 and 8 are used to determine task schedulability.

Definition 5: An application is schedulable according to a TAS algorithm if all of its tasks are schedulable.

Definition 6: An application is said to be feasible with respect to a given heterogeneous many-core system if there exists at least one TAS solution that can schedule all possible sequences of tasks that may be generated by the application on that system without missing any deadlines or violating any inter-task dependency.

Definition 7: Given two tasks T_i and T_j, a third task T_k can be composed out of T_i and T_j for mapping purposes if T_i and T_j are sequential, with no intermediate observable state, and cannot be pipelined.

B. ILP formulation of tasks-based scheduling

An application task graph $G = (A, E)$ is a directed acyclic graph (DAG) in which each vertex $T_i \in A$ represents a task and each edge $e(T_i, T_j) \in E$ represents a dependency between tasks T_i and T_j. Given a DAG $G = (A, E)$, we want to find a schedule that minimizes the finishing time of the last critical task. For the formulation, let us assume that all tasks in A are critical and T_k is the last task. We want to assign to each task $T_i \in A$ a pair (t_s, t_f) where t_s and t_f represent the starting and finishing time of task T_i under a given schedule θ. For all edge $e(T_i, T_j) \in E, w_{i,j}$ presents the amount of data transferred from T_i to T_j during execution $(O(o_1, o_2, ..., o_m)_{T_i} \cap I(i_1, i_2, ..., i_n)_{T_j} = w_{i,j})$. For k tasks and n processors, the exhaustive listing of schedules will produce $\frac{k!}{(k-n)!}$ schedules. This is prohibitive for large

problems. The user can limit the ILP runtime and find a solution over a subset.

C. ILP formulation

The objective function is:

$$minimize \quad T_k(t_f) \tag{1}$$

Subject to:

$$T_i(t_f) \leq d_j \tag{2}$$

$$T_i(t_s) \leq T_i(t_f) \tag{3}$$

$$if\ e(T_i, T_j) \in E \qquad T_i(t_f) < T_j(t_s) \tag{4}$$

$$\forall\ e(T_i, T_j) \in E, \quad T_j(t_s) - T_i(t_f) = d_{i,j} \tag{5}$$

$$\forall\ (P(T_i) = p_u\ ,\ P(T_j) = p_v), \quad w_{i,j} \times b_{p_u,p_v} \leq d_{i,j} \tag{6}$$

$$\begin{aligned} \forall\ T_i \in A, \forall p_u \in P, & \\ E_{(T_i,p_u)} \quad &= \quad (\wp_u \times c_i) \\ &+= \quad (w_i \times hit_{rate_{i,u}} \times l_{min_u}) \\ &+= \quad (w_i \times (1 - hit_{rate_{i,u}}) \times l_{max_u}) \end{aligned} \tag{7}$$

$$For \quad T_i \in A,\ p_u \in P, \quad T_i(t_f) - T_i(t_s) \geq E_{(T_i,p_u)} \tag{8}$$

$$\forall\ T_i \in A,\ p_u \in P, \quad M(T_i, p_u) - (E_{(T_i,p_u)} \times b_{i,u}) = 0 \tag{9}$$

$$\forall\ T_i \in A, \quad \sum_{u=1}^{n} b_{i,u} = 1 \tag{10}$$

$$\forall\ p_u \in P, \quad \sum_{i=1}^{k} M(T_i, p_u) \leq T_k(t_f) \tag{11}$$

D. Heuristic task-based scheduling algorithms

For applications with a large number of tasks and tight constraints where the convergence of the ILP formulation onto a satisfiable solution may take too long, we examine a set of heuristics to provide a fairly effective alternative to the ILP formulation. Heuristic H1 is very simple and converges quickly. It assigns to each task a set of processing units based on execution affinity, and tries to minimize processor-sharing among those sets. Its output is a set of processors that can be used for a task. In general it is good to select the fastest processing unit out of the set as the final mapping processor. Our second heuristic H2 takes into account task deadlines in addition to processor affinity in mapping task to processors. It tries to minimizes the finishing time per processor as opposed to the global finishing time as done in the ILP formulation.

Algorithm 1 Minimizes intersections across all mapping sets (H1).

Assumption: An application A composed of a number of tasks, $A = \{T_1, T_2, ..., T_k\}$ and a system with a list of processing elements $P = \{p_1, p_2, ..., p_n\}$.

Objective: Find a set of task-processor mapping $S = \{S(T_1), S(T_2), ..., S(T_k)\}$ such that: $\forall\ T_i \in A,\ S(T_i) = \{p_u, ..., p_v\}$ with $1 \leq u < v \leq n$ while minimizing $\forall (i,j)\ S(T_i) \cap S(T_j)$.

