Available online at www.sciencedirect.com
ScienceDirect
J. Parallel Distrib. Comput. 67 (2007) 154–169
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

Dynamically mapping tasks with priorities and multiple deadlines in a heterogeneous environment☆

Jong-Kook Kim[a,*], Sameer Shivle[b], Howard Jay Siegel[b,c], Anthony A. Maciejewski[b], Tracy D. Braun[b], Myron Schneider[b,d], Sonja Tideman[c], Ramakrishna Chitta[c], Raheleh B. Dilmaghani[b], Rohit Joshi[b], Aditya Kaul[b], Ashish Sharma[b], Siddhartha Sripada[b], Praveen Vangari[b], Siva Sankar Yellampalli[e]

[a] *Samsung SDS, IT R & D Center, 159-9 Gumi-Dong Bundang-Gu Seongnam-Si, Gyeonggi-Do, South Korea*
[b] *Electrical and Computer Engineering Department, Colorado State University, Fort Collins, CO 80523-1373, USA*
[c] *Computer Science Department, Colorado State University, Fort Collins, CO 80523-1373, USA*
[d] *Agilent Technologies, Loveland, CO 80537, USA*
[e] *Electrical and Computer Engineering School, Louisiana State University, Baton Rouge, LA 70802, USA*

Received 14 April 2005; received in revised form 20 April 2006; accepted 28 June 2006
Available online 13 November 2006

Abstract

In a distributed heterogeneous computing system, the resources have different capabilities and tasks have different requirements. To maximize the performance of the system, it is essential to assign the resources to tasks (*match*) and order the execution of tasks on each resource (*schedule*) to exploit the heterogeneity of the resources and tasks. Dynamic mapping (defined as matching and scheduling) is performed when the arrival of tasks is not known a priori. In the heterogeneous environment considered in this study, tasks arrive randomly, tasks are independent (i.e., no inter-task communication), and tasks have priorities and multiple soft deadlines. The value of a task is calculated based on the priority of the task and the completion time of the task with respect to its deadlines. The goal of a dynamic mapping heuristic in this research is to maximize the value accrued of completed tasks in a given interval of time. This research proposes, evaluates, and compares eight dynamic mapping heuristics. Two static mapping schemes (all arrival information of tasks are known) are designed also for comparison. The performance of the best heuristics is 84% of a calculated upper bound for the scenarios considered.
© 2006 Elsevier Inc. All rights reserved.

Keywords: Deadlines; Distributed computing; Dynamic mapping; Heterogeneous computing; Priority; Resource allocation; Resource management; Scheduling; Static mapping

1. Introduction

Heterogeneous computing (*HC*) is the coordinated use of various resources with different capabilities to satisfy the requirements of varying task mixtures. The heterogeneity of the resources and tasks in an HC system is exploited to maximize the performance or the cost-effectiveness of the system (e.g., [8,11,15]). To exploit the different capabilities of a suite of heterogeneous resources, typically, a *resource management system* (*RMS*) allocates the resources to the tasks and the tasks are ordered for execution on the resources. In this research, heuristics are proposed that can be used in such an RMS.

An important research problem is how to assign resources to tasks (*match*) and order the execution of tasks on the resources (*schedule*) to maximize some performance criterion of an HC system. This procedure of matching and scheduling is called *mapping* or *resource allocation*. Two different types of mapping are static and dynamic. *Static* mapping is performed when

☆ This research was supported by the Colorado State University George T. Abell Endowment.
* Corresponding author.
E-mail addresses: jongkook@ieee.org (J.-K. Kim), sameer.shivle@gmail.com (S. Shivle), hj@colostate.edu (H.J. Siegel), aam@colostate.edu (A.A. Maciejewski), tdbraun@yahoo.com (T.D. Braun), myron_schneider@agilent.com (M. Schneider), stidema@sandia.gov (S. Tideman), ramacmr@cs.colostate.edu (R. Chitta), rdilmaghani@ucsd.edu (R.B. Dilmaghani), aditya21@lycos.com (A. Kaul), ashish@engr.colostate.edu (A. Sharma), siddhu@engr.colostate.edu (S. Sripada), praveen@engr.colostate.edu (P. Vangari), syella1@lsu.edu (S.S. Yellampalli).

0743-7315/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2006.06.005

the applications are mapped in an off-line planning phase [7], e.g., planning the schedule for a set of production jobs. *Dynamic* mapping is performed when the applications are mapped in an on-line fashion [25], e.g., when tasks arrive at unknown intervals and are mapped as they arrive (the workload is not known a priori). In both cases, the mapping problem has been shown, in general, to be NP-complete (e.g., [10,12,18]). Thus, the development of heuristic techniques to find near-optimal solutions for the mapping problem is an active area of research (e.g., [2,6,5,13,27,38]).

In this research, the dynamic mapping of tasks onto machines is studied. Simulation is used for the evaluation and comparison of the dynamic heuristics developed in this research. As described in [25], dynamic mapping heuristics can be grouped into two categories, immediate mode and batch mode. Each time a mapping is performed (*mapping event*), *immediate mode* heuristics only consider the new task for mapping, whereas *batch mode* considers a subset of tasks for mapping, thus having more information about the task mixture before mapping the tasks. As expected, the study in [25] showed that the immediate mode heuristics had shorter running times than those of the batch mode heuristics, but the batch mode heuristics gave higher performance. The heuristics proposed in this research are batch mode schemes.

In this study we assume that tasks are independent (i.e., no inter-task communication). For example, these tasks can be generated by different users. Furthermore, each task has a priority level and multiple soft deadlines.

The target hardware platform assumed is a dedicated cluster of heterogeneous machines (as opposed to a geographically dispersed, loosely connected grid). Such a cluster may be found in a military command post.

The HC environment considered is oversubscribed. While most computing environments are designed to handle the expected computational load, there are important cases where this is not possible. For example, this may occur in defense environments where battle damage reduces the available resources or in catastrophic events where requests greatly exceed the expected load. In these scenarios, it is important to have a mechanism by which a resource management scheme can determine which tasks must be completed in a timely fashion. This study attempts to reflect this by a weighted priority scheme in conjunction with multiple soft deadlines, i.e., the value of a task is determined by its priority level and when the task is completed with respect to its deadlines. This environment will be useful when tasks have different importance and a task's value depends on when it is completed.

As an example of how priority schemes are used, consider a military environment. High priority tasks may involve the execution of defensive maneuvers, medium priority tasks may involve the control of offensive weapons, and low priority tasks may involve ordering supplies.

As an example of how multiple soft deadlines are used, consider a disaster management scenario. In particular, if a tsunami is reported (task) as soon as an earthquake is detected, then it will have full value to the people where the tsunami is expected to hit (i.e., a lot of people can evacuate). If the tsunami is reported when it is in visual range, then it will have some value (i.e., only some people can take cover). However, if the people of the area are warned as the tsunami hits the area it would have very little value (i.e., there will be a lot of casualties).

The goal of a dynamic mapping heuristic in this research is to maximize the sum of the values of completed tasks in a given interval of time. We designed eight dynamic mapping schemes to solve this problem. Two static heuristics are used to provide benchmarks against which the performance of the dynamic heuristics is compared. These two static methods are based on simulated annealing (SA) and genetic algorithm (GA) approaches.

The contributions of this research are: (1) the design of eight dynamic mapping heuristics for the proposed HC system model, (2) the comparison of the performance of the heuristics, and (3) a method for calculating an upper bound (UB) on the performance of a resource allocation, i.e., an UB on the maximum possible sum of values of completed tasks in a given interval of time.

The next section provides details of the problem statement. In Section 3, the literature related to this work is discussed. Section 4 presents the dynamic mapping heuristics studied in this research. The static mapping heuristics used for comparison to the performance of the dynamic methods are described in Section 5. Section 6 presents the simulation setup and the results of the simulation experiments are analyzed in Section 7. The last section gives a brief summary of this research.

2. Problem statement

2.1. Task model

The tasks considered are assumed to be independent, i.e., no communication or dependency between tasks. Each task has a priority level (i.e., high, medium, and low) and soft deadlines. The worth of a submitted task may degrade according to when it completes execution. The performance metric is the sum of the worth of tasks that complete in an interval of time.

The estimated time to complete (*ETC*) values of each task on each machine is assumed to be known based on user-supplied information, experiential data, task profiling and analytical benchmarking, or other techniques (e.g., [1,15,16,21,26,40]). Determination of ETC values is a separate research problem, and the assumption of such ETC information is a common practice in mapping research (e.g., [16,20,21,24,31,39]).

In the simulation experiments, the mapping heuristics only have knowledge of the ETC values and these ETC values are used to make the mapping decision. The $ETC(i, j)$ is the estimated execution time of task i on machine j, where i is the task number and j is the machine number. These estimated values may differ from actual times, e.g., actual times may depend on input data. Therefore, for the simulation studies, the actual time to complete (*ATC*) values are calculated using the ETC values as the mean. The ATC values are used only for the evaluation of the heuristics after the heuristics are used in the simulation of the system. The details of the simulation environment and the calculation and use of the ATC values are presented in Section 6.

2.2. Performance metric

2.2.1. Priorities

Each task i has a *priority level* that indicates the importance of a task relative to other tasks: high, medium, or low. To quantify the relative importance of priority levels in our study, we use a polynomial weighting scheme. The weights are used to compute a task's value. In particular, the *weighted priority* of task i is p_i, where

$$p_i = \begin{cases} x^2 & \text{for high priority tasks,} \\ x & \text{for medium priority tasks,} \\ 1 & \text{for low priority tasks.} \end{cases}$$

For this research, $x = 2$ or 4 (*light* or *heavy* priority weighting schemes, respectively). The weighted priority of a task is the maximum value it can contribute to the evaluation function. Clearly, this is just one example of a method for assigning priority values, and a different set of values can be used depending on the application domain.

