# HYPER-HEURISTICS FOR PERFORMANCE OPTIMIZATION OF SIMULTANEOUS MULTITHREADED PROCESSORS

by

İsa Ahmet Güney

Submitted to the Institute of Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science
in
Computer Engineering

Yeditepe University

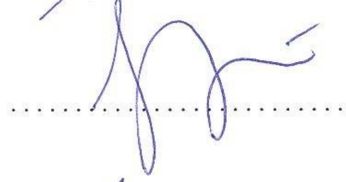
2014

HYPER-HEURISTICS FOR PERFORMANCE OPTIMIZATION OF SIMULTANEOUS MULTITHREADED PROCESSORS

APPROVED BY:

Assistant Prof. Dr. Gürhan Küçük ..............................
(Thesis Supervisor)

Associate Prof. Dr. Sezer Gören Uğurdağ ..............................



Associate Prof. Dr. Alper Şen ..............................

DATE OF APPROVAL: ..../..../2014

# ABSTRACT


## HYPER-HEURISTICS FOR PERFORMANCE OPTIMIZATION OF SIMULTANEOUS MULTITHREADED PROCESSORS

In Simultaneous Multi-Threaded processor datapaths, there are many resources that are concurrently shared by multiple threads. A few number of heuristic approaches, which explicitly distribute those resources among threads with the goal of an improved overall performance, have been proposed. A selection hyper-heuristic is a high level search methodology which mixes a predetermined set of heuristics under an iterative framework to exploit their strengths while solving a given problem. In this study, we propose a set of learning selection hyper-heuristics for predicting, choosing and running the best performing heuristic at periodic time intervals that we name epochs. The empirical results show that hyper-heuristics are capable of improving the performance of the studied workloads. The peak performance improvement is observed to be around 25 per cent over a previously proposed Hill Climbing heuristic and around 11 per cent over Adaptive Resource Partitioning Algorithm. Our best hyper-heuristic, HH4, performs better than either of the state-of-the art heuristics on almost 72 per cent of the simulated workloads. HH4 also beats both of the heuristics on around 30 per cent of the simulated workloads.

# ÖZET

## EŞZAMANLI ÇOKLU İŞPARÇACIKLI İŞLEMCİLERİN BAŞARIM ENİYİLENMESİ İÇİN ÜST-SEZGİSELLER

Eşzamanlı Çoklu İşparçacıklı işlemcilerin veri yollarında birçok kaynak eş zamanlı olarak birden çok iş parçacığı tarafından paylaşılmaktadır. Literatürde, performansı artırmak amacıyla bu kaynakları iş parçacıkları arasında doğrudan bölüştüren sezgisel yaklaşımlar mevcuttur. Seçici-üst-sezgiseller, önceden belirlenmiş bir sezgisel kümesinden sezgiselleri seçerek bir problemin çözümünde bu sezgisellerin avantajlarından faydalanan bir metottur. Bu çalışmada, çağ olarak adlandırdığımız periyodik zaman aralıklarında en iyi çalışan sezgiseli tahmin etmek ve seçmek için bir grup öğrenme tabanlı üst sezgisel sunuyoruz. Deneysel sonuçlar üst sezgisellerin test edilen iş yüklerinin performansını artırma potansiyeline sahip olduğunu göstermektedir. Çalışmamızda gözlemlenen en yüksek performans kazançları, literatürde sunulmuş olan Hill Climbing sezgiseli için yaklaşık yüzde 25, Adaptive Resource Partitioning Algorithm sezgiseli için ise yaklaşık yüzde 11'dir. En iyi üst sezgiselimiz olan HH4, literatürdeki en başarılı sezgisellerden en az birinde, test edilen iş yüklerinin yaklaşık yüzde 72'sinde daha iyi sonuç vermektedir. En iyi üst-sezgiselimiz olan HH4 aynı zamanda test edilen iş yüklerinin yaklaşık yüzde 30'unda ise her iki sezgiselden de daha yüksek performans göstermektedir.

# TABLE OF CONTENTS


ABSTRACT ........ iii
ÖZET ........ iv
TABLE OF CONTENTS ........ v
LIST OF FIGURES ........ vii
LIST OF TABLES ........ x
LIST OF ALGORTIHMS ........ xii
LIST OF SYMBOLS / ABBREVIATIONS ........ xiii
1. INTRODUCTION ........ 1
  1.1. Background ........ 5
2. RELATED WORK ........ 11
  2.1. Heuristics for SMT Resource Management ........ 11
  2.2. Hyper-Heuristics ........ 17
3. PRELIMINARY ANALYSIS ........ 20
  3.1. Motivation ........ 20
    3.1.1. [bzip2-cactusADM-hmmer] ........ 20
    3.1.2. [lbm-milc-gobmk] ........ 24
    3.1.3. [art-mcf-mgrid] ........ 28
    3.1.4. Long Run ........ 31
4. PROPOSED DESIGN ........ 33
  4.1. Combining the Heuristics ........ 34
  4.2. Implementation Details of Heuristics ........ 36
    4.2.1. Implementation Details of HILL ........ 36
    4.2.2. Implementation Details of ARPA ........ 37
    4.2.3. Parameters ........ 37
  4.3. Proposed Hyper-Heuristics ........ 38
    4.3.1. HH1: IPC-based Hyper-Heuristic ........ 38
    4.3.2. HH2: Commit-over-fetch-based Hyper-Heuristic ........ 39
    4.3.3. HH3: IPC-and-Commit-over-fetch-based Hyper-Heuristic ........ 39
    4.3.4. HH4: Round-based Hyper-Heuristic ........ 40

4.4. Hardware Complexity ........ 41
5. EXERIMENTAL METHODOLOGY ........ 46
5.1. Processor Specifications ........ 46
5.2. Benchmarks ........ 47
5.3. Heuristic Parameters ........ 47
5.4. Metrics ........ 48
6. TESTS AND RESULTS ........ 50
6.1. Sensitivity Analysis ........ 50
6.2. Results and Discussion ........ 53
7. CONCLUSIONS AND FUTURE WORK ........ 61
APPENDIX A: RESULTS OF TWO-THREADED BENCHMARKS ........ 62
APPENDIX B: RESULTS OF THREE-THREADED BENCHMARKS ........ 65
APPENDIX C: RESULTS OF FOUR-THREADED BENCHMARKS ........ 69
REFERENCES ........ 73

## LIST OF FIGURES

Figure 1.1. ARPA and HILL performance comparison on a few SMT workloads..... 4

Figure 1.2. An example scalar datapath........................................................ 6

Figure 1.3. An example pipelined datapath................................................. 6

Figure 1.4. An example superscalar datapath.............................................. 7

Figure 1.5. An example out-of-order superscalar datapath........................... 9

Figure 1.6. An example SMT datapath....................................................... 10

Figure 2.1. The significance of domain barrier in hyper-heuristic design.................. 18

Figure 3.1. Cumulative IPC values for ARPA, BEST and HILL in [bzip2-cactusADM-hmmer]........................................................................ 21

Figure 3.2. Individual epoch IPC values of ARPA, BEST and HILL in [bzip-cactusADM-hmmer]........................................................................ 22

Figure 3.3. Individual IPC values of cactusADM when run in standalone mode and in a mixture with HILL ................................................................. 23

Figure 3.4. Total number of instructions committed by cactusADM when run in standalone mode and in a mixture with HILL.......................................... 24

Figure 3.5. Individual epoch IPC values of ARPA, BEST and HILL in [lbm-milc-gobmk]........................................................................................ 25

| | | |
|---|---|---|
| Figure 3.6. | Performance result of individual threads in ARPA in [lbm-milc-gobmk] | 26 |
| Figure 3.7. | Performance results of individual threads in HILL in [lbm-milc-gobmk] | 27 |
| Figure 3.8. | Performance results of individual threads in BEST in [lbm-milc-gobmk] | 28 |
| Figure 3.9. | Individual epoch IPC values of ARPA, BEST and HILL in [art-mcf-mgrid] | 29 |
| Figure 3.10. | IPC values of all 1024 possible permutations in workload [bzip2-cactusADM-hmmer] | 30 |
| Figure 3.11. | IPC values of all 1024 possible permutations in workload [lbm-milc-gobmk] | 30 |
| Figure 3.12. | IPC values of all 1024 possible permutations in workload [art-mcf-mgrid] | 31 |
| Figure 3.13. | Cumulative performances of ARPA, HILL and LIMIT in [gobmk-lbm-mcf-sjeng] | 32 |
| Figure 4.1. | The proposed design | 34 |
| Figure 4.2. | An example timeline with heuristics running in arbitrary order | 35 |
| Figure 4.3. | An example timeline where the hyper-heuristic decides HILL, ARPA and HILL should be run on an SMT processor with three threads | 35 |
| Figure 4.4. | Detailed design of hyper-heuristic controlled resource allocation | 45 |
| Figure 6.1. | Percentage of workloads for which hyper-heuristics perform better or | 56 |

equivalent to HILL and ARPA in terms of IPC ........................................

Figure 6.2. Normalized performance results of two-thread workloads utilizing HILL, ARPA, and HH4 .................................................................. 56

Figure 6.3. Normalized QoS results of two-thread workloads utilizing HILL, ARPA, and HH4 ............................................................................ 57

Figure 6.4. Normalized fairness results of two-threaded workloads utilizing HILL, ARPA, and HH4 ............................................................................ 57

Figure 6.5. Normalized performance results of three-thread workloads utilizing HILL, ARPA, and HH4 .................................................................. 58

Figure 6.6. Normalized QoS results of three-thread workloads utilizing HILL, ARPA, and HH4 ............................................................................ 58

Figure 6.7. Normalized fairness results of three-thread workloads utilizing HILL, ARPA, and HH4 ............................................................................ 59

Figure 6.8. Normalized performance results of four-thread workloads utilizing HILL, ARPA, and HH4 .................................................................. 59

Figure 6.9. Normalized QoS results of four-thread workloads utilizing HILL, ARPA, and HH4 ............................................................................ 60

Figure 6.10. Normalized fairness results of four-thread workloads utilizing HILL, ARPA, and HH4 ............................................................................ 60

## LIST OF TABLES


Table 4.1. Transistor cost of HILL ........ 43

Table 4.2. Transistor cost of ARPA ........ 43

Table 4.3. Transistor cost of hyper-heuristic design ........ 45

Table 5.1. Processor Specifications ........ 46

Table 5.2. Heuristic Parameters ........ 47

Table 6.1. Performance gains of various threshold values (th.v.s) in HH3 against STATIC ........ 51

Table 6.2. Percentages of ARPA run in various threshold values (th.v.s) in HH3 ........ 52

Table 6.3. Performance gains of HH4 in various round lengths (r.l.s) against STATIC ........ 52

Table 6.4. Performance gains of heuristic against STATIC ........ 54

Table 6.5. Peak gains of proposed hyper-heuristics over HILL and ARPA ........ 54

Table A.1. IPC results of two-threaded benchmarks ........ 62

Table A.2. Average Weighted IPC results of two-threaded benchmarks ........ 63

Table A.3. Harmonic Mean of Weighted IPC results of two-threaded benchmarks ........ 64

Table B.1. IPC results of three-threaded benchmarks ........ 66

Table B.2. Average Weighted IPC results of three-threaded benchmarks ................ 67

Table B.3. Harmonic Mean of Weighted IPC results of three-threaded benchmarks. 68

Table C.1. IPC results of four-threaded benchmarks ................................................ 70

Table C.2. Average Weighted IPC results of four-threaded benchmarks .................. 71

Table C.3. Harmonic Mean of Weighted IPC results of four-threaded benchmarks .. 72

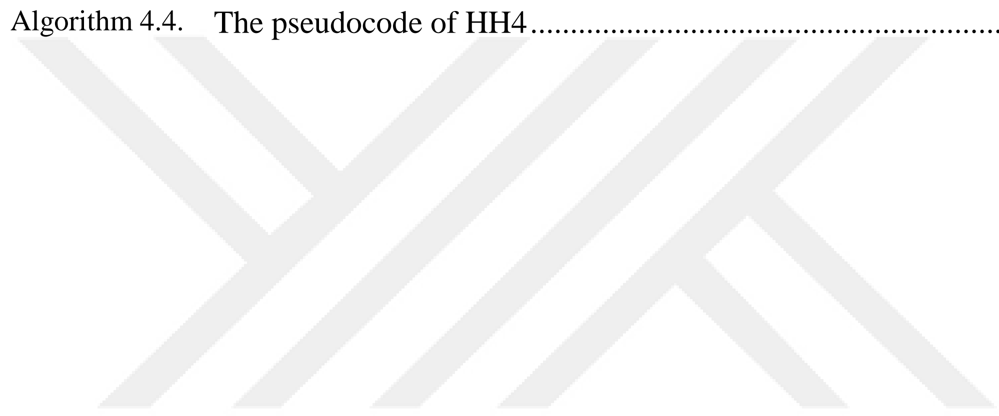
# LIST OF ALGORITHMS


Algorithm 4.1. The pseudocode of HH1 ........ 39

Algorithm 4.2. The pseudocode of HH2 ........ 39

Algorithm 4.3. The pseudocode of HH3 ........ 40

Algorithm 4.4. The pseudocode of HH4 ........ 41

# LIST OF SYMBOLS / ABBREVIATIONS

| | |
|---|---|
| ARPA | Our implementation of adaptive resource partitioning algorithm |
| BEST | Best performing permutation in limit studies |
| CIPRE | Committed instructions per resource entry |
| DCRA | Dynamically controlled resource allocation |
| D-TLB | Translation lookaside buffer |
| $FIPCepoch_i$ | Fetched instructions per cycle per epoch |
| HILL | Our implementation of hill climbing |
| HH1: | Proposed hyper-heuristic 1: IPC-based hyper-heuristic |
| HH2: | Proposed hyper-heuristic 2: Commit-over-fetch-based hyper-heuristic |
| HH3: | Proposed hyper-heuristic 3: IPC-and-Commit-over-fetch-based hyper-heuristic |
| HH4: | Proposed hyper-heuristic 4: Round-based hyper-heuristic |
| ILP | Instruction level parallelism |
| IPC | Instruction per cycle |
| $IPCepoch_i$ | Committed instructions per cycle per epoch |
| IFQ | Instruction fetch queue |
| IQ | Issue queue |
| L2 | Level two of cache |
| LIMIT | Best performing permutation in long run |
| LLSR | Long latency shift register |
| LSQ | Load/Store queue |
| MLP | Memory level parallelism |
| OARPA | Original implementation of adaptive resource partitioning algorithm |
| OHILL | Original implementation of hill climbing |
| Opcode | Operation code |
| PC | Program counter |
| QoS | Quality of service |
| ROB | Re-order buffer |
| Single_IPC | Instruction per cycle of threads when they are run standalone |
| SIWW | Speculative instruction window weighting |

SMT Simultaneous multi threading

STATIC Static resource partitioning algorithm

T Number of threads that run simultaneously

TLP Thread level parallelism

# 1. INTRODUCTION

Simultaneous Multi-Threaded (SMT) processors aim to improve the system throughput by issuing instructions from multiple threads within a single clock cycle [1]. The SMT architecture is simply a modified superscalar processor with many shared datapath resources, such as Issue Queue, Re-Order Buffer, Load/Store Queue, Physical Register Files, Arithmetic Logic Units, and caches. In a typical superscalar processor, the datapath resources are also shared by multiple threads but each thread waits it's time for the total control of all datapath 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 those resources among threads while maximizing the processor throughput is a major challenge; and today, most of the research effort in the field is focused on this issue.

There are various strategies to improve the efficiency of SMT processors. First, there are fetch policies that attempt to regulate the stream of instructions introduced to 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. Most famous examples of these techniques are *Instruction Count*, which gives fetch priority to threads with less resource occupancy, *Branch Count*, which favors threads with the fewest unresolved branches, *Data Cache Miss Count*, 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 recovers by flushing the stalled instructions when it happens.

Beside the implicit techniques, there are *explicit* resource partitioning methods that partition the shared resources according to runtime behavior of the running threads. Basically, these techniques explicitly decide how a shared resource is to be partitioned and distributed. The most well known example of these techniques is the Dynamically

Controlled Resource Allocation (DCRA) [4]. In DCRA, each thread and 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, and when a resource is not used by a thread for a number of cycles within a predetermined threshold, the thread for that resource becomes *inactive*. Then, the DCRA mechanism attempts to give more resources to the *slow* threads by stealing 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 stealing a few resource entries from then and giving them to the *slow* threads. Similarly, when a thread is labeled as *inactive* for a resource, then there is no harm giving its share on that resource to the threads that actually need it.

SMT resource distribution via hill climbing is another explicit resource partitioning mechanism that runs in epochs (periodic intervals) [5]. Hill Climbing heuristic assumes that there is a certain maxima in the performance graph and it attempts to reach to that peak by dynamically changing resource distributions in a greedy fashion. In the initial trial epochs, each thread is given a 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 the additional resources. Then, these trial epochs and the consequent resource distributions are periodically applied.

Finally, the Adaptive Resource Partitioning Algorithm introduces the efficiency metric for distributing the resources [6]. Similar to Hill Climbing, Adaptive Resource Partitioning Algorithm tries to give more resources to the most deserving thread by stealing resource entries 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 the algorithm gives more resources to that thread. In Hill Climbing, a thread can show the best performance and be chosen to receive more resources every epoch regardless of it efficiency. As a result, a thread may starve to its death, since it cannot perform better than some other thread. Adaptive Resource Partitioning Algorithm 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 improve afterwards.

Heuristics are inexact, rule of thumb computational methods, tailored for a specific problem in hand. DCRA, Hill Climbing and Adaptive Resource Partitioning Algorithm are examples of heuristic approaches. 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 results across the given instances. For each instance, a different heuristic might perform the best. Hyper-heuristics have emerged as general high level methods searching the space generated by a set of low level heuristics rather than solutions directly to solve a given problem [7]. The main goal is to combine the strengths of multiple 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 problems within dynamic environments [8, 9].

Our literature survey, preliminary studies and the variety of problem instances and the observed dynamic changes show us that hyper-heuristics are very suitable 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 analysis for the proposed heuristics. In this study, we aim to optimize the performance of SMT processors by partitioning datapath resources among running threads using hyper-heuristics under a multistage framework. Since, the Hill Climbing and Adaptive Resource Partitioning Algorithm heuristics have similar periodic nature; we studied combining both under several hyper-heuristic approaches throughout this study In order for our model to work, the processor should be able to run Hill Climbing and Adaptive Resource Partitioning Algorithm at each epoch, interchangeably. To make both heuristics fully compatible, we made some minor changes in our implementation of these heuristics. We will refer to our implementations of Hill Climbing and Adaptive Resource Partitioning Algorithm as HILL and ARPA, for the rest of the study. To evaluate our work correctly, all performance results of Hill Climbing and Adaptive Resource Partitioning Algorithm are gathered by running our implementations of these heuristics.

Figure 1.1 shows the performance graphs for some of the well known benchmark workloads[1] that we studied [10] in terms of IPC (Instruction Per Cycle). In [art-gcc-mgrid] and [art-parser-vortex] workloads ARPA performs better than HILL (12 per cent and eight per cent, 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 five per cent better. Note that while ARPA is successfully running [art-parser-vortex] workload, when parser benchmark is replaced with mesa benchmark everything turns 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 works on hyper-heuristics to dynamically select the proper heuristic at run time on SMT processors.

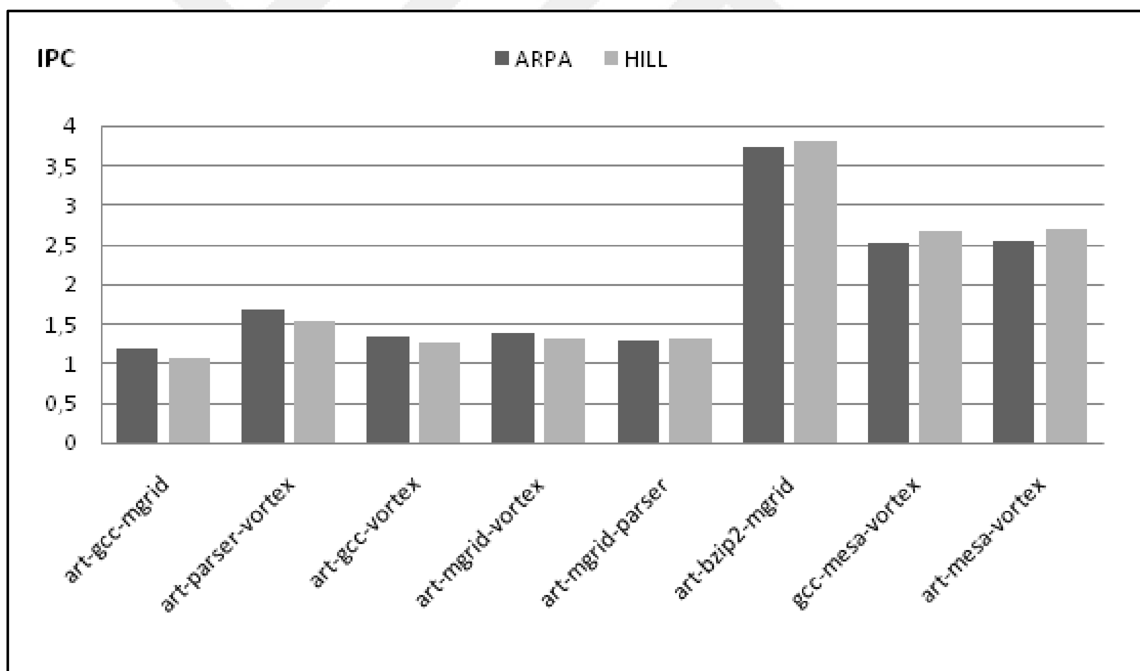



Figure 1.1. ARPA and HILL performance comparison on a few SMT workloads

[1] Throughout the study, we focus on two-, three- and four-threaded workloads. For instance, we use [a-b-c] notation to show that we run three applications a, b, and c; on a three-thread SMT processor. The details of the processor are given in Chapter 5.1.

### 1.1. Background

Execution of a single instruction consists of four main stages: *fetch*, *decode*, *execute*, and *writeback*. In the fetch stage, the instruction is retrieved from the main memory. In the decode stage, the instruction is resolved into its operation code (opcode) and operands (either literal values or register numbers). The execute stage is where the instruction is sent to the function units (as known as Arithmetic Logic Units) and actually executed. When the execution of the instruction is done, the output is written into the destination register in the writeback stage.

Processors use *clock signals* to synchronize circuitry. A *clock cycle* is the time elapsed between the two consecutive clock signals. In earlier implementations; fetch, decode, execute, and writeback stages were handled within a single clock cycle. With the help of instruction pipelining in processors, these stages were separated by pipeline latches where each stage could be handled within a clock cycle (Figure 1.2). Instruction pipelining increased throughput by handling multiple instructions within a single clock cycle; such as writing back an instruction while executing the next one, and decoding and fetching the instructions back to back at the same time. The reduced time needed to complete individual stages allowed processors to decrease the duration of a single clock cycle and increase the clock frequency. In some designs, some stages may be further divided into multiple stages, allowing further reduction of the clock cycle time.

Instruction pipelining also enabled the output of instructions to travel directly to the decode stage while being written into the destination register, if an instruction is waiting for the output. This allowed processors to avoid pipeline stalls due to instructions waiting for their source operands to be written into the register, only so they can read them. This type of forwarding an instruction output is allowed by the *bypass circuitry* (or *forwarding bus*). Figure 1.3 shows an example diagram of an instruction pipeline.

Superscalar processor are able to fetch, decode, execute, and writeback multiple instructions in a single clock cycle. A superscalar processor which can handle *n* instructions in a single cycle is typically described as an *n-way* processor. The superscalar architecture introduces the Issue Queue (IQ), which is a first-in first-out queue for

instructions waiting to be executed. The oldest instruction in the IQ may be sent to the function units for execution if its source operands are available. The bypass circuitry, again, forwards the output of instructions to the waiting instructions in the IQ. The additional logic needed to move instructions into IQ slot adds delay to the decode stage, putting it on the critical path of the processor. Therefore, an additional *dispatch* stage is introduced between decode and execute stages to handle this task.

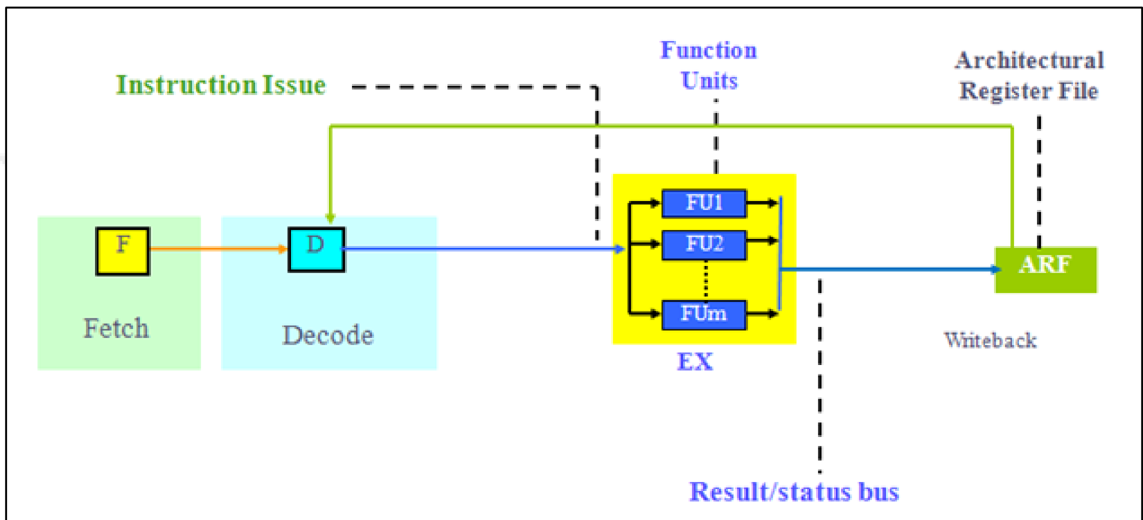



Figure 1.2. An example scalar datapath

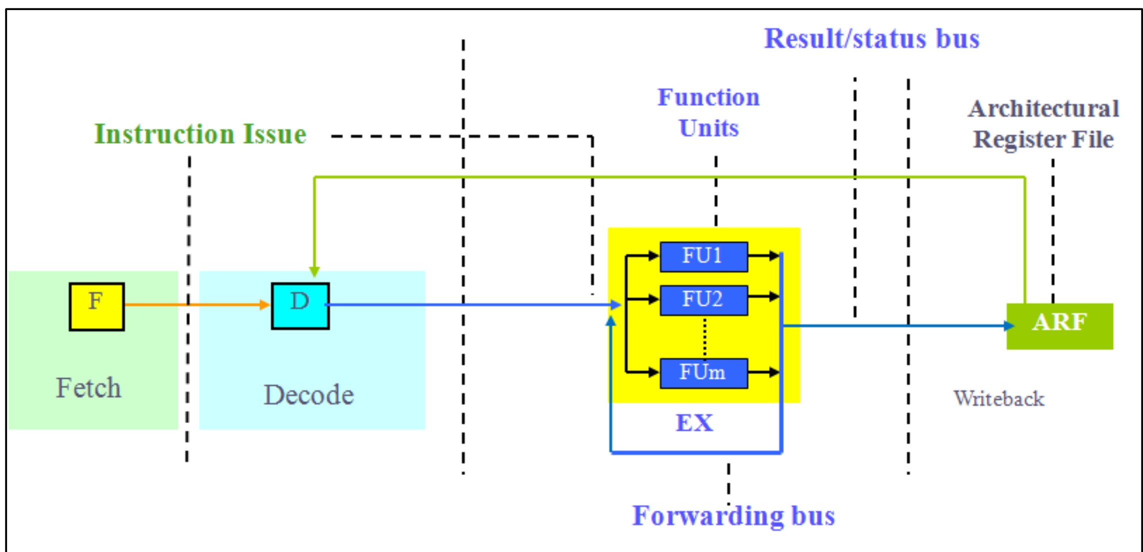



Figure 1.3. An example pipelined datapath

The purpose of superscalar architecture is to increase throughput and the utilization of function units by executing multiple instructions simultaneously by exploiting *instruction level parallelism* (ILP) hidden in the instruction streams. However, the ability of superscalar processors to exploit ILP is limited. Since instructions are executed in-order, any instruction in the stream which waits for the result of a prior instruction stalls the whole pipeline. Figure 1.4 shows an example diagram of a superscalar pipelined datapath.

Out-of-order superscalar processors are capable of executing instructions out of the program order. In out-of-order architectures, independent instructions in a window (IQ becomes a Dispatch Buffer) are identified and issued for execution, allowing further exploitation of ILP. To conserve the original program state, the retirement of instructions are put back in the program order by a circular first-in first-out unit called the Re-order Buffer (ROB). The retirement of instructions take place in a new stage called *commit*. Another task to be handled is register renaming, in which false data dependencies such as write-after-write and write-after-read are eliminated. The elimination of false dependencies prevents the processor from relating instructions inaccurately as dependent and reducing the ILP exploitable by the processor. Register renaming is handled in decode and dispatch stages.

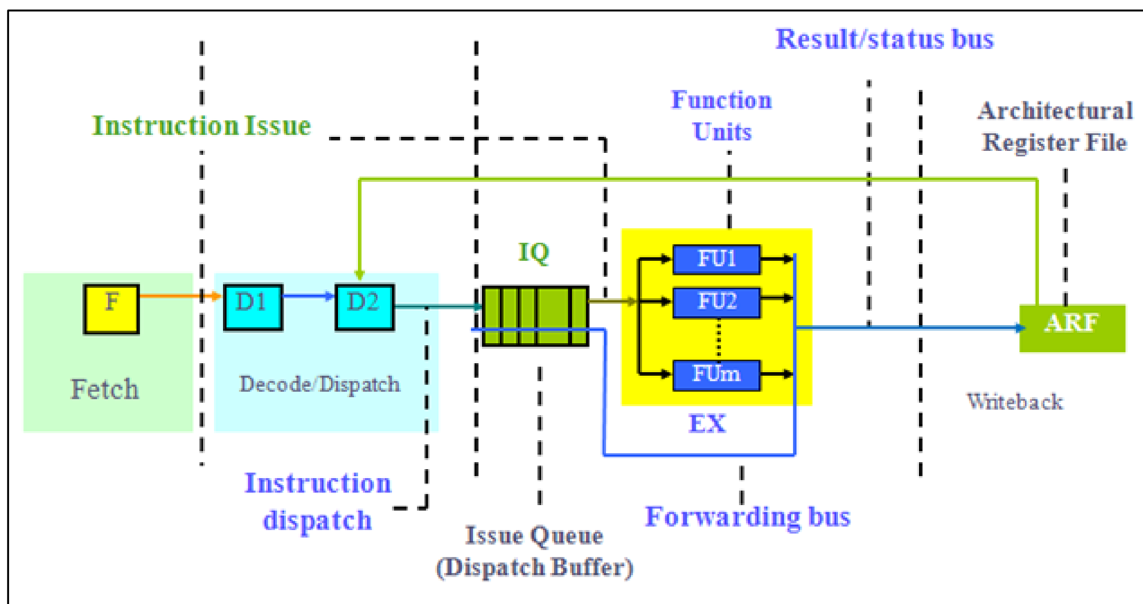



Figure 1.4. An example superscalar datapath

Most of the designs also introduce a queue structure called Load/Store Queue (LSQ), which is dedicated to memory instructions. When a memory instruction is dispatched, entries from IQ, LSQ, and ROB are allocated to that instruction. The purpose of LSQ is to assure that dependent memory instructions run in program order. LSQ also provides store to load forwarding and load bypassing mechanisms to improve the processor throughput.

An out-of-order datapath usually consists of three parts: in-order frontend, out-of-order core, and in-order backend. In the in-order frontend, instructions are fetched, decoded and dispatched in program order. Once an instruction is dispatched, an entry in the ROB is allocated for it, allowing the program order to be preserved. In the out-of-order core, instructions are issued for execution from the IQ, regardless of their program order. The out-of-order issue needs wakeup and selection logic to be implemented. Wakeup logic marks instructions as ready when their source operands become available. Then, the selection logic selects the instructions among the ready instructions for execution.

In the in-order backend, instructions are committed in program order. This is achieved by committing instructions from the head of the ROB structure one by one. Even if the execution of an instruction is finished, it cannot commit until all older instructions are committed. An example diagram of an out-of-order superscalar datapath is shown in Figure 1.5.

Although out-of-order superscalar architectures improve throughput by exploiting ILP; there is not enough parallelism inside a single application to fully utilize superscalar resources most of the time. SMT allows multiple threads to be run on the same datapath, exploiting Thread Level Parallelism (TLP). Instructions from multiple threads can be fetched, dispatched, executed, or committed in a single clock cycle. Resources such as fetch/dispatch/issue/writeback/commit bandwidths and IQ, LSQ, ROB, register file, and function units can be either shared among threads or replicated. The partitioning may be dynamic as well as it can be static. An example diagram of an SMT processor is given in Figure 1.6.

In an uncontrolled environment where all resources are shared among threads, some threads may dominate these shared resources. For example, if instructions are fetched from

threads in a round robin fashion, threads which are stalled due to high-latency instructions (usually memory instructions experiencing cache-misses) will not commit and release resources, but will continue allocating new entries. Thus, in time, most of the shared resources will be allocated to a stalled thread, which does not use those extra resources but prevents the other threads from making any progress. This problem is well recognized in the literature and is referred as *resource clogging*.

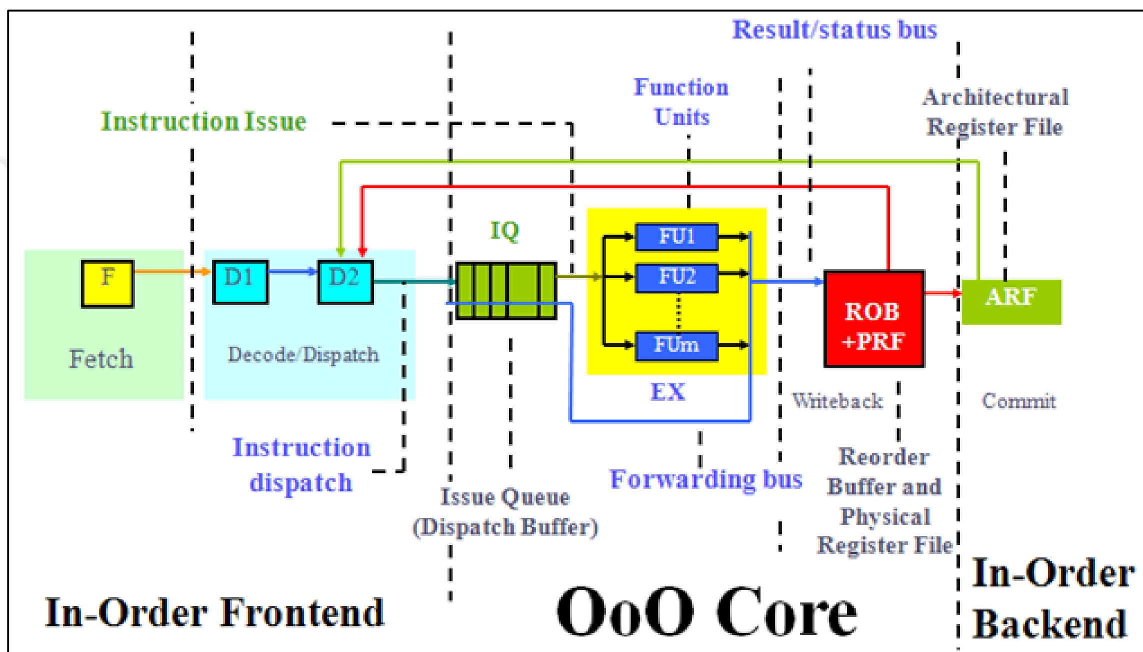



Figure 1.5. An example out-of-order superscalar datapath

Many solutions have been proposed to overcome the degrading effects of this problem, which are mentioned in Chapter 2.1. One of the most popular approaches is ICOUNT, which has been widely used in SMT studies. ICOUNT is a fetching mechanism, which fetches instructions from the thread with least amount of instructions in the decode and rename stages, and IQ. ICOUNT prevents stalled threads from dominating the resources to a degree. It also improves throughput by rewarding fast threads by fetching instructions. Since it is a fetch mechanism, it can be orthogonally applied to explicit resource partitioning algorithms, as well.

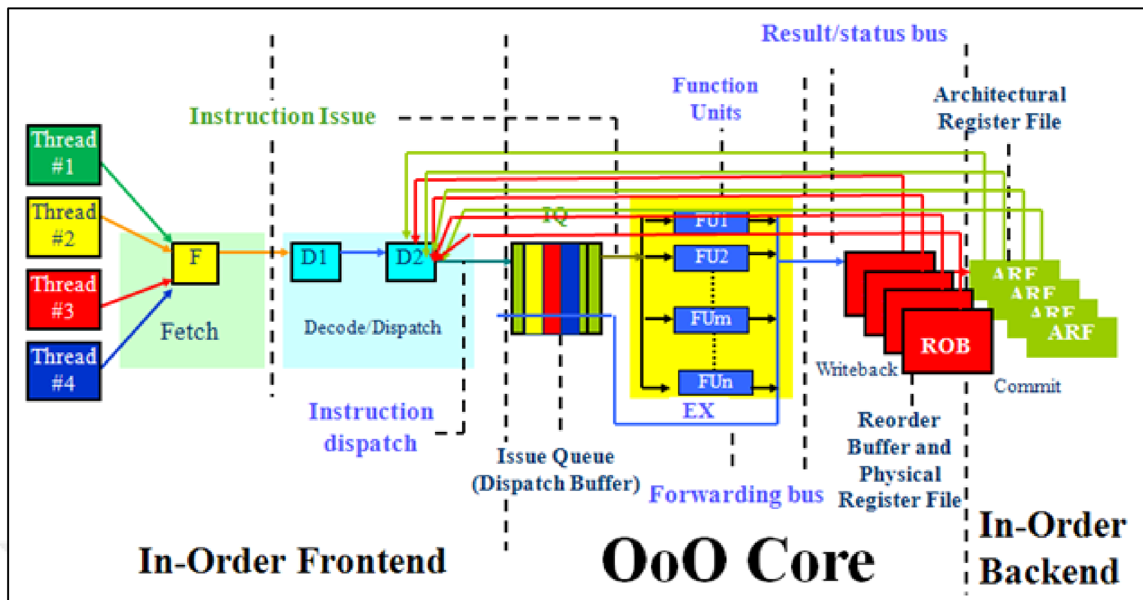



Figure 1.6. An example SMT datapath

# 2. RELATED WORK

## 2.1. Heuristics for SMT Resource Management

One of the major design challenges in SMT processors is on the pipeline frontend. 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 of these metrics are satisfied. In [2], the baseline architecture fetches instructions from threads in a round robin fashion. This can be done either by fetching as many instructions as possible from a single thread in a cycle; or by fetching fewer instructions from multiple threads. An improvement on instruction fetching is to fetch as many instructions as possible from one thread, and fill the rest of the available slots with instructions from subsequent threads.

The aforementioned study shows that selecting right instructions to be fetched has a great effect on performance. Hence, the study proposes different algorithms to select suitable threads for fetching instructions. Instructions from mispredicted branches occupy resources without improving the performance of a thread. BRCOUNT gives priority to threads with fewest unresolved branch instructions, therefore trying to reduce the number of speculative instructions fetched from mispredicted branches and allow these resources to be used by instructions that can make progress. The study shows that BRCOUNT indeed reduces the number of wrong path instructions and improves performance; however, the problem of mispredicted branch instructions on SMT processors is not a big deal since the nature of SMT already alleviates the impact of such instructions.

Instructions which need to wait a long time before they can be executed hold up IQ slots, preventing an issuable instruction from entering the IQ. MISSCOUNT aims to mitigate the effects of IQ-clog by reducing the number of instructions in the IQ that are dependent to long-latency memory instructions. To do so, MISSOUCNT prioritizes threads with fewest outstanding D-cache misses. The results show that although MISSCOUNT improves performance, the IQ-clog condition still occurs in a significant percentage of time, indicating that long-latency memory instructions are not the sole reason of IQ-clogs.

IQPSN acts on the intuition that threads with oldest instructions are more prone to cause IQ-clog. The heuristic gives a lower priority to threads which have their instructions closest to the head of IQ. ICOUNT is another mechanism proposed in [2], which favors more efficient threads. ICOUNT favors threads with the least number of instructions on-the-fly. This allows ICOUNT to give higher priority to more efficient threads, as well as to achieve a more balanced distribution of IQ slots among threads.

The results obtained in [2] shows that all heuristics yield a performance gain over the round robin policy. BRCOUNT and MISSCOUNT provide reasonable performance improvements only when there are enough threads. The most significant improvements are achieved through IQPSN and ICOUNT. Although IQPSN shows similar performance results, it never surpasses ICOUNT. The success and simplicity of ICOUNT has led the following studies in this area to widely use it as the sole instruction fetching mechanism, or the baseline heuristic to be compared against.

Tullsen and Brown discuss the effects of long-latency instructions, with the focus on long-latency load instructions, on SMT processors in [3]. According to the study, the power of SMT architecture comes from its ability to exploit inter-thread parallelism. However, when a thread issues an instruction which cannot be executed shortly, it diminishes the ability of other threads to find instructions to issue by holding onto resources when thread level parallelism is most needed. This is also backed up by the preliminary experiments.

The study proposes two mechanisms to detect long-latency load instructions: *trigger on miss* and *trigger on delay*. *Trigger on miss* identifies a long-latency load instruction whenever an L2 cache miss occurs whereas *trigger on delay* identifies one whenever an instruction spends more than a predetermined number of cycles in the load queue.

Once a long-latency load instruction is identified, instructions from that thread are evicted from the IQ, freeing resources to be used by other threads. The study also considers different policies for the starting point of a flush operation. *Next* flushes all instructions beginning from the next instruction after the load. *First use* flushes instructions beyond the first use of the loaded data. *After 10* and *After 20* flush starting from 10 and 20 instructions

after the load instruction. *Next branch* flushes after the next branch. *Trigger on delay* and *first use* are selected for further experiments in that study (FLUSH).

Other methods are considered in the same study, as well. *Stall fetch* (referred as STALL in the literature) stalls threads from fetching instructions once a long-latency load is identified. *Stall and flush* mechanism mixes FLUSH with STALL by stalling the thread when a long-latency load is detected, but only flushes if a resource is exhausted. *Pseudo-static* puts an upper limit on the number of instructions a thread can have in the queue stage or earlier.

The results show that STALL improves performance over the baseline scheme, but not as much as the other mechanisms. *Pseudo-static* does not achieve as high performance gains as others due to the inherent inefficiencies of placing artificial limits on the queue usage of threads. *Stall and flush* provides an improvement over FLUSH; however, the study argues that FLUSH achieves a good balance between implementation complexity and performance on a wide variety of workloads.

Cazorla et al. introduces a resource distribution mechanism called Dynamically Controlled Resource Allocation (DCRA) in [4]. Unlike the previous work, DCRA is not an implicit fetch policy. Instead, it limits the amount of resources usable by each thread. If a thread tries to use resources beyond its allowable limit, it is blocked from fetching further instructions.

The resource distribution logic of DCRA is based on two simple ideas: 1) the processor should not allocate resources to threads which do not need them, and 2) slow threads need more resources than fast threads to exploit the ILP hidden inside instruction streams. To distribute resources according to these considerations, DCRA categorizes each thread every cycle. A thread is identified as *slow* if it has a pending L2 cache miss, and as *fast*, otherwise. The reason for that is threads with long-latency loads are more likely to need a larger instruction window to find independent instructions. Additionally, a thread is labeled as *inactive* for a type of resource if it does not use that resource within a threshold number of cycles.

Upon the categorization of threads, each type of resource is distributed separately according to the number of active/inactive and slow/fast threads. Slow threads get significantly more resources than fast ones, whereas inactive threads do not get any resources that they would not use at all.

SMT resource distribution via hill climbing (hereafter, we will use the term OHILL, Original Hill Climbing for referring to this study) is another resource partitioning mechanism that runs in epochs (periodic intervals) [5]. OHILL assumes that there is a certain peak 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 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 (hereafter OARPA for Original Adaptive Resource Partitioning Algorithm) introduces efficiency metric into the picture [6]. Similar to OHILL, OARPA tries to give more resources to the most deserving thread by stealing resources from the others. The efficiency metric, 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 OARPA gives more resources to that thread. On the contrary, in OHILL, a thread can show the best performance and be chosen to receive more resources every epoch regardless of its efficiency.

Eyerman and Eeckhout point out that previous fetch policies do not take Memory Level Parallelism (MLP) into account [11]. As a matter of fact, by stalling fetch or flushing instructions, these previous fetch 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. The study consists of three parts: identifying long-latency loads, predicting MLP, and the MLP-Aware fetch policy.

Long-latency loads are identified using two different methods: they are either identified whenever a Last Level Cache or D-TLB miss occurs, or when a load instruction is predicted to be a long-latency one. The study uses the miss pattern predictor proposed in [12]. The prediction is done by comparing the number of load hits between the two most recent long-latency loads and the number of load hits since the last long-latency load. If these numbers are equal, the load instruction is predicted to be a long-latency one. These statistics are kept distinctly for each static load instruction in a table indexed by the Program Counter (PC).

The second task handled is to predict the MLP. In [11], MLP is referred as the average number of outstanding long-latency loads when there is at least one. To be able to determine if MLP is available at a given time, and how further should the processor go in order to exploit it; the MLP is predicted by an MLP distance predictor. Each thread is provided with a dedicated MLP distance predictor, which is a table indexed by the PC. The entries of this table store the number of instruction that must be executed in order to achieve the maximum MLP. This table is updated by using a vector called Long-Latency Shift Register (LLSR). Each LLSR has the size of the ROB dedicated to each thread. When an instruction is committed from a ROB, LLSR is shifted once; “one” is inserted from the other end if the instruction committed is a long-latency one, and “zero” is inserted, otherwise. The MLP distance predictor is updated for the long-latency load which reaches the head of LLSR (once a “one” reaches the head), equal to the distance to the farthest long-latency instruction in the LLSR.

Once the MLP distance is predicted, fetching mechanism can determine if there is MLP to exploit or not. The study proposes two different schemes at this point: *stall fetch* and *flush*, similar to the ones proposed in [3]. In *stall fetch*, upon the prediction of a long-latency load, *m* more instructions are fetched, and then the thread is fetch stalled, where *m* is the MLP distance prediction for that load instruction. *Flush* evicts instructions from the pipeline beyond MLP distance prediction once a long-latency load is detected. There is no prediction involved in *flush*. Both policies use ICOUNT when there are no long-latency load instructions.

Vainderendonck and Seznec propose a framework called Speculative Instruction Window Weighting (SIWW) for applying different fetch policies and resource limitations on the SMT architecture [13]. In this framework, SIWW fetch policy gives priority to instructions from threads with least amount of work left in the pipeline. The work identifies several types of instructions and each type is assigned a weight. The amount of work left for each thread is calculated by adding the weights of all instructions that belongs to that thread in the pipeline. Resource limitations can be applied by defining an upper limit for the "amount of work". A thread which exceeds this limit cannot fetch any more instructions until it commits and releases some instructions.

The study categorizes instructions into the following types: simple instructions, branch instructions and load instructions. Simple instructions consist of ALU instructions. Although the execution time needed by more complex instructions such as floating point division are higher than simpler instructions such as integer addition; all ALU instructions are categorized into a single type since the execution latencies of medium-latency instructions are well hidden by the out-of-order execution. Fetching wrong path instructions are costly to the performance; therefore branch instructions constitute another category. However, considering that the framework needs to take the contribution of the instructions beyond the branch to the amount of work left into account immediately and the framework can only predict the outcome of these branches until they are actually executed; high-confidence predictions can be assigned lower weights than low-confidence branches in order to compensate for the inaccuracy of predictors. The impact of load instructions with cache misses on performance another issue to handle. SIWW allows assigning different weights to predicted load hits and misses. Additionally, isolated cache misses should be treated differently than non-isolated ones. Load instructions which are predicted to miss, but are predicted to have MLP can be assigned lower weights than the ones that are predicted to have no MLP. SIWW uses the MLP predictor proposed in [11] for that purpose.

Assigning different weights to different instruction types allow SIWW to apply different fetch policies without making any changes to the hardware. For example, assigning a weight of one to all instruction types will result in a fetch policy equivalent to ICOUNT.

Additionally, the study proposes several different weight configurations which provide state-of-the-art throughput, fairness and harmonic mean performance.

Küçük et al. propose an improvement over SIWW, called HPIWW [14]. Instead of implementing a number of power-hungry speculation circuitry, HPIWW predicts the total amount of work left in the queue by keeping history of time spent by instructions in a circular queue for each thread. History-based predictions are already used in processor structures such as caches, branch predictors and load value predictors. Every time an instruction is issued into execution, two oldest entries are removed from the queue and its weight, which is calculated by subtracting IQ entrance cycle from the current cycle, is inserted into the queue. Whenever an instruction enters IQ, the head of the queue moves one entry away from the tail. The total amount of work left can be predicted by calculating the summation of all elements in the queue. By removing the speculative circuitry, HPIWW achieves great decrease in power consumption compared to SIWW, which becomes a hotspot in the processor. Also, by providing more accurate predictions, HPIWW provides higher throughput than SIWW.

## 2.2. Hyper-Heuristics

Hyper-heuristics are high level methodologies that operate 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 60s [15]. There are two main types of hyper-heuristics managing a set of low level heuristics: *selection* and *generation* hyper-heuristics [16]. Currently, hyper-heuristics are designed based on the notion of a *domain barrier* which separates the problem domain from the high level method. The barrier (depicted in Figure 2.1) acts as a filter disallowing no problem specific information from the problem domain to the hyper-heuristic level. This approach provides 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 hyper-heuristic, which is the focus of this study, is often an iterative search method, consisting of two components that are invoked successively at each step: *heuristic*

*selection* and *move acceptance* methods [17]. This type of framework mixes and controls a predefined fixed set of low level heuristics. Most of the simple selection hyper-heuristic components are introduced in [18]. 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.

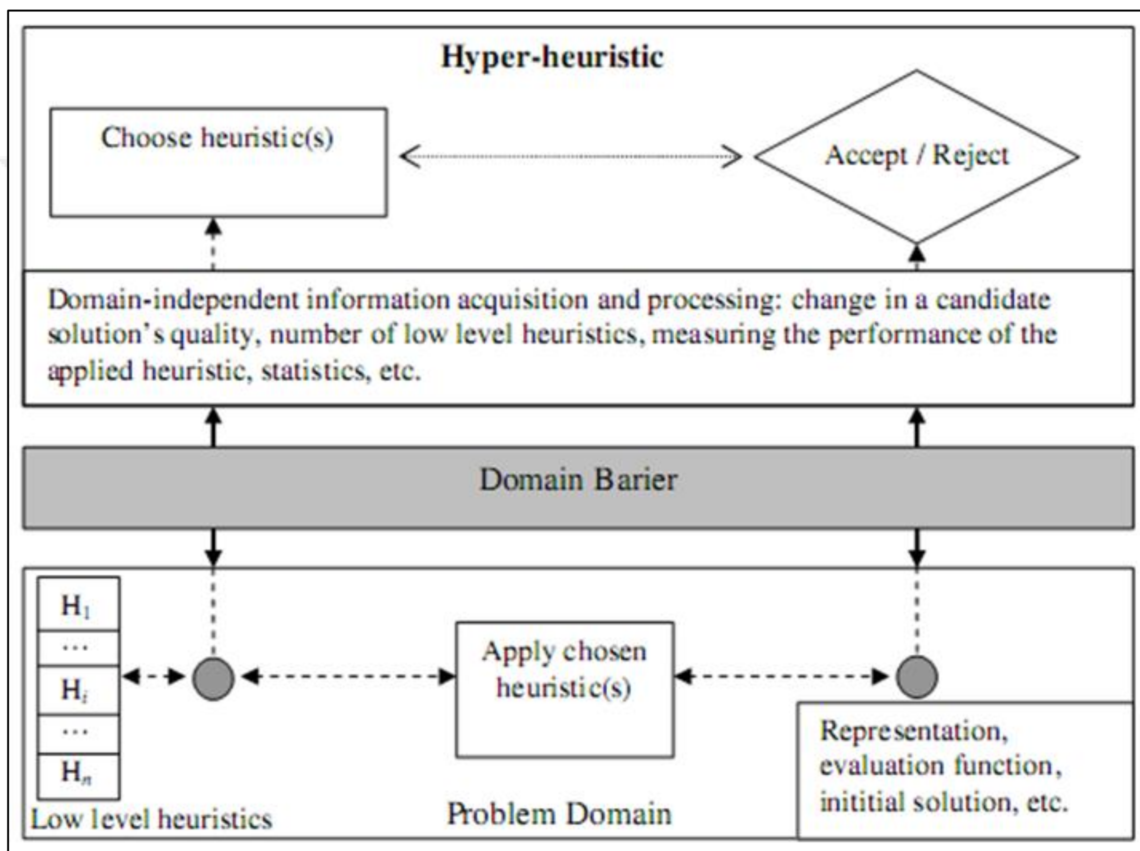



Figure 2.1. The significance of domain barrier in hyper-heuristic design (Figure taken from *Hyper-heuristics: A Survey of the State of the Art* by Burke et al.)

There are more elaborate hyper-heuristics making use of machine learning techniques. Learning within hyper-heuristics takes place in an *online* or *offline* manner. Offline learning hyper-heuristics are employed in a train-and-test fashion, where the feedback from the search process is obtained during the training stage on some sample problem instances. Online learning hyper-heuristics get feedback during the search process for guidance. 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 [19, 20]. 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 a maximum score. Moreover, a hyper-heuristic 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. [21] successfully applies a reinforcement based delayed learning hyper-heuristic on a timetabling problem as well as bin packing. There is a theoretical [22] as well as an empirical evidence [23, 24] that hyper-heuristics 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 hyper-heuristics can adapt and result with high quality solutions [8, 9].

More on hyper-heuristics can be found in [7, 25, 26]. Sharing the SMT processor datapath resources among the threads of a given workload is a challenging task which needs to be addressed in a dynamically changing environment. Moreover, even a small change in a workload, for example, swapping the order of two programs, could lead to a large change in the overall characteristics of the instance and so making the problem even more difficult to handle. In this study, we use the previous work on selection hyper-heuristics as an inspiration to design a set of learning selection hyper-heuristics to mix well-known heuristics for improving the SMT resource distribution.

# 3. PRELIMINARY ANALYSIS

## 3.1. Motivation

This section lays out our motivation through a preliminary analysis over some experiments that we have performed. It is shown in previous studies [4-6] that explicit partitioning of pipeline resources gives promising results in SMT processors, and there is no single best heuristic which performs better than the others in all workloads. Both previous studies and our initial tests confirm that phenomenon.

Our goal in this study is to combine these heuristics via a selection hyper-heuristic in order to capture the strength of each heuristic. By doing this, we aim to achieve a performance level close to the best performance available through these heuristics, for most of the workloads that we use to evaluate our design.

To show that mixing heuristics would indeed enable us to obtain the performance of the better performing heuristic, we run a number of workloads that contain SPEC benchmarks [10], using the same processor configuration as in the tests that we use to evaluate our proposed designs. The tests are run for a time interval where 10 heuristic selections are carried out. Since the limit study is run for three-thread workloads, the duration of the limit study is 30 epochs (see Section 4.1 for Big Epochs). During this interval all possible selection permutations are run, exhaustively. Due to the exponential cost of time needed to run these tests, we did not go beyond 10 decision points for this limit study. Although this time interval is too short to make definitive deductions, the results provided us with some important insights.

### 3.1.1. [bzip2-cactusADM-hmmer]

One of the workloads used in our limit study is [bzip2-cactusADM-hmmer]. Cumulative IPC values for ARPA, HILL and the best performing permutation (BEST) are shown in Figure 3.1. This figure also represents the total number of instructions committed. The best performing permutation performs 0.63 per cent better than ARPA and 3.5 per cent better

than HILL. An improvement of only 0.63 per cent may seem insignificant; however, it must be considered that 1) the duration of limit study is too short to achieve very high performance gains, 2) ARPA does not perform better then HILL in all workloads, therefore the best performing permutation would outperform a processor with a design choice of using HILL by 3.5 per cent, and, most importantly, 3) the results show that combining these heuristics may achieve a performance even beyond what can be achieved by running them in isolation. Moreover, 96 per cent of the permutations perform better than HILL and 11 per cent of the permutations perform better than ARPA.

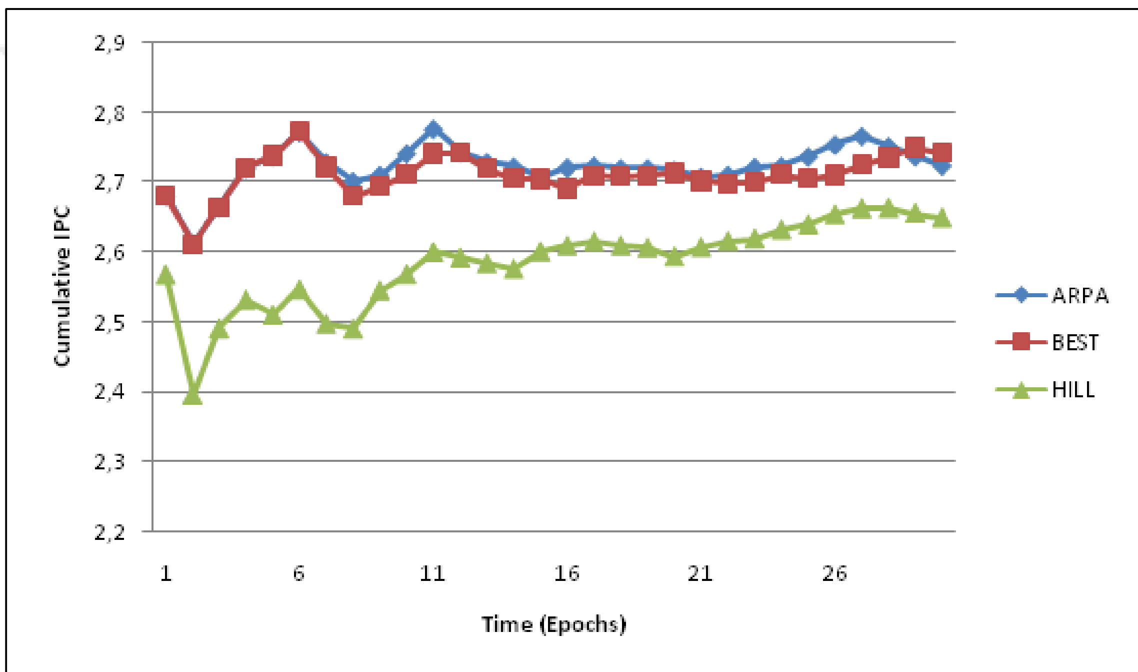



Figure 3.1. Cumulative IPC values for ARPA, BEST and HILL in [bzip2-cactusADM-hmmer]

Figure 3.2 shows how heuristics perform in each epoch. It can be seen that in some epochs, HILL performs better than both ARPA and BEST. This may be misleading in terms of "capturing the strengths of both heuristics", as in even the best performing permutation, BEST, cannot reach the performance level of HILL in some epochs. This would be a wrong assessment since all possible permutations are tested and BEST reached the highest performance level overall among all these permutations, including HILL and ARPA.

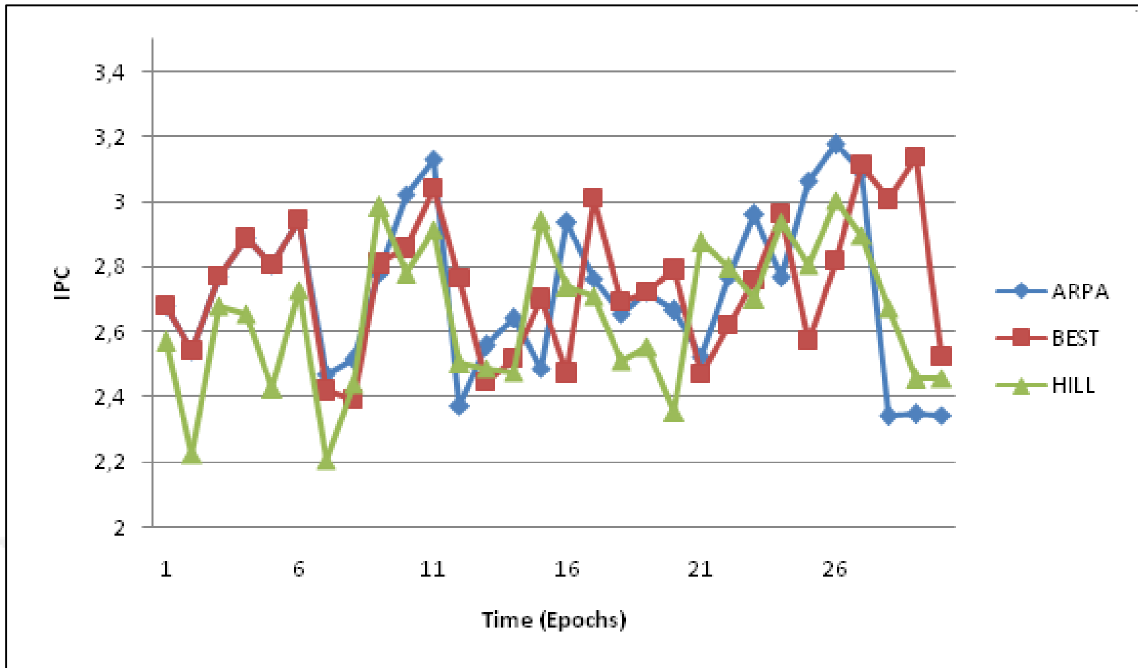



Figure 3.2. Individual epoch IPC values of ARPA, BEST and HILL in [bzip-cactusADM-hmmer]

The reason why HILL outperforms even the best performing permutation in some epochs can be explained as follows: when threads run alone, they run faster; therefore, they reach to the upcoming program phases with different behaviors faster. When threads run together and share resources, they run slower and it takes them longer to reach to these subsequent phases. Figure 3.3 shows the performance of cactusADM when it is run standalone and when it is run in a workload using HILL. In the standalone mode, cactusADM experiences a great drop in performance in epoch four. Since the thread is able to use all resources available in the processor, there are no external factors and the performance drop is caused by cactusADM entering a phase with lower IPC. Meanwhile, in HILL, cactusADM cannot reach that execution phase where it suffers the performance drop in epoch four. This can be confirmed by cactusADM in standalone mode performing better than cactusADM in HILL, and committing more instructions until epoch four (Figure 3.4). Later on, cactusADM in HILL experiences a similar performance drop in epoch seven. In epoch four, cactusADM in HILL is still in the relatively fast execution phase, and cactusADM in standalone mode has entered a slow execution phase. Thus, in epoch four, HILL can achieve a level of performance higher than the standalone mode.

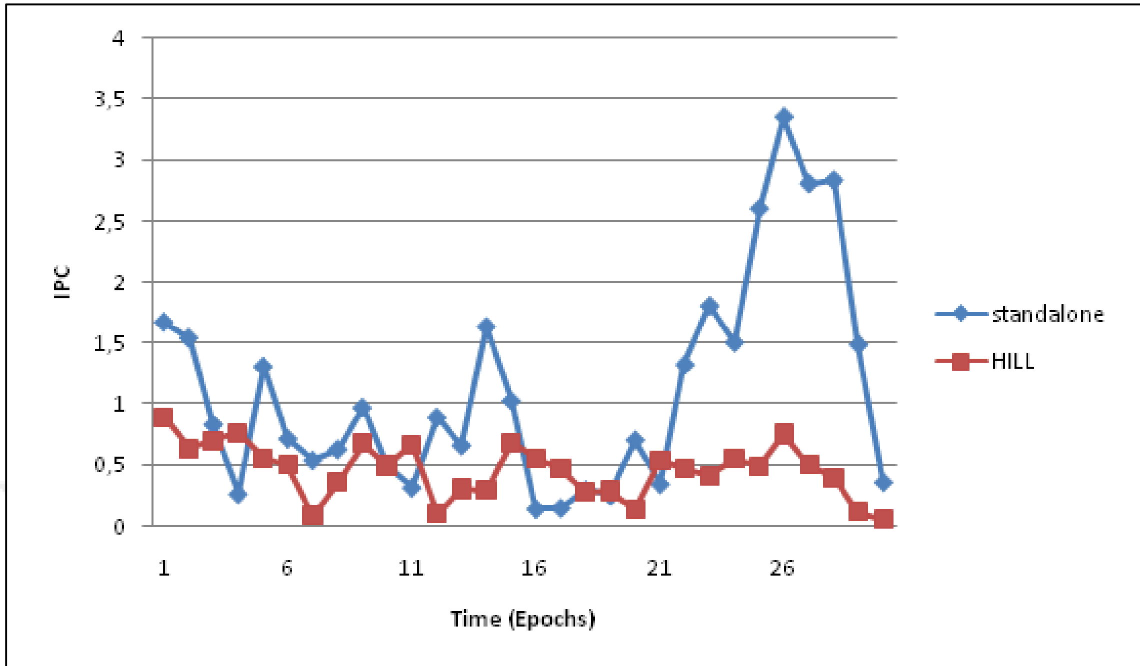



Figure 3.3. Individual IPC values of cactusADM when run in standalone mode and in a mixture with HILL

In Figure 3.3, it can be observed that the IPC of cactusADM oscillates greatly in time intervals of two-three epochs, even in standalone mode where the performance drops are not caused by sharing resources with co-running threads. This shows how fast the execution phases of threads can change. This also explains the anomaly of HILL outperforming BEST in some epochs.

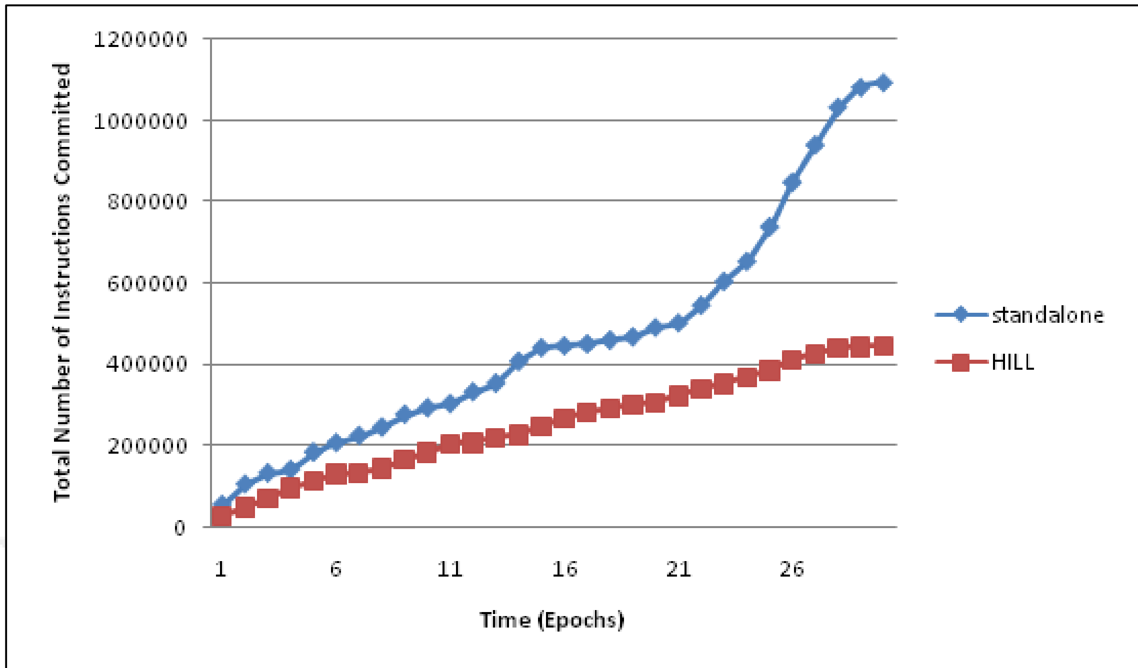



Figure 3.4. Total number of instructions committed by cactusADM when run in standalone mode and in a mixture with HILL

### 3.1.2. [lbm-milc-gobmk]

Second workload that is tested in our limit study is [lbm-milc-gobmk]. The performance results of ARPA, HILL and BEST per epoch are given in Figure 3.5. BEST outperforms HILL by 3.96 per cent and ARPA by 5.02 per cent. If we bring the performance gains of BEST in these two workloads together, it means that if there were a hyper-heuristic which could find the best performing permutation, it would outperform HILL by 3.5 per cent and 3.96 per cent; and ARPA by 0.63 per cent and 5.02 per cent. The importance of performing at least at a similar level, if not better, to the better performing heuristic becomes more obvious when the performance gains in different workloads are considered together. Moreover, in [lbm-milc-gobmk], around six per cent and 26 per cent of the tested permutations perform better than HILL and ARPA, respectively.

When the performances of threads in [lbm-milc-gobmk] are examined individually, it can be seen that gobmk experiences three phases with different behaviors. First, gobmk starts with a slow phase where extra resources do not add much to its performance. Then, gobmk

enters a phase where it can increase its performance with extra resources. Finally, it enters a phase where it can run extremely fast.

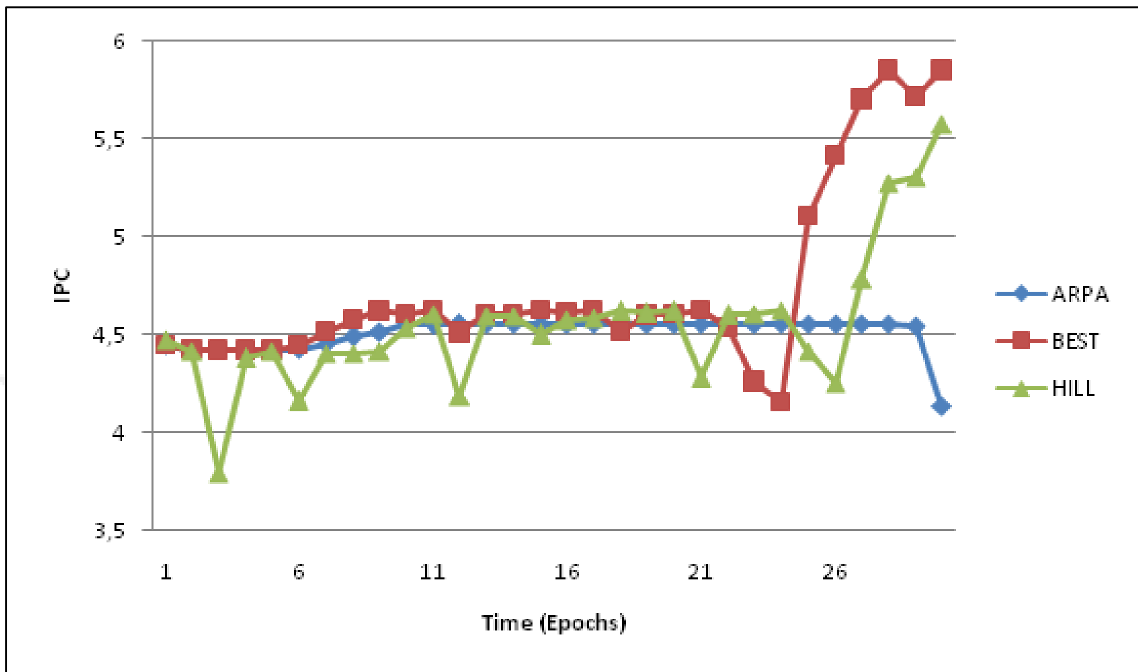



Figure 3.5. Individual epoch IPC values of ARPA, BEST and HILL in [lbm-milc-gobmk]

In the beginning, ARPA starts giving resources to milc, since it is the most efficient thread in terms of CIPRE metric. While resources are being shifted towards milc, it increases its performance, greatly. Meanwhile, the other threads experience great performance degradations. IPC values for threads in ARPA, HILL and BEST per epoch can be seen in Figures 3.6-3.8. In epoch six, it can be seen that the performance of gobmk slightly increases in ARPA. Since gobmk does not get any extra resources, this is due to gobmk entering its second phase, which is a relatively faster phase compared to the first one. However, since milc still has a higher efficiency value, gobmk still does not get any extra resources, and cannot improve its performance. In the end, in ARPA, gobmk does not commit enough instructions to reach to its third phase within the given amount of time.

What ARPA fails detecting is whether a thread can make good use of extra resources or not. ARPA makes decisions only based on a thread performance in the current distribution. If a thread is not the most efficient one already, ARPA does not give any extra resources to

it, and cannot determine if that thread can make use of these extra resources or not. This issue is also discussed in detail in [6].

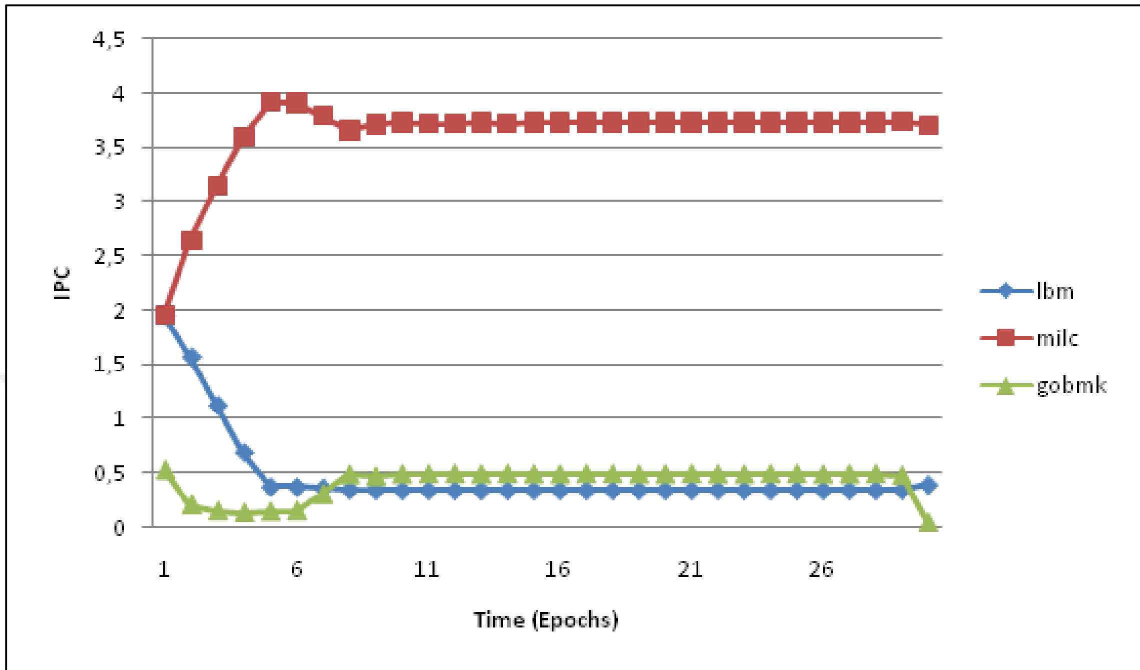



Figure 3.6. Performance result of individual threads in ARPA in [lbm-milc-gobmk]

In general, the efficiency of a thread is expected to increase when the amount of resources allocated to it is decreased, since ILP available in threads usually do not allow IPC to increase in the same rate with resources. However, when these resources available to a thread get low, the efficiency may drop even further. When the number of resource entries decrease, the window to search for independent instructions in order to exploit ILP gets smaller, converging to an in-order processor performance. In such cases, even only a few long latency instructions may stall a thread and degrade its performance. This is the case in gobmk in the first few epochs of ARPA. When the amount of resources allocated to gobmk decrease, the performance decreases even more. Hence, efficiency of gobmk decreases, which prevents it from getting any extra resources.

HILL, on the other hand, runs trial periods where each thread gets its fair chance with extra resources, which allows HILL to determine whether threads can increase their performance when they are provided with extra resources or not. However, periodically allowing each

thread to use extra resources comes with a cost. It can be seen from Figure 3.5 that HILL experiences performance drops in epochs divisible to three, which are the epochs where gobmk gets extra resources. Since HILL favors gobmk by giving extra resources even when gobmk cannot improve its performance significantly, throughput decreases in these epochs. This exposes another weakness of HILL: it spends most of its time in non-optimal distributions (according to its own evaluation). This may not be an issue when the performance improvement in the favored thread can make up for the performance loss in other threads (such as when resources are shifted among milc and lbm, in this example), but in some cases this may prove to be wasteful.

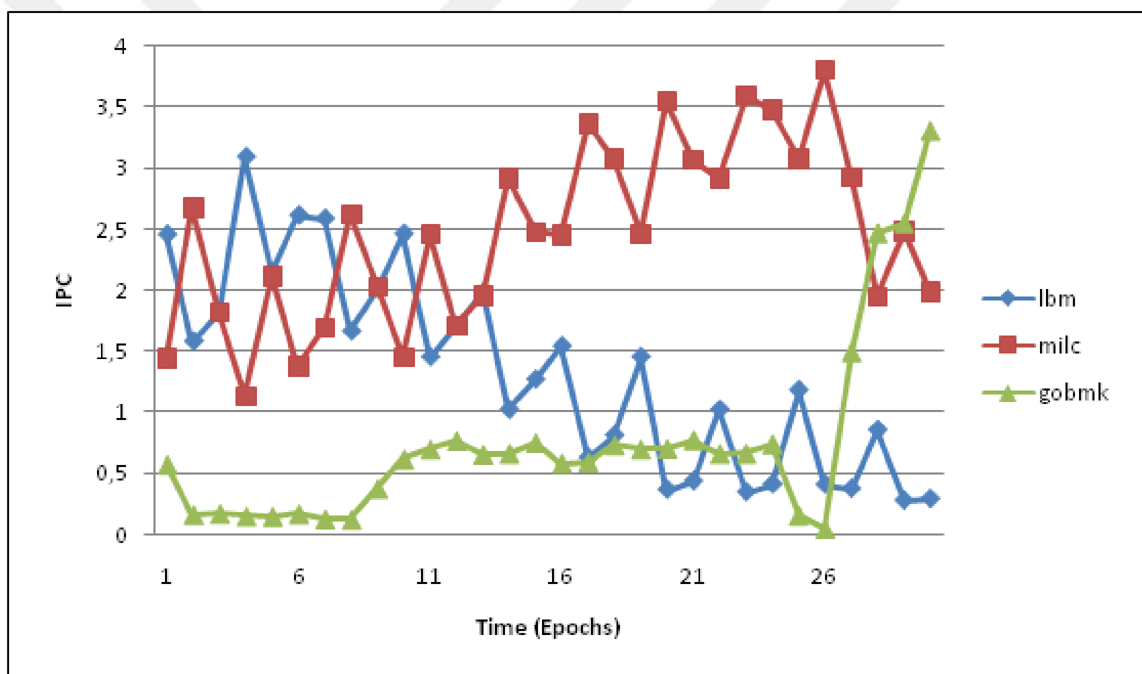



Figure 3.7. Performance results of individual threads in HILL in [lbm-milc-gobmk]

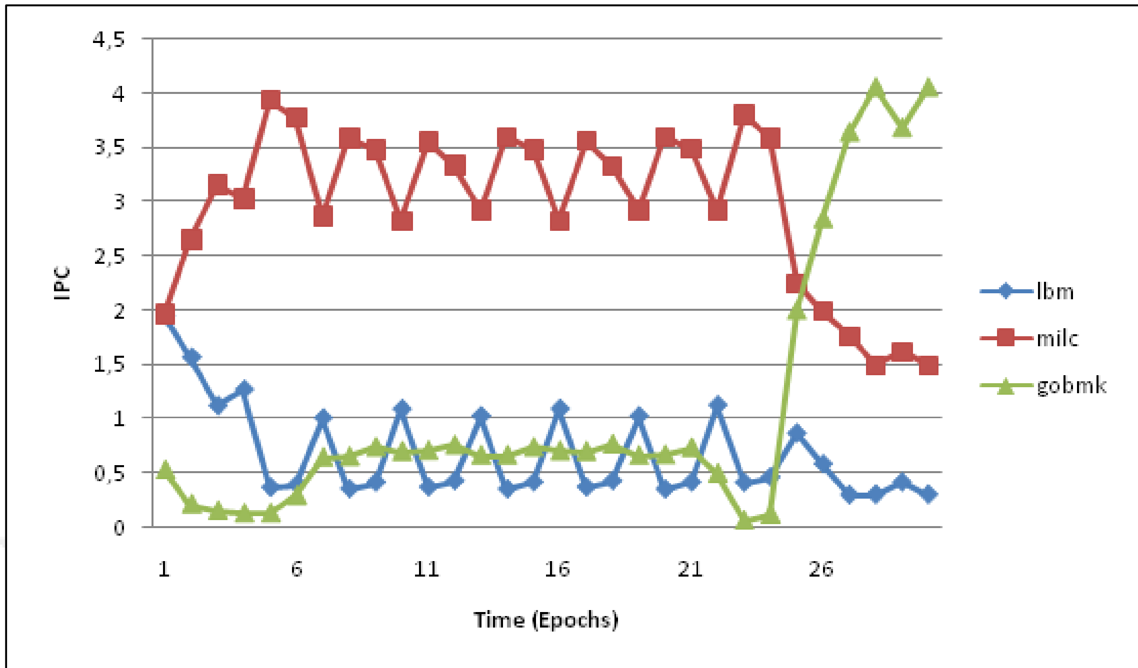



Figure 3.8. Performance results of individual threads in BEST in [lbm-milc-gobmk]

### 3.1.3. [art-mcf-mgrid]

[art-mcf-mgrid] is another workload that is studied. In this workload, BEST scored an IPC value 2.4 per cent higher than HILL and 10 per cent higher than ARPA. The performance results of ARPA, BEST and HILL per epoch are given in Figure 3.9. 24 per cent and 53 per cent of permutations outperform HILL and ARPA, respectively.

Another interesting observation that can be done from this limit study is the importance of the initial distribution decisions. In [bzip2-cactusADM-hmmer], among 109 permutations managed to perform with a higher IPC than ARPA, all 109 permutations started with running ARPA as the first heuristic. In [lbm-milc-gobmk], 63 permutations outperformed HILL and 61 of them started with running ARPA as the first heuristic. Finally, in [art-mcf-mgrid], 243 permutations outperformed HILL and all 243 started with running HILL as the first heuristic.

Figures 3.10-3.12 show the performance geography of all 1024 permutations in these three workloads. The horizontal axis represents the list of heuristics run as the first five

consecutive heuristics, depth axis represents the list of heuristics run as the five consecutive heuristics and the vertical axis represents the performance. It can be clearly seen that there are regular oscillations along the horizontal axis, implying the importance of the first heuristic to be run. If cells on the horizontal axis are assumed to be in pairs; these pairs would represent the exact same heuristic list, except for the first heuristic to be run. The obvious performance difference between such pairs shows that the initial distribution decision is of utmost importance, even when the permutations which performed below the better performing heuristic are considered.

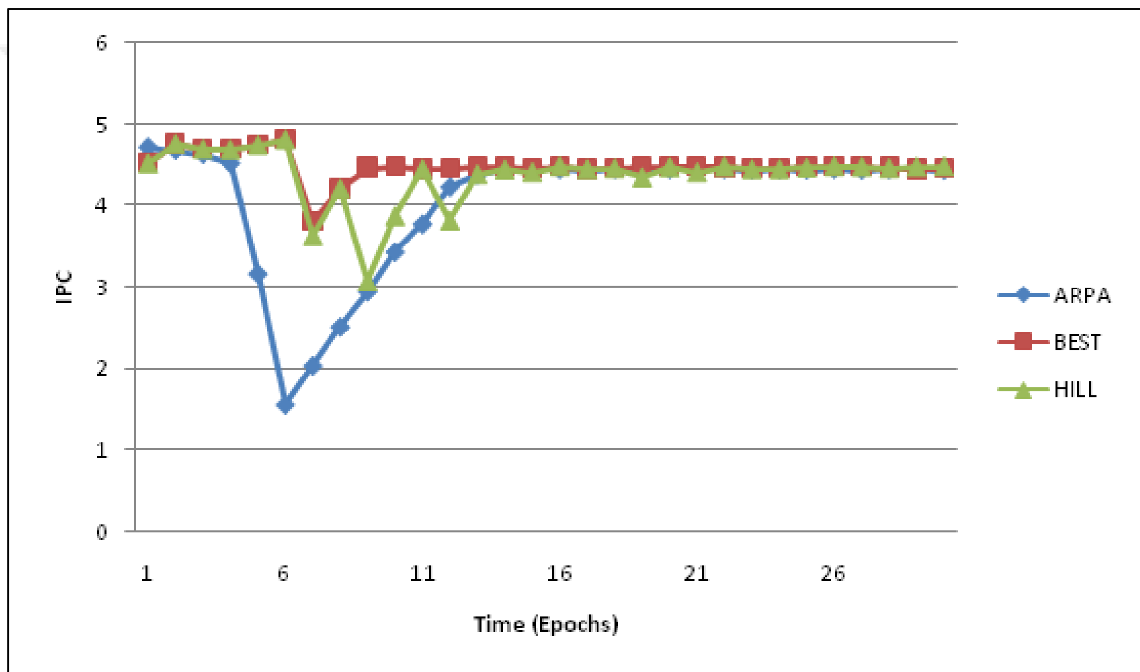



Figure 3.9. Individual epoch IPC values of ARPA, BEST and HILL in [art-mcf-mgrid]

Figures 3.10-3.12 provide us with the information of probability of success. In [bzip2-cactusADM-hmmer], 980 out of 1024 permutations and 109 out of 1024 permutations manage to outperform HILL and ARPA, respectively. 63 permutations outperform HILL and 265 permutations outperform ARPA in [lbm-milc-gobmk]. [art-mcf-mgrid] yields the most promising results: 243 permutations perform better than HILL and 543 permutations perform better than ARPA. These numbers show the probability of performing in a higher level than the current heuristics *without* any careful analysis or rational decision making.

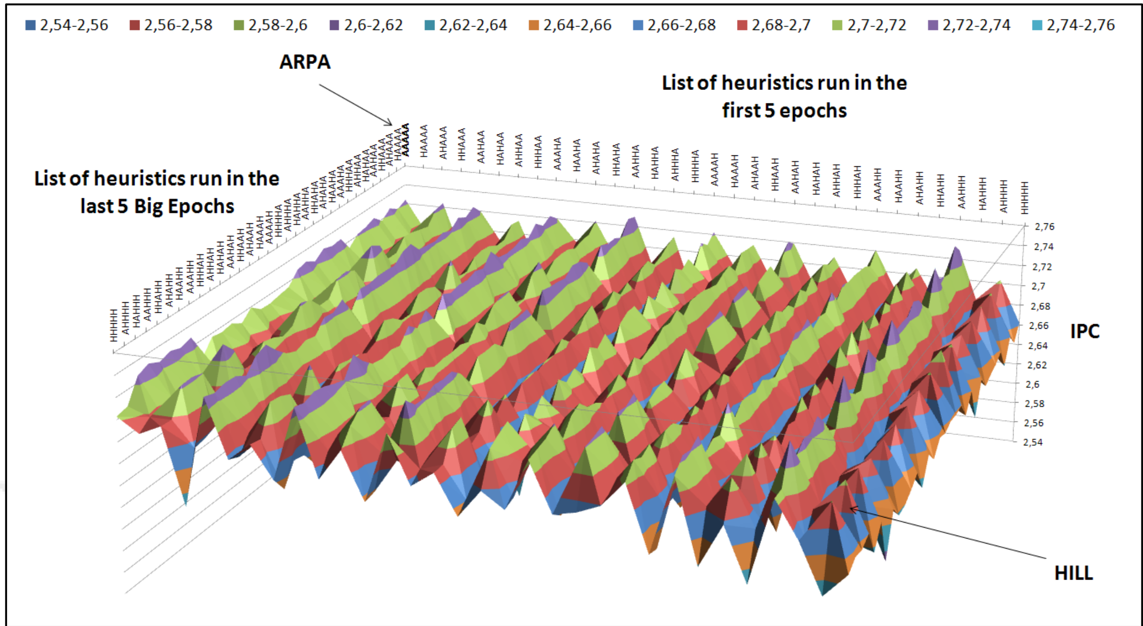



Figure 3.10. IPC values of all 1024 possible permutations in workload [bzip2-cactusADM-hmmer]

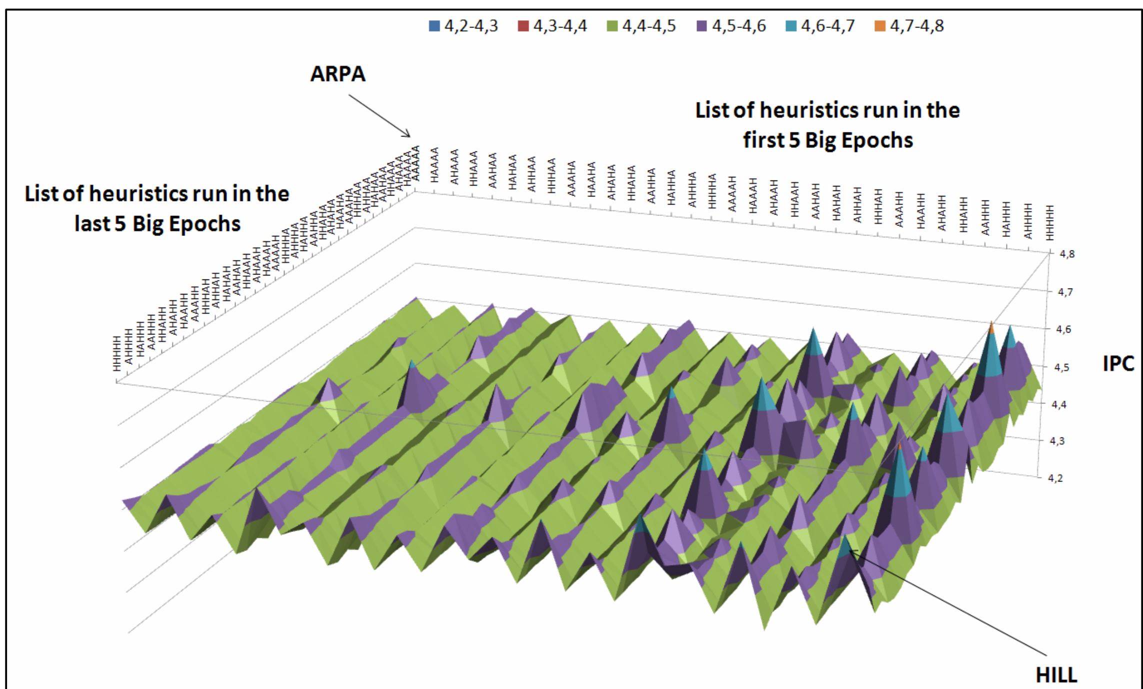



Figure 3.11. IPC values of all 1024 possible permutations in workload [lbm-milc-gobmk]

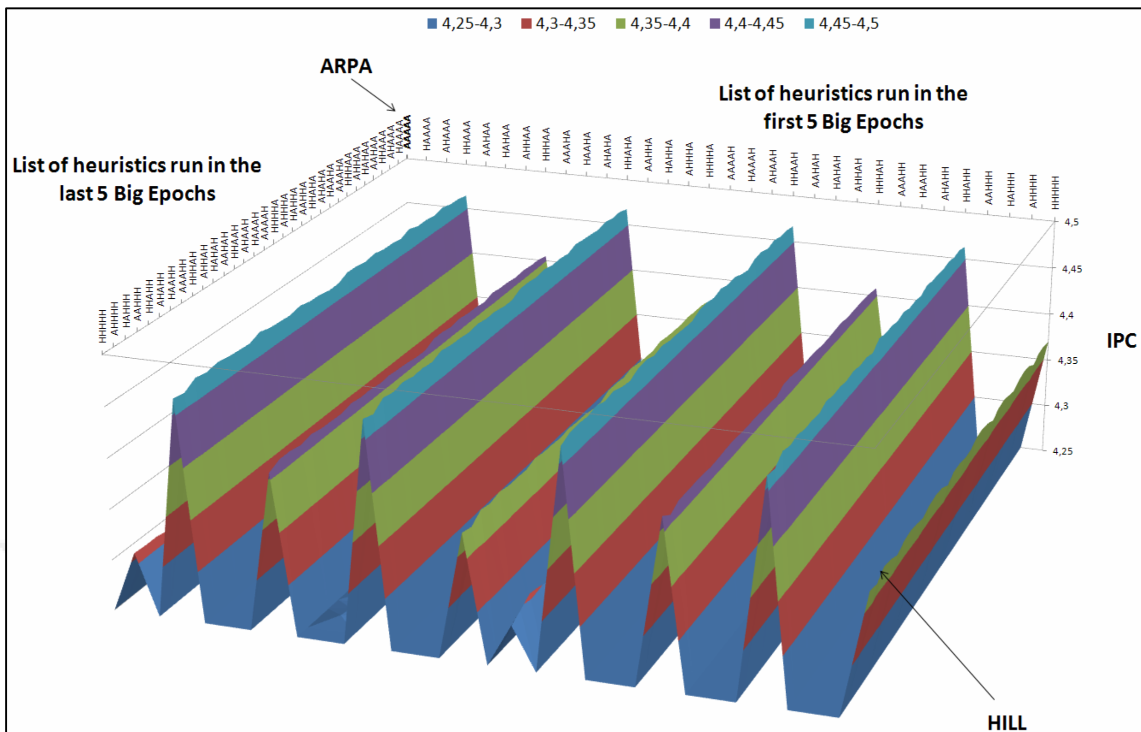



Figure 3.12. IPC values of all 1024 possible permutations in workload [art-mcf-mgrid]

### 3.1.4. Long Run

In order to gain a better insight on the potential performance improvement available by utilizing hyper-heuristics, another test, lasting approximately 13M cycles (100 Big Epochs) on the workload [gobmk-lbm-mcf-sjeng], was carried out. However, due to exponential time cost, not all possible permutations were tested. Instead, the simulation is divided into time periods of 10 Big Epochs. For each period, brute force search is applied for 1024 possible permutations. Each new period is tested on top of the list of best performing permutations found that far. Figure 3.13 shows the cumulative performance values of ARPA, HILL and the best performing permutation found (LIMIT) during the simulation. In the end, LIMIT outperforms HILL by one per cent and ARPA by 5.4 per cent.

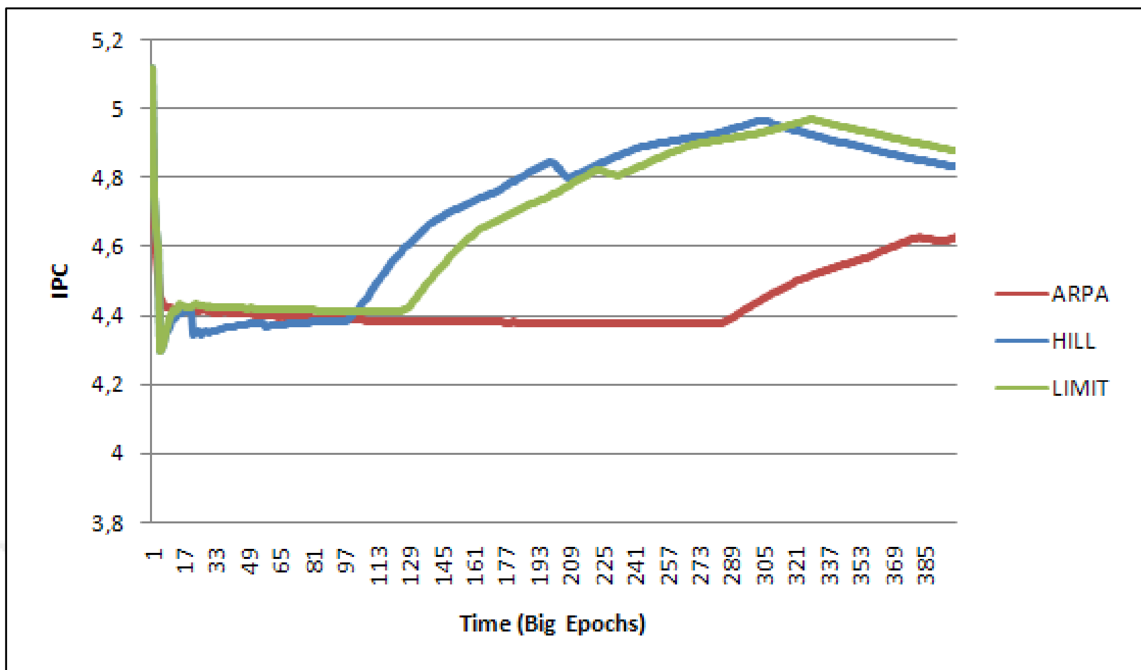



Figure 3.13. Cumulative performances of ARPA, HILL and LIMIT in [gobmk-lbm-mcf-sjeng]

# 4. PROPOSED DESIGN

The initial design starts by creating a habitat that may interchangeably run both OARPA and OHILL. ARPA and HILL heuristics are our faithfully implemented versions of these heuristics on hardware. 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 if the CIPRE value of a thread is greater than the others. Therefore, as we emphasize the details in the later sections, the resulting circuitry that runs both heuristics is much less complex than what one may expect.

As shown in Figure 4.1, our proposed design brings ARPA and HILL 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 evaluation functions or heuristics (and the hyper-heuristic). 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.

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

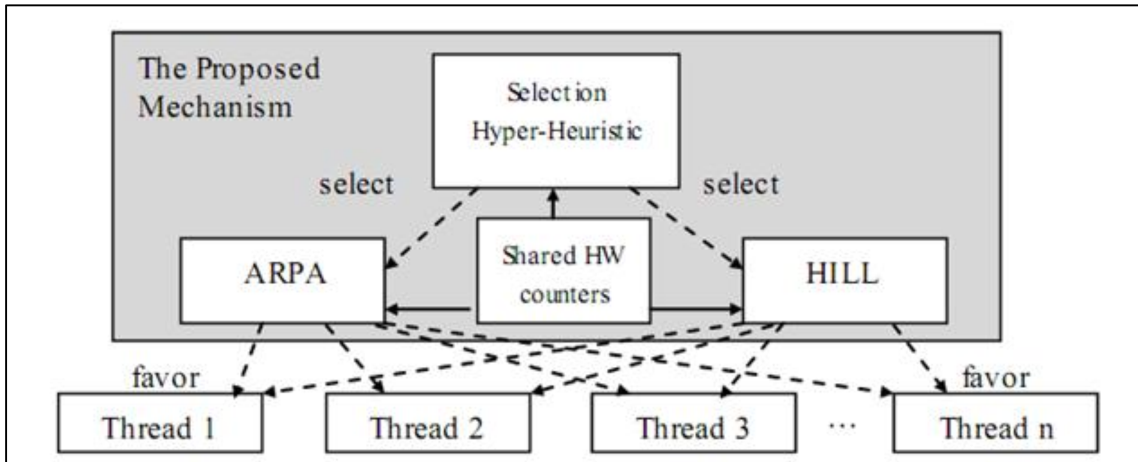



Figure 4.1. The proposed design

## 4.1. Combining the Heuristics

Our proposed hyper-heuristic needs to be able to run both heuristics, interchangeably. However, there are substantial differences 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 changing its behavior between the two trial periods, which leads HILL into comparing trial performances of two different program phases and renders it to be a totally different heuristic. This can be observed better in the timeline given in Figure 4.2. In this example lets assume that HILL is run in the first epoch, and HILL runs its first trial round. Then, the hyper-heuristic 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 zero and eight, which are quite far away from each other, causing inaccurate evaluations that OHILL 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.

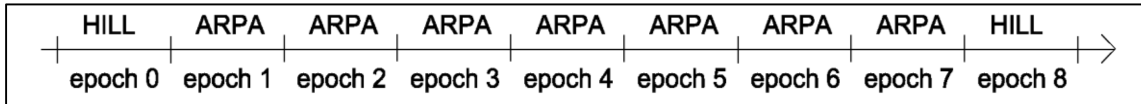



Figure 4.2. An example timeline with heuristics running in arbitrary order

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 4.3. Therefore, the hyper-heuristic makes decisions only at the beginning of Big Epochs. To provide the hyper-heuristic with accurate data on how heuristic performs, all evaluations are done using performance values of heuristic in Big Epochs; although, the heuristics still make their decisions in the traditional epoch granularity.

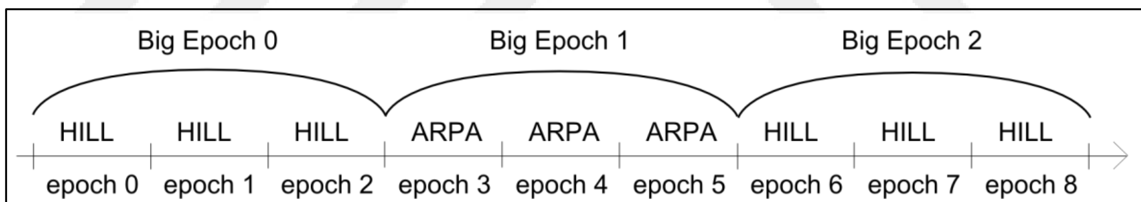



Figure 4.3. An example timeline where the hyper-heuristic decides HILL, ARPA and HILL should be run on an SMT processor with three threads

## 4.2. Implementation Details of Heuristics

### 4.2.1. Implementation Details of HILL

OHILL partitions integer IQ, integer rename registers, and ROB. These resources are partitioned, proportionately. The authors state that by partitioning these resources, they indirectly control how much floating point resources each thread uses. Hence, OHILL does not explicitly partition floating point resources. In our implementation, HILL partitions IQ, ROB, and Load/Store Queue (LSQ), proportionately.

The original study does not refer to a lower limit of resources a thread can have, which may cause resource starvation. We implement a lower limit to the resources, which prevents from any thread having less than a certain amount of resources. This lower limit is equal to the amount of resources taken away from each thread in trial periods of HILL.

The authors of the original study propose running the algorithm in software level. With a pessimistic approach taking context switches also into account, the original study applies a 200-cycle penalty per epoch when evaluating the work. Our implementation assumes all computations are done in hardware level, and, therefore, applies no such penalty. However, since out work is compared against our implementation, HILL, both algorithms are penalty-free and thus the comparison is to be fair.

The optimization goal of OHILL can be altered by changing the feedback metric. In the original study, OHILL is optimized for three different metrics: average IPC, average weighted IPC, and harmonic mean of weighted IPC. In order for OHILL to be able to compute these values, it must know the IPC of threads when they are run standalone in the processor (single_IPC). To gather the single_IPC value at runtime, OHILL runs one thread alone every 40 epochs. This sampling cost is taken into account in the results published in the original work. Since we are only interested in optimizing throughput at this point, we do not do any sampling, and, therefore, do not incur any sampling costs in our results; neither when HILL runs alone or as a part of the hyper-heuristic.

### 4.2.2. Implementation Details of ARPA

OARPA partitions Instruction Fetch Queue (IFQ), IQ, and ROB. The original study states that partitioning the ROB enables control over rename registers; therefore the original implementation does not explicitly partition the register file. Furthermore, IFQ and ROB are not partitioned separately; instead an upper limit to how many entries a thread can use from these resources in total is applied. IQ is partitioned, proportionately. For compatibility purposes, our implementation partitions LSQ, IQ, and ROB proportionately and independently, as HILL does.

The CIPRE value of each thread is calculated by dividing the number of committed instructions in the current epoch to the number of resource entries allocated to each thread. In the original implementation, in harmony with how IFQ and ROB are distributed together, the divisor (number of resources allocated) is taken as the upper bound of IFQ and ROB entries a thread can use in total. Since our implementation does not partition IFQ, we use the total number of IQ and ROB entries allocated to each thread when computing CIPRE values.

OARPA applies a lower bound to the number of resources allocated to each thread, to prevent resource starvation. This lower limit is 25 per cent of the initial amount that is allocated [6]. For compatibility with HILL; ARPA applies a lower limit equal to the amount shifted away from each thread at decision making points.
The original study also suggests implementing the algorithm in software level as in OHILL, and applies the same performance penalties. However, we assume the algorithm is implemented in hardware level and apply no penalties, as we do not apply any penalties to HILL. Thus, comparisons between HILL, ARPA, and our proposed hyper-heuristic would be fair in this manner.

### 4.2.3. Parameters

There are two common (and also critical) parameters for both ARPA and HILL: *epoch size* and *delta*. The *epoch size* is the length of and epoch in cycles. Both studies agree that the *epoch size* is an important parameter. A very small *epoch size* may lead to a very dynamic

inter-epoch behavior and cause the heuristic to make faulty and unstable decisions. On the other hand, a very large *epoch size* would make the heuristics unresponsive and prevent them from reacting quickly to changing needs of the workloads.

The *delta* parameter determines how many resource entries is to be taken from or given to threads. A very small *delta* value would cause the heuristics to act slowly, increasing the time needed to reach the optimal distribution. A very large *delta* value, on the other hand, may turn heuristics into coarse-grain solutions and cause them to miss some optimal distributions. In our design, this value also determines the minimum number of resource entries that can be allocated to each thread.

## 4.3. Proposed Hyper-Heuristics

In this study, we propose a variety of hyper-heuristics ranging from the simplest random-gradient-based (HH1) hyper-heuristic to the more complex reinforcement learning-based (HH4) one.

### 4.3.1. HH1: IPC-based Hyper-Heuristic

Our first hyper-heuristic is based on the *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 immediately conclude that something is not going right in terms of performance. In such cases, HH1 punishes the heuristic that was being used in the previous epoch and changes it with an alternative heuristic for the incoming epoch. In our study, we only utilize two heuristics (ARPA and HILL), and, hence, we choose the alternative heuristic in such cases. Algorithm 4.1 shows the pseudocode for this hyper-heuristic.

Algorithm 4.1. The pseudocode of HH1

```
if IPCBigEpoch_i >= IPCBigEpoch_i-1 then
    Use the current heuristic in the next epoch
else
    Change the heuristic
end if
```

### 4.3.2. HH2: Commit-over-fetch-based Hyper-Heuristic

The second hyper-heuristic is based on a different metric which we call *commit over fetch*, as shown in Algorithm 4.2. 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 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 a mispredicted path are all flushed. Here, in this metric, we measure if the number of flushed (or wasteful) instructions is increasing. When this happens, HH2 punishes the previously utilized heuristic by selecting the alternative one.

Algorithm 4.2. The pseudocode of HH2

```
CommitOverFetch ← IPCBigEpoch_i / FIPCBigEpoch_i

if CommitOverFetch_i >= CommitOverFetch_i-1 then
    Use the current heuristic in the next epoch
else
    Change the heuristic
end if
```

### 4.3.3. HH3: IPC-and-Commit-over-fetch-based Hyper-Heuristic

In this third algorithm, we propose a slightly more 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 directly related to the performance of the running heuristic. Secondly, in our study, we observed that the overall performance may radically drop within a number of epochs. To stabilize our algorithm further, we give one more chance to a running heuristic only if it is ARPA, even when the drop in $IPCepoch_i$ is below a threshold value. In our experiments, we found that ARPA is a more successful heuristic compared to HILL, and this is to ensure not to punish a well-performing heuristic at its first fault. 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 pseudocode of the algorithm is given in Algorithm 4.3.

Algorithm 4.3. The pseudocode of HH3

```
if IPCBigEpoch_i / IPCBigEpoch_i-1 >= thresholdValue then
    Keep the current heuristic running for the next epoch
    oneMoreChance ← 0
else
    oneMoreChance ← oneMoreChance + 1
    if oneMoreChance is 1 and the current heuristic is ARPA then
        Give one more chance to the current heuristic
    else
        CommitOverFetch_i ← IPCBigEpoch_i / FIPCBigEpoch_i
        if CommitOverFetch_i >= CommitOverFetch_i-1 then
            Keep the current heuristic running for the next epoch
        else
            Change the heuristic
            oneMoreChance ← 0
        end if
    end if
end if
```

### 4.3.4. HH4: Round-based Hyper-Heuristic

This hyper-heuristic tries to learn the success rate of each heuristic by keeping a running score for each of them. The score is set according to the performance improvement or degradation achieved by the running thread at the end of each Big Epoch. We also introduce a round-based scoring, and we reset heuristic scores every *Round* in HH4. This approach guarantees in the hyper-heuristic a degree of stability. Additionally, by altering the Round length in the algorithm, a limit to the history of successes of each heuristic can be set. For example, a Round length of 20 Big Epochs indicates that HH4 is not to take

successes beyond 20 Big Epochs into account. At the beginning of each round, where scores are reset, the heuristic which completes the previous round with the highest score is selected. The pseudocode of HH4 is given in Algorithm 4.4.

Algorithm 4.4. The pseudocode of HH4

```
if it is the beginning of a Round then
    if ARPA_score >= HILL_score then
        last_winner ← ARPA
    else
        last_winner ← HILL
    end if
    ARPA_score ← 0
    HILL_score ← 0
    Use last_winner as the next heuristic
else
    if current heuristic is ARPA then
        ARPA_score ← ARPA_score + (IPCBigEpoch_i - IPCBigEpoch_i-1)
    else
        HILL_score ← HILL_score + (IPCBigEpoch_i - IPCBigEpoch_i-1)
    end if
    if ARPA_score >= HILL_score then
        Use ARPA
    else
        Use HILL
    end if
end if
```

## 4.4. Hardware Complexity

In order to implement a hyper-heuristic utilizing predefined heuristics, the circuitry needed for running the heuristics and the hyper-heuristic must be integrated into the hardware. Since both the heuristics and the hyper-heuristics do similar computations to a degree, some elements can be implemented once and can be dynamically dedicated to the algorithm which really needs them.

Both ARPA and HILL need per-thread counters for counting the number of committed instructions within an epoch, which are already utilized in most of the contemporary processors. These counters do not need to be duplicated since the input gathered by these counters can be directed to HILL, to ARPA and to the hyper-heuristic at the same time. Some of our proposed hyper-heuristics need to know the number of instructions fetched each epoch per-thread. Similar counters can be implemented for feeding this information to the hyper-heuristic.

Both heuristics need comparators. ARPA needs them for comparing the efficiency (CIPRE) of the threads whereas HILL needs them to compare the actual performance values (number of committed instructions in our case since we optimize HILL only for throughput). Since ARPA and HILL are not planned to be run simultaneously, not only these comparators can be shared among ARPA and HILL but also they can be used for comparing the current epoch's performance to the previous one by the hyper-heuristic. There is no need for replicating these resources.

In explicit partitioning of resources, control logic must be implemented to prevent threads from using more resources than they are allowed to. This can be handled by allocating two registers per thread, per resource; one to keep the maximum number of entries the thread is allowed to use, and one to keep the number of entries the thread is currently using. If the number of currently used resource entries for a thread exceeds its limit, fetching or dispatching is blocked until either the thread frees some resources, or its limit is increased. In addition, a mechanism to prevent threads from losing resources beyond a lower bound must be implemented. This control logic is needed for any kind of explicit resource partitioning mechanism, and our design do not cause any additional complexity over a rival approach for that reason.

HILL needs approximately 1600 transistors, except for the resource control logic if implemented according to our experimental setup (See Sections 5.1 and 5.3). HILL needs four commit counters for keeping track of each thread's progress in a four-threaded processor. The processor is eight-way, which means a thread can commit a maximum of eight instructions per cycle. Considering that the epoch size is 32768 cycles, the maximum number of instructions that can be committed by a single thread can be kept in an 18-bit counter. HILL also needs comparators to find out in which epoch the processor performed best. Assuming that all comparisons are done in parallel, HILL needs three comparators, each being 18-bit wide. The summary of hardware complexity of HILL in terms of transistor number can be seen in Table 4.1.

Table 4.1. Transistor cost of HILL

| Unit type | Number needed | Bits per unit | Number of transistors |
|---|---|---|---|
| Commit counters | 4 | 18-bits | 432 |
| Comparators | 3 | 18-bits | 1134 |
| | | | 1566 |

ARPA, in addition to HILL, needs four division units for the computation of CIPRE value. CIPRE is computed by dividing the number of committed instructions to the total number of IQ and ROB entries allocated to the thread. Since the number of committed instructions is almost always much greater than total number of entries allocated, division units must be aligned with respect to the number of committed instructions; which means division units must be 18-bits wide. ARPA needs roughly 5500 transistors for implementation except for the resource control logic. Details of this transistor cost can be seen in Table 4.2.

Table 4.2. Transistor cost of ARPA

| Unit type | Number needed | Bits per unit | Number of transistors |
|---|---|---|---|
| Commit Counters | 4 | 18-bits | 432 |
| Comparators | 3 | 18-bits | 1134 |
| Division Units | 4 | 18-bits | 3888 |
| | | | 5454 |

Our proposed hyper-heuristic design requires hardware support for both its own decision mechanism, and for the control mechanism for running heuristics based on those decisions. Hyper-heuristics keep track of thread performances in terms of Big Epochs; therefore, the size of counters needed for hyper-heuristics are greater. In an eight-way processor where the epoch size is 32768 cycles; each thread can commit a maximum of $2^{18}$ instructions. The duration of Big Epochs is equal to the number of threads running in parallel. In the case of four threads running simultaneously, the size of commit counters should be 20-bits. Also, 20-bit adders are needed to add the commit values of threads in sequential epochs, in order

to compute IPCBigEpochs. Same amount of logic units are also needed for keeping track of total number of fetched units and FIPCBigEpochs.

The hardware cost caused by the decision mechanism varies to the type of hyper-heuristic utilized. In the following approximation; the most costly hyper-heuristic for each type of logic unit is considered. In terms of comparators, HH3 is the most costly hyper-heuristic: it needs two comparators; one for comparing the rate of IPCBigEpochs to the threshold value, and one for comparing CommitOverFetch values. 20-bits are sufficient for each comparator since both comparisons are done between 20-bit values. HH3 also has the most complex logic in terms of division. Two division units are needed for HH3: one for dividing IPCBigEpochs and one for computing CommitOverFetch value. Both division units should be 20-bits. HH4 needs two score registers; for HILL and ARPA. It also needs two subtractors for computing the IPPCBigEpoch difference between last two Big Epochs, and two adders for adding this difference to the appropriate score register. The maximum amount of instructions that can be committed in a single BigEpoch is $2^{20}$. Since our design uses five rounds as Round Duration in HH4 (see Section 6.1.); the size of these score registers and adders should be 23-bits, whereas 20-bits are sufficient for subtractors. The summary of hardware cost in terms of transistors is given in Table 4.3.

The calculation in Table 4.3 is a rough approximation. Considering the extra latches, selection logic, and other overlooked costs; we triple the hardware cost and round it up to 40,000 transistors. One of the most up-to-date processors, 15-Core Xeon Ivy Bridge-Ex (which will be released in 2014) contains 4.31 billion transistors, which means there are approximately 287 million transistors per core. Our hyper-heuristic design amount to 0.01 per cent of a single Xeon Ivy Bridge-Ex core, which can be easily ignored.

Figure 4.4 shows a more detailed design scheme. Green units are required for both HILL and ARPA. Yellow units are required only by ARPA. These units are also utilized in hyper-heuristic design since hyper-heuristics need to be able to run both HILL and ARPA. Red units in the figure are only required by hyper-heuristics. The center of control is the Hyper-heuristic Control Logic; which includes hardware for control signals, selection, score registers, and additional arithmetic operations.

Table 4.3. Transistor cost of hyper-heuristic design

| Unit type | Number needed | Bits per unit | Number of transistors |
|---|---|---|---|
| Commit Counters | 4 | 20-bits | 480 |
| Commit Adders | 4 | 20-bits | 2240 |
| Fetch Counters | 4 | 20-bits | 480 |
| Fetch Adders | 4 | 20-bits | 2240 |
| Comparators (HH3) | 2 | 20-bits | 840 |
| Division Units (HH3) | 2 | 20-bits | 2160 |
| Registers (HH4) | 2 | 23-bits | 1656 |
| Subtractors (HH4) | 2 | 20-bits | 1120 |
| Adders (HH4) | 2 | 23-bits | 1288 |
| | | | 12504 |

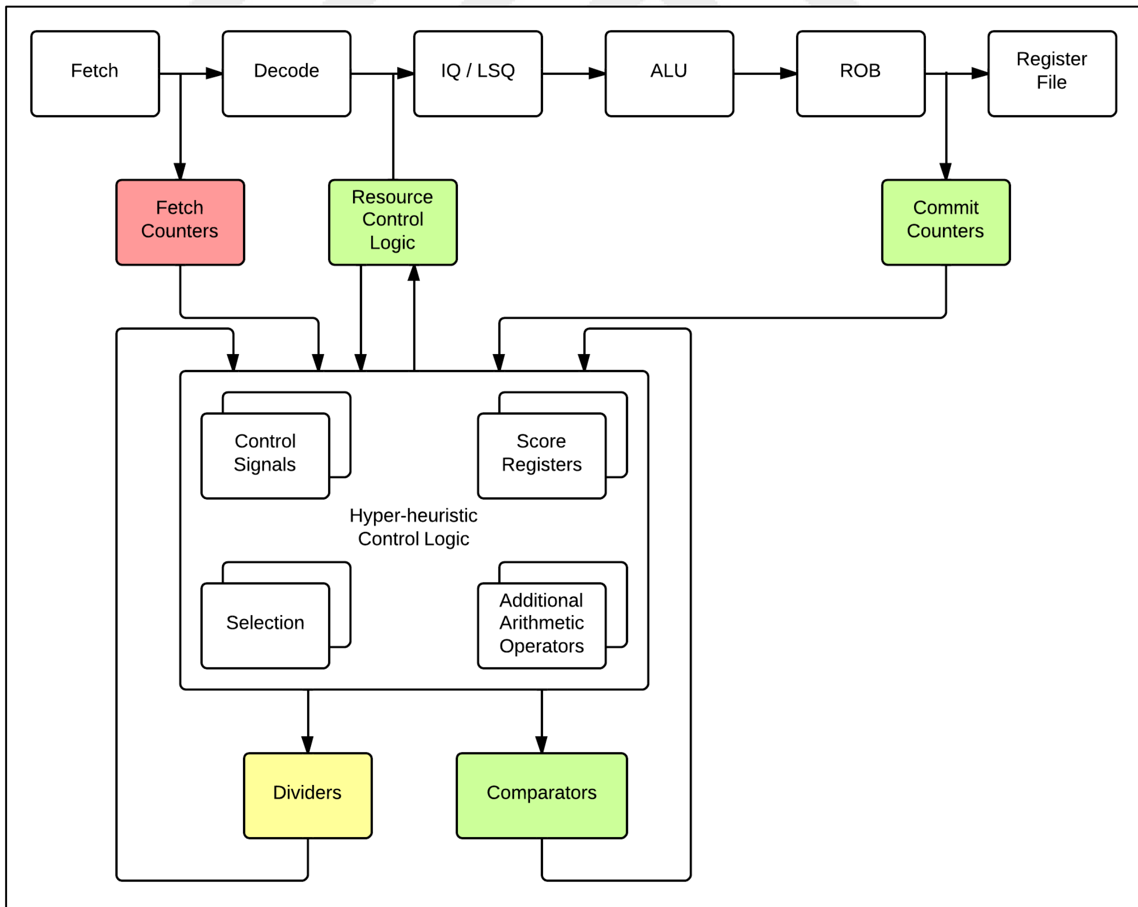



Figure 4.4. Detailed design of hyper-heuristic controlled resource allocation

# 5. EXPERIMENTAL METHODOLOGY

## 5.1. Processor Specifications

We use M-Sim to evaluate our design [27]. M-Sim supports SMT along with the cycle-accurate simulation of the instruction pipeline and the memory hierarchy. The datapath resources are kept as scarce as possible in our simulated processor since having a very large datapath is not realistic due to high power consumption and complexity overhead. A larger IQ would increase wakeup and selection logic complexity, thus increasing access latency. Since IQ is on the critical path of an instruction pipeline, higher access latency may easily degrade processor performance. Table 5.1 gives the details of the simulated processor.

Table 5.1. Processor Specifications

| Decode / Issue / Commit bandwidth | 8 instructions / cycle |
|---|---|
| Register file | 256 int, 256 floating point |
| Reorder buffer (ROB) size | 64 entries |
| Issue Queue (IQ) size | 40 entries |
| Load / Store queue (LSQ) size | 32 entries |
| Number of integer ALUs | 6 |
| Number of integer multiplier / dividers | 3 |
| Number of floating point ALUs | 3 |
| Number of floating point multiplier / dividers | 3 |
| L1 instruction cache | 32 KB, 2-way, LRU |
| L1 data cache | 32 KB, 4-way, LRU |
| L2 unified cache | 512 KB, 4-way, LRU |
| L1 cache hit time | 1 cycle |
| L2 cache hit time | 20 cycles |
| Main memory access time | 300 cycles |

### 5.2. Benchmarks

We select seven benchmarks from the SPEC2006 benchmark suite according to their behaviors provided in [10], with the concern creating a variety of unique workload behaviors. The selected benchmarks are hmmer, lbm, mcf, milc, namd, sjeng, and zeusmp.

Workloads with two, three, and four threads are tested in our study and almost all possible combinations are tested (21 workloads for two-thread mixtures, 35 workloads for three-thread mixtures and 35 workloads for four-thread mixtures). All benchmarks are fast-forwarded for 100 M instructions, and then cycle-accurate simulations are run for 200 M cycles.

### 5.3. Heuristic Parameters

Two important parameters for both ARPA and HILL are *epoch size* and the amount of resources shifted among threads, *delta*.

Both heuristics perform best with a moderate *epoch size*. OHILL sets this parameter to 64 K cycles whereas OARPA sets it to 32 K cycles. In our tests, both heuristics performed best with 32 K cycles of epoch size.

The effect of *delta* values varies with the size of datapath resources. Therefore, comparing the delta values would be misleading. The delta values that we used in our test are empirically chosen, and they are the best performing ones in our setup. Design parameters are given in Table 5.2.

Table 5.2. Heuristic Parameters

| Epoch size | 32768 cycles |
|---|---|
| Delta (ROB) | 4 |
| Delta (IQ) | 2 |
| Delta (LSQ) | 2 |

## 5.4. Metrics

We use three metrics for evaluating the proposed hyper-heuristics: *IPC*, *Average Weighted IPC*, and *Harmonic Mean of Weighted IPC*. IPC quantifies the throughput of the processor. However, IPC alone does not tell if a heuristic fairly treats each running thread. An improvement in IPC can be achieved by increasing the performance of fast threads and decreasing the performance of slow threads. The formula for calculating IPC of *T* threads is given in Equation 5.1.

$$IPC = \sum_{i=1}^{T} \frac{Number\ of\ instructions\ committed\ by\ thread_i}{Number\ of\ cycles} \tag{5.1}$$

Average Weighted IPC shows how much of the performance is achieved by threads compared to their standalone performance. The formula for calculating Average Weighted IPC for *T* threads is given in Equation 5.2. In an ideal run, Average Weighted IPC is expected to be one, indicating that no performance degradation took place due to resource sharing nature of SMT. However, this metric may be greatly affected by even minor performance changes in slow threads. Average Weighted IPC represents the Quality of Service (QoS) information. For example, an Average Weighted IPC of 0.8 or above means that the system can guarantee an average performance level that is only less than 20 per cent of the standalone performance. Therefore, Average Weighted IPC will be referred as *QoS* for the rest of the thesis.

$$Average\ Weighted\ IPC = \frac{\sum_{i=1}^{T} \frac{IPC_i}{standaloneIPC_i}}{T} \tag{5.2}$$

The third metric that we use in our study is the Harmonic Mean of Weighted IPC. It simply takes the harmonic mean of normalized IPCs as shown in Equation 5.3. This metric provides throughput-fairness balance since it is affected by both how fast threads run, and how much of the standalone performance is achieved by each thread. Harmonic Mean of Weighted IPC will be referred as *Hmean* or *fairness* for the rest of the thesis.

$$Harmonic\ mean\ of\ Weighted\ IPC = \frac{T}{\sum_{i=1}^{T} \frac{standaloneIPC_i}{IPC_i}} \tag{5.3}$$

# 6. TESTS AND RESULTS

Throughout this section, we show the effectiveness of our proposed method in terms of various metrics that are described in the previous section. We first start with the sensitivity analysis of our parametric hyper-heuristics, HH3 and HH4.

## 6.1. Sensitivity Analysis

Hyper-heuristics HH3 and HH4 contain variable parameters such as *threshold value* and *round length*. It is quite a challenging task to develop a complete model and foresee the impact of these parameters in SMT processors. Therefore, a sensitivity analysis is carried out by a series of simulations. The effect of *threshold values* of 0.95, 0.98, 1.00, and 1.02; and *round lengths* of five, 10, 15, 20, 25, and 30 Big Epochs are tested on HH3 and HH4, respectively. Full set of tests are run for both hyper-heuristics for every variation of the parameters tested. HH1 and HH2 are not included in the sensitivity analysis due to their non-parametric nature. The results are compared with a static resource partitioning algorithm (STATIC), which partitions the resources equally among running threads, and are presented for HH3 and HH3 in Table 6.1 and Table 6.3, respectively. The equation used in calculating gains of heuristics and hyper-heuristics against static partitioning is given in Equation 6.1.

$$Performance\ Gain = \left(\frac{IPC_{hyper-heuristic}}{IPC_{STATIC}} - 1\right) * 100 \tag{6.1}$$

Table 6.1 shows that HH3 operates best with the threshold value of 1.00 in two-threaded and three-threaded benchmarks. In four-threaded benchmarks, the threshold value of 1.02 performs best in terms of IPC, threshold value of 1.00 being the second. However, the threshold value of 1.00 also provides the best results in QoS and Hmean in four-threaded benchmarks. Therefore, we decided to report the results of HH3 with the threshold value of 1.00 through the rest of the thesis.

The threshold value used in HH3 affects the heuristic selection process by altering the threshold IPC, which determines whether the heuristic is considered to be successful or not during the previous Big Epoch. Therefore, decreasing the threshold IPC results in such a case that a single heuristic proclaims its dominance. Considering that ARPA is favored by being given a second chance when it is unsuccessful; it is inevitable that HH3 converges to running only ARPA if threshold value is set too low. Obviously, this conflicts with our purpose of harvesting the advantages of multiple heuristics by using them in a mixture. Table 6.2 shows the percentages of ARPA run in HH3. As expected, the utilization percentage of ARPA increases as the threshold value drops. The ARPA-o column in the table shows the percentage of workloads that only run ARPA. It can also be seen that a significant number of workloads start running only ARPA when the threshold value gets lower. Similarly, when the threshold value increases too much, the threshold IPC value simply becomes too high for any heuristic to satisfy, causing the hyper-heuristic to go into an unstable phase. As the threshold value increases, the percentage of ARPA selections converges to 66.6 per cent (since ARPA is given two chances per failure, whereas HILL is given only one). The fact that the performance of HH3 drops when the threshold value is increased beyond its optimal value suggests that using heuristics regularly in turns of making clever decisions may not be sufficient to reach optimal performance values.

Table 6.1. Performance gains of various threshold values (*th.v.*s) in HH3 against STATIC

| | Two-thread benchmarks | | | Three-thread benchmarks | | | Four-thread benchmarks | | |
|---|---|---|---|---|---|---|---|---|---|
| *th.v.* | IPC | QoS | Hmean | IPC | QoS | Hmean | IPC | QoS | Hmean |
| 0.95 | 2.2% | -1.9% | -6.3% | 5.4% | -3.7% | -12.4% | 5.9% | -7.7% | -22.5% |
| 0.98 | 2.1% | -2.0% | -6.4% | 5.4% | -3.9% | -11.4% | 5.9% | -7.6% | -22.4% |
| 1.00 | 2.7% | -1.1% | -3.2% | 5.6% | -2.9% | -8.7% | 5.9% | -5.8% | -16.3% |
| 1.02 | 2.5% | -1.9% | -4.8% | 5.5% | -3.3% | -9.5% | 5.9% | -6.4% | -17.4% |

All hyper-heuristics proposed in this study except HH4 are binary hyper-heuristics, in the sense that these hyper-heuristics identify heuristics merely as either successful or not successful. Simply, they do not take the degree of success into account. HH4, on the other hand, is a score based hyper-heuristic, in which the past performance results of heuristics affect future decisions of the hyper-heuristic. However, without an intervention to the

score-based approach, a single heuristic may dominate the processor due to its past successes, disregarding the current performance of the processor and the recent needs of the workloads. To overcome this problem, HH4 resets the scores periodically at the start of each *round*. This prevents HH4 from considering rather old heuristic performances, making it a more responsive hyper-heuristic to adapt to the recent changes of the workload behavior. Setting the *round length* to a very large amount causes HH4 to perform as a simple score-based hyper-heuristic, whereas setting the round length to zero causes HH4 to mimic HH1 by comparing only the last two Big Epoch's performances. The performance of HH4 in various round lengths is given in Table 6.3. HH4 with a round length of five performs best among all tested round lengths in terms of IPC; therefore, we decided to report the results for this configuration through the rest of the thesis.

Table 6.2. Percentages of ARPA run in various threshold values (*th.v.*s) in HH3

| | Two-thread benchmarks | | Three-thread benchmarks | | Four-thread benchmarks | |
|---|---|---|---|---|---|---|
| *th.v.* | ARPA | ARPA-o | ARPA | ARPA-o | ARPA | ARPA-o |
| 0.95 | 86.1% | 4.8% | 91.9% | 20.0% | 97.1% | 40.0% |
| 0.98 | 85.8% | 4.8% | 91.9% | 11.4% | 93.0% | 14.3% |
| 1.00 | 70.7% | 0.0% | 75.5% | 0.0% | 76.6% | 0.0% |
| 1.02 | 63.4% | 0.0% | 66.0% | 0.0% | 66.7% | 0.0% |

Table 6.3. Performance gains of HH4 in various round lengths (*r.l.*s) against STATIC

| | Two-threaded benchmarks | | | Three-threaded benchmarks | | | Four-threaded benchmarks | | |
|---|---|---|---|---|---|---|---|---|---|
| *r.l.* | IPC | QoS | Hmean | IPC | QoS | Hmean | IPC | QoS | Hmean |
| 5 | 3.0% | -0.4% | -2.0% | 5.6% | -2.4% | -6.9% | 5.9% | -4.5% | -12.8% |
| 10 | 2.7% | -1.4% | -3.6% | 5.5% | -2.1% | -6.5% | 5.7% | -4.4% | -12.2% |
| 15 | 2.7% | -1.2% | -3.2% | 5.6% | -2.1% | -6.4% | 5.6% | -4.4% | -12.2% |
| 20 | 2.7% | -1.1% | -3.1% | 5.5% | -2.1% | -6.3% | 5.5% | -4.5% | -12.3% |
| 25 | 2.7% | -1.2% | -3.2% | 5.5% | -2.2% | -6.6% | 5.4% | -4.4% | -12.3% |
| 30 | 2.9% | -0.4% | -2.2% | 5.5% | -2.1% | -6.4% | 5.5% | -4.6% | -12.6% |

## 6.2. Results and Discussion

In this section, the performance results of HILL, ARPA, HH1, HH2, HH3, and HH4 in various workloads are discussed. Static Partitioning (STATIC) is chosen as a baseline algorithm to provide a basis for comparison. Table 6.4 reports the performance results of the algorithms in terms of IPC, QoS, and Hmean metrics. Peak gains of hyper-heuristics over HILL and ARPA are also shown on Table 6.5.

The results show that none of the heuristics can outperform static partitioning in QoS or Hmean (except for HILL in QoS, in two-threaded benchmarks). This is an expected result, since we are using IPC as a performance goal for HILL and ARPA, which tends to monopolize resources to one thread in most of the cases. In ARPA, a thread that loses a portion of its resources improves its efficiency whereas the thread which receives extra resources loses its efficiency. ARPA is built upon the assumption that such resource exchanges among threads are carried out until a point is reached where the efficiencies of the threads are close to each other or no more resources can be exchanged anymore. This is the actual case in some workloads; however, in some workloads, the thread which gives resources away also loses efficiency due to the reduction of its out-of-order execution window size. The latter case is much more common, and ARPA can outperform static partitioning in Hmean, in seven out of 21 workloads in two-threaded benchmarks; 11 out of 35 workloads in three-threaded benchmarks and five out of 35 workloads in four-threaded benchmarks. Additionally, the Hmean loss in workloads that ARPA experiences is far more pronounced than the Hmean gain in workloads that ARPA performs better than static partitioning. On the most extreme cases in four-threaded benchmarks, ARPA performs 4.5 per cent better and 53.8 per cent worse than static partitioning in terms of Hmean. Additionally, it can be seen that the fairness metric suffers more greatly in benchmarks that run more simultaneous threads. This is an expected result especially when the resources are monopolized by a few threads. The reason HILL performs better than ARPA in terms of the fairness metric, despite the fact that it is optimized for performance, is that HILL spends most of its time comparing the performance of threads in consecutive trial epochs. Therefore, HILL inevitably gives some extra resources to each thread compared to ARPA, regardless of resource monopolization. It could also be argued that the implementation difference between ARPA and OARPA in terms of decreased lower limit

of resources per thread causes the fairness degradation in ARPA (the lower limit is 25 per cent of initial share in OARPA, and is equal to step size *delta* in ARPA). However, in out experimental setup, the lower limits would be identical for four-thread benchmarks in OARPA and ARPA. For example, 25 per cent of the initial share in a processor with 64-entry ROB which runs four threads simultaneously would be four entries per thread, which is equal to the *delta* value used in our tests. The same argument is valid of IQ and LSQ, as well.

Table 6.4. Performance gains of heuristic against STATIC

| | Two-thread benchmarks | | | Three-thread benchmarks | | | Four-thread benchmarks | | |
|---|---|---|---|---|---|---|---|---|---|
| Heur. | IPC | QoS | Hmean | IPC | QoS | Hmean | IPC | QoS | Hmean |
| HILL | 2.9% | 0.1% | -1.0% | 5.0% | -1.6% | -4.6% | 3.5% | -3.8% | -10.9% |
| ARPA | 1.8% | -2.9% | -7.8% | 5.0% | -4.3% | -13.7% | 5.9% | -7.8% | -22.8% |
| HH1 | 3.0% | -0.6% | -2.3% | 5.6% | -2.7% | -8.0% | 6.1% | -6.0% | -15.5% |
| HH2 | 2.6% | -1.7% | -4.3% | 5.5% | -3.2% | -9.0% | 5.9% | -6.0% | -16.4% |
| HH3 | 2.7% | -1.1% | -3.2% | 5.6% | -2.9% | -8.7% | 5.9% | -5.8% | -16.3% |
| HH4 | 3.0% | -0.4% | -2.0% | 5.6% | -2.4% | -6.9% | 5.9% | -4.5% | -12.8% |

Table 6.5. Peak gains of proposed hyper-heuristics over HILL and ARPA

| | Peak Gains | | | | | |
|---|---|---|---|---|---|---|
| | Two-thread benchmarks | | Three-thread benchmarks | | Four-thread benchmarks | |
| Heur. | HILL | ARPA | HILL | ARPA | HILL | ARPA |
| HH1 | 3.0% | 6.1% | 4.5% | 5.1% | 19.2% | 4.0% |
| HH2 | 3.2% | 6.1% | 4.4% | 5.1% | 25.0% | 10.4% |
| HH3 | 3.0% | 5.3% | 4.5% | 4.7% | 19.5% | 5.4% |
| HH4 | 2.5% | 6.6% | 3.9% | 4.2% | 19.1% | 4.2% |

The performance results of hyper-heuristics proposed in this study show that HH4 is the clear winner. In two-threaded benchmarks, HH1 and HH4 perform best, with performance levels of IPC almost identical. However, HH4 is better overall in two-threaded benchmarks, due to slight performance gains in QoS and Hmean in two-threaded benchmarks. This is also the case in three-threaded benchmarks. In four-threaded

benchmarks, HH1 performs best in terms of IPC, but HH4 offers 1.5 per cent and 2.7 per cent less QoS and Hmean loss, respectively, against static partitioning, compared to HH1. Actually, HH3 always performs better than HH1 when the QoS and fairness metrics are considered.

Figure 6.1 shows the percentage of workloads where the hyper-heuristics manage to outperform HILL and ARPA in terms of IPC. HH4 outperforms either of the heuristics on 72 per cent of the overall workloads, and is the clear winner. HH1 is the second best with 70 per cent and HH3 is the third best with 68 per cent. HH2 performs worst by achieving 66 percent in these tests. Besides, HH4 outperforms both of the heuristics on around 27 per cent of the simulated workloads. Although, HH4 is the most successful hyper-heuristics in terms of almost all metrics, HH1 shows a quite noticeable performance and demonstrates that even a very simple hyper-heuristic may give promising results.

HH4 drops almost never below the worst performer among HILL and ARPA in all metrics. Figures 6.2-6.10 present the normalized performance, QoS and Hmean results of HILL, ARPA, and HH4 in all tested workloads. The results in these figures are normalized to the best performer for each workload. It can clearly be seen that HH4 is between HILL and ARPA most of the time, outperforming both heuristics time to time and rarely dropping below both. In this sense, HH4 achieves our purpose of this study: harnessing the strengths of both heuristics, performing on a level close to the better performing heuristic on most of the tested workloads. In fact, HH4 manages to outperform both heuristics in some workloads. These results also indicate that HH4 provides a more stable performance among workloads compared to HILL and ARPA.

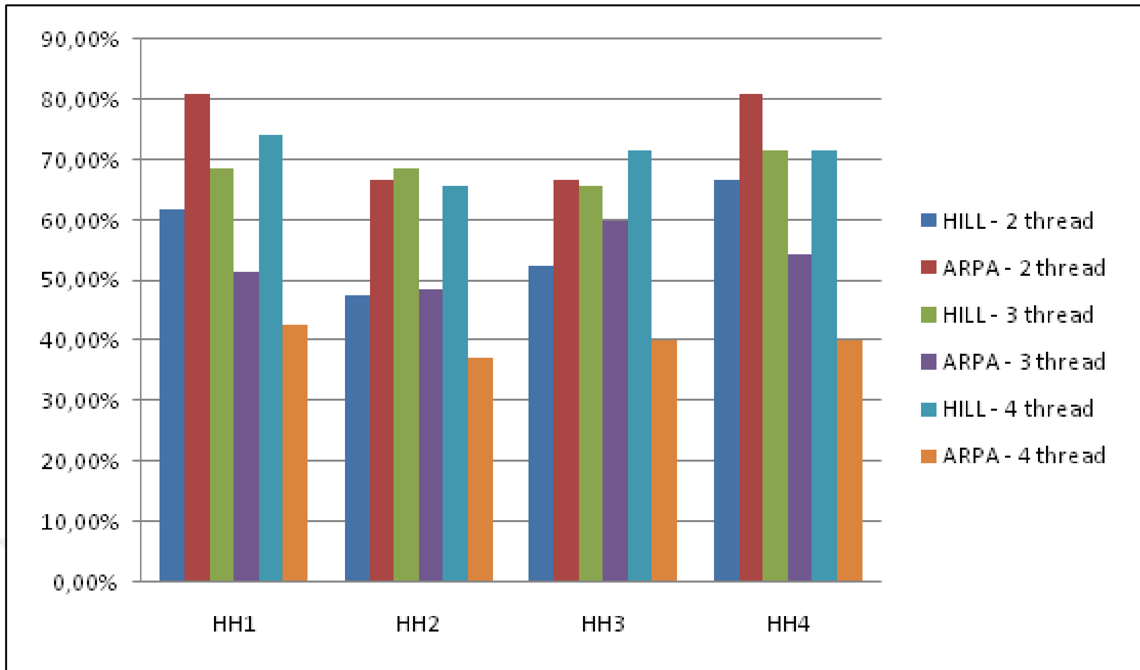



Figure 6.1. Percentage of workloads for which hyper-heuristics perform better or equivalent to HILL and ARPA in terms of IPC

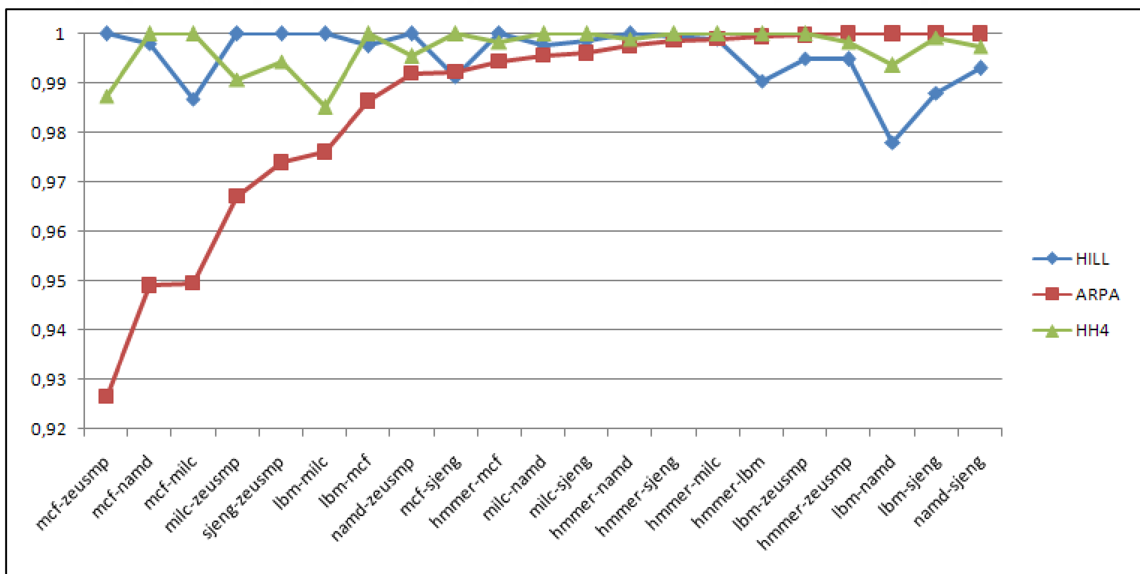



Figure 6.2. Normalized performance results of two-thread workloads utilizing HILL, ARPA, and HH4

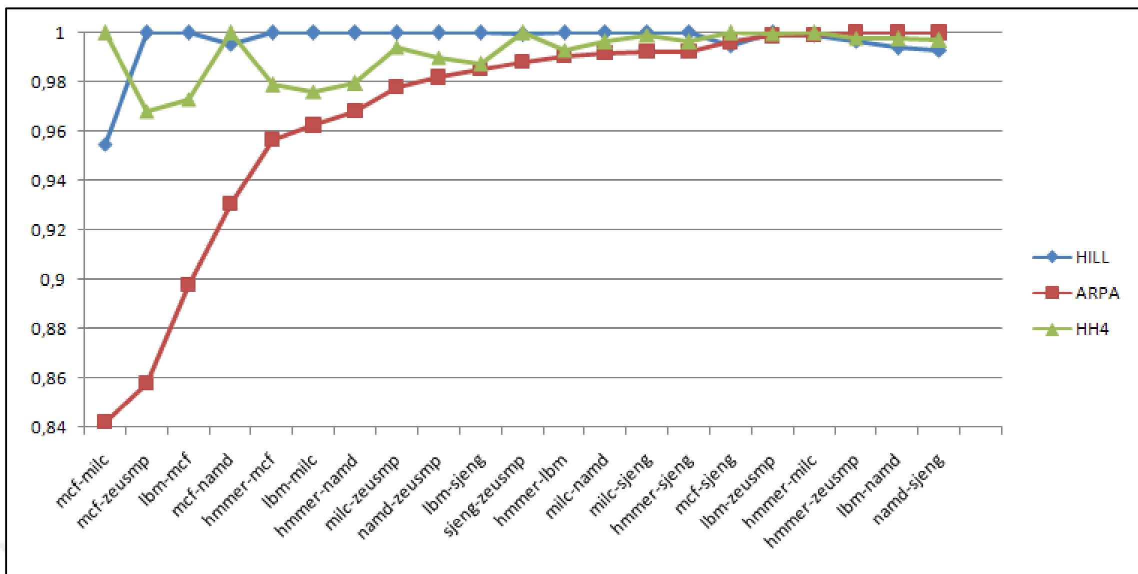



Figure 6.3. Normalized QoS results of two-thread workloads utilizing HILL, ARPA, and HH4

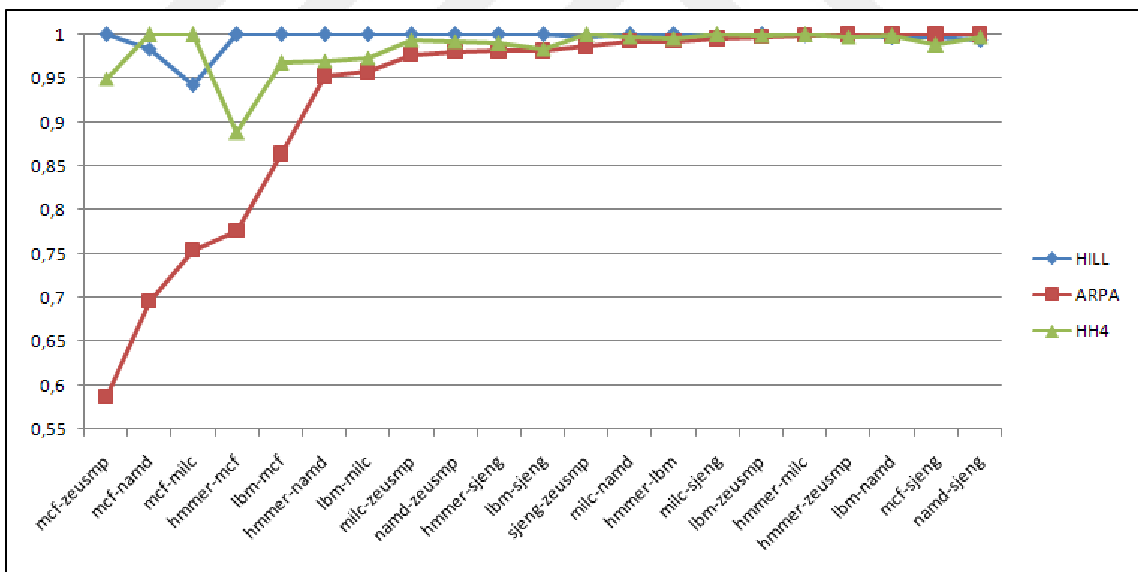



Figure 6.4. Normalized fairness results of two-threaded workloads utilizing HILL, ARPA, and HH4

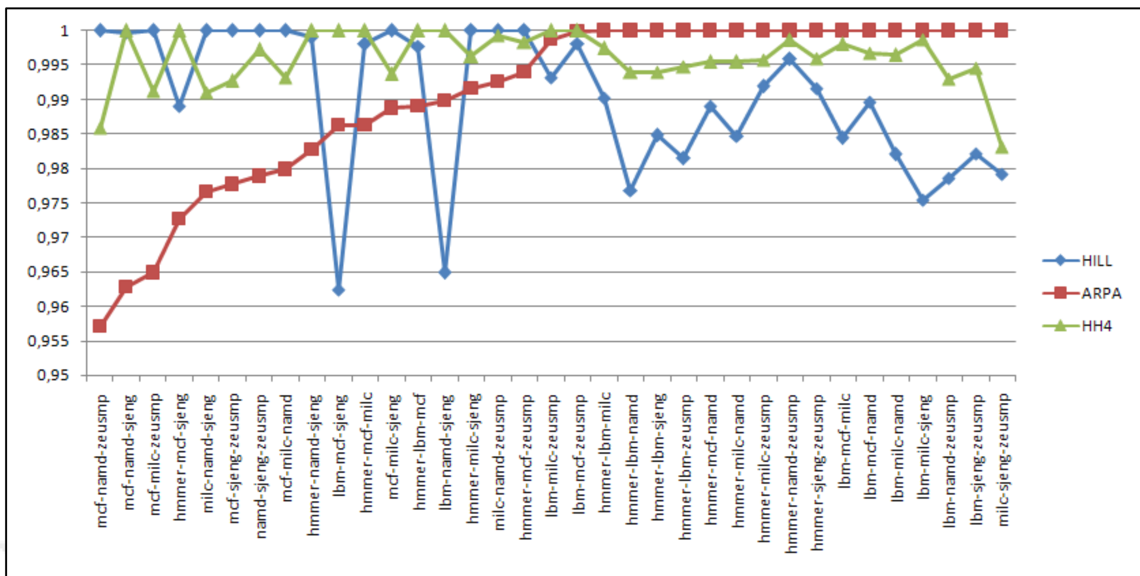



Figure 6.5. Normalized performance results of three-thread workloads utilizing HILL, ARPA, and HH4

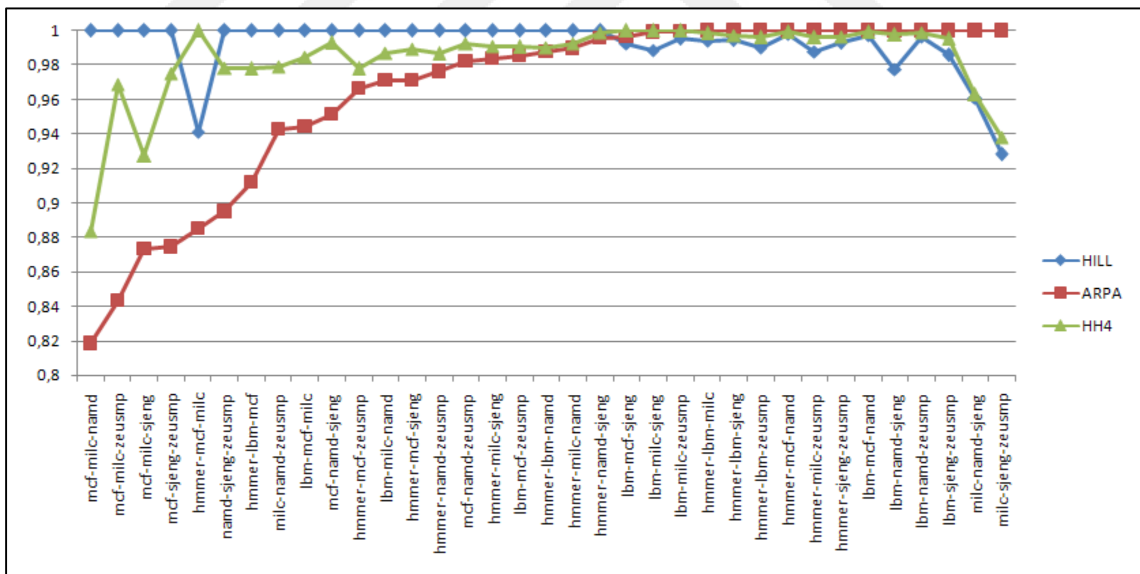



Figure 6.6. Normalized QoS results of three-thread workloads utilizing HILL, ARPA, and HH4

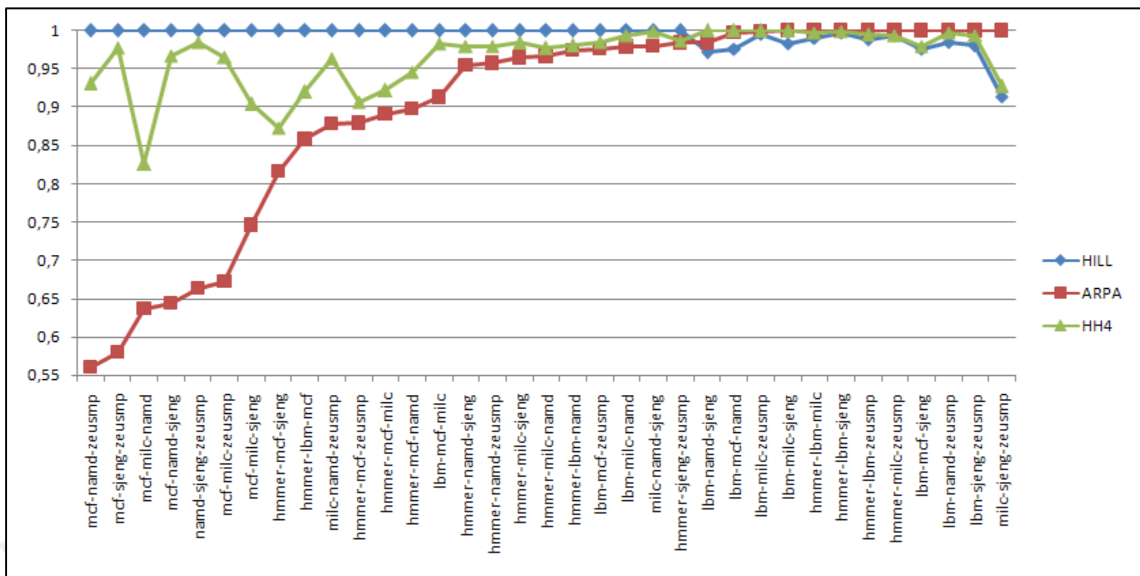



Figure 6.7. Normalized fairness results of three-thread workloads utilizing HILL, ARPA, and HH4

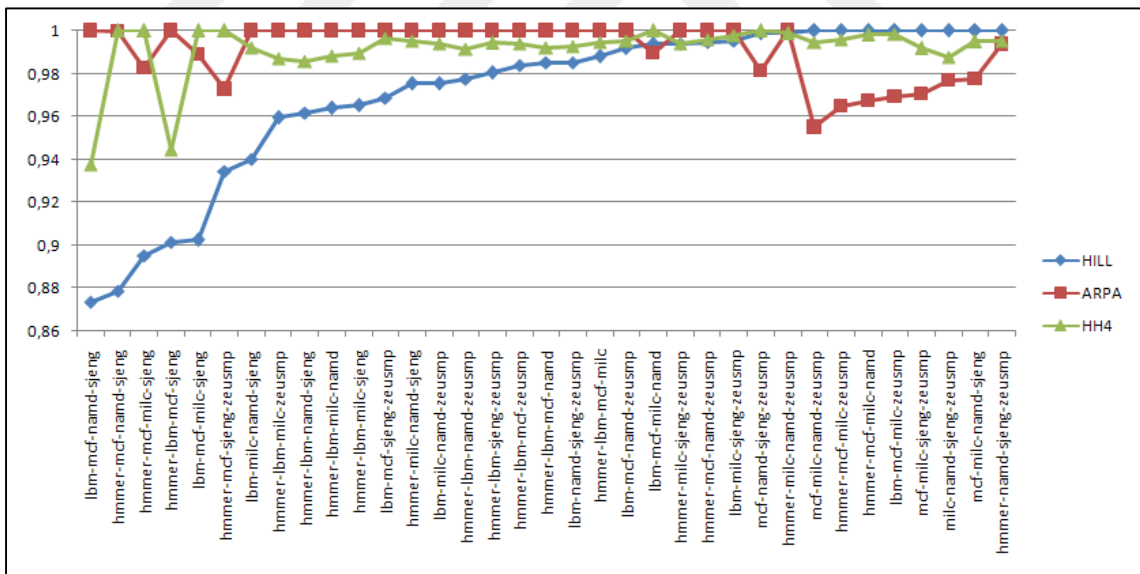



Figure 6.8. Normalized performance results of four-thread workloads utilizing HILL, ARPA, and HH4

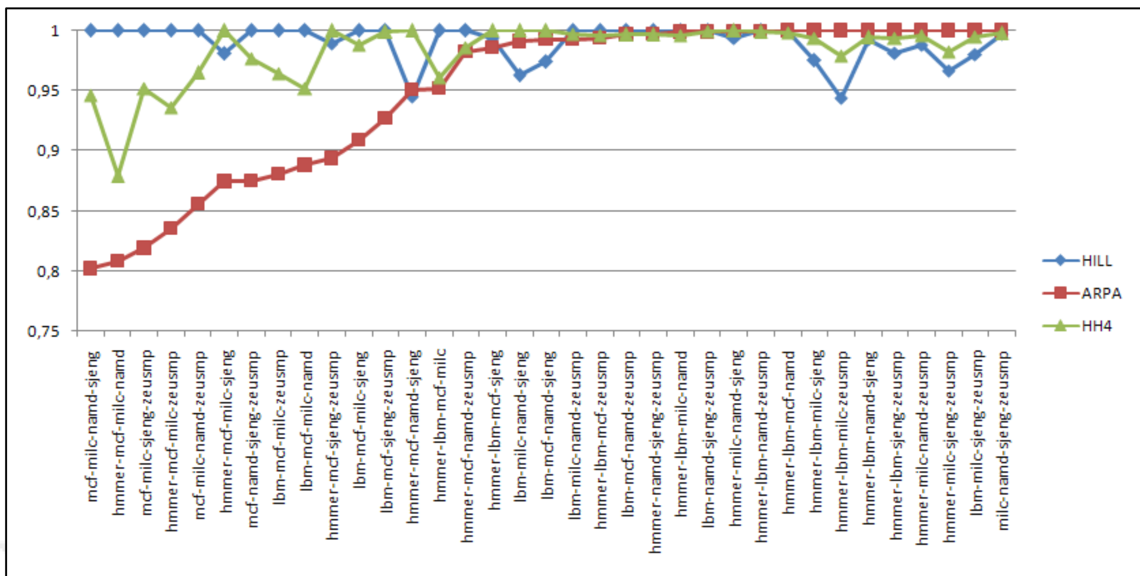



Figure 6.9. Normalized QoS results of four-thread workloads utilizing HILL, ARPA, and HH4

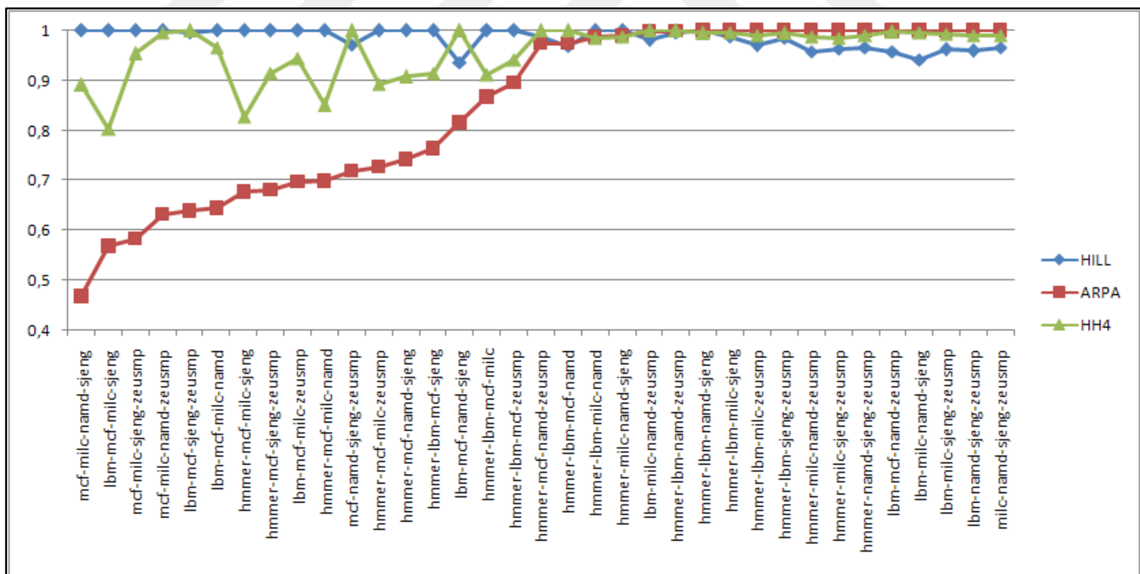



Figure 6.10. Normalized fairness results of four-thread workloads utilizing HILL, ARPA, and HH4

# 7. CONCLUSIONS AND FUTURE WORK

Resource distribution in SMT processors is a key issue for obtaining a high level of performance. In a processor where the resources are uncontrollably shared among threads, threads with long latency instructions may monopolize resources without making any progress. Meanwhile, in a processor where resources are statically partitioned among threads, the resources may not be fully utilized. There are various studies on optimizing SMT performance by controlling the resource distribution via different heuristics. However, each proposed heuristic has its cons and pros, and, in this study, we propose the utilization of the right heuristic at the right time with the help of a hyper-heuristic.

Hyper-heuristics have been applied to different real world problems. However, to the best of our knowledge, this is the first time it is applied to the processor domain. As a matter of fact, creating an environment where different heuristics may coexist on the same hardware is a difficult task by itself, if not impossible. For the heuristics that can coexist together, compromises must be made at each side of adaptation. Moreover, each heuristic that is implemented increases both the hyper-heuristic logic cost and its hardware complexity. With all these limitations in mind, we utilized Hill Climbing and Adaptive Resource Partitioning algorithms as our base heuristics. By using various approaches inspired from the existing hyper-heuristics, we managed to develop new hyper-heuristics mixing two heuristics, which perform better than each individual heuristic run on their own in terms of processor throughput. Moreover, all proposed hyper-heuristics perform better than the Adaptive Resource Partitioning Algorithm in terms of both Average Weighted IPC and the fairness metrics.

The performance of hyper-heuristics can be further improved by introducing new heuristics into the mix, which can increase the throughput of the processor where the present two heuristics fail to make accurate assessments and degrade throughput. Finally, another possible improvement over this study may be to use other processor statistics that are already available. Exploiting additional statistics may provide a better insight to hyper-heuristics ob the decision making process.

# APPENDIX A: RESULTS OF TWO-THREADED BENCHMARKS

Table A.1. IPC results of two-threaded benchmarks

| | STATIC | HILL | ARPA | HH1 | HH2 | HH3 | HH4 |
|---|---|---|---|---|---|---|---|
| hmmer-lbm | 1.98 | 1.95 | 1.97 | 1.97 | 1.97 | 1.97 | 1.97 |
| hmmer-mcf | 3.96 | 4.25 | 4.23 | 4.25 | 4.24 | 4.23 | 4.25 |
| hmmer-milc | 2.45 | 2.46 | 2.46 | 2.46 | 2.46 | 2.46 | 2.46 |
| hmmer-namd | 3.71 | 3.78 | 3.77 | 3.77 | 3.77 | 3.77 | 3.77 |
| hmmer-sjeng | 2.85 | 2.85 | 2.85 | 2.85 | 2.85 | 2.85 | 2.86 |
| hmmer-zeusmp | 3.61 | 3.69 | 3.71 | 3.71 | 3.71 | 3.71 | 3.71 |
| lbm-mcf | 4.17 | 4.39 | 4.34 | 4.40 | 4.40 | 4.40 | 4.40 |
| lbm-milc | 2.06 | 2.08 | 2.03 | 2.05 | 2.04 | 2.05 | 2.05 |
| lbm-namd | 3.58 | 3.91 | 4.00 | 4.00 | 3.99 | 3.98 | 3.97 |
| lbm-sjeng | 2.56 | 2.58 | 2.61 | 2.61 | 2.61 | 2.61 | 2.61 |
| lbm-zeusmp | 3.13 | 3.45 | 3.46 | 3.46 | 3.45 | 3.46 | 3.46 |
| mcf-milc | 4.30 | 4.48 | 4.31 | 4.54 | 4.23 | 4.43 | 4.54 |
| mcf-namd | 4.76 | 4.76 | 4.52 | 4.77 | 4.76 | 4.77 | 4.77 |
| mcf-sjeng | 4.15 | 4.31 | 4.31 | 4.35 | 4.34 | 4.33 | 4.35 |
| mcf-zeusmp | 4.91 | 4.89 | 4.53 | 4.81 | 4.80 | 4.77 | 4.82 |
| milc-namd | 4.03 | 4.22 | 4.22 | 4.24 | 4.24 | 4.24 | 4.23 |
| milc-sjeng | 2.95 | 2.97 | 2.97 | 2.98 | 2.97 | 2.97 | 2.98 |
| milc-zeusmp | 3.59 | 3.84 | 3.71 | 3.79 | 3.78 | 3.78 | 3.80 |
| namd-sjeng | 4.00 | 4.03 | 4.06 | 4.05 | 4.05 | 4.05 | 4.05 |
| namd-zeusmp | 4.90 | 4.85 | 4.81 | 4.82 | 4.83 | 4.83 | 4.83 |
| sjeng-zeusmp | 4.03 | 4.13 | 4.03 | 4.10 | 4.10 | 4.09 | 4.11 |

Table A.2. Average Weighted IPC results of two-threaded benchmarks

| | STATIC | HILL | ARPA | HH1 | HH2 | HH3 | HH4 |
|---|---|---|---|---|---|---|---|
| hmmer-lbm | 0.85 | 0.84 | 0.83 | 0.83 | 0.83 | 0.83 | 0.83 |
| hmmer-mcf | 0.57 | 0.60 | 0.58 | 0.59 | 0.58 | 0.58 | 0.59 |
| hmmer-milc | 0.83 | 0.84 | 0.84 | 0.84 | 0.84 | 0.84 | 0.84 |
| hmmer-namd | 0.73 | 0.70 | 0.68 | 0.68 | 0.68 | 0.68 | 0.68 |
| hmmer-sjeng | 0.76 | 0.76 | 0.76 | 0.76 | 0.76 | 0.76 | 0.76 |
| hmmer-zeusmp | 0.79 | 0.77 | 0.77 | 0.77 | 0.77 | 0.77 | 0.77 |
| lbm-mcf | 0.85 | 0.87 | 0.78 | 0.84 | 0.84 | 0.78 | 0.84 |
| lbm-milc | 0.90 | 0.91 | 0.88 | 0.89 | 0.89 | 0.88 | 0.89 |
| lbm-namd | 0.82 | 0.83 | 0.84 | 0.84 | 0.84 | 0.84 | 0.84 |
| lbm-sjeng | 0.86 | 0.86 | 0.85 | 0.85 | 0.85 | 0.85 | 0.85 |
| lbm-zeusmp | 0.83 | 0.85 | 0.85 | 0.85 | 0.85 | 0.85 | 0.85 |
| mcf-milc | 0.81 | 0.76 | 0.67 | 0.80 | 0.63 | 0.81 | 0.80 |
| mcf-namd | 0.61 | 0.60 | 0.56 | 0.61 | 0.61 | 0.56 | 0.61 |
| mcf-sjeng | 0.62 | 0.61 | 0.61 | 0.61 | 0.61 | 0.61 | 0.61 |
| mcf-zeusmp | 0.68 | 0.68 | 0.58 | 0.65 | 0.65 | 0.58 | 0.66 |
| milc-namd | 0.82 | 0.83 | 0.82 | 0.83 | 0.83 | 0.83 | 0.83 |
| milc-sjeng | 0.84 | 0.85 | 0.84 | 0.84 | 0.84 | 0.85 | 0.85 |
| milc-zeusmp | 0.82 | 0.85 | 0.83 | 0.85 | 0.84 | 0.85 | 0.85 |
| namd-sjeng | 0.70 | 0.70 | 0.70 | 0.70 | 0.70 | 0.70 | 0.70 |
| namd-zeusmp | 0.72 | 0.72 | 0.70 | 0.71 | 0.71 | 0.71 | 0.71 |
| sjeng-zeusmp | 0.78 | 0.79 | 0.78 | 0.79 | 0.79 | 0.78 | 0.79 |

Table A.3. Harmonic Mean of Weighted IPC results of two-threaded benchmarks

| hmmer-lbm | STATIC | HILL | ARPA | HH1 | HH2 | HH3 | HH4 |
|---|---|---|---|---|---|---|---|
| hmmer-mcf | 0.85 | 0.83 | 0.83 | 0.83 | 0.83 | 0.83 | 0.83 |
| hmmer-milc | 0.43 | 0.42 | 0.33 | 0.36 | 0.34 | 0.33 | 0.37 |
| hmmer-namd | 0.82 | 0.83 | 0.83 | 0.83 | 0.83 | 0.83 | 0.83 |
| hmmer-sjeng | 0.72 | 0.69 | 0.66 | 0.67 | 0.67 | 0.66 | 0.67 |
| hmmer-zeusmp | 0.75 | 0.75 | 0.74 | 0.74 | 0.74 | 0.74 | 0.74 |
| lbm-mcf | 0.79 | 0.75 | 0.75 | 0.75 | 0.75 | 0.75 | 0.75 |
| lbm-milc | 0.85 | 0.86 | 0.75 | 0.83 | 0.83 | 0.75 | 0.84 |
| lbm-namd | 0.90 | 0.91 | 0.87 | 0.88 | 0.88 | 0.87 | 0.89 |
| lbm-sjeng | 0.82 | 0.83 | 0.84 | 0.84 | 0.84 | 0.84 | 0.84 |
| lbm-zeusmp | 0.86 | 0.86 | 0.85 | 0.85 | 0.85 | 0.85 | 0.85 |
| mcf-milc | 0.83 | 0.85 | 0.85 | 0.85 | 0.85 | 0.85 | 0.85 |
| mcf-namd | 0.81 | 0.75 | 0.60 | 0.80 | 0.51 | 0.81 | 0.80 |
| mcf-sjeng | 0.61 | 0.59 | 0.42 | 0.60 | 0.60 | 0.42 | 0.60 |
| mcf-zeusmp | 0.58 | 0.50 | 0.50 | 0.49 | 0.49 | 0.50 | 0.49 |
| milc-namd | 0.68 | 0.68 | 0.40 | 0.63 | 0.63 | 0.40 | 0.65 |
| milc-sjeng | 0.81 | 0.83 | 0.82 | 0.83 | 0.83 | 0.83 | 0.83 |
| milc-zeusmp | 0.84 | 0.84 | 0.84 | 0.84 | 0.84 | 0.84 | 0.84 |
| namd-sjeng | 0.82 | 0.85 | 0.83 | 0.85 | 0.84 | 0.85 | 0.85 |
| namd-zeusmp | 0.69 | 0.69 | 0.70 | 0.70 | 0.70 | 0.70 | 0.70 |
| sjeng-zeusmp | 0.72 | 0.71 | 0.70 | 0.71 | 0.71 | 0.71 | 0.71 |
| hmmer-lbm | 0.78 | 0.79 | 0.78 | 0.79 | 0.79 | 0.78 | 0.79 |

# APPENDIX B:  RESULTS OF THREE-THREADED BENCHMARKS

Table B.1. IPC results of three-threaded benchmarks

| | STATIC | HILL | ARPA | HH1 | HH2 | HH3 | HH4 |
|---|---|---|---|---|---|---|---|
| hmmer-lbm-mcf | 3.73 | 4.49 | 4.46 | 4.51 | 4.51 | 4.51 | 4.50 |
| hmmer-lbm-milc | 2.80 | 2.86 | 2.88 | 2.88 | 2.88 | 2.88 | 2.88 |
| hmmer-lbm-namd | 3.70 | 3.80 | 3.89 | 3.87 | 3.86 | 3.88 | 3.87 |
| hmmer-lbm-sjeng | 3.05 | 3.12 | 3.16 | 3.15 | 3.15 | 3.16 | 3.15 |
| hmmer-lbm-zeusmp | 3.65 | 3.89 | 3.96 | 3.94 | 3.93 | 3.95 | 3.94 |
| hmmer-mcf-milc | 3.79 | 4.54 | 4.49 | 4.55 | 4.56 | 4.55 | 4.55 |
| hmmer-mcf-namd | 4.34 | 4.58 | 4.63 | 4.62 | 4.61 | 4.63 | 4.61 |
| hmmer-mcf-sjeng | 3.44 | 3.95 | 3.89 | 4.01 | 4.01 | 4.00 | 4.00 |
| hmmer-mcf-zeusmp | 4.46 | 4.85 | 4.82 | 4.83 | 4.83 | 4.83 | 4.84 |
| hmmer-milc-namd | 3.99 | 4.03 | 4.09 | 4.07 | 4.07 | 4.08 | 4.08 |
| hmmer-milc-sjeng | 3.30 | 3.37 | 3.34 | 3.36 | 3.35 | 3.36 | 3.36 |
| hmmer-milc-zeusmp | 4.01 | 4.16 | 4.20 | 4.18 | 4.16 | 4.18 | 4.18 |
| hmmer-namd-sjeng | 3.88 | 3.87 | 3.80 | 3.87 | 3.86 | 3.84 | 3.87 |
| hmmer-namd-zeusmp | 4.72 | 4.80 | 4.82 | 4.81 | 4.81 | 4.82 | 4.81 |
| hmmer-sjeng-zeusmp | 4.19 | 4.29 | 4.33 | 4.31 | 4.30 | 4.32 | 4.31 |
| lbm-mcf-milc | 4.02 | 4.62 | 4.70 | 4.69 | 4.67 | 4.65 | 4.69 |
| lbm-mcf-namd | 4.68 | 4.77 | 4.82 | 4.81 | 4.81 | 4.81 | 4.80 |
| lbm-mcf-sjeng | 3.55 | 4.07 | 4.17 | 4.25 | 4.25 | 4.25 | 4.23 |
| lbm-mcf-zeusmp | 4.57 | 4.94 | 4.95 | 4.95 | 4.94 | 4.95 | 4.95 |
| lbm-milc-namd | 4.11 | 4.32 | 4.40 | 4.40 | 4.40 | 4.40 | 4.38 |
| lbm-milc-sjeng | 3.27 | 3.26 | 3.35 | 3.34 | 3.33 | 3.35 | 3.34 |
| lbm-milc-zeusmp | 3.88 | 4.12 | 4.14 | 4.14 | 4.13 | 4.14 | 4.15 |
| lbm-namd-sjeng | 3.85 | 3.78 | 3.88 | 3.91 | 3.90 | 3.92 | 3.92 |
| lbm-namd-zeusmp | 4.70 | 4.85 | 4.95 | 4.92 | 4.91 | 4.93 | 4.92 |
| lbm-sjeng-zeusmp | 4.09 | 4.25 | 4.32 | 4.30 | 4.29 | 4.31 | 4.30 |
| mcf-milc-namd | 4.73 | 4.76 | 4.66 | 4.71 | 4.72 | 4.68 | 4.72 |
| mcf-milc-sjeng | 4.10 | 4.48 | 4.43 | 4.45 | 4.44 | 4.45 | 4.46 |
| mcf-milc-zeusmp | 4.74 | 4.93 | 4.76 | 4.84 | 4.79 | 4.80 | 4.89 |
| mcf-namd-sjeng | 4.45 | 4.58 | 4.41 | 4.58 | 4.58 | 4.54 | 4.58 |
| mcf-namd-zeusmp | 5.13 | 5.08 | 4.87 | 4.99 | 5.00 | 4.99 | 5.01 |
| mcf-sjeng-zeusmp | 4.58 | 4.85 | 4.74 | 4.79 | 4.76 | 4.78 | 4.81 |
| milc-namd-sjeng | 4.16 | 4.14 | 4.04 | 4.09 | 4.08 | 4.08 | 4.10 |
| milc-namd-zeusmp | 4.87 | 4.93 | 4.89 | 4.92 | 4.92 | 4.92 | 4.92 |
| milc-sjeng-zeusmp | 4.32 | 4.36 | 4.45 | 4.44 | 4.41 | 4.44 | 4.37 |
| namd-sjeng-zeusmp | 4.88 | 4.94 | 4.83 | 4.91 | 4.92 | 4.92 | 4.92 |

Table B.2. Average Weighted IPC results of three-threaded benchmarks

| | STATIC | HILL | ARPA | HH1 | HH2 | HH3 | HH4 |
|---|---|---|---|---|---|---|---|
| hmmer-lbm-mcf | 0.57 | 0.61 | 0.56 | 0.60 | 0.60 | 0.56 | 0.60 |
| hmmer-lbm-milc | 0.76 | 0.77 | 0.77 | 0.77 | 0.77 | 0.77 | 0.77 |
| hmmer-lbm-namd | 0.67 | 0.65 | 0.65 | 0.64 | 0.65 | 0.65 | 0.65 |
| hmmer-lbm-sjeng | 0.69 | 0.70 | 0.70 | 0.70 | 0.70 | 0.70 | 0.70 |
| hmmer-lbm-zeusmp | 0.70 | 0.70 | 0.71 | 0.71 | 0.71 | 0.71 | 0.71 |
| hmmer-mcf-milc | 0.55 | 0.55 | 0.52 | 0.58 | 0.55 | 0.58 | 0.58 |
| hmmer-mcf-namd | 0.44 | 0.42 | 0.42 | 0.42 | 0.42 | 0.42 | 0.42 |
| hmmer-mcf-sjeng | 0.43 | 0.42 | 0.41 | 0.42 | 0.41 | 0.42 | 0.42 |
| hmmer-mcf-zeusmp | 0.49 | 0.48 | 0.46 | 0.47 | 0.47 | 0.46 | 0.47 |
| hmmer-milc-namd | 0.64 | 0.62 | 0.62 | 0.62 | 0.62 | 0.62 | 0.62 |
| hmmer-milc-sjeng | 0.65 | 0.67 | 0.66 | 0.66 | 0.66 | 0.66 | 0.66 |
| hmmer-milc-zeusmp | 0.68 | 0.67 | 0.68 | 0.68 | 0.68 | 0.68 | 0.68 |
| hmmer-namd-sjeng | 0.54 | 0.53 | 0.53 | 0.53 | 0.53 | 0.53 | 0.53 |
| hmmer-namd-zeