Hyper-heuristics for Performance Optimization of Simultaneous Multithreaded Processors

İsa Ahmet Güney[1], Gürhan Küçük[1], Ender Özcan[2]

[1]Department of Computer Engineering, Yeditepe University, Istanbul, Turkey
{iguney,gkucuk}@cse.yeditepe.edu.tr
[2]School of Computer Science, University of Nottingham, Jubilee Campus, Nottingham, UK
ender.ozcan@nottingham.ac.uk

Abstract. In Simultaneous Multi-Threaded (SMT) processor datapaths, there are many datapath resources that are shared by multiple threads. Currently, there are a few heuristics that distribute these resources among threads for better performance. A selection hyper-heuristic is a search method which mixes a fixed set of heuristics to exploit their strengths while solving a given problem. In this study, we propose learning selection hyper-heuristics for predicting, choosing and running the best performing heuristic. Our initial test results show that hyper-heuristics may improve the performance of the studied workloads by around 2%, on the average. The peak performance improvement is observed to be 41% over the best performing heuristic, and more than 15% over all heuristics that are studied. Our best hyper-heuristic performs better than the state-of-the art heuristics on almost 60% of the simulated workloads.

1 Introduction

Today, the Simultaneous Multi-Threaded (SMT) processors aim to increase the system throughput by executing instructions from different threads in a single clock cycle. These processors are widely utilized in both high-end (e.g. Intel core i7) and low-end (e.g. Intel Atom) computers, and they try to satisfy the high system throughput requirements in high-end machines and the efficient and effective utilization of system resources in low-end machines, together. SMT processors are also utilized in highly popular contemporary chip multi processors (e.g. Intel Xeon servers).

In SMT processors, there are many datapath resources (Issue Queue, Re-Order Buffer, Load/Store Queue, Physical Register Files, Arithmetic Logic Units and cache structures) that are shared by multiple running threads. In an uncontrolled environment, threads assume that all the shared datapath resources are solely dedicated to themselves, and, inadvertently, they go into a race for stealing datapath resources from each other. In such case, a single thread may take over the Issue Queue, pollute the caches and fill the Physical Register Files by the instructions that are introduced into the processor from a mispredicted path even though the branch misprediction rate

is known to be high. As a result, today, the throughput obtained from SMT processors are much lower than the potential throughput that can be actually obtained.

There are various strategies to improve the efficiency of these processors. First, there are fetch policies that try to regulate the stream of instructions that are introduced to the pipeline. These techniques change the distribution of shared resources, indirectly. Most famous examples of these are ICOUNT, which gives fetch priority to threads with less resource occupancy, BCOUNT, which favors threads with the fewest unresolved branches, MISSCOUNT, which gives priority to threads with fewest outstanding D-cache misses, STALL, which triggers fetch-lock when a load operation stays to be outstanding beyond some threshold number of cycles and FLUSH, which measures resource clog and when it happens recovers by flushing the stalled instructions [1][2].

Beside these fetch throttling techniques, there are resource partitioning techniques that directly distribute shared resource partitions among running threads. Basically, these techniques dynamically decide how a shared resource is to be partitioned and distributed. The most famous example of these techniques is known as Dynamically Controlled Resource Allocation (DCRA) [3]. In DCRA, each thread and the datapath resource are dynamically tracked by a number of hardware counters. For example, when a thread has a pending cache miss, it is immediately labeled as a slow thread, or when a resource is not used by a thread for a threshold number of cycles, then the thread for that resource becomes inactive. Then, the DCRA mechanism tries to give more resources to the slow threads by stealing from fast or inactive threads.

SMT resource distribution via hill climbing (HILL) is another resource partitioning mechanism that runs in epochs (periodic intervals) [4]. HILL assumes that there is a certain optimum in the performance graph and it tries to reach to that peak by dynamically changing resource distributions in a greedy fashion. In the initial trial epochs, each thread gets its chance to show its performance with extra resources. At the end of these trial epochs, the performance of each thread is compared and the best performing (and the most deserving) thread is selected for receiving additional resources. Then, these trial epochs and the consequent resource distribution are done inside an infinite loop as long as the processor is running.

The Adaptive Resource Partitioning Algorithm (ARPA) introduces efficiency metric into the picture [5]. Similar to HILL, ARPA tries to give more resources to the most deserving thread by stealing resources from the others. The efficiency metric, committed instructions per resource entry (CIPRE), is a thread specific metric which is evaluated at the end of each epoch. When a thread does a great job and commits many instructions with limited number of resources, its CIPRE value becomes high, and ARPA gives more resources to that thread. In HILL, a thread can show the best performance and be chosen to receive more resources every epoch regardless of its efficiency. As a result, a thread may starve to its death, since it cannot perform better than some other thread. ARPA solves this issue implicitly by its efficiency metric. When a thread receives more resources its CIPRE value gets lower and lower if it commits similar amount of instructions every epoch. In such cases, a thread with worse performance may get its share, since its efficiency may go up after a while.

Vandierendonck and Seznec [6] propose a new fetch throttling mechanism called Speculative Instruction Window Weighting. This mechanism fetches instructions from the thread with least amount of work left in the pipeline. The amount of work left for each thread is predicted by assigning weights to instructions. These weights are determined by the instruction type, confidence level of branch instructions and confidence level, prediction result and memory-level parallelism for memory instructions. By limiting the maximum amount of work of a thread, distribution of datapath resources is also achieved.

Another fetch policy by Eyerman and Eeckhout [7] takes memory-level parallelism into consideration. Their design predicts long-latency loads and the number of instructions a thread must go down in the instruction stream in order to exploit memory-level parallelism. The algorithm stalls fetching if the thread has reached the number of instructions predicted by the MLP-predictor, or flushes instructions beyond the predicted number of instructions in case a long-latency load is identified.

Heuristics (meta-heuristics) are problem specific inexact, rule of thumb computational methods. DCRA, HILL and ARPA are examples of such heuristics. In literature, different heuristics with different performances for almost all “hard” problems could be found. It has been observed that each heuristic may be successful in solving different problem instances. Hyper-heuristics are general methods which search the space generated by a set of heuristics rather than solutions to directly solve a given problem. A goal is designing intelligent and automated approaches enabled to combine the strengths of heuristics while avoiding their weaknesses for solving not only the instances in hand, but also the unseen ones. Hyper-heuristics have been applied to many static problems, whereas there are a few studies on their applications to dynamic environment problems. In these studies, either a theoretical problem or a benchmark function is used, where the changes in the environment can be controlled.

As a result, the heuristics, described up to this point, try to achieve a better performance compared to the one that we can observe in a baseline configuration, in which resource sharing is not regulated at all. The motivation behind this study is very simple. We show that there is no single algorithm that performs best in all the SMT workloads. In some workloads, DCRA works best, in some other workloads HILL works better than others, and in the rest of the workloads ARPA performs best. Our literature survey, preliminary studies and the variety of problem instances and the observed dynamic changes show us that hyper-heuristics are a very suitable choice for the performance optimization on SMT processors. Although, in the literature, there are only a few heuristics proposed for solving this problem, there is no study showing how general these heuristics are or providing a thorough performance analyses for the proposed heuristics. More importantly, there is no real world application of hyper-heuristics to such a dynamic environment problem in hand.

In this study, we aim to optimize the performance of SMT processors by partitioning datapath resources among running threads by using hyper-heuristics. Since, HILL and ARPA heuristics have similar periodic nature; we studied combining both under several hyper-heuristics throughout this study.

Figure 1 shows the performance (Instruction Per Cycle) graphs for some of the workloads that we studied. In art-gcc-mgrid and art-parser-vortex workloads ARPA

performs better than HILL (12% and 8%, respectively). However, there are some other workloads in which ARPA performs worse than HILL. For instance, in gcc-mesa-vortex and art-mesa-vortex workloads HILL performs more than 5% better. Note that while ARPA is successfully running art-parser-vortex workload, when parser benchmark is replaced with mesa benchmark everything changes upside down and HILL starts to become more successful. This graph shows us that some heuristics can be successful in some of the workloads and some others can be successful in some other workloads. To the best of our knowledge, there is no such study that work on hyper-heuristics to dynamically select the proper heuristic at run time, and, in this study, we are aiming to fill this gap.

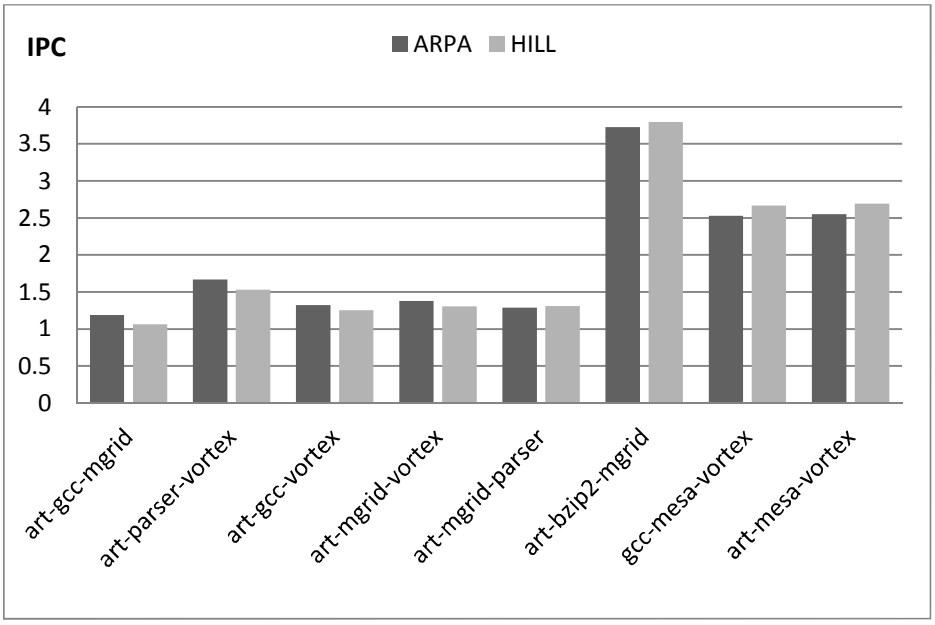

Fig.1. ARPA and HILL performance comparison on a few SMT workloads.

2 Proposed Design

The initial design starts by creating a habitat that may run both ARPA and HILL, interchangeably. Both ARPA and HILL track down runtime statistics collected by a number of hardware counters. For instance, both of them require the number of committed instructions for each thread in time periodic intervals called *epochs*. They also need a comparator circuitry to decide if the performance of a trial epoch is greater than the performance value experienced by the other trial epochs or the CIPRE value of a thread is greater than the others. The resulting circuitry that runs both heuristics is less complex than what one may expect.

As shown in Figure 2, our proposed design brings ARPA and HILL SMT partitioning heuristics, together. The job of these heuristics is to favor one of the running

threads and to award it with more resources. The shared hardware counters keep runtime statistics that are required by the heuristics' (and the hyper-heuristic's) evaluation functions. A few example counters are *committed instructions per cycle per epoch* ($IPCepoch_i$, for the i^{th} epoch), CIPRE and *fetched instructions per cycle per epoch* ($FIPCepoch_i$). The main responsibility of our proposed hyper-heuristic is the careful selection of the heuristic that is to be utilized for the next epoch.

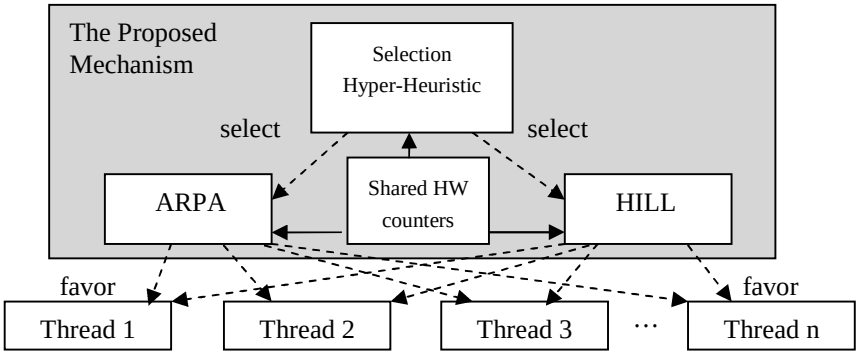

Fig.2. The proposed design.