Begin
$\forall\ T_i \in A, \quad S(T_i) = \phi$
for $i = 1;\ i \leq k;\ i++$: **do**
  for $j = 1;\ j \leq n;\ j++$: **do**
    if $(T_i \triangleright p_j)$ **then**
      $S(T_i) = S(T_i) \cup \{p_j\}$
    end if
  end for
end for
while $(\forall\ T_i \in A,\ |S(T_i)| > 1$ and $\forall\ (i,j)\ S(T_i) \cap S(T_j) \neq \phi)$ **do**
  if $(\exists\ (T_i, T_j) \mid S(T_i) \cap S(T_j) \neq \phi)$ **then**
    if $((|S(T_i)| > 1) \wedge (|S(T_j)| > 1))$ **then**

$$S(T_i) = \begin{cases} S(T_i) - \{S(T_{min}) \cap S(T_i)\}\ where \\ S(T_i) \subsetneq \{S(T_{min}) \cap S(T_i)\} \\ \\ \{p_e\}\ for\ any\ p_e \in S(T_i)\ otherwise \end{cases}$$

    end if
  end if
end while
End

V. Evaluation Methodology

To demonstrate the effectiveness of our scheduling algorithms, we use the *Heracles* multicore system framework [27], [28] to construct eight different processor core configurations with different execution power, enumerated below: 16-bit microprocessor (Processor1), single-cycle MIPS core (Processor2), 7-stage single-threaded MIPS core (Processor3), 7-stage 2-way threaded MIPS core (Processor4), Single lane vector machine (Processor5), 2-lane vector machine (Processor6), 4-lane vector machine (Processor7), 8-lane vector machine (Processor8). Power electronics is one of the key physical layers of the *smart grid* that enables highly efficient and fully controllable flow of electric power, and promises to deliver up to 30% electric energy savings across all aspects of the conversion of primary energy into electricity. A typical power electronics system consists of an energy source interface, a power converter, and a control unit. We take two representative power electronics system applications: namely, a utility grid connected wind turbine converter system and a hybrid electric vehicle motor drive system.

A. Utility Grid Connected Wind Turbine

The general control strategy in a wind turbine consists of limiting the power delivery to the grid under high wind by doing a stall control, or an active stall, or even a pitch control.

Algorithm 2 Minimizes the finishing time on each processor (H2).

Assumption: An application A composed of a number of tasks, $A = \{T_1, T_2, ..., T_k\}$ and a system with a list of processing elements $P = \{p_1, p_2, ..., p_n\}$.

Objective: Find a set of task-processor mapping $\{S(p_1), S(p_2), ..., S(p_k)\}$ such that $\forall\ p_i \in P,\ S(p_i) = \{T_a, ..., T_b\}$ where $D(p_i) = minimum(\sum_{u=1}^{k} d'_u)$.

$$d'_u = \begin{cases} d_u & where\ T_u \triangleright p_i \\ 0 & otherwise \end{cases}$$

Begin
$\forall\ p_i \in P,\ S(p_i) = \phi$ and $D(p_i) = 0$
for $i = 1;\ i \leq n;\ i++$: **do**
  for $j = 1;\ j \leq k;\ j++$: **do**
    if $(T_j \triangleright p_i)$ **then**
      $S(p_i) = S(p_i) \cup \{T_j\}$
      $D(p_i) += d_j$
    end if
  end for
end for
for $i = 1;\ i \leq n;\ i++$: **do**
  for $j = 1;\ j \leq n;\ j++$: **do**
    $\forall\ T_u \in S(p_i)$:
    if $((T_u \in S(p_j)) \wedge (D(p_j) < D(p_i)))$ **then**
      $S(p_i) = S(p_i) - \{T_u\}$
      $D(p_i) -= d_u$
    end if
  end for
end for
End

To converge to the proper strategy, the control algorithm may need a reference emulator of the physical system to check system responses, interfaces, and failure modes. Figure 3 shows the profiled task graph of our wind turbine application. Figure 2 reports the execution time of all tasks on each processor type.

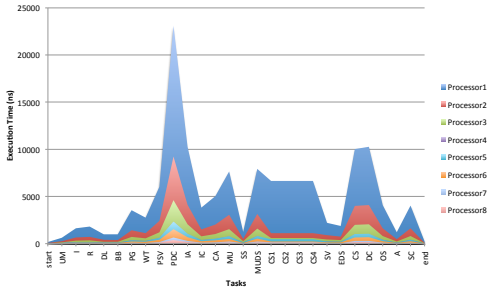


Fig. 2. Total execution time for all tasks in the wind turbine application per processor.

