Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2020) 28: 821 – 835

doi:10.3906/elk-1904-49

Hyperheuristics for explicit resource partitioning in simultaneous multithreaded processors

İsa Ahmet GÜNEY[1,*], **Kemal POYRAZ**[1], **Gürhan KÜÇÜK**[1], **Ender ÖZCAN**[2]
[1]Department of Computer Engineering, Faculty of Engineering, Yeditepe University, İstanbul, Turkey
[2]School of Computer Science, University of Nottingham, Jubilee Campus, Nottingham, United Kingdom

Received: 05.04.2019 • **Accepted/Published Online:** 31.10.2019 • **Final Version:** 28.03.2020

Abstract: In simultaneous multithreaded (SMT) processors, various data path resources are concurrently shared by many threads. A few heuristic approaches that explicitly distribute those resources among threads with the goal of improved overall performance have already been proposed. A selection hyperheuristic is a high-level search methodology that mixes a predetermined set of heuristics in an iterative framework to utilize their strengths for solving a given problem instance. In this study, we propose a set of selection hyperheuristics for selecting and executing the heuristic with the best performance at a given stage. To the best of our knowledge, this is one of the first studies implementing a hyperheuristic algorithm on hardware. The results of our experimental study show that hyperheuristics are indeed capable of improving the performance of the studied workloads. Our best performing hyperheuristic achieves better throughput than both baseline heuristics in 5 out of 12 workloads and gives about 15% peak performance gain. The average performance gains over the well-known hill-climbing and adaptive resource partitioning heuristics are about 5% and 2%, respectively.

Key words: Hyperheuristics, simultaneous multithreading, resource partitioning

1. Introduction

Simultaneous multithreaded (SMT) processors aim to improve system throughput by issuing instructions from multiple threads within a clock cycle [1]. The SMT architecture is simply a modified superscalar processor with shared issue queue (IQ), re-order buffer (ROB), physical register files (PRF), load/store queue (LSQ), arithmetic logic units (ALUs), and caches. In a typical superscalar processor, the data path resources are also shared by multiple threads, but each thread waits its time for the total control of all data path resources. At the end of each context switch, the running thread is suspended and the next scheduled thread starts executing its instructions. In such a scheme, between two context switches, only one thread can claim its monopoly on all resources. However, in SMT processors, multiple threads must simultaneously share the available resources. Fair sharing of the resources among threads while maximizing the processor throughput is a major challenge, and today, most of the research effort is focused on this issue.

There are many ways to improve the effectiveness of SMT processors. For instance, various SMT fetch policies attempt to load-balance the stream of instructions introduced at the processor pipeline [2, 3]. These fetch-oriented techniques are known as implicit methods for improving resource utilization since they implicitly manage the distribution of shared resources among working threads. Beside these implicit techniques, there

*Correspondence: iguney@cse.yeditepe.edu.tr

are explicit resource partitioning methods that partition shared resources according to runtime behavior of running threads. These are examples of explicit techniques applying heuristics over resource partitioning and distribution problems.

Heuristics are inexact, rule of thumb computational methods, tailored for a specific problem in hand. Dynamically controlled resource allocation (DCRA) [4], hill climbing [5], and the adaptive resource partitioning algorithm (ARPA) [6] are examples of heuristic approaches for partitioning SMT resources. There are many different heuristics for many different computationally *hard* problems in the literature. Considering a single problem domain, it has been frequently observed that different heuristics yield different performance figures across given instances. For each instance, a different heuristic might perform the best. Hyperheuristics have emerged as general high-level methods searching the space generated by a set of low-level heuristics compared to solutions that target a direct solution for a given problem [7]. Here, the main strategy is combining the strengths of various heuristics and avoiding their weaknesses for both solving seen and, most importantly, unseen problem instances. Hyperheuristics are already successfully applied for numerous static problems. However, there are only a few examples of their use in dynamically and continuously changing problems [8, 9].

Our preliminary studies gave us very promising results on performance improvements of SMT processors using hyperheuristics. In the literature, there are two major heuristics proposed for solving the resource partitioning problem in SMT processors. Here, we aim to improve the throughput of SMT processors by partitioning shared resources among threads using hyperheuristics in a multistage framework. Since both hill climbing and adaptive resource partitioning heuristics have periodic nature, we study mixing them using several hyperheuristic approaches in this study.

Our main motivation in this study is to combine the power of these two algorithms by integrating them into a hyperheuristic framework. In the end, we show that our proposed hyperheuristic algorithms successfully switch to the best performing heuristic when really needed.

The organization of the manuscript is as follows: Section 2 provides an overview of related work covering SMT resource management and hyperheuristics in general. Section 3 presents results from preliminary experiments, which provides the motivation of using hyperheuristics for SMT resource partitioning. Section 4 elaborates our proposed design. In Section 5, experimental results are presented. The last section, Section 6, concludes our study.

2. Related work

2.1. Heuristics for SMT resource management

One of the major design challenges in SMT processors is on the pipeline front end. Here, the fetch stage has to decide what to fetch next so that the resource utilization is improved and either throughput or fairness (or maybe both) criteria are satisfied. Tullsen et al. showed that fetching the right instructions has a great effect on performance and proposed a variety of algorithms for selecting instructions for the fetch stage [3]. BRCOUNT promotes threads with the fewest unresolved branches to mitigate the effects of mispredicted branch instructions; MISSCOUNT prioritizes threads with the fewest outstanding D-cache misses to reduce IQ-clogging; IQPOSN gives a lower priority to threads with instructions closest to the head of IQ; and ICOUNT favors efficient threads by prioritizing threads with the least number of instructions on the fly.

Tullsen and Brown discussed the effects of long-latency instructions, with a focus on long-latency load instructions, on SMT processors in [2]. Once a long-latency instruction is identified (by either an L2 cache miss or the instruction spending more than a predetermined number of cycles), either instructions from the thread

that owns the long-latency instruction are flushed or the thread is prevented from fetching more instructions for a certain amount of time.

Eyerman and Eeckhout pointed out that previous fetch policies do not take memory level parallelism (MLP) into account [10]. As a matter of fact, by stalling fetch or flushing instructions, these previous policies serialize the penalty of the long-latency load instructions. The study proposes an MLP-aware fetching mechanism, which aims to overlap the penalties of long-latency loads by executing them simultaneously, if possible. Once a long latency instruction is identified, the MLP-distance is predicted to determine how much further the processor should go to exploit MLP. Threads with such load instructions are either prevented from fetching more instructions than the MLP-distance or instructions beyond the MLP-distance are flushed.