2.2.2. Deadlines

The deadlines of a task reflect the importance of the response time to the user. In this research, each task have three soft deadlines (i.e., 100%, 50%, and 25% deadlines). The *deadline factor*, d_i, indicates the degradation scheme of the worth of a task. In particular,

$$d_i = \begin{cases} 1.00 & \text{if task } i \text{ finished at or before its 100\% deadline,} \\ 0.50 & \text{if task } i \text{ finished at or before its 50\% deadline,} \\ 0.25 & \text{if task } i \text{ finished at or before its 25\% deadline,} \\ 0.05 & \text{if task } i \text{ finished after its 25\% deadline,} \\ 0 & \text{if task } i \text{ is never started during the time period evaluated.} \end{cases}$$

2.2.3. Performance metric

The performance metric described in this section is used to evaluate the performance of a heuristic designed for the environment described in Section 1 during a fixed time period (referred to as the *evaluation period*). This performance metric builds on the idea of the FISC measure in [23]. For the evaluation, the tasks that partially execute within the evaluation period are prorated. Let B denote the beginning of the evaluation period and let E denote the end of the evaluation period. Let j be the machine assigned to task i by the mapping heuristic. The simulated actual execution time for task i on machine j is $ATC(i, j)$. The start time of task i on machine j is $st(i, j)$ and the finish time of task i on machine j is $ft(i, j)$. Then, b_i gives the boundary weighting for each task i, i.e.,

$$b_i = \begin{cases} (ft(i,j) - B)/\mathrm{ATC}(i,j) & \text{if } st(i,j) < B \text{ and } B < ft(i,j) \leqslant E, \\ 1.00 & \text{if } st(i,j) \geqslant B \text{ and } ft(i,j) \leqslant E, \\ (E - st(i,j))/\mathrm{ATC}(i,j) & \text{if } B \leqslant st(i,j) < E \text{ and } ft(i,j) > E, \\ (E - B)/\mathrm{ATC}(i,j) & \text{if } st(i,j) \leqslant B \text{ and } ft(i,j) \geqslant E, \\ 0 & \text{if } ft(i,j) \leqslant B \text{ or } st(i,j) \geqslant E. \end{cases}$$

Let T be the total number of tasks that are mapped (i.e., the total number of tasks in the ETC matrix). Then the *evaluation value*, V used to evaluate each mapping is defined as

$$V = \sum_{i=0}^{T-1} p_i \times d_i \times b_i.$$

Thus, the value associated with a mapping is the sum of the weighted priority of tasks executed during the evaluation period, reduced if the 100% deadline is not met and prorated if tasks are not started and/or completed during the evaluation period.

2.3. Upper bound (UB)

The *UB* on the evaluation value uses the arrival time of tasks, priority of tasks, the deadline of the tasks, and the time interval between the arrivals of tasks. The tasks that have arrived before or at the mapping event are called *selectable tasks*. At any mapping event, only the selectable tasks are considered for the calculation of the UB. Let Q_i be equal to the priority weighting of task i divided by the minimum $\mathrm{ATC}(i, j)$ over all machines (i.e., priority weight per unit actual computation time).

The UB starts by initializing all task's *remaining ATC* values, $rATC(i, j)$, to the minimum $\mathrm{ATC}(i, j)$ over all machines. When a new task arrives, the UB follows the procedure below.

(1) At a mapping event, determine the *total aggregate computation time* (*TACT*) until the next task arrives. That is, TACT = time interval between arrival times of the new task and the next task multiplied by the number of machines.
(2) Put all selectable tasks with $\mathrm{rATC}(i, j) > 0$ in a task list.
(3) Select the task a that has the highest Q_i from the task list.
(4) If $\mathrm{TACT} \leqslant \mathrm{rATC}(a, j)$
    add ($Q_a \times \mathrm{TACT}$) to the evaluation value
    subtract TACT from $\mathrm{rATC}(a, j)$
    done (i.e., TACT = 0)
  if $\mathrm{TACT} > \mathrm{rATC}(a, j)$
    add ($Q_a \times \mathrm{rATC}(a, j)$) to the evaluation value
    subtract $\mathrm{rATC}(a, j)$ from TACT
    (this becomes the new TACT)
    repeat steps 3 and 4.
(5) Repeat steps (1)–(4) until the end of evaluation period.

3. Related work

There have been many previous research studies concerned with the dynamic mapping of independent tasks onto heterogeneous machines to minimize the completion time of the last finishing task (e.g., [14,18,24,25]). Our research has a different task model (with priorities and multiple deadlines) and a different performance metric (described in Section 2.2.3) that complicate the mapping problem.

The environment in [30] has randomly arriving tasks with a hard deadline. The concept of moving a task if its deadline may not be satisfied (presented in [30]) is used in one of the

heuristics in our research. However, the environment in our research is different because it includes task with priorities and multiple deadlines, heterogeneous machines, and a more complex performance metric, all of which complicate the scheduling problem.

The DeSiDeRaTa project (e.g., [9,17,33–36]) focuses on dynamically reallocating resources for applications, but the system model is very different. The system model in DeSiDeRaTa includes sets of heterogeneous machines, sensors, applications, and actuators. The applications in the DeSiDeRaTa project are continuously running ones where data inputs to an application are processed and output to another application or an actuator. In contrast, the tasks in this research are independent, are randomly arriving, have priorities, and have multiple deadlines.

The work in [41] focuses on the dynamic mapping of independent tasks onto machines in an environment that is similar to the one in this study (i.e., randomly arriving tasks, heterogeneous machines, and heterogeneous tasks). Our research and the research in [41] include different heuristics based on Min–Min and Max–Min from [18] designed for the particular environments in the two different studies. The difference between our study and the work in [41] is that the tasks in our study are assigned a weighted priority, each task has multiple deadlines, and the performance metric is the value of tasks completed in an interval of time instead of completion rate, defined as the number of tasks completed in an interval of time. The idea of "fine-tuning" in [41] is used in two of the heuristics in our research as "rescheduling" after tasks are mapped.

4. Dynamic mapping heuristics

4.1. Mapping event

Each dynamic mapping approach was designed to compute the new mapping faster than the anticipated average arrival rate of the tasks. Therefore, the heuristics that are developed have a limit on the maximum time each computation of a new mapping (*mapping event*) can take. A mapping event occurs when a new task arrives in the system and the previous mapping event has ended. If tasks arrive while a mapping event is in progress, the current mapping event is not disturbed, but the next mapping event includes any tasks that had arrived.

At any mapping event, the new task and the tasks in the machine queues still awaiting execution are considered together for machine assignment, i.e., previously mapped but unexecuted tasks can be *remapped*. The exception is that the first task in each machine's wait queue is not considered for remapping. The reason for this is to reduce the chance of a machine becoming idle if during a mapping event the currently executing task finishes. While it is still possible that a machine may become idle, it is highly unlikely for the assumptions in this research (the average execution time of a task is 180 s while the average execution time of a mapping event is less than 0.5 s).

4.2. Max–Max

The *Max–Max* method is based on the Min–Min (greedy) concept in [18]. The Max–Max finds the "best" machine for each task that is considered for mapping, and then among these task/machine pairs it selects the "best" pair to map first. To determine which machine or which task/machine pair is the best, a fitness value is used. The *fitness value* for the task on a given machine is the worth of the task divided by the estimated execution time of the task on that machine, where the *worth* of the task is the priority weighting of the task multiplied by the deadline factor of the task.

Max–Max can be summarized by the following procedure, which starts when a new task arrives and generates a mapping event. The *mappable tasks* are tasks that are waiting to be executed in the machine wait queue (except the first task) and the new task. When the mapping event begins, it is assumed that none of the mappable tasks are mapped, i.e., they are not in any machine wait queue.

(1) Generate a task list that includes all the mappable tasks.
(2) For each task in the task list, find the machine that gives the task its maximum fitness value (the first "Max"), ignoring other tasks in the mappable task list.
(3) Among all the task/machine pairs found from above, find the pair that gives the maximum fitness value (the second "Max").
(4) Remove the above task from the mappable task list and map the task to its maximum fitness value machine.
(5) Update the machine available times.
(6) Repeat steps (2)–(5) until all tasks are mapped.

The availability status of the machine selected in step (3) is updated in step (5) and then used in calculating the deadline factors in the next iteration. The *completion time* for task i on machine j is the sum of the *machine available time* ($mat(j)$) (i.e, the time machine j is available to execute task i), and $\mathrm{ETC}(i, j)$. The deadline factor for a given task/machine pair is determined using the task's estimated completion time on that machine. In step (2), the worth for a task/machine pair is recalculated if the mat was updated for the machine.

4.3. Max–Min and Min–Min

Part of the *Max–Min* heuristic also is based on the greedy concept in [18]. This scheme finds the machine with the minimum completion time for each task. Then, from these task/machine pairs, the heuristic selects the pair that has the maximum completion time. This method maps tasks that take more time first because these tasks typically have a higher probability of not completing before their deadline if not mapped as soon as possible.

Max–Min can be summarized by the following procedure, which starts when a new task arrives and generates a mapping event. When the mapping event begins, it is assumed that none of the mappable tasks are mapped, i.e., they are not in any machine wait queue.

(1) Generate a task list that includes all the mappable tasks.
(2) For each task in the mappable task list, find the minimum completion time machine (the "Min"), ignoring other tasks in the mappable task list.