There are two types of high level hyper-heuristic methodologies managing a set of low level heuristics: *selection* and *generation* [8]. Selection hyper-heuristics frequently consist of two successive stages of heuristic selection and move acceptance [9]. Most of the simple selection hyper-heuristic components are introduced in [10]. For example, *random permutation gradient* heuristic selection creates a permutation list of low level heuristics and chooses a low level heuristic in the given order one by one at each step to apply on the current solution. If a chosen heuristic makes an improvement, the same heuristic is utilized.

There are more elaborate hyper-heuristics making use of machine learning techniques. For example, a reinforcement learning based hyper-heuristic assigns a utility score for each heuristic which is increased using a rewarding mechanism after improvement or decreased as a punishment mechanism after a worsening move [11]. A heuristic is chosen based on this score which then gets updated at each step. Different strategies can be utilized for heuristic selection, one of them being selection of the low level heuristic with maximum score. There is theoretical [12] as well as empirical evidence [13] that hyper-heuristics are effective solution methodologies.

We investigated different heuristic selection methods and hyper-heuristics. A simplified variant of a reinforcement learning based hyper-heuristic is employed, in this study. This variant uses different success criteria based on two successive stages and also different heuristic selection mechanisms to choose a low level heuristic at each step. The success measure is used as the utility score of a heuristic.

Hyper-Heuristic 1 (HH1): Our first hyper-heuristic is based on *committed instructions per cycle per epoch* ($IPCepoch_i$) metric. This success measure is a good indicator for the processor performance during an epoch. When this value is decreasing in the current epoch, we can directly say that something is going wrong. In such cases,

HH1 punishes the heuristic that is used in the previous epoch and changes it with an alternative heuristic for the incoming epoch. In our study, we only utilized two heuristics (ARPA and HILL), and, hence, we choose the alternative heuristic in such cases. Figure 3 shows the pseudo code for this hyper-heuristic.

```
IF IPCepoch(i) >= IPCepoch(i-1) THEN
    Keep the current heuristic running for the next epoch
ELSE
    Change the heuristic
ENDIF
```

Fig.3. The pseudo code for HH1.

Hyper-Heuristic 2 (HH2): The second hyper-heuristic is based on a different metric which we call *commit over fetch*, as shown in Figure 4. Generally, the number of instructions that enters the processor may not match the number of instructions that exits the processor by a successful completion. $IPCepoch_i$ value can be equal to but generally much less than the *fetched instructions per cycle per epoch* ($FIPCepoch_i$). The main reason for this phenomenon is due to the speculative nature of today's processors. To improve the processor throughput, the processors run instructions in an out of program order and have hardware branch predictors that may fill the processor pipeline from speculative paths. When the branch outcome is incorrectly predicted, instructions that are fetched from the wrong path are all flushed. Here, in this metric, we measure if the number of flushed instructions are increasing. When this happens, HH2 punishes the previously utilized heuristic by selecting the alternative heuristic.

Hyper-Heuristic 3 (HH3): Our final proposed hyper-heuristic design for performance optimization of simultaneous multithreaded processors is based on the random permutation gradient hyper-heuristic. Here, we propose a slightly complex evaluation function. First, we check if the $IPCepoch_i$ improves as we do in HH1 with a minor twist. By adding a *threshold value* to the algorithm, we want to tolerate the small fluctuations in the performance due to external factors (phase changes in threads, increased cache steals among threads, etc.), which are not related to the performance of the running heuristic. Secondly, in our study, we observed that the overall performance may radically drop in a number of epochs. To stabilize our algorithm further, we give one more chance to the running heuristic, if it is ARPA, even when the drop in $IPCepoch_i$ is below our threshold value. In our experiments, we found that ARPA

```
commitOverFetch(i) ← IPCepoch(i) / FIPCepoch(i)
IF commitOverFetch(i) >= commitOverFetch(i-1) THEN
    Keep the current heuristic running for the next epoch
ELSE
    Change the heuristic
ENDIF
```

Fig.4. The pseudo code for HH2.

is a more successful heuristic compared to HILL, and this is for an insurance not to punish a well-performing heuristic, mistakenly. Finally, as in HH2, we check if the efficiency of the last epoch is not worse than the efficiency of its predecessor. If this is the case, then we continue using the same heuristic; otherwise, we change the heuristic. The pseudo code of the algorithm is given in Figure 5.

```
IF IPCepoch(i) / IPCepoch(i-1) > thresholdValue THEN
    Keep the current heuristic running for the next epoch
    oneMoreChance ← 0
ELSE
    oneMoreChance++
    IF oneMoreChance is 1 AND the current heuristic is ARPA THEN
        Giving one more chance to the current heuristic
    ELSE
        commitOverFetch(i) ← IPCepoch(i) / FIPCepoch(i)
        IF commitOverFetch(i) >= commitOverFetch(i-1) THEN
           Keep the current heuristic running for the next epoch
        ELSE
           Change the heuristic
           oneMoreChance ← 0
        ENDIF
    ENDIF
ENDIF
```

Fig. 5. The pseudo code for HH3.

3 Computational Experiments

3.1 Processor Specifications

M-Sim [13] is used in our study to simulate the SMT processor. M-Sim is modified to support ARPA and HILL in any epoch in order to run these heuristics in mixed order. We arbitrarily chose 7 benchmarks from SPEC2000 benchmark suite in order to evaluate our work. We ran our tests for all 35 possible 3-thread mixtures consisting of these benchmarks. 10M instructions from each thread are skipped before per-cycle simulation begins and it stops after 5Mcycles. The epoch size is set to 32 Kcycles.

The simulated processor can decode/issue/commit 8 instructions per cycle. Reorder buffer, issue queue and load/store queue sizes are 64, 40 and 32 entries, respectively. There are 128 integer and 128 floating point registers. L1 instruction cache is 2-way with 32KB capacity and L1 data cache is 4-way with 32KB capacity. The L2 cache is unified; 512KB in size and it is 4-way. All caches use least recently used replacement policy. Access to L2 cache is 20 cycles. The main memory has 2 ports and access time is 300 cycles for the first chunk and inter-chunk access delay is 6 cycles.

3.2 Tests and Results

Figure 6 shows the average and peak results of our proposed hyper-heuristics compared to ARPA. We are not comparing our results to HILL, since its results are

generally worse than the results of ARPA. As the result clearly indicates, none of the hyper-heuristic is no worse than ARPA, and, suprisingly, HH3 performs 1.8% better than ARPA and 2.4% better than HILL (not shown on graph), on the average. The peak performance values on a few workloads seem to be very promising, as well (41% IPC gain on *art-bzip2-mgrid* mixture).

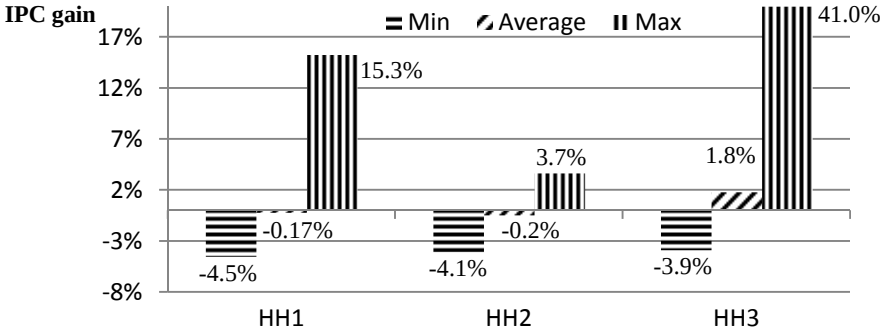

Fig. 6. The worst, the average and the best results of hyper-heuristics over ARPA.

In Figure 7, we show the number of workloads that perform better than ARPA (the leftmost bar) and number of workloads better than both ARPA and HILL (the rightmost bar) out of 35 workloads. Again, the best performing hyper-heuristic is HH3 which performs better on 25 workloads over ARPA, and 20 workloads over both ARPA and HILL.

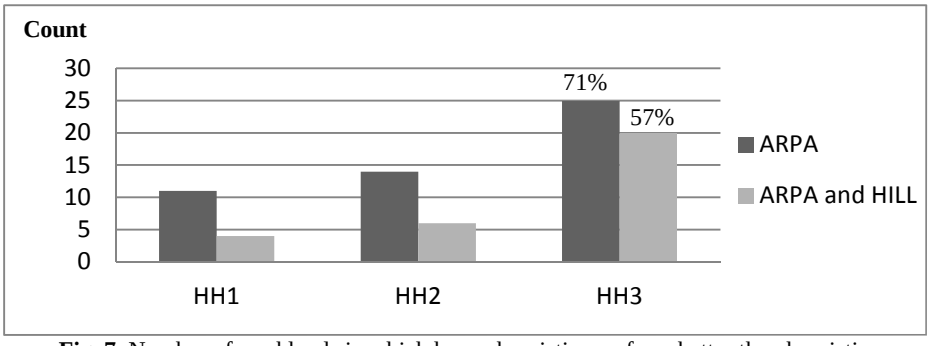

Fig. 7. Number of workloads in which hyper-heuristics perform better than heuristics.

4 Conclusion and Future Work

In this study, we investigate the performance of some selection hyper-heuristics that mix multiple heuristics with different abilities. The results show that best performing hyper-heuristic is better than the well-known approaches in almost 60% of the studied workloads. Moreover, it generates as much as 41% of performance improvement in some workloads with 1.8% to 2.4% over ARPA and HILL on average. The results are very promising, and we believe that there is still room for research to design better hyper-heuristics in this area.

References

1. D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In: *23rd annual International Symposium on Computer Architecture*, pages 191-202, New York, NY, USA, 1996. ACM.
2. D. M. Tullsen, J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In: *34th annual ACM/IEEE International Symposium on Microarchitecture*, pages 318-327, Washington, DC, USA, 2001. IEEE Press.
3. F. J. Cazorla, A. Ramirez, M. Valero, E. Fernandez. Dynamically controlled resource allocation in SMT processors. In: *37th annual IEEE/ACM International Symposium on Microarchitecrure*, pages 171-182, Washington, DC, USA, 2004. IEEE Press.
4. S. Choi, D. Yeung. Learning-based SMT processor resource distribution via hill-climbing. In: *33rd annual International Symposium on Computer Architecture*, pages 239-251, Washington, DC, USA, 2006. IEEE Press.
5. H. Wang, I. Koren, C. M. Krishna. An adaptive resource partitioning algorithm for SMT processors. In: *17th International Conference on Parallel Architectures and Compilation Techniques*, pages 230-239, New York, NY, USA, 2008. ACM.
6. H. Vandierendonck, A. Seznec. Managing SMT resource usage through speculative instruction window weighting. *ACM Trans. on Arch. and Code Opt.*, 8(3), 2011. ACM.
7. S. Eyerman, L. Eeckhout. Memory-level parallelism aware fetch policies for simultaneous multithreading processors. *ACM Trans. on Arch. and Code Opt.*, 6(1), 2009. ACM.
8. E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, J. R. Woodward. A classification of hyper-heuristic approaches. Handbook of Metaheuristics. Vol. 146 of *Intl Series in Op. Res. and Man. Sci.*, 2010, pages 449–468, 2010. Springer.
9. E.Özcan, B. Bilgin, and E. E. Korkmaz. A comprehensive analysis of hyper-heuristics. *Intell. Data Anal.*, 12(1):3–23, 2008.
10. P.Cowling, G. Kendall, E. Soubeiga. A hyperheuristic approach to scheduling a sales summit. In: *Selected papers from the Third International Conference on Practice and Theory of Automated Timetabling*, pages 176–190, London, UK, 2001. Springer-Verlag.
11. A. Nareyek. Choosing search heuristics by non-stationary reinforcement learning. In:*Metaheuristics: Computer Decision-Making*, ch. 9, pages 523–544, 2003.Kluwer
12. P.K. Lehre, E. Özcan. A runtime analysis of simple hyper-heuristics: To mix or not to mix operators. In*: Pre-Conf. Proc. of Foundations of Gen. Algorithms XII*, pages 91–98, 2013.
13. E. K. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, R. Qu. Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society*, to appear, 2013.
14. J. J. Sharkey, D. Ponomarev, K. Ghose. M-SIM: A Flexible, Multithreaded Architectural Simulation Environment. Department of Computer Science, Binghamton University, *Technical Report No.*CS-TR-05-DP01, 2005.