Vandierendonck and Seznec proposed a framework called speculative instruction window weighting (SIWW) for applying different fetch policies and resource limitations on the SMT architecture [11]. In this framework, SIWW fetch policy gives priority to instructions from threads with the minimum amount of work remaining in the pipeline. The work identifies several types of instructions and each type is assigned a weight. The amount of work remaining for an individual thread is computed by adding predetermined weights of remaining instructions that belong to that thread in the pipeline. Resource limitations can be applied by defining an upper limit for the amount of work. A thread that exceeds this limit cannot fetch any more instructions until it commits and releases some instructions. Assigning different weights to different instruction types allows SIWW to apply different fetch policies without making any changes to the existing hardware. For example, assigning a weight of 1 to all instruction types results in a fetch policy equivalent to ICOUNT [3].

Besides these implicit techniques, there are explicit resource partitioning methods that partition shared resources according to the runtime behavior of each thread. The mechanism known as dynamically controlled resource allocation (DCRA) is one of these explicit methods [4]. DCRA dynamically tracks down the behavior of each thread and the use of data path resources with the help of various hardware counters. Here, a data path resource becomes inactive when it is not referenced for a predetermined timeout period. Meanwhile, a thread becomes a slow thread when it has a pending cache miss. Then DCRA allocates more resources to slow threads by taking some portion of resources from fast or inactive threads. The rationale behind this mechanism is as follows: a fast thread is already fast, and so there is no harm in stealing a few resource entries from it and giving them to the slow threads. Similarly, when a thread is labeled as inactive for a resource, then there is no harm in giving its share of that resource to a thread that actually needs it.

Another explicit mechanism, which proposes hill climbing for solving SMT resource partitioning problems, runs in epochs (periodic intervals) [5]. The hill climbing heuristic is a greedy algorithm that aims to gradually climb to a peak performance point by changing resource allocations at certain decision points. The execution is divided into trial epochs, and at each trial epoch, an arbitrary thread receives more resources than its actual portion. After running trial epochs for each thread, the decision point selects the best performing thread with extra resources and gives that extra resource to that selected thread. These trial epochs and decisions are run in a continuous manner in the SMT processor.

The adaptive resource partitioning algorithm (ARPA) utilizes a resource efficiency metric known as committed instructions per resource entry (CIPRE) [6]. At the end of each epoch, the CIPRE of each thread is calculated, and the thread with the highest CIPRE value receives proportionately more resources, whereas the thread with the lowest CIPRE value receives the least amount of resources. This mechanism has a self-balancing nature so that none of the threads can always dominate or starve. For instance, if a thread is provided with more resources, its new calculated CIPRE becomes lower as long as its commit rate does not change. As a result, a

thread with the worst CIPRE value can indirectly improve its efficiency later and receive extra resources in the end.

In another work from the literature, Eyerman and Eeckhout proposed a method that estimates the execution time of threads in SMT architecture if threads are run alone [12]. Weng and Liu provided higher fetching priority to threads with fewer utilized resources by examining early stages of the pipeline as well as low-level data cache misses [13]. Zhang and Lin limited the number of entries each thread can have in the issue queue according to the previous allocation's impact on performance [14]. Zhang and Lin improved SMT performance by partitioning shared register files among threads [15]. Güngörer and Küçük utilized a hill-climbing algorithm to dynamically partition the physical register files among threads in an SMT processor [16]. Finally, Sheikh and Lin also proposed a dynamic physical register file capping scheme that allows thread-based individual capping in SMT processors [17].

2.2. Hyperheuristics

Hyperheuristics are high-level methodologies that work on top of the heuristic search space for solving computationally difficult problems. The basic idea is to exploit the strength of multiple heuristics (move/neighborhood operators), which dates back to early 1960s [18]. Mainly, there are *selection* and *generation* hyperheuristics, which manage a set of low-level heuristics [19]. Currently, hyperheuristics are designed based on the notion of a *domain barrier*, which separates the problem domain from any high-level method. The barrier acts as a filter disallowing no problem-specific information from the problem domain to pass to the hyperheuristic level. This approach provides a basis for an automated, adaptive, modular, easy-to-maintain, and flexible software design that is enabled for reuse while solving an unseen instance from a domain and even other problem domains without necessitating any modification.

A selection hyperheuristic is often an iterative search method, consisting of *heuristic selection* and *move acceptance* methods that are invoked successively at each step [20]. This type of framework manages by mixing and controlling a fixed set of low-level heuristics. Cowling et al. introduced almost all of the simple selection hyperheuristic components [21]. For instance, the *random permutation gradient* selection heuristic first generates a list of permutations of low-level heuristics. Consequently, it selects a low-level heuristic in that predetermined order at any step to run on the current solution. Once a selected heuristic gives an improvement, it is utilized one more time.

Some of the hyperheuristics can also make use of various machine learning techniques. Learning within hyperheuristics takes place in an *online* or *offline* manner. Offline learning hyperheuristics are employed in a train-and-test order, where the feedback from the search process is obtained during the training stage on some sample problem instances. Online learning hyperheuristics receive feedback during the ongoing search process for guidance. One of the best examples of this category of hyperheuristics is known as the reinforcement learning-based hyperheuristic. In this approach, each heuristic is assigned a utility score that can either increase as a rewarding mechanism after an improving move or decrease as a punishment mechanism after a worsening move [22, 23]. The utility score is updated after each step of the algorithm, and the heuristic with the best score can be selected as the default strategy. Moreover, a hyperheuristic can embed a delayed learning mechanism, which, for example, scores low-level heuristics in a stage and then using those scores for choosing heuristics in the following stage. Bai et al. successfully applied a reinforcement-based delayed learning hyperheuristic on a timetabling problem as well as bin packing [24]. There is theoretical [25] as well as empirical evidence [26, 27] that hyperheuristics are effective solution methodologies for solving combinatorial optimization problems. Even

if the environment changes dynamically for a given problem, it has been shown that the hyperheuristics can adapt and result in high-quality solutions [8, 9].

More on hyperheuristics can be found in [7, 28, 29]. Sharing the SMT processor data path resources among the threads of a given workload is a challenging task that needs to be addressed in a dynamically changing environment. Moreover, even a small change in a workload, such as swapping the order of two programs, could lead to a large change in the overall characteristics of the instance, making the problem even more difficult to handle. Here, we use the previous work on selection hyperheuristics as an inspiration to design a set of learning selection hyperheuristics to mix well-known SMT heuristics to improve the SMT throughput.