(3) Among all the task/machine pairs found from above, select the pair that gives the maximum completion time (the "Max").
(4) Remove the task identified above from the mappable task list and assigned to its minimum completion time machine.
(5) Update the machine available times to calculate the minimum completion time in step (2).
(6) Repeat steps (2)–(5) until all the tasks are mapped.
(7) For each machine, if there are tasks in the machine wait queue, reschedule these tasks in descending order according to their worth.

The rescheduling of tasks in step (7) for each machine can be summarized by the following procedure.

(a) Initialize the machine available time to the completion time of the first task in the wait queue (i.e., assume that none of the mappable tasks are mapped).
(b) Group the tasks using their priority levels.
(c) For the tasks in the high priority level group, keeping their relative ordering from the machine wait queue, one by one, in order, insert each task that can finish by its 100% deadline into the machine wait queue. Once scheduled, a task is removed from the group and the machine available time is updated.
(d) Repeat step (c) for the 50% deadline and then the 25% deadline.
(e) Repeat steps (c) and (d) for medium priority tasks and then repeat for low priority tasks.
(f) High priority tasks that cannot finish by their 25% deadline are added to end of the machine wait queue. Medium priority tasks that cannot finish by their 25% deadline are added next and then low priority tasks are added to the machine wait queue.

The *Min–Min* heuristic, which is a variation of Max–Min, was also implemented. The difference is in step (3), where instead of selecting the pair that gives the maximum completion time, the pair that gives the minimum completion time is selected. The goal of this method is to attempt to complete as many tasks as possible.

4.4. Percent Best

The first part of the *Percent Best* heuristic is a variation of the k-percent best heuristic found in [25]. Let M be the total number of machines within the HC suite. The idea behind Percent Best is to, in general, assign a task to one of the $m \leqslant M$ machines with the best execution time. However, limiting the number of machines to which a task can be mapped may cause the system to become unbalanced, therefore the completion times are also considered.

Percent Best can be summarized by the procedure below, which starts when a new task arrives and generates a mapping event. When the mapping event begins, it is assumed that none of the mappable tasks are mapped, i.e., they are not in any machine wait queue.

(1) Generate a task list that includes all the mappable tasks.
(2) Tasks are grouped according to their priority levels.
(3) For each task in the high priority level group, find the top $m = 3$ machines that give the best execution time for that task (the total number of machines used in the simulation studies in this research is eight).
(4) For each task, find the minimum completion time machine from the machines found in step (3) and the machines that are idle.
(5) Map tasks with no contention (i.e., there are no other tasks with the same minimum completion time machine) and remove them from the group.
(6) For tasks with contention (tasks having the same minimum completion time machine), map the task with the earliest 100% deadline and remove it from the group.
(7) Update the availability status of all machines assigned tasks.
(8) Repeat steps (3)–(7) until all tasks in the group are mapped.
(9) Repeat steps (3)–(8) for tasks in the medium and low priority level groups, using $m = 4$ and 8, respectively. Note that the values of m are determined experimentally.

4.5. Queueing Table

The *Queueing Table* heuristic uses a lookup table (see Table 1) constructed based on priority, relative execution time (*RET*), and urgency. The RET is the ratio of the average execution time of a task across all machines to the overall average task execution time for all tasks across all machines in the HC system. The Queueing Table heuristic uses the above definition and a heuristic constant (RET cutoff) to classify tasks into one of two categories: "slow" and "fast." If a task's RET>RET cutoff, then it is considered to be slow; if a task's RET$\leqslant$RET cutoff, then it is considered to be fast. The RET cutoff was determined experimentally.

Let δ be the 100% deadline of a given task i minus the current time. If δ is positive then the *urgency* of a given task i equals (average ETC(i, j) over all $j)/\delta$.

It is considered more urgent if the ratio is larger. If δ is zero or negative (i.e., the current time passes the 100% deadline

Table 1
The lookup table constructed based on priority, relative execution time, and urgency for the Queueing Table heuristic

Queueing rank	Priority level	Relative execution time	Urgency
1	High	Slow	Sooner
2	High	Fast	Sooner
3	High	Slow	Later
4	High	Fast	Later
5	Med	Fast	Sooner
6	Low	Fast	Sooner
7	Med	Fast	Later
8	Low	Fast	Later
9	Med	Slow	Sooner
10	Med	Slow	Later
11	Low	Slow	Sooner
12	Low	Slow	Later

of a task), the task's urgency is set to negative infinity. The method uses the above definition of urgency and a heuristic constant (*urgency cutoff*) to classify task into two categories. If a task's urgency $\leqslant$ urgency cutoff this indicates that the task can be started "later"; if a task's urgency $>$ urgency cutoff, then the task needs to be started "sooner." The urgency cutoff was determined experimentally.

Queueing Table can be summarized by the following procedure, which starts when a new task arrives and generates a mapping event. In contrast to other heuristics, this method does not generate a task list that includes all the mappable tasks and initially maps only the new task to a machine. Steps (1)–(4) assign the new task to a machine; steps (5)–(8) consider moving any task that will miss its 100% deadline.

(1) For all mappable tasks, calculate the urgency.
(2) For the new task, calculate the RET.
(3) For each of the machines, compare the new task with the tasks on that machine's wait queue, starting from the front of the queue (lowest rank number). If there are no tasks with the same queueing rank as the new task, then the new task's position is in front of the first task with the higher numbered queueing rank. If there are tasks with the same queueing rank as the new task, then the new task's position is in front of the first task that is less urgent.
(4) Using the position on each machine wait queue found from above, the completion time on all machines is calculated and the new task is mapped to its minimum completion time machine.
(5) For the first task in machine 1's wait queue that will miss its 100% deadline (if any), find machines where (a) the priority of the task $\geqslant$ the highest priority of any task on that machine, (b) moving the task to the front of that machine wait queue does not cause any task to miss its 100% deadline (tasks already missing their 100% deadline are not checked), and (c) the task can finish by its 100% deadline on that machine.
(6) Among the machines identified above, find the machine with the minimum completion time for the task and move the task to the front of that machine's wait queue. (If no machines are found in step (5), the task is not moved.)
(7) Update the machine available times.
(8) Repeat steps (5)–(7) until all machines are checked (the order in which the machines are checked is from machine 1 to machine M).

As indicated in step (8), the search of tasks missing their 100% deadline is done on all machines. This is because there may be tasks in machine wait queues other than the one the new task is mapped to that miss their 100% deadline. At any mapping event, at most one task from each machine is allowed to be moved to limit the heuristic execution time.

4.6. Relative Cost

The *Relative Cost* heuristic loosely builds on the sufferage idea in [25] and the relative cost idea in [37] to map tasks. The *relative cost* (*RC*) value calculated for this heuristic is similar to the one in [37]. However, the Relative Cost heuristic in [37] uses RC as the fitness value for a Min–Min type heuristic.

For each mappable task considered, the RC is calculated by computing the minimum completion time of that task over all machines divided by the average completion time of that task on all machines. When the RC is high, the minimum completion time is similar to the average and most of the completion times on all machines are similar. When the RC is low, the minimum completion time is very different from the average. Assume tasks a and b prefer the same machine (best machine) for execution. Task a is considered to suffer more than task b, when there is a larger difference between the completion times of the best and the second best machines for task a than for task b. The RC is an approximation of this difference. If a task's RC is high then there is a low probability that the task will suffer more than a task with a low RC.

The RC method can be summarized by the following procedure, which starts when a new task arrives and generates a mapping event. When the mapping event begins, it is assumed that none of the mappable tasks are mapped, i.e., they are not in any machine wait queue.

(1) Generate a task list that includes all the mappable tasks.
(2) For each task in the mappable task list calculate the RC.
(3) For all the tasks in the task list, calculate their worth (as described below) and sort the tasks according to their worth (highest first).
(4) Consider the tasks having the highest worth. Determine the minimum completion time machines for each of these tasks, ignoring other tasks on the mappable task list. If two or more tasks have the same minimum completion time machine then map the task with the lowest RC value to its minimum completion time machine (i.e., tasks with a unique minimum completion time machine are assigned to that machine). Else, map all tasks to their minimum completion time machine.
(5) Remove mapped task(s) from the mappable task list.
(6) Update the machine available times.
(7) Repeat steps (2)–(6) until all tasks are mapped.

In step (3), the deadline factor for each task is calculated using the minimum completion time of that task over all machines found from the current mapping, the current time, and the deadline for the tasks, ignoring other tasks in the mappable task list. Using the deadline factor found for each task, the worth is recalculated every time tasks are mapped. The availability status of machines selected for mapping are updated in step (6) to calculate the completion time of all tasks on the machines and the deadline factor.

4.7. Slack Sufferage

The *Slack Sufferage* heuristic also builds on the sufferage concept in [25], as described in Section 4.6. However, rather than using an RC value to capture the sufferage concept, this method uses the percentage slack described below to determine which task suffers most if it is not mapped onto its "best" machine, where the slack is an indicator of how much the ATC

entry can differ from the corresponding ETC entry without violating the deadline.