B. Hybrid Electric Vehicle

Hybrid electric vehicles (HEVs) use two power sources; internal combustion and electric. They are fuel efficient because of their electric motor drive. The motor drive, with a controlled inverter system, is needed to deliver powerful and efficient drive to the electric motor. In general, HEV motor drive systems can operate in two modes; namely, conventional mode and regenerative breaking mode. In conventional mode, the internal combustion engine supplies the voltage that passes through the inverter block to drive the induction motor; whereas in the regenerative breaking mode, the induction machine acts as a generator and the inverter block acts as a three-phase rectifier.

C. Results

In this section, we present the output of our three scheduling algorithms. In general, H1performs poorly when compared to the ILP-based mapping, because it does not take into account many of the system architectural features. If a compute-intensive task is inadvertently mapped into a weaker core it can impact the whole application execution time. H2 performs better then H1, and it is within 20% to 70% of the ILP-based mapping efficiency.

VI. Conclusion

We present in this work an ILP formulation and two non-iterative heuristics that can handle different application performance targets and architectural features for scheduling a task-based application onto a heterogeneous many-core architecture. We use RTL-based heterogeneous many-core systems and real power electronics applications for evaluations of mapping algorithms. In the future we will expand on these algorithms and compare them with closely related schemes.

References

[1] http://www.tilera.com/, "Tilera tile-gx processor," 2012.

[2] D. Pham, S. Asano, M. Bolliger, M. Day, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa, "The design and implementation of a first-generation cell processor," in *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pp. 184 –592 Vol. 1, Feb. 2005.

[3] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, pp. 56–67, Oct. 2009.

[4] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-isa heterogeneous multi-core architectures for multithreaded workload performance," in *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, (Washington, DC, USA), pp. 64–, IEEE Computer Society, 2004.

[5] H. Kopetz, *Real-Time Systems : Design Principles for Distributed Embedded Applications (The International Series in Engineering and Computer Science)*. Springer, Apr. 1997.

[6] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (pie)," in *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, pp. 213 –224, June 2012.

[7] J. Chen and L. John, "Efficient program scheduling for heterogeneous multi-core processors," in *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pp. 927 –930, July 2009.

[8] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn, "Efficient operating system scheduling for performance-asymmetric multi-core architectures," in *Supercomputing, 2007. SC '07. Proceedings of the 2007 ACM/IEEE Conference on*, pp. 1 –11, Nov. 2007.

[9] M. Garey and D. Johnson, "Complexity results for multiprocessor scheduling under resource constraints," *SIAM Journal on Computing*, vol. 4, no. 4, pp. 397–411, 1975.

[10] D. Karger, C. Stein, and J. Wein, "Scheduling algorithms," 1997.

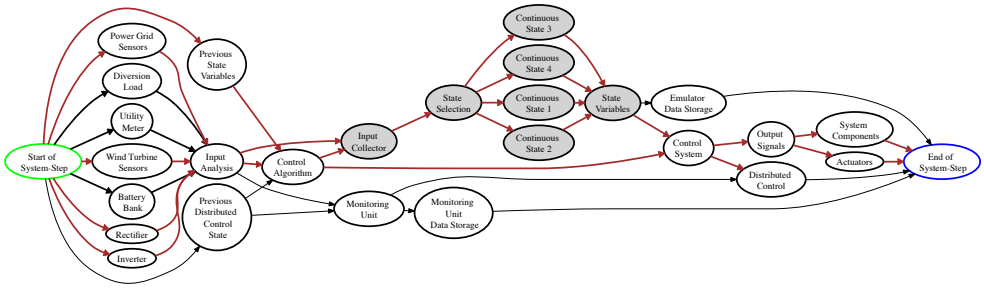
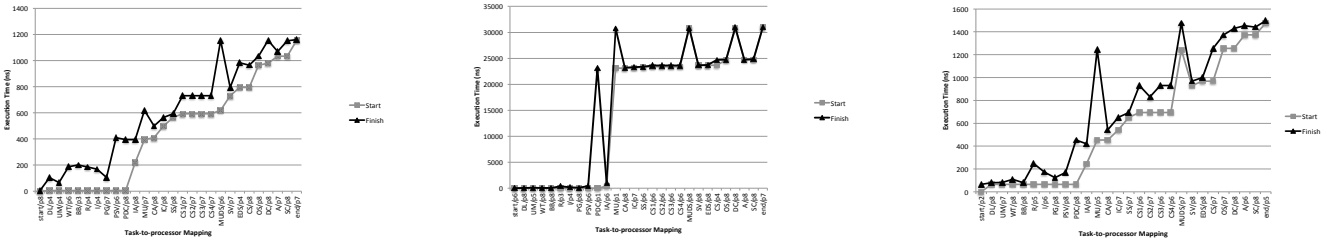


Fig. 3. Task-graph of the wind turbine system application.

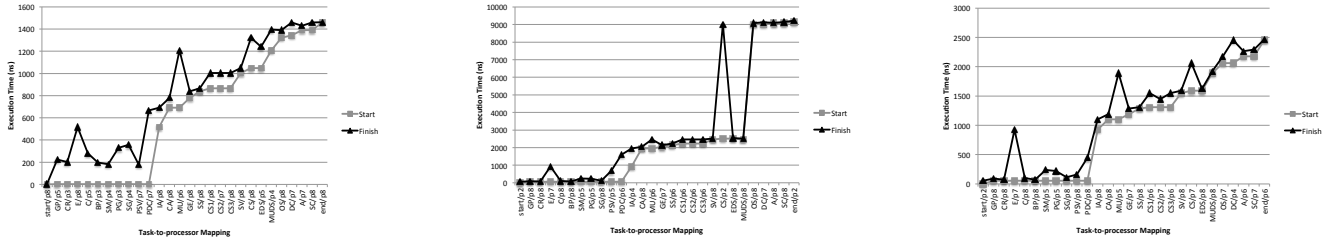


(a) ILP schedule solution for the wind turbine system application.

(b) H1 schedule solution for the wind turbine system application.

(c) H2 schedule solution for the wind turbine system application.

Fig. 4. Scheduling of wind turbine system application



(a) ILP schedule solution for the hybrid electric vehicle application.

(b) H1 schedule solution for the hybrid electric vehicle application.

(c) H2 schedule solution for the hybrid electric vehicle application.

Fig. 5. Scheduling of hybrid electric vehicle application

[11] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, K. C, and P. Wong, "Theory and practice in parallel job scheduling," 1994.

[12] T. Yang and A. Gerasoulis, "Dsc: Scheduling parallel tasks on an unbounded number of processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 951–967.

[13] M. R. Garey and D. S. Johnson, "Two-processor scheduling with start-times and deadlines," *SIAM Journal on Computing*, vol. 6, pp. 416–426, 1977.

[14] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," 1999.

[15] M. Becchi and P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures," in *Proceedings of the 3rd conference on Computing frontiers*, CF '06, pp. 29–40, ACM, 2006.

[16] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pp. 125–138, ACM, 2010.

[17] B. B. Brandenburg, J. M. Calandrino, and J. H. Anderson, "On the scalability of real-time scheduling algorithms on multicore platforms: A case study," 2008.

[18] K. Ramamritham and J. Stankovic, "Scheduling algorithms and operating systems support for real-time systems," *Proceedings of the IEEE*, vol. 82, pp. 55 –67, Jan. 1994.

[19] A. Burns, "Scheduling hard real-time systems: a review," *Software Engineering Journal*, vol. 6, pp. 116 –128, May 1991.

[20] H. Topcuoglu and M. you Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, pp. 260–274, 2002.

[21] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *In Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), Puerto Vallarta*, pp. 119–129, 1998.

[22] S. Chaudhuri, R. A. Walker, and J. E. Mitchell, "Analyzing and exploiting the structure of the constraints in the ilp approach to the scheduling problem," *IEEE Transactions on VLSI Systems*, vol. 2, pp. 456–471, 1994.

[23] Y. Yi, W. Han, X. Zhao, A. Erdogan, and T. Arslan, "An ilp formulation for task mapping and scheduling on multi-core architectures," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pp. 33 –38, April 2009.

[24] N. B. Lakshminarayana, J. Lee, and H. Kim, "Age based scheduling for asymmetric multiprocessors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pp. 25:1–25:12, ACM, 2009.

[25] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar, "Hass: a scheduler for heterogeneous multicore systems," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 66–75, Apr. 2009.

[26] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, pp. 35:1–35:44, Oct. 2011.

[27] M. Kinsy, M. Pellauer, and S. Devadas, "Heracles: Fully synthesizable parameterized mips-based multicore system," in *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pp. 356 –362, Sept. 2011.

[28] M. A. Kinsy, M. Pellauer, and S. Devadas, "Heracles: A tool for fast rtl-based design space exploration of multicore processors," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pp. 125–134, ACM, 2013.