3. Motivation

Although we can try utilizing as many of the aforementioned heuristics as possible in our hyperheuristic framework, there are two major obstacles ahead. First, a selection hyperheuristic asks for freedom to select and utilize any of the heuristics that are under its control at any given time. However, each heuristic has its own running strategy. For instance, all those in [5, 6, 16] have a periodic nature, whereas that of [4] is based on instant decision and repartitioning. Second, the heuristics that we focus on may not always have the same mindset. For instance, all those in [4–6] focus on resource partitioning, whereas those of [14–17] focus on resource capping, which might be a quite different task compared to partitioning from time to time. In this study, we focus on two well-known heuristics (i.e. hill climbing and adaptive resource partitioning) that fall into the same category (i.e. resource partitioning with a periodic nature) to demonstrate the viability and feasibility of hyperheuristics in the SMT domain.

We present the results obtained in our preliminary study, which revealed that neither of the heuristics examined (hill climbing and adaptive resource partitioning algorithm) is better than the other in every case.[1] Figure 1 shows the performance charts for some of the well-known benchmark workloads [30]. These results present both cases where HILL outperforms ARPA and ARPA outperforms HILL, indicating that there is no single heuristic that performs better than the others when all workloads are considered. The figure also presents the BEST value, which depicts the throughput of an oracle selection hyperheuristic that successfully selects the best performing heuristic in all cases. The duration of this preliminary analysis is only 8-heuristic periods long, but it is sufficient to show the potential of our approach for improving throughput as long as the correct sequence of heuristics is chosen.

As a case study, we examine how HILL, ARPA, and BEST perform when the three-threaded lbm-milc-gobmk workload is run. Figure 2a shows the throughput of three threads when the optimal permutation of heuristics (BEST) is used, and Figures 2b and c show the throughput results when ARPA and HILL are used, respectively. By examining the results for HILL, it can be seen that giving extra resources to gobmk during the first few epochs does not improve throughput. However, this is exactly what HILL does: it wastes resources on threads that are unable to improve throughput with additional resources. Another shortcoming of HILL is that it is slower than ARPA. In the first few epochs, the best allocation decision is to take as many resources from lbm and gobmk and give them to milc. ARPA can reach this state as fast as in six epochs, whereas it takes three times longer for HILL since there are three threads in this workload.

On the other hand, ARPA has its shortcomings, too. ARPA evaluates threads by their CIPRE value and takes resources away from threads with low CIPRE values and allocates these resources to the one with the

[1] In order for our model to work, the processor should be able to run hill climbing and ARPA heuristics at each stage interchangeably. To make both heuristics fully compatible, we made a few minor changes in our implementation of these heuristics. From now on, we refer to our implementations of hill climbing and the ARPA as HILL and ARPA, respectively.

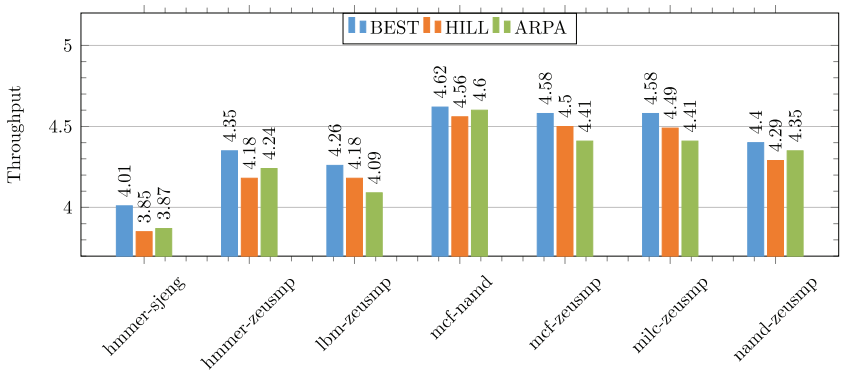


Figure 1. Throughput of HILL, ARPA, and maximum possible throughput achievable by a perfect selection hyper-heuristic, which we name BEST.

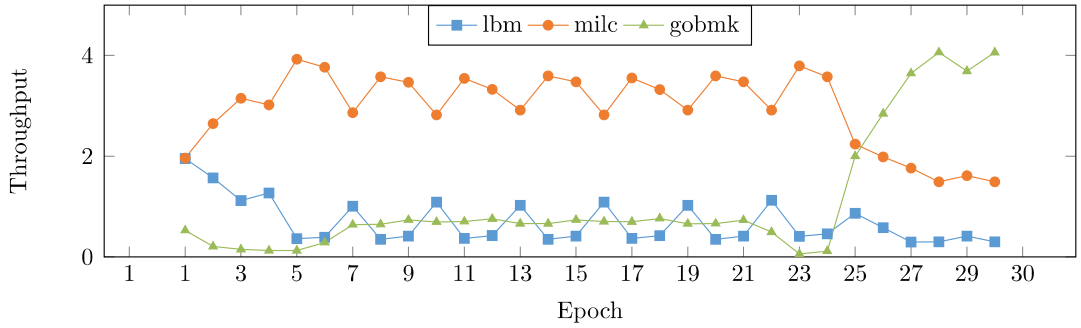
highest CIPRE. However, when ARPA takes resources away from a thread, the window in which that thread can search for data-independent instructions shrinks and the thread starts to lose its ability to exploit instruction level parallelism. The problem ARPA experiences in this workload is that it becomes harder for both lbm and gobmk to improve their CIPRE values as they lose their resources, creating a harmful feedback loop for these threads. It can be seen in Figure 2b that after epoch nine, ARPA is stuck with its allocation decision and cannot increase the number of resources allocated to gobmk, which can improve throughput with additional resources as can be seen in epochs 24 to 30 in Figure 2.

Figure 3 shows the throughput results of all possible heuristic permutations for the bzip2-cactusADM-hmmer workload for the first ten epochs. In the graph, the x-axis represents the first five heuristics selected in the first five epochs, and the y-axis represents the last five heuristics selected in the last five epochs. Here, H stands for HILL and A stands for ARPA. The colors represent the overall throughput in terms of instructions per cycle (IPCs), where red means more IPCs and green means fewer. Lower-left and upper-right corners show the actual throughputs of the original heuristics, ARPA and HILL, respectively. Simulation results show that less than 11% of permutations are able to improve throughput compared to ARPA in this workload, meaning that more than 89% of permutations hurt the performance. These results indicate that randomly selecting heuristics to utilize is more likely to degrade system performance, and smarter selection of algorithms is needed, instead.