The Slack Sufferage method can be summarized by the following procedure, which starts when a new task arrives and generates a mapping event. When the mapping event begins, it is assumed that none of the mappable tasks are mapped, i.e., they are not in any machine wait queue. In this heuristic, the *percentage slack* for task *i* on machine *j* using a given deadline *d* is defined as

$$PS(i, j, d) = 1 - (\mathrm{ETC}(i, j)/(d - \mathrm{mat}(j))).$$

(1) Generate a task list that includes all the mappable tasks.
(2) For each task in the mappable task list, for each machine calculate the $\mathrm{PS}(i, j, d)$, where d is task i's 100% deadline. $\mathrm{PS}(i, j, d) = -1$ for a machine if the task misses its deadline on that machine.
    For a given task i, if $\mathrm{PS}(i, j, d) < 0$ for all machines
      recalculate $\mathrm{PS}(i, j, d)$ for each machine using $d = 50\%$ deadline
      if $\mathrm{PS}(i, j, d) < 0$ for all machines
        recalculate $\mathrm{PS}(i, j, d)$ for each machine using $d = 25\%$ deadline
        if $\mathrm{PS}(i, j, d) < 0$ for all machines
          recalculate $\mathrm{PS}(i, j, d)$ for each machine using $d =$ end of evaluation period.
(3) For each task, determine the maximum percentage slack machine.
(4) Sort tasks by their worth (worth is calculated using the deadline factor associated with *d*).
(5) If there is more than one task with the current highest worth, check if tasks have the same maximum percentage slack machine (*contention*).
(6) If there is no contention, map all highest worth tasks. If there is contention among the highest worth tasks, map the most critical task (the task with the largest difference of percentage slack between the best percentage slack and the second best percentage slack machines).
(7) Remove mapped tasks from the task list.
(8) Update machine available times for step (2)'s calculation.
(9) Repeat steps (2)–(8) until all tasks are mapped.

4.8. Switching Algorithm

Part of the *Switching Algorithm* heuristic builds on the concept underlying the switching algorithm in [25]. The basic idea is to first try to map tasks onto their best machine according to a metric. Then, when the load on the machines becomes unbalanced, the strategy is changed to balance the load. After the load becomes balanced then the mapping method is changed back to the original scheme. Switching Algorithm can be summarized by the following procedure, which starts when a new task arrives and generates a mapping event. The *load balance ratio* for the system in the heuristic is the ratio of the earliest machine available time over all machines to the latest machine available time. A high threshold and a low threshold were determined experimentally for this ratio (high threshold>low threshold). Initially, new tasks are mapped onto their minimum completion time machine. Tasks are always inserted at the end of the chosen machine wait queue and then they are moved if necessary.

(1) Calculate the load balance ratio for the system.
(2) If the load balance ratio>high threshold, switch the method to its minimum execution time machine to map the new task.
    If the load balance ratio<low threshold, switch the method to map the new task to its minimum completion time machine.
    If low threshold $\leqslant$ load balance ratio $\leqslant$ high threshold, method to map the new task is not changed.
(3) The mappable tasks in the machine wait queue where the new task is mapped are reordered using their priority. If tasks have the same priority then order the tasks using their 100% deadlines (with earliest 100% deadline task coming first).

5. Comparing the performance of dynamic mapping heuristics to the static mapping heuristics

5.1. Overview

This research is a dynamic resource management research, therefore the two static mapping techniques described in this section are used to compare to the dynamic methods in Section 4 only. These static techniques assume complete a priori knowledge of when all tasks arrive and priority levels and deadlines of all tasks. Thus, these methods are not viable to solve this mapping problem dynamically, as must be done.

5.2. Two phase simulated annealing

The *SA* technique is an iterative improvement process of local search to obtain the global optimum of some given function. The SA technique has proven to be quite effective in approximating global optima and variations on the SA are used for many different NP-hard combinatorial problems. The *two phase simulated annealing* (*TPSA*) heuristic described here builds on earlier SA research (e.g., [8,29]).

The TPSA technique starts with an initial temperature, a function to decrease temperature, and an initial mapping solution. In this research, the initial temperature was arbitrarily selected to be 100,000 because the temperature had to be sufficiently large at the start of the TPSA method so that the mapping solution does not converge quickly and fall into an early local minimum. The temperature T was decreased at each iteration using the formula, $T = \alpha \times T$, where α is set to 0.99. For the initial mapping solution, a solution generated by the best dynamic mapping heuristic for each scenario was used. The machine assignment and the ordering of the tasks on a machine wait queue were used to determine the start times and the finish times of the tasks. These start and finish times are used in the calculation of the worth of each task.

At each iteration, (1) the current mapping solution is changed to make a new mapping solution, (2) the new mapping is compared to the current mapping, and (3) the temperature is

lowered. If the new mapping solution has a higher evaluation value V (from Section 2.2.3), then the new mapping is chosen to be the current solution. When the new mapping is worse than the current mapping, it is probabilistically chosen to be the current mapping. The uphill probability is determined using $e^{-\frac{|value_{\text{new}} - value_{\text{current}}|}{T}}$, where the equation decides the probability of going uphill. For example, if the equation $= 0.2$, then there is a 20% chance that the new solution will be chosen over the better current solution even though it is a worse solution. The TPSA heuristic runs until it meets a certain predetermined stopping criteria. The stopping criteria are when the temperature goes below 10^{-200}, or when the current mapping solution is unchanged for a predetermined number of iterations.

When generating a new mapping from the current mapping, two methods are used. The first method randomly chooses a task and maps it onto a randomly chosen machine and position in the machine wait queue (*mutation*). The second method randomly chooses two tasks and swaps their machine assignments and queue positions in the machine wait queue (*swap*). In TPSA, there are two phases. In the first phase, the mutation method is used for the first 4000 iterations or until the solution does not change for 400 iterations. Then, in the second phase the swap method is used until the mapping solution is not changed for 1000 iterations or until the temperature is zero (i.e., 10^{-200}). For each trial, the heuristic is run five times and the mapping with the best solution is selected. The intuition behind the TPSA method is that after some number of mutations, a near-optimal number of tasks per machine will be found. Then swapping two tasks will maintain the number of tasks on two machines (or on a machine) while trying to search for a better solution.

At every iteration, when the current solution is changed, the machine assignment and/or the order of the tasks may be changed. To compare the current solution and the new solution, the new solution must be evaluated. To evaluate a solution, the start times and the finish times of tasks must be determined. The following method is used to determine the start and finish times.

(1) For machine 1, move all tasks from the machine wait queue into a task list, retaining the same ordering.
(2) One by one, the tasks are taken from this task list and inserted to start as early as possible (e.g., at the task's arrival time or after another task).
(3) Determine the start and finish times of all tasks.
(4) Do for all machines.

Our research group also used a modified version of the TPSA for a different HC environment in [22].

5.3. Genetic algorithm

The *GA* is based on biological evolution and is used for searching large solution spaces. The GA method shown here is based on [28] and [32]. The general GA starts by generating an initial population and evaluating the population. Then while the stopping criteria are not met, selection, crossover, mutation, and evaluation are done in this order.

The GA implemented in this research starts with an initial population of 100 chromosomes (possible solutions or mappings). One cycle of selection, crossover, mutation, and evaluation is called a *generation*. The population size is maintained at 100 for all generations. Each chromosome is a matrix that has the assignment of tasks onto machines and the order of the tasks to be considered for execution on each machine wait queue. Among the 100 chromosomes, one chromosome is the mapping from the best dynamic mapping solution for each scenario (*seeding*) and the rest of the chromosomes are generated randomly. When randomly generating the initial 99 chromosomes, with equal probability, a task is picked from the list of tasks. Then, a machine is determined with equal probability and the selected task is put at the end of the machine queue. The process continues until all tasks are put into any of the machines. The chromosomes are evaluated by the value function (V) shown at the end of Section 2.2.3. The starting and stopping times of all tasks are determined using the method discussed in Section 5.2.

In the selection phase, a rank-based roulette wheel scheme [4] is used. This method probabilistically duplicates some chromosomes while deleting others, where better solutions have a higher probability of surviving the process and being duplicated in the next generation. *Elitism*, the property of guaranteeing that the best chromosome remains in the population, is implemented.

After the population for the next generation is determined, the crossover operation is performed. Going through the population once, parents are selected randomly with some probability (determined empirically to be 90% in this research). When two parents are determined (e.g., Fig. 1(a) and (b)), the two parents are used for crossover. When a parent is selected, it is used only that one time. Using a randomly chosen task, both parents are divided into a head and a tail (Fig. 1(c) and (d)). When determining the two children, the head of one parent, the tail of the other parent, and the machine assignment and the position information are used. When two tasks (one from the head and the other from the tail) have the same machine assignment and the same position in the machine wait queue, the order of these tasks are randomly determined (shown in Fig. 1(e) and (f) as two tasks in one position slot).

After the crossover is done, the mutation operation is performed. Going through the population once, a chromosome is considered for mutation with a probability of 20% (empirically determined). For the chromosome that is selected, a random task's machine assignment and machine wait queue position are randomly changed. For both crossover and mutation, random operations select values from a uniform distribution. Finally, the chromosomes are evaluated and this completes one generation of the GA.

The GA stops when any one of three conditions are met: (a) 1000 total generations, (b) no change in the elite chromosome for 200 iterations, or (c) all chromosomes converge to the same mapping. The stopping criteria that occurred most was (b) and the second was (a). Even with an increase in the maximum total generation allowed to 2000, there was no significant increase (less than 1%) in the value of the best solution.

machine \ position	0	1	2
0	1	4	3
1	2	5	
2	6		

(a)

machine \ position	0	1	2
0	2	6	5
1	1	4	
2	3		

(b)

	head			tail		
task	1	2	3	4	5	6
machine	0	1	0	0	1	2
position	1	1	3	2	2	1

(c)

	head			tail		
task	1	2	3	4	5	6
machine	1	0	2	1	0	0
position	0	0	0	1	3	2

(d)

machine \ position	0	1	2
0	1	6	5, 3
1	2	4	
2			

(e)

machine \ position	0	1	2
0	2	4	
1	1	5	
2	3, 6		

(f)

Fig. 1. The crossover procedure starts by picking two parents: (a) parent 0 and (b) parent 1. A random task is picked (task 3) and the head and tail of the two parents are determined in (c) and (d). The resulting two children are shown in (e) child 0 and (f) child 1.

6. Simulation setup

The simulated HC system is considered oversubscribed such that not all tasks can finish by their 100% deadline. To model such an environment, the arrival rates of tasks are determined such that there are enough tasks in the system.

The system is simulated for a 250-min period. The period from 0 to 10 min is the *system start-up period*, where the mean task inter-arrival time is fast to populate the system (using a Poisson distribution with a mean task inter-arrival time of 3.5 s). The period between 10 and 250 min is the *evaluation period* (i.e., the period where the heuristics' performance is measured). During this period the mean task inter-arrival time is 14 s. In addition, three (10 min) bursty periods are introduced randomly during the evaluation period, where the arrival rate is increased. These periods do not overlap with each other and have a mean task inter-arrival time of 7 s. The HC system consists of eight machines and an average of 1276 tasks. A *trial* is defined as one such simulation of the HC system.

The estimated execution times of all tasks taking heterogeneity into consideration are generated using the gamma distribution method described in [3]. Two different cases of ETC heterogeneities are used in this research, the high task and high machine heterogeneity (*high heterogeneity*) case and the low task and low machine heterogeneity (*low heterogeneity*) case. For both heterogeneity cases, a task mean and *coefficient of variation* (*COV*) are used. (The COV is defined as the standard deviation divided by the mean.) The high heterogeneity cases use a mean task execution time of 3 min and a COV of 0.9 (task heterogeneity) to calculate the values for all of the elements in a task vector (where the number of elements equal the total number of tasks). Then using the *i*th element of the vector as the mean and a COV of 0.9 (machine heterogeneity), the ETC values for task i on all the machines are calculated. The low heterogeneity cases use a mean task execution time of 3 min and a COV of 0.3 for task heterogeneity and 0.3 for machine heterogeneity.

The ATC values are generated for the purpose of determining how well the heuristics perform when the actual task execution times on the machines vary from the ETC values. When task i starts executing on machine j, $\mathrm{ATC}(i, j)$ is used for the calculation of the actual end time of that task. The calculation of the machine available time of a machine is the actual end time of the task that is executing plus the ETC values of tasks that are waiting in the machine queue.

For a given ETC matrix, $\mathrm{ATC}(i, j)$ is computed using $\mathrm{ETC}(i, j)$ as the mean and a COV of 0.1. The average difference of the ETC values and ATC values is 8%. The minimum difference is 0% and the maximum difference is 50%. Each ETC/ATC pair corresponds to one trial.

There are two types of weightings that are assigned to high, medium, and low priority level tasks, namely, 16, four, and one for the *heavy priority weighting* and four, two, and one for the *light priority weighting*. Of all the tasks that arrive, approximately one third will be of each priority level.

The deadline of each task is calculated using the following process. A deadline for a task is the arrival time of the task, plus the median execution time of the task across all machines, plus a multiplier times the median execution time of all tasks (i.e., 2.4 min in this study). Two types of deadlines, i.e., *loose* and *tight*, are used in the simulation. The multiplier was used to differentiate between the two types of deadlines. For the loose deadline, the multiplier is four, eight, and 12 for the 100%, 50%, and 25% deadlines, respectively. For the tight deadline, the multiplier is one, two, and four for the 100%, 50%, and 25% deadlines, respectively.

A *scenario* is one combination of the two types of heterogeneity, two types of priority weighting, and two types of deadline. Therefore, there are a total of eight scenarios. For each of the scenarios, 50 trials are run.

7. Results

The simulation results are shown in Figs. 2 and 3 for the two different types of deadlines. Each figure consists of four scenarios (all combinations of high/low heterogeneity and heavy/light priority weighting). Two static mapping heuristics were run for comparison to the dynamic mapping heuristics. The static mapping heuristics used the ATC matrices and all

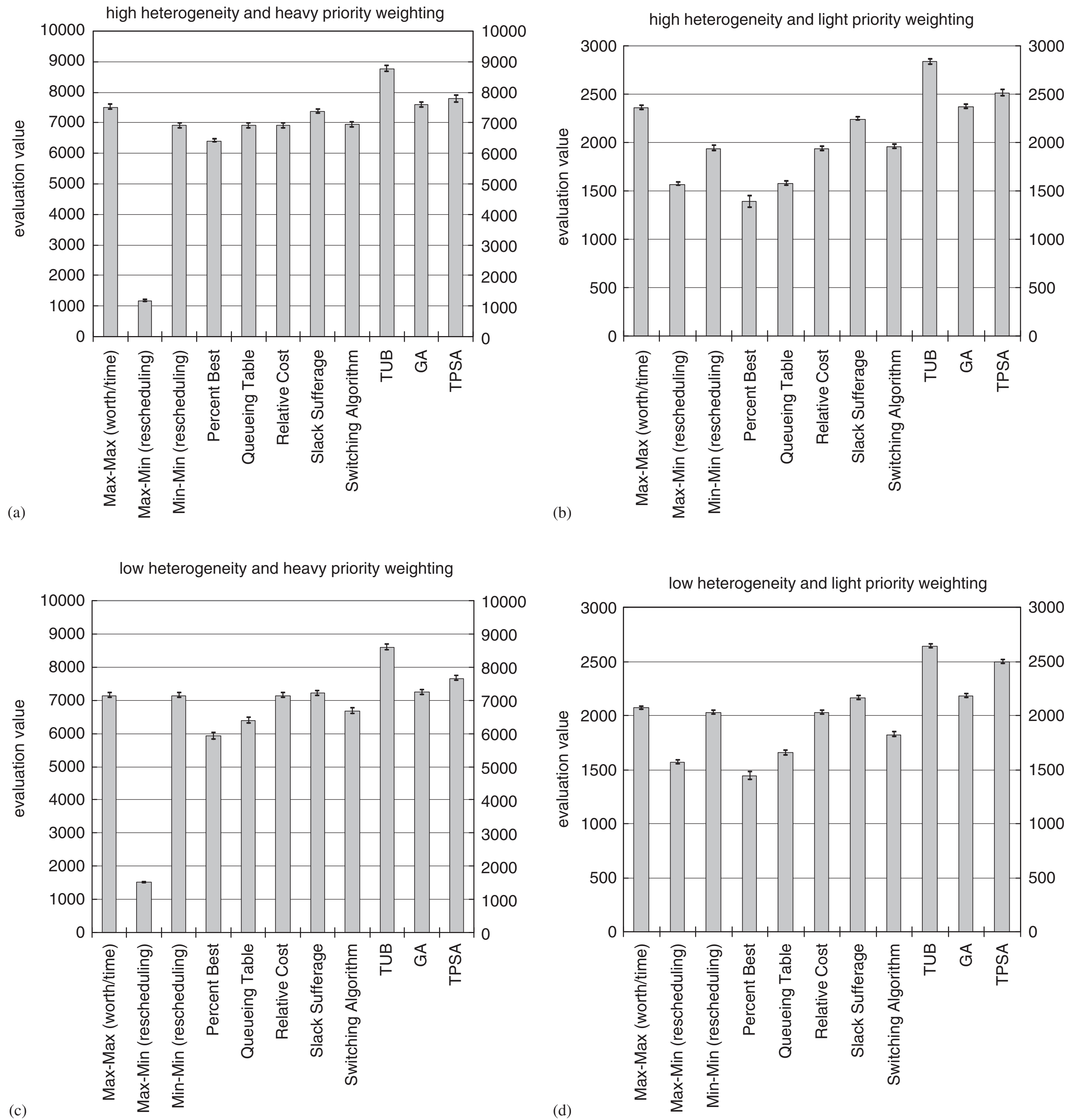


Fig. 2. The simulation results using loose deadlines for (a) high heterogeneity with the heavy priority weighting of 16, four, and one for high, medium, and low priority levels, (b) high heterogeneity with the light priority weighting of four, two, and one for high, medium, and low priority levels, (c) low heterogeneity with the heavy priority weighting of 16, four, and one for high, medium, and low priority levels, and (d) low heterogeneity with the light priority weighting of four, two, and one for high, medium, and low priority levels.

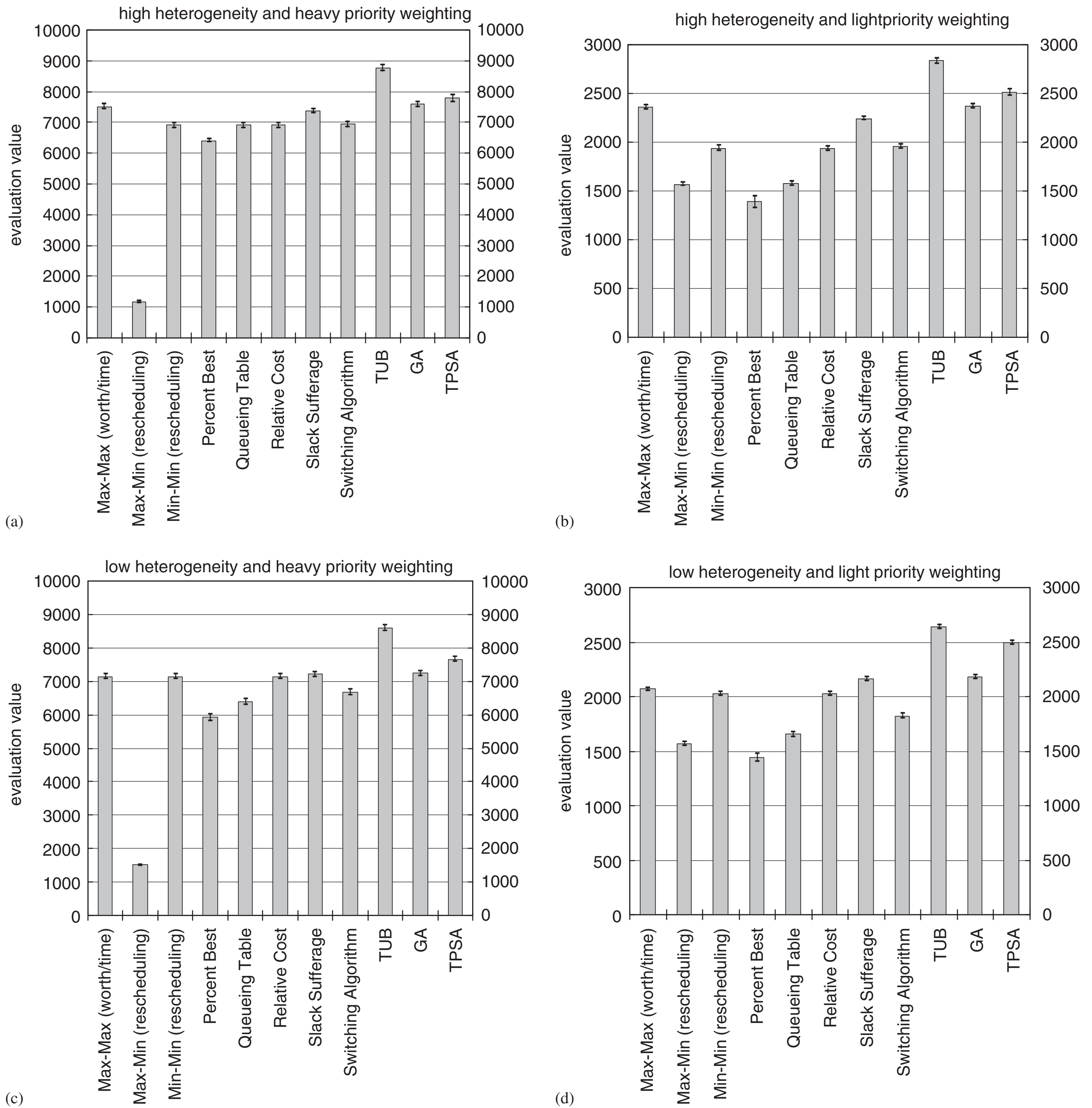


Fig. 3. The simulation results using tight deadlines for (a) high heterogeneity with the heavy priority weighting of 16, four, and one for high, medium, and low priority levels, (b) high heterogeneity with the light priority weighting of four, two, and one for high, medium, and low priority levels, (c) low heterogeneity with the heavy priority weighting of 16, four, and one for high, medium, and low priority levels, and (d) low heterogeneity with the light priority weighting of four, two, and one for high, medium, and low priority levels.

the arrival information of the tasks. The averages over 50 trials and the 95% confidence intervals [19] are shown (most of the intervals are very close to the mean).

In Fig. 2, simulation results using loose deadlines are shown. For the high heterogeneity cases, Max–Max performed the best among the dynamic heuristics (86% and 83% of the UB for heavy and light priority weightings, respectively), while Slack Sufferage was the best in the low heterogeneity cases (84% and 81% of the UB for high and low priority weighting, respectively) with the Max–Max heuristic a close second. The relative performance among the rest of the heuristics was similar in all the scenarios, with Max–Min performing the worst. In the high

priority cases, there is less performance difference among all heuristics (excluding Max–Min) than in the low priority cases. This is because all heuristics map the high priority tasks that can meet their 100% deadline first and if the weighting of the high priority task is dominant, then there is less difference in performance.

In Fig. 3, as can be expected, the performance of all heuristics degraded as tasks are more likely to miss their deadlines because of the tight deadlines. The relative performance among the heuristics remained the same (i.e., Max–Max and Slack Sufferage performed well) except for the Queueing Table method. Queueing Table was the best in the low heterogeneity cases and the performance of the Queueing Table heuristic degraded the least from Figs. 2 to 3 for each of the scenarios. Queueing Table is one of the heuristics that uses urgency to order the execution of tasks in a machine wait queue and this accounts for the limited degradation. Percent Best and Switching Algorithm also use urgency to order the execution of tasks in a machine wait queue (ties are broken using the method of earlier 100% deadline first, see Sections 4.4 and 4.8), but they do not determine whether a task can finish before its 100% deadline or not. In their mapping process, assuming tasks have the same priority weighting, tasks that cannot finish by their 25% deadline may be scheduled to execute in front of a task that can meet its 100% deadline. This has a higher probability of occurring in the scenarios that use the tight deadline than in those that use the loose deadline, because tasks with the tight deadline have a higher probability of violating their deadlines. However, in the Queueing Table heuristic this will not happen because, if a task misses its 100% deadline, then the urgency is set to negative infinity.

It is interesting that the Max–Max and Slack Sufferage heuristics generally perform comparably while the relative performance of the two changes according to the heterogeneity. For both deadline types, in the high heterogeneity cases, Max–Max performs better than Slack Sufferage. However, in the low heterogeneity cases, Slack Sufferage performs better than Max–Max.

The following is an example of a high heterogeneity case where Max–Max will do better than Slack Sufferage. Assume that there are two tasks ($t1$ and $t2$) with the same priority and two machines ($m1$ and $m2$), where the machine available times are 5 and 155 s, respectively, and the estimated execution times and deadlines are as shown in Table 2. Assume that when the 100% deadline is not met, the 50% deadline will be met.

Using the information from the previous paragraph, the two tasks will miss their 100% deadline on $m2$. Thus, the deadline factor will be 0.5 for both tasks on $m2$ and the worth (priority weighting multiplied by deadline factor) will be half of that on $m1$. Therefore, after calculating the fitness value, Max–Max will determine the two task/machine pairs $t1/m1$ and $t2/m1$ in the first phase and pick the $t2/m1$ pair first to map and then pick the $t1/m1$ pair to map. Slack Sufferage will first calculate the percentage slack for all task/machine pairs as shown in Table 3. Because the best machine is the same for both tasks, the task that is more critical is picked first. In this case, $t1$ is picked and mapped to $m1$. Another calculation of the percentage

Table 2
An example of tasks with high heterogeneity estimated execution times in seconds

Tasks	Machines		100% deadline
	$m1$	$m2$	
$t1$	38	20	160
$t2$	3	10	10

Table 3
The calculation of the percentage slack values using Table 2

Tasks	Machines		100% deadline
	$m1$	$m2$	
$t1$	0.75	−1	160
$t2$	0.4	−1	10

Table 4
An example of tasks with low heterogeneity estimated execution times in seconds

Tasks	Machines		100% deadline
	$m1$	$m2$	
$t1$	9	4.4	16
$t2$	5	4	13

slack value for $t2$ after $t1$ is mapped indicates that $t2$ will miss its 100% deadline on both $m1$ and $m2$. The task $t2$ for Slack Sufferage will violate its 100% deadline, while Max–Max completes both tasks before their 100% deadline.

The following is a low heterogeneity case where Slack Sufferage will do better than Max–Max. The ETC values of tasks have a higher probability of being similar in low versus high heterogeneity cases. The fitness value of a task on all machines calculated by Max–Max may be similar. In the first phase of Max–Max, the task/machine pair that has the maximum fitness value is determined. Assume that the worth is the same on all machines (i.e., the deadline factor is the same for all machines). In this example, selecting the machine with the higher percentage slack for a task may give the task a higher probability of not violating its deadline rather than picking the most worth per unit time machine. As an example of a low heterogeneity case where Slack Sufferage does better than Max–Max, assume that there are two tasks ($t1$ and $t2$) with the same priority and two machines ($m1$ and $m2$), where the machine available times are 4 and 8 s, respectively, and that estimated execution times and deadlines are as shown in Table 4. Assume that when the 100% deadline is not met, the 50% deadline will be met.

Max–Max will determine the two task/machine pairs $t1/m2$ and $t2/m2$ in the first phase (the worth of both tasks on both machines are the same) and pick the $t2/m2$ pair first to map. After mapping $t2$ and the machine available time is updated, if

Table 5
The calculation of the percentage slack values using Table 4

Tasks	Machines		100% deadline
	$m1$	$m2$	
$t1$	0.25	0.45	16
$t2$	0.44	0.2	13

$t1$ is mapped on $m1$, it does not violate its 100% deadline and if $t1$ is mapped on $m2$, it misses its 100% deadline. However, the fitness value (calculated using the deadline factor of 0.5 for $m2$) is higher for $t1$ on $m2$. Therefore, $t1$ is mapped on $m2$. Slack Sufferage will first calculate the percentage slack for all task/machine pairs as shown in Table 5. In this case, $t1$ is mapped onto $m2$ and $t2$ is mapped onto $m1$. Slack Sufferage finishes both tasks by their 100% deadline and by time 12.4. Max–Max completes both tasks by time 16.4 and $t1$ misses its 100% deadline.

The reason for Max–Max generally outperforming the next tier of heuristics (i.e., Percent Best, Queueing Table, Min–Min, and Relative Cost) is because Max–Max uses the worth per unit time fitness function to determine which task to map first. Even if a task has very low worth it may still be picked to be mapped first if its execution time is very fast. Other heuristics use the worth or the weighted priority of the task as the main factor for decision making.

The fastest heuristics are Queueing Table and Switching Algorithm because these heuristics basically map only the new task arriving in the system. The average execution times of a mapping event for the heuristics Max–Max, Max–Min, Min–Min, Percent Best, Queueing Table, Relative Cost, Slack Sufferage, and Switching Algorithm are 0.11, 0.45, 0.35, 0.44, 0.0004, 0.36, 0.28, and 0.0002 s, respectively.

8. Summary

For the heterogeneous computing (HC) environment described in this research, eight dynamic heuristics were designed, developed, and simulated. Dynamically arriving tasks with priorities and multiple deadlines were mapped using the heuristics proposed in this research.

When loose deadlines were used, Max–Max and Slack Sufferage were the two best dynamic approaches and performed comparably. In many scenarios, these heuristics achieve over 80% of the upper bound (UB) that was derived. When tight deadlines were used, the performance of all heuristics is degraded. In the high heterogeneity cases, Max–Max and Slack Sufferage are still the heuristics of choice, however, in the low heterogeneity cases, Queueing Table (that uses urgency in its mapping process) performed the best. The fastest heuristics were the Queueing Table and the Switching Algorithm.

Static heuristics (two phase simulated annealing (TPSA) and genetic algorithm (GA)) were used to compare to the performance of the dynamic mapping heuristics. The static heuristics did improve the best solution found for each scenario. Both heuristics performed comparably in most cases with the TPSA doing slightly better in the low heterogeneity and low priority weighting scenarios (12.5% increase in performance over the best mapping determined by the dynamic heuristics). The execution time for GA was about 23.5 times that of TPSA. An optimal mapping falls between the TPSA/GA solutions and the UB.

Acknowledgments

The authors thank Prasanna Sugavanam for his comments. A preliminary version of portions of this paper was presented at the 12th Heterogeneous Computing Workshop.

References

[1] S. Ali, T.D. Braun, H.J. Siegel, A.A. Maciejewski, N. Beck, L. Boloni, M. Maheswaran, A.I. Reuther, J.P. Robertson, M.D. Theys, B. Yao, Characterizing resource allocation heuristics for heterogeneous computing systems, in: A.R. Hurson (Ed.), Advances in Computers Volume 63, Parallel, Distributed, and Pervasive Computing, Elsevier, Amsterdam, The Netherlands, 2005, pp. 91–128.

[2] S. Ali, J.-K. Kim, Y. Yu, S.B. Gundala, S. Gertphol, H.J. Siegel, A.A. Maciejewski, V. Prasanna, Utilization-based techniques for statically mapping heterogeneous applications onto the HiPer-D heterogeneous computing system, Parallel Distrib. Comput. Practices 5 (4) (2002) (Special Issue on Parallel Numeric Algorithms on Faster Computers).

[3] S. Ali, H.J. Siegel, M. Maheswaran, D. Hensgen, S. Ali, Representing task and machine heterogeneities for heterogeneous computing systems, Tamkang J. Sci. Eng. 3 (3) (2000) 195–207 (invited, special 50th Anniversary Issue).

[4] J.E. Baker, Reducing bias and inefficiency in the selection algorithm, in: Second International Conference on Genetic Algorithms and Their Application, Lawrence Erlbaum Associates, Inc., Hillsdale, NJ, 1987, pp. 14–21.

[5] I. Banicescu, V. Velusamy, Performance of scheduling scientific applications with adaptive weighted factoring, 10th IEEE Heterogeneous Computing Workshop (HCW 2001), in: Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS 2001), paper HCW 06, April 2001.

[6] H. Barada, S.M. Sait, N. Baig, Task matching and scheduling in heterogeneous systems using simulated evolution, 10th IEEE Heterogeneous Computing Workshop (HCW 2001), in: Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS 2001), paper HCW 15, April 2001.

[7] T.D. Braun, H.J. Siegel, N. Beck, L. Boloni, R.F. Freund, D. Hensgen, M. Maheswaran, A.I. Reuther, J.P. Robertson, M.D. Theys, B. Yao, A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems, J. Parallel Distrib. Comput. 61 (6) (2001) 810–837.

[8] T.D. Braun, H.J. Siegel, A.A. Maciejewski, Heterogeneous computing: goals, methods, and open problems, 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001), June 2001, pp. 1–12 (invited keynote paper).

[9] C.D. Cavanaugh, L.R. Welch, B.A. Shirazi, E. Huh, S. Anwar, Quality of service negotiation for distributed, dynamic real-time systems, in: J. Rolim et al. (Ed.), Parallel and Distributed Processing, Lecture Notes in Computer Science, vol. 1800, Springer, New York, NY, 2000, pp. 757–765.

[10] E.G. Coffman Jr. (Ed.), Computer and Job-Shop Scheduling Theory, Wiley, New York, NY, 1976.

[11] M.M. Eshaghian (Ed.), Heterogeneous Computing, Artech House, Norwood, MA, 1996.

[12] D. Fernandez-Baca, Allocating modules to processors in a distributed system, IEEE Trans. Software Eng. SE-15 (11) (1989) 1427–1436.

[13] I. Foster, C. Kesselman (Eds.), The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, San Fransisco, CA, 1999.
[14] R.F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J.D. Lima, F. Mirabile, L. Moore, B. Rust, H.J. Siegel, Scheduling resources in multiuser, heterogeneous, computing environments with SmartNet, Seventh IEEE Heterogeneous Computing Workshop (HCW 1998), March 1998, pp. 184–199.
[15] R.F. Freund, H.J. Siegel, Heterogeneous processing, IEEE Comput. 26 (6) (1993) 13–17.
[16] A. Ghafoor, J. Yang, A distributed heterogeneous supercomputing management system, IEEE Comput. 26 (6) (1993) 78–86.
[17] E. Huh, L.R. Welch, B.A. Shirazi, B. Tjaden, C.D. Cavanaugh, Accommodating QoS prediction in an adaptive resource management framework, in: J. Rolim et al. (Ed.), Parallel and Distributed Processing, Lecture Notes in Computer Science, vol. 1800, Springer, New York, NY, 2000, pp. 792–799.
[18] O.H. Ibarra, C.E. Kim, Heuristic algorithms for scheduling independent tasks on non-identical processors, J. ACM 24 (2) (1977) 280–289.
[19] R. Jain, The Art of Computer Systems Performance Analysis, Wiley, New York, NY, 1991.
[20] M. Kafil, I. Ahmad, Optimal task assignment in heterogeneous distributed computing systems, IEEE Concurrency 6 (3) (1998) 42–51.
[21] A. Khokhar, V.K. Prasanna, M.E. Shaaban, C. Wang, Heterogeneous computing: challenges and opportunities, IEEE Comput. 26 (6) (1993) 18–27.
[22] J.-K. Kim, Resource management in heterogeneous computing systems: continuously running applications, tasks with priorities and deadlines, and power constrained mobile devices, Ph.D. Thesis, Department of Electrical and Computer Engineering, School of Engineering, Purdue University, August 2004.
[23] J.-K. Kim, D.A. Hensgen, T. Kidd, H.J. Siegel, D.St. John, C. Irvine, T. Levin, N.W. Porter, V.K. Prasanna, R.F. Freund, A flexible multi-dimensional QoS performance measure framework for distributed heterogeneous systems, Cluster Comput. 9 (3) (2006) 281–296.
[24] C. Leangsuksun, J. Potter, S. Scott, Dynamic task mapping algorithms for a distributed heterogeneous computing environment, Fourth IEEE Heterogeneous Computing Workshop (HCW ’95), 1995, pp. 30–34.
[25] M. Maheswaran, S. Ali, H.J. Siegel, D. Hensgen, R.F. Freund, Dynamic mapping of a class of independent tasks onto heterogeneous computing systems, J. Parallel Distrib. Comput. 59 (2) (1999) 107–121.
[26] M. Maheswaran, T.D. Braun, H.J. Siegel, Heterogeneous distributed computing, in: J.G. Webster (Ed.), Encyclopedia of Electrical and Electronics Engineering, vol. 8, Wiley, New York, NY, 1999, pp. 679–690.
[27] Z. Michalewicz, D.B. Fogel, How to Solve It: Modern Heuristics, Springer, New York, NY, 2000.
[28] M. Mitchell, An Introduction to Genetic Algorithms, MIT Press, Cambridge, MA, 1996.
[29] M.D. Natale, J.A. Stankovic, Scheduling distributed real-time tasks with minimum jitter, IEEE Trans. Comput. 49 (4) (2000) 303–316.
[30] K. Ramamritham, J.A. Stankovic, W. Zhao, Distributed scheduling of tasks with deadlines and resource requirements, IEEE Trans. Comput. 38 (8) (1989) 1110–1123.
[31] H. Singh, A. Youssef, Mapping and scheduling heterogeneous task graphs using genetic algorithms, in: Fifth IEEE Heterogeneous Computing Workshop (HCW 1996), 1996, pp. 86–97.
[32] L. Wang, H.J. Siegel, V.P. Roychowdhury, A.A. Maciejewski, Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach, J. Parallel Distrib. Comput. 47 (1) (1997) 8–22 (Special Issue on Parallel Evolutionary Computing).
[33] L.R. Welch, B. Ravindran, B.A. Shirazi, C. Bruggeman, Specification and modeling of dynamic, distributed real-time systems, 19th IEEE Real-Time Systems Symposium (RTSS ’98), December 1998, pp. 72–81.
[34] L.R. Welch, B.A. Shirazi, A dynamic real-time benchmark for assessment of QoS and resource management technology, Fifth IEEE Real-Time Technology and Applications Symposium (RTAS ’99), June 1999, pp. 36–45.
[35] L.R. Welch, B.A. Shirazi, B. Ravindran, C. Bruggeman, DeSiDeRaTa: QoS management technology for dynamic, scalable, dependable, real-time systems, in: F. De Paoli, I.M. MacLeod (Eds.), Distributed Computer Control Systems 1998, Elsevier Science, Kidlington, UK, 1999, pp. 7 –12, (Proceedings volume from the 15th International Federation of Automatic Control (IFAC) Workshop, September 1998).
[36] L.R. Welch, P.V. Werme, B. Ravindran, L.A. Fontenot, M.W. Masters, D.W. Mills, B.A. Shirazi, Adaptive QoS and resource management using a posteriori workload characterizations, Fifth IEEE Real-Time Technology and Applications Symposium (RTAS ’99), June 1999, pp. 266–275.
[37] M.-Y. Wu, W. Shu, A high-performance mapping algorithm for heterogeneous computing systems, 15th International Parallel and Distributed Processing Symposium (IPDPS 2001), April 2001.
[38] M.-Y. Wu, W. Shu, H. Zhang, Segmented min–min: a static mapping algorithm for meta-tasks on heterogeneous computing systems, Ninth IEEE Heterogeneous Computing Workshop (HCW 2000), May 2000, pp. 375–385.
[39] D. Xu, K. Nahrstedt, D. Wichadakul, QoS and contention-aware multi-resource reservation, Cluster Comput. 4 (2) (2001) 95–107.
[40] J. Yang, I. Ahmad, A. Ghafoor, Estimation of execution times on heterogeneous supercomputer architectures, International Conference on Parallel Processing, August 1993, pp. I-219–I-226.
[41] V. Yarmolenko, J. Duato, D.K. Panda, P. Sadayappan, Characterization and enhancement of dynamic mapping heuristics for heterogeneous systems, IEEE International Workshop on Parallel Processing, August 2000, pp. 437–444.

Jong-Kook Kim received his M.S. degree in Electrical Engineering and his Ph.D. degree in Electrical and Computer Engineering from Purdue University in May 2000 and August 2004, respectively. He received his B.S. degree in electronic engineering from Korea University, Seoul, Korea in 1998. In 2005, he joined Samsung SDS, Information Technology Research and Development Center as a Senior Researcher. His research interests include heterogeneous distributed computing, computer architecture, performance measures, resource management, evolutionary heuristics, energy-aware computing, grid computing, and resource virtualization. He is a member of the IEEE, IEEE Computer Society, and ACM.

Sameer Shivle received his M.S. degree in Electrical and Computer Engineering from Colorado State University. He received a B.E. degree in Electrical Engineering from the Government College of Engineering, Pune, India. His fields of interest are heterogeneous computing, computer architecture and digital system design.

Howard Jay Siegel holds the endowed chair position of Abell Distinguished Professor of Electrical and Computer Engineering at Colorado State University (CSU), where he is also a Professor of Computer Science. He is the Director of the CSU Information Science and Technology Center (ISTeC). ISTeC a university-wide organization for promoting, facilitating, and enhancing CSU’s research, education, and outreach activities pertaining to the design and innovative application of

computer, communication, and information systems. Prof. Siegel is a Fellow of both the IEEE and the ACM. From 1976 to 2001, he was a professor in the School of Electrical and Computer Engineering at Purdue University. He received a B.S. degree in Electrical Engineering and a B.S. degree in management from the Massachusetts Institute of Technology (MIT), and the M.A., M.S.E., and Ph.D. degrees from the Department of Electrical Engineering and Computer Science at Princeton University. He has co-authored over 300 technical papers. His research interests include heterogeneous parallel and distributed computing, communication networks, parallel algorithms, parallel machine interconnection networks, and reconfigurable parallel computer systems. He was a Coeditor-in-Chief of the Journal of Parallel and Distributed Computing, and has been on the Editorial Boards of both the IEEE Transactions on Parallel and Distributed Systems and the IEEE Transactions on Computers. He was Program Chair/Co-Chair of three major international conferences, General Chair/Co-Chair of four international conferences, and Chair/Co-Chair of five workshops. He is currently on the Steering Committees of five continuing conferences/workshops. He is a member of the Eta Kappa Nu electrical engineering honor society, the Sigma Xi science honor society, and the Upsilon Pi Epsilon computing sciences honor society.

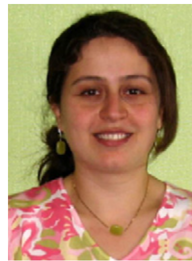
Anthony A. Maciejewski received the B.S.E.E, M.S., and Ph.D. degrees in Electrical Engineering in 1982, 1984, and 1987, respectively, all from The Ohio State University under the support of an NSF graduate fellowship. From 1985 to 1986 he was an American Electronics Association Japan Research Fellow at the Hitachi Central Research Laboratory in Tokyo, Japan where he performed work on the development of parallel processing algorithms for computer graphic imaging. From 1988 to 2001, he was a Professor of Electrical and Computer Engineering at Purdue University. In 2001, he joined Colorado State University as a Professor of Electrical and Computer Engineering. Prof. Maciejewski's primary research interests relate to the analysis, simulation, and control of robotic systems and he has co-authored over 100 published technical articles in these areas. He is an Associate Editor for the IEEE Transactions on Robotics and Automation, a Regional Editor for the journal Intelligent Automation and Soft Computing, and was co-guest editor for a special issue on "Kinematically Redundant Robots" for the Journal of Intelligent and Robotic Systems. He serves on the IEEE Administrative Committee for the Robotics and Automation Society and was the Program Co-Chair (1997) and Chair (2002) for the International Conference on Robotics and Automation, as well as serving as the Chair and on the Program Committee for numerous other conferences.

Tracy D. Braun received his Ph.D. in Electrical and Computer Engineering from the School of Electrical and Computer Engineering at Purdue University in 2001. In 1997, he received his M.S.E.E. from the School of Electrical and Computer Engineering at Purdue University. He received a B.S. in Electrical and Computer Engineering with Honors and High Distinction from the University of Iowa in 1995. He has been a CISSP since 2003. He is also a member of AFCEA, IEEE, IEEE Computer Society, and Eta Kappa Nu honorary society. His research interests include information assurance and computer forensics.

Myron Schneider currently works as a hardware design engineer for Agilent Technologies Manufacturing Test Business Unit in Loveland, CO. He received a B.S. in Electrical Engineering from the University of Illinois at Urbana-Champaign in August 1996 and a Masters of Electrical Engineering from Colorado State University in December 2002. His technical interests and work experience include FPGA/CPLD design, hardware description languages, high-speed digital system design, reconfigurable computing, computer architecture, heterogeneous computing, adaptive algorithms, and automated manufacturing test systems.

Sonja Tideman received a B.S. in Computer Science from the University of New Mexico and a M.S. degree in Computer Science at Colorado State University. She is employed by Sandia National Laboratory in Albuquerque, NM. Her research interests include embedded systems, networking, and operating systems.

Ramakrishna Chitta is a Computer Science major pursuing his M.S. at Colorado State University, where he is currently a Graduate Teaching Assistant. He received his B.Tech degree in Computer Science and Engineering from Jawaharlal Nehru Technological University, Hyderabad, India in 2001. His fields of specialization are compilers and computer architecture. He is a member of ACM.



Raheleh B. Dilmaghani received her B.S. in Electrical Engineering from University of Tehran, Iran in 1996 (graduated Supra Cum Laude). Since graduation, she has acted as a technical lead for the Moshanir Engineering Consulting Firm. She is currently a Master's degree candidate with the Electrical and Computer Engineering Department at Colorado State University. Her current interests and areas of research are in computer networking, security, and heterogeneous computing environments.

Rohit S. Joshi received his M.S. degree in Electrical and Computer Engineering at Colorado State University. He received his B.S. in Electrical Engineering from University of Pune, India in May 2000. His research interests include VLSI system design, microprocessor based systems, and software testing.

Aditya Kaul is currently pursuing his Ph.D. in Industrial Engineering and Operations Research at the Harold Inge Marcus Department of Industrial and Manufacturing Engineering at The Pennsylvania State University. He received his Master's degree in Electrical Engineering from Colorado State University in August 2002 and B.S. in Electrical Engineering from Regional Engineering College, Surat, India in August 2000.

Ashish Sharma received his B.E. degree in Electronics and Power Engineering from Nagpur University, India in 2000 and his M.S. degree in Electrical Engineering from Colorado State University, Fort Collins, CO in 2004. He is currently pursuing his Ph.D. degree in Electrical Engineering at Colorado State University. His research is focused on bio-medical applications of non-thermal atmospheric pressure plasmas.

Siddhartha Sripada is a graduate student in the Department of Electrical and Computer Engineering at Colorado State University. He received a B.Tech degree in Electrical and Electronics Engineering from Nagarjuna University, India. His research interests include computer architecture, digital design, and heterogeneous computing. He has done projects in the fields of digital system design, heterogeneous computing, and fault tolerant computing. He is a student member of IEEE and an active member of the Nagarjuna University alumni.

Praveen Vangari is a graduate student of Colorado State University pursuing an M.S. degree in Electrical and Computer Engineering. He received his B.E. degree in the Vasavi College of Engineering (affiliated to the Osmania University, Hyderabad) in the field of Electronics and Communication Engineering. His research interests include computer architecture, system level hardware design, and VLSI.

Siva Sankar Yellampalli received his B.Tech in Electrical and Electronics Engineering from Jawaharlal Nehru Technological University in 2001. He is currently pursuing an M.S. degree in VLSI design at Louisiana State University. His research interests include IDDQ testing in nanometer technology, mixed signal design, and computer architecture.