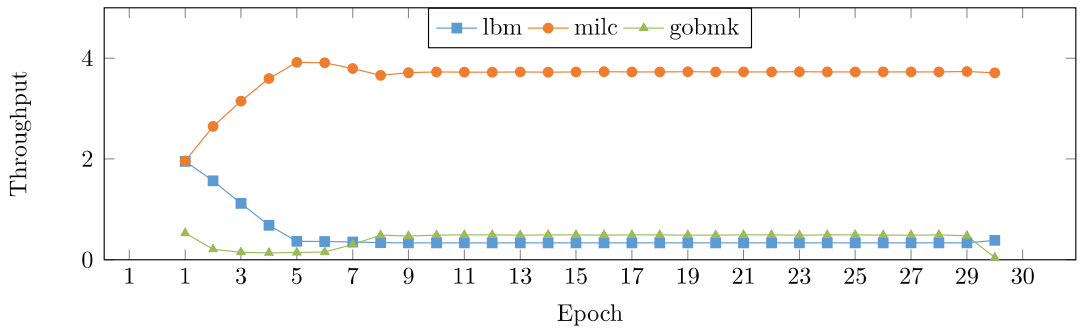
Based on the insights gained by our preliminary analysis and previous work, which concur that there is no single heuristic that will perform better than the rest in all cases, we propose using hyperheuristics to take advantage of multiple heuristics. Furthermore, Figure 1 suggests that the performance of hyperheuristics can even surpass the individual performance of each utilized heuristic.

4. The proposed hyperheuristic design

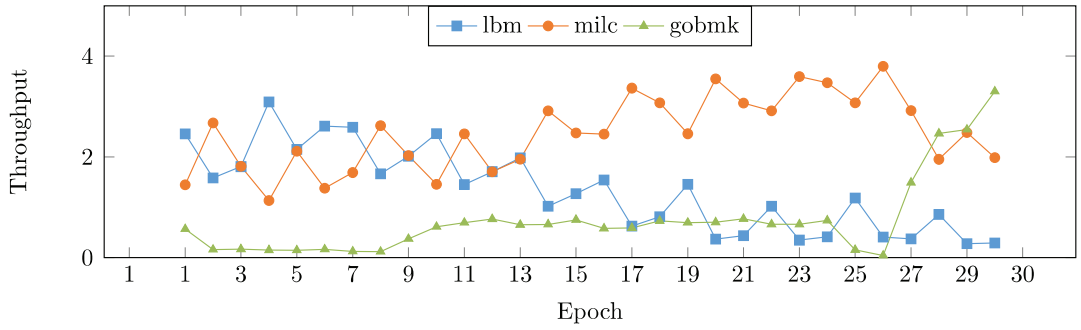
We start by creating a mechanism that can interchangeably run either adaptive resource partitioning or hill climbing algorithms. ARPA and HILL heuristics are our faithfully implemented versions of these heuristics on hardware. The relationship between the hyperheuristic and the underlying heuristics is shown in Figure 4. The aim of ARPA and HILL heuristics is to find the thread that deserves extra resources the most. The hardware counters are shared by all heuristics and the hyperheuristic that utilizes those heuristics. These per epoch counters are IPC, CIPRE, and fetched instructions per cycle (FIPC). When the performance of a low-level heuristic deteriorates and the indicator value reduces below a certain threshold, our hyperheuristic selects the



(a) Throughput of individual threads when the best performing combination of HILL and ARPA is used for the workload lbm-milc-gobmk



(b) Throughput of individual threads when ARPA is used as the heuristic for the workload lbm-milc-gobmk



(c) Throughput of individual threads when HILL is used as the heuristic for the workload lbm-milc-gobmk

Figure 2. Throughputs of individual threads under various heuristics for the workload lbm-milc-gobmk.

next one from the other available low-level heuristics. Therefore, we can say that our proposed hyperheuristic presents similar characteristics of a permutation gradient hyperheuristic. In this study, we investigate various runtime statistics as well as heuristic selection methods.

4.1. Mixing ARPA and HILL

The proposed hyperheuristic should be able to run both heuristics interchangeably and adaptively in epochs. This is a challenging task, as additionally, this needs to be done in real time based on hardware entries. There is a slight difference between HILL and ARPA: HILL needs a number of trial epochs to decide, whereas ARPA can make permanent decisions at the end of a single epoch. If the system allows ARPA to run between two trial epochs of HILL, this will have two severe consequences. First, it will increase the chances of a workload

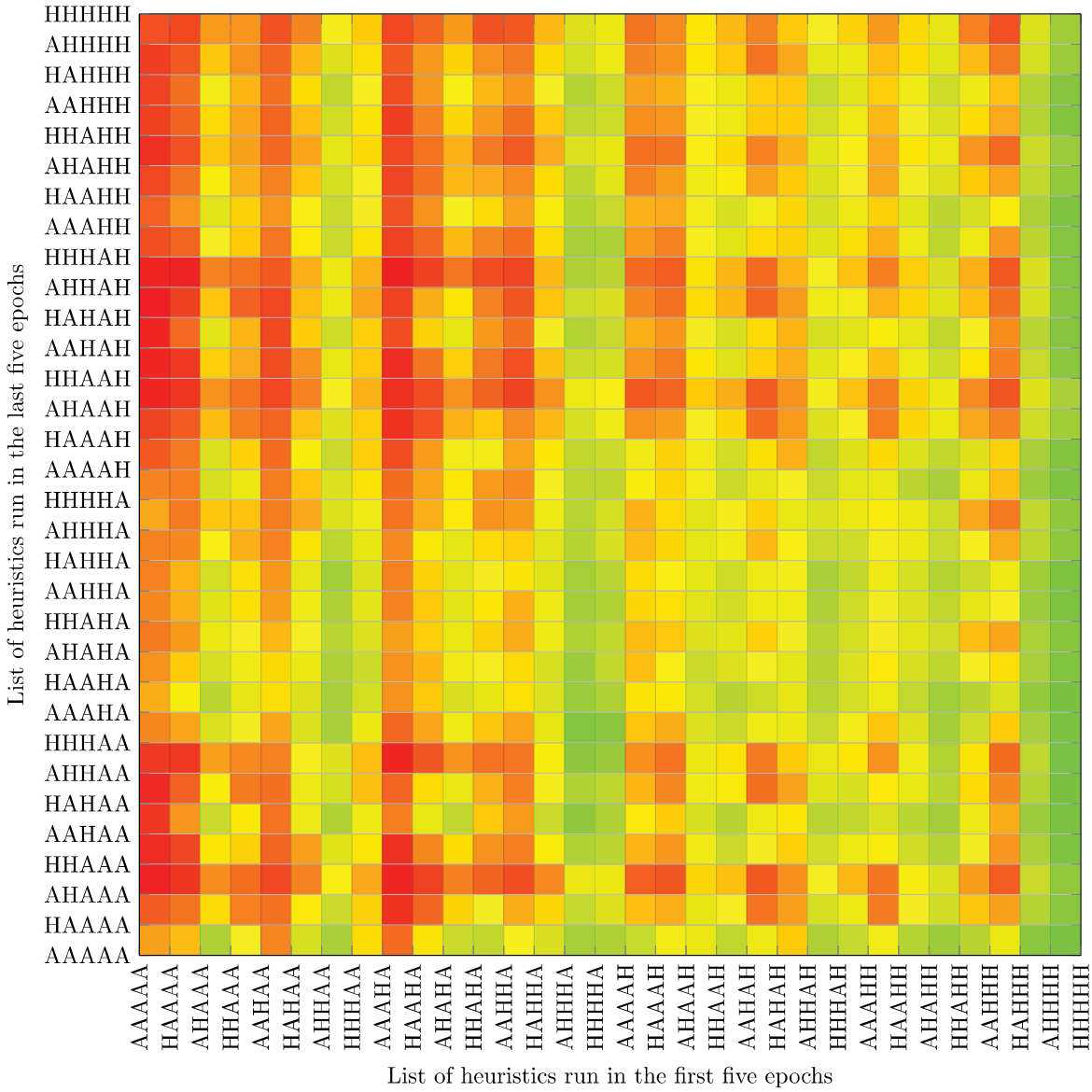


Figure 3. Throughput results of all possible heuristic permutations in bzip2-cactusADM-hmmer workload.

changing its behavior between the two trial periods, which leads HILL to compare trial performances of two different program phases and renders it to be a totally different heuristic. This can be observed better in the time line given in Figure 5. In this example, HILL is run in the first epoch, and HILL runs its first trial round. Then the hyperheuristic decides that ARPA should be run for the next seven epochs. When HILL finishes its trials and makes a decision, it has to compare performance results of epochs 0 and 8, which are quite far away from each other, causing inaccurate evaluations that the original algorithm does not experience at all.

The second problem that may occur when ARPA is allowed to be run between two trial epochs of HILL is that the processor may have to make radical changes in resource distribution if ARPA keeps changing the distribution in a particular direction and HILL wants to return to its anchor state. This phenomenon would cause the processor to act in a way against the nature of both heuristics.

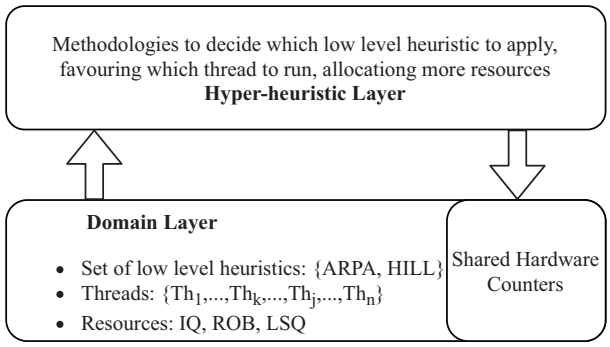


Figure 4. The proposed design.

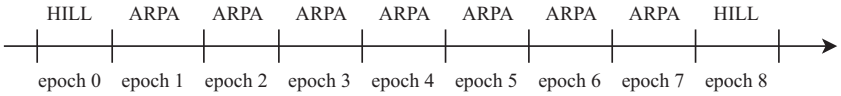


Figure 5. An example time line with heuristics running in arbitrary order.

Throughout this process, we tried our best to faithfully implement the algorithmic behavior of each heuristic to be consistent with its original implementation. To overcome the problems described above, we define *big epochs*. Big epochs consist of T epochs, where T is the number of threads running simultaneously in the system. Only a single type of heuristic runs within a big epoch, as shown in Figure 6. Therefore, the hyperheuristic makes decisions only at the beginning of big epochs. To provide the hyperheuristic with accurate data on how heuristics perform, all evaluations are done using performance values of heuristics gathered in big epochs, although the heuristics still make their decisions in the traditional epoch granularity.

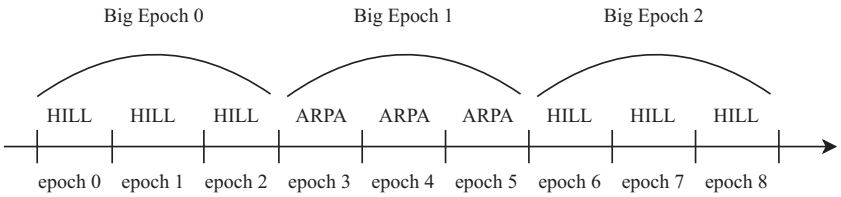


Figure 6. An example time line where the hyperheuristic decides HILL, ARPA, and HILL should be run on the SMT processor with 3 threads.

As stated earlier, the nature of utilized heuristics must have a certain similarity so the system can accommodate all heuristics at the same time without major difficulties. For example, DCRA is another promising heuristic with different qualities compared to HILL and ARPA, but it is different from the others due to its nonperiodic nature [4]. Thus, DCRA and such other heuristics are left out in this research. However, it is certainly a prominent future research direction to include heuristics of different natures into the mix.

4.2. Proposed hyperheuristics

In this section, we focus on hyperheuristics for selecting a heuristic that allocates SMT resources among concurrently running threads. For the selection process, simpler rules usually produce more effective results as more complex hyperheuristics are more likely to introduce many corner cases where the hyperheuristic fails to

choose the right heuristic for the job. Although we tested dozens of hyperheuristics with different metrics and for different reasons, we only present three hyperheuristics that sufficiently represent the effect of hyperheuristic usage on SMT resource partitioning.

4.2.1. HH1: A throughput-oriented hyperheuristic

Our first proposed hyperheuristic is the most straightforward one as it directly uses the end result as a decision parameter. At the end of a decision period, HH1 toggles the heuristic that is being used if the current heuristic causes a throughput drop compared to the previous decision period. By introducing a threshold rate, HH1 can be more forgiving to heuristics, which cause a small throughput drop, or more greedy, by punishing heuristics that fail to improve throughput above the threshold. The algorithm for HH1 is shown in Algorithm 1.

Algorithm 1 Algorithm for HH1.

IPC_i : *Throughput of ith Big Epoch*
procedure HH1(bigepoch: current big epoch number)
    $target\_ipc \leftarrow IPC_{bigepoch\ -\ 1} * threshold$
    if $IPC_{bigepoch} >= target\_ipc$ **then**
        Keep heuristic
    else
        Switch heuristic

4.2.2. HH2: An efficiency-oriented hyperheuristic

Our second hyperheuristic, HH2, evaluates heuristics based on their ability to help threads commit most of their instructions that they fetch. Speculative instructions from mispredicted paths do not make any contribution to the throughput metric. These instructions are fetched into the pipeline but are not committed. Hence, such instructions cause shared pipeline resources to be wasted. HH2 aims to keep the ratio of the number of instructions committed to the number of instructions fetched as small as possible by switching to the next heuristic if the current heuristic fails to improve this ratio above a certain threshold. The algorithm for HH2 is shown in Algorithm 2.

Algorithm 2 Algorithm for HH2.

$commit_i$: *Number of instructions committed in ith Big Epoch*
$fetch_i$: *Number of instructions fetched in ith Big Epoch*
procedure HH2(bigepoch: current big epoch number)
    $FIPC_{bigepoch\ -\ 1} \leftarrow commit_{bigepoch\ -\ 1}\ /\ fetch_{bigepoch\ -\ 1}$
    $FIPC_{bigepoch} \leftarrow commit_{bigepoch}\ /\ fetch_{bigepoch}$
    $target\_fipc \leftarrow FIPC_{bigepoch\ -\ 1} * threshold$
    if $FIPC_{bigepoch} >= target\_fipc$ **then**
        Keep heuristic
    else
        Switch heuristic

4.2.3. HH3: A hybrid solution

Finally, HH3 borrows metrics from both HH1 and HH2; thus, it is a mixture of these two heuristics. A heuristic may help improve the number of instructions committed but degrade the ratio of the number of instructions committed to the number of instructions fetched, or vice versa. HH3, therefore, constitutes a more aggressive

approach by switching the heuristic if it causes performance degradation on either of these metrics. Similar to HH1 and HH2, HH3 utilizes a threshold ratio to adjust the level of tolerance (or intolerance) against performance drops. The algorithm for HH3 is shown in Algorithm 3.

Algorithm 3 Algorithm for HH3.

$commit_i$: *Number of instructions committed in ith Big Epoch*
$fetch_i$: *Number of instructions fetched in ith Big Epoch*
procedure HH3(bigepoch: current big epoch number)
    $FIPC_{bigepoch\ -\ 1} \leftarrow commit_{bigepoch\ -\ 1} / fetch_{bigepoch\ -\ 1}$
    $FIPC_{bigepoch} \leftarrow commit_{bigepoch}\ /\ fetch_{bigepoch}$
    $target\_fipc \leftarrow FIPC_{bigepoch\ -\ 1}\ *\ threshold$
    $target\_commit \leftarrow commit_{bigepoch\ -\ 1}\ *\ threshold$
    if $FIPC_{bigepoch} >= target\_fipc$ **and** $commit_{bigepoch} >= target\_commit$ **then**
        Keep heuristic
    else
        Switch heuristic

5. Computational experiments

5.1. Experimental settings

In our study, we used M-Sim as our simulation environment.[2] The configuration of the simulated processor is shown in Table 1. hmmer (Hidden Markov Models for protein sequence analysis), lbm (Lattice Boltzmann method for simulating incompressible fluids in 3D), mcf (single-depot vehicle scheduling in public mass transportation), milc (Multiple Instruction Multiple Data Lattice Computation for quantum chromodynamics), namd (a parallel program for simulating large biomolecular systems), sjeng (a chess application for exploring the tree of variations resulting from a given position), and zeusmp (a program for solving the equations of nonresistive, nonrelativistic hydrodynamics and magnetohydrodynamics) benchmarks are randomly chosen from the SPEC2006 CPU suite to create multithreaded workloads. Twelve workloads are created using these benchmarks. For a fair comparison, these workloads are organized as two 6-workload groups: WLHILL, which consists of workloads for which HILL performs better than ARPA, and WLARPA, which consists of workloads for which ARPA performs better than HILL. These workloads are shown in Table 2. For all workloads, the system is fast-forwarded for 100M cycles as a warm-up period followed by a cycle-accurate simulation for 200M cycles.

Epoch duration is chosen as 32K cycles. This is the epoch duration used in the previous work [5, 6]. We also empirically determined that both HILL and ARPA work best with this duration. Since a big epoch consists of epoch times number of threads, the big epoch duration becomes 128K cycles. As both heuristics move ROB, IQ, and LSQ entries between threads, the number of entries moved for these resources at each step must be proportional to their sizes. We determined these values as 4, 2, and 2 entries, respectively. The threshold values for all hyperheuristics are empirically set as 1.0.

5.2. Results

In this section, we present the experimental results obtained from our simulation environment. The throughput of the proposed hyperheuristics is examined in two different workload sets: 1) workloads for which HILL achieves higher throughput compared to ARPA (WLHILL), and 2) workloads for which ARPA gives higher throughput

[2]M-SIM (2005): A Flexible, Multithreaded Architectural Simulation Environment [online]. Website https://www.cs.binghamton.edu/ msim/documentation/msim_tr.pdf [accessed 11 November 2019].

Table 1. Specifications of the simulated system.

Number of concurrent threads	4
Decode / Issue / Commit bandwidth	8
Number of memory ports	8
Register file	256 integer, 256 floating point
Re-order buffer size	64 entries
Issue queue size	32 entries
Load/Store queue size	40 entries
Number of integer ALUs	6
Number of integer multipliers	3
Number of floating point ALUs	3
Number of floating point multipliers	1
L1 Instruction cache size	32 KB, 2-way, LRU
L1 Data cache size	32 KB, 4-way, LRU
L1 Cache hit time	1 cycle
L2 Cache size	512 KB, 4-way, LRU
L2 Cache hit time	20 cycles
Main memory access time	300 cycles

Table 2. Workloads.

Workload no.	WLHILL	WLARPA
1	hmmer-mcf-milc-namd	hmmer-lbm-mcf-sjeng
2	hmmer-mcf-milc-zeusmp	hmmer-mcf-milc-sjeng
3	lbm-mcf-milc-zuesmp	hmmer-mcf-namd-sjeng
4	mcf-milc-namd-zeusmp	lbm-mcf-milc-sjeng
5	mcf-milc-sjeng-zeusmp	lbm-mcf-namd-sjeng
6	mcf-namd-sjeng-zeusmp	lbm-milc-namd-sjeng

compared to HILL (WLARPA). Figures 7 and 8 show the throughput obtained by utilizing HILL, ARPA, and our proposed hyperheuristics on WLARPA and WLHILL workload sets, respectively.

It was shown in the previous work [5, 6] that both heuristics have their advantages and disadvantages and may outperform each other in different workloads. This can also be seen in the results shown in Figures 7 and 8. In WLHILL, HILL outperforms ARPA by more than 3%, and, in WLARPA, ARPA outperforms HILL by about 10% on average. Here, our main motivation is to be able to select and utilize the better performing heuristic in a timely manner, so that we do not lose too much performance. From Figure 1 we also know that we can even achieve a higher performance than both ARPA and HILL if we make this selection right and timely.

The results show that our proposed hyperheuristics can achieve throughput results quite close to the results of the better performing heuristic, and, in almost all of the WLARPA workloads, they can even perform slightly better than both heuristics. However, we also see that devising a perfect hyperheuristic, which always gives higher performance than the heuristics that it operates on, is a challenging (if not impossible) task. As a result, our best performing hyperheuristic (HH3) outperforms HILL by about 5% and ARPA by about 2% on

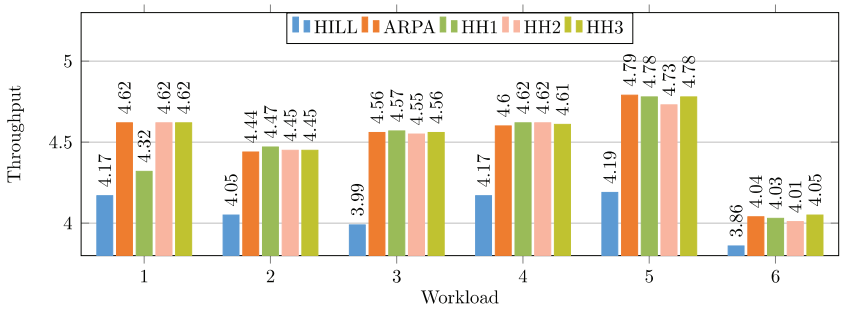


Figure 7. Throughput of HILL, ARPA, and hyperheuristics for workload set WLARPA.

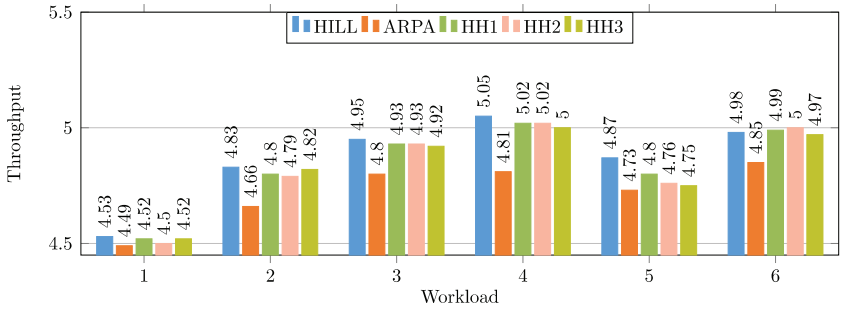


Figure 8. Throughput of HILL, ARPA, and hyperheuristics for workload set WLHILL.

average. The peak performance gain over HILL is about 15% in the third workload of WLARPA, whereas the peak performance gain over ARPA is slightly lower (5%) in the fourth workload of WLHILL.

We were also expecting that our hyperheuristics perform better than HILL and ARPA in occasional cases. In the second, the third, the fourth, and the sixth workloads of WLARPA, we clearly observe this phenomenon. Although the performance gain over both heuristics is small, these results are important to demonstrate that hyperheuristics are tools for not only tracking better performing heuristics but also achieving optimal or close-to-optimal results even better than the best heuristic that is under the hood.

In terms of hardware complexity, all hyperheuristics induce insignificant complexity for the system. HH3, the most complex hyperheuristic among all proposed hyperheuristics, requires only one register to store the FIPC value for the previous epoch. The computation required by HH3 consists of one division, two multiplication, and two comparison operations, which can be carried out by the processor's already existing functional units in less than one hundred cycles. Considering that the control logic for both heuristics themselves introduce little complexity, it can be said that the proposed hyperheuristics can be applied to SMT processors without any significant costs in terms of hardware complexity.

6. Conclusion

It is shown in the literature that resource partitioning in SMT processors has an important impact on throughput. In this research, we show that there is no single solution that works best in all situations, and we investigate utilizing hyperheuristics to exploit the advantages of two well-known SMT resource partitioning algorithms.

Among various hyperheuristics that use different heuristic selection logic and feedback metrics evaluated in this research, the three most prominent hyperheuristics are presented. The best performing hyperheuristic

improves performance by 5% compared to hill climbing and by 2% compared to the ARPA on average across all workloads that are studied. The peak performance gain reaches up to 15%.

We studied the usage of hardware hyperheuristics, and this is one of the first studies that implements a selection hyperheuristic on such a restricted environment with limited resources. Implementing the hyperheuristic on hardware requires the decision logic to be fairly simple due to increasing hardware complexity and power considerations. A future research direction would be to implement hyperheuristics as kernel modules, allowing more complex hyperheuristics to be utilized.

Apart from using different selection algorithms and feedback metrics, one way to improve the potential and effective throughput gain would be to introduce new low-level heuristics into the system. Such extra heuristics can enable hyperheuristics to cover a larger search space with richer and better performance options.

Acknowledgment

This work was supported by the Scientific and Technological Research Council of Turkey under Grant No. 117E866.

References

[1] Tullsen DM, Eggers SJ, Levy HM. Simultaneous multithreading: maximizing on-chip parallelism. In: 25 Years of the International Symposia on Computer Architecture (Selected Papers); Barcelona, Spain; 1998. pp. 533-544.

[2] Tullsen DM, Brown JA. Handling long-latency loads in a simultaneous multithreading processor. In: Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture; Austin, TX, USA; 2001. pp. 318-327.

[3] Tullsen DM, Eggers SJ, Emer JS, Levy HM, Lo JL et al. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. ACM SIGARCH Computer Architecture News 1996; 24 (2): 191-202. doi: 10.1145/232974.232993

[4] Cazorla FJ, Ramirez A, Valero M, Fernandez E. Dynamically controlled resource allocation in SMT processors. In: Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture; Portland, OR, USA; 2004. pp. 171-182.

[5] Choi S, Yeung D. Hill-climbing SMT processor resource distribution. ACM Transactions on Computer Systems 2009; 27 (1): 1-47. doi: 10.1145/1482619.1482620

[6] Wang H, Koren I, Krishna CM. Utilization-based resource partitioning for power-performance efficiency in SMT processors. IEEE Transactions on Parallel and Distributed Systems 2011; 22 (7): 1150-1163. doi: 10.1109/TPDS.2010.199

[7] Burke EK, Gendreau M, Hyde M, Kendall G, Ochoa G et al. Hyper-heuristics: A survey of the state of the art. Journal of the Operational Research Society 2013; 64 (12): 1695-1724. doi: 10.1057/jors.2013.71

[8] Kiraz B, Etaner-Uyar AS, Özcan E. Selection hyper-heuristics in dynamic environments. Journal of the Operational Research Society 2013; 64 (12): 1753-1769. doi: 10.1057/jors.2013.24

[9] Uludağ G, Kiraz B, Etaner-Uyar AS, Özcan E. A hybrid multi-population framework for dynamic environments combining online and offline learning. Soft Computing 2013; 17 (12): 2327-2348. doi: 10.1007/s00500-013-1094-7

[10] Eyerman S, Eeckhout L. Memory-level parallelism aware fetch policies for simultaneous multithreading processors. ACM Transactions on Architecture and Code Optimization 2009; 6 (1): 1-33. doi: 10.1145/1509864.1509867

[11] Vandierendonck H, Seznec A. Managing SMT resource usage through speculative instruction window weighting. ACM Transactions on Architecture and Code Optimization 2011; 8 (3): 1-20. doi: 10.1145/2019608.2019611

[12] Eyerman S, Eeckhout L. Per-thread cycle accounting in SMT processors. ACM SIGPLAN Notices 2009; 44 (3): 133-144. doi: 10.1145/1508284.1508260

[13] Weng L, Liu C. A resource utilization based instruction fetch policy for SMT processors. Microprocessors and Microsystems 2015; 39 (1): 1-10. doi: 10.1016/j.micpro.2014.10.001

[14] Zhang Y, Hays M, Lin WM, John E. Autonomous control of issue queue utilization for simultaneous multi-threading processors. In: Proceedings of the High Performance Computing Symposium; Tampa, FL, USA; 2014. pp. 1-8.

[15] Zhang Y, Lin WM. Efficient resource sharing algorithm for physical register file in simultaneous multi-threading processors. Microprocessors and Microsystems 2016; 45 (PB): 270-282. doi: 10.1016/j.micpro.2016.06.002

[16] Güngörer H, Küçük G. Dynamic capping of physical register files in simultaneous multi-threading processors for performance. In: 32nd International Symposium (ISCIS 2018); Poznan, Poland; 2018. pp. 41-48.

[17] Sheikh MN, Lin WM. Dynamic capping of rename registers for SMT processors. Journal of Systems Architecture 2019; 99: 1-20. doi: 10.1016/j.sysarc.2019.101637

[18] Fisher H, Thompson GL. Probabilistic learning combinations of local job-shop scheduling rules. In: Muth JF, Thompson GL (editors). Industrial Scheduling. Hoboken, NJ, USA: Prentice Hall, 1963, pp. 225-251.

[19] Burke EK, Hyde M, Kendall G, Ochoa G, Özcan E et al. A classification of hyper-heuristics approaches. In: Gendreau M, Potvin JY (editors). Handbook of Metaheuristics. New York, NY, USA: Springer, 2010, pp. 449-468.

[20] Özcan E, Bilgin B, Korkmaz E. A comprehensive analysis of hyper-heuristics. Intelligent Data Analysis 2008; 12 (1): 3-23. doi: 10.3233/ida-2008-12102

[21] Cowling P, Kendall G, Soubeiga E. A hyperheuristic approach to scheduling a sales summit. In: Burke E, Erben W (editors). Practice and Theory of Automated Timetabling III. Berlin, Germany: Springer, 2001, pp. 176-190.

[22] Nareyek A. Choosing search heuristics by non-stationary reinforcement learning. In: Resende MGC, de Sousa JP (editors). Metaheuristics: Computer Decision-Making. New York, NY, USA: Springer, 2003, pp. 523-544.

[23] Özcan E, Mısır M, Ochoa G, Burke EK. A reinforcement learning - great-deluge hyper-heuristic for examination timetabling. International Journal of Applied Metaheuristic Computing 2010; 1 (1): 39-59. doi: 10.4018/jamc.2010102603

[24] Bai R, Blazewicz J, Burke EK, Kendall G, McCollum B. A simulated annealing hyper-heuristic methodology for flexible decision support. 4OR 2012; 10 (1): 43-66. doi: 10.1007/s10288-011-0182-8

[25] Lehre PK, Özcan E. A runtime analysis of simple hyper-heuristics: to mix or not to mix operators. In: Proceedings of the Twelfth Workshop on Foundations of Genetic Algorithms XII; Adelaide, Australia; 2013. pp. 97-104.

[26] Kheiri A, Özcan E. An iterated multi-stage selection hyper-heuristic. European Journal of Operational Research 2016; 250 (1): 77-90. doi: 10.1016/j.ejor.2015.09.003

[27] Shahriar A, Karapetyan D, Kheiri A, Özcan E, Parkes AJ. Combining Monte-Carlo and hyper-heuristic methods for the multi-mode resource-constrained multi-project scheduling problem. Information Sciences 2016; 373: 476-498. doi: 10.1016/j.ins.2016.09.010

[28] Cowling P, Kendall G, Soubeiga E. Hyperheuristics: a tool for rapid prototyping in scheduling and optimisation. In: Applications of Evolutionary Computing: Proceedings of Evo Workshops 2002; Kinsale, Ireland; 2002. pp. 269-287.

[29] Ross P. Hyper-heuristic. In: Burke EK, Kendall G (editors). Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques. New York, NY, USA: Springer, 2005, pp. 529-556.

[30] Prakash TK, Peng L. Performance characterization of SPEC CPU2006 benchmarks on Intel Core 2 Duo processor. ISAST Transactions on Computers and Software Engineering 2008; 2 (2): 36-41. doi: 10.1109/TPDS.2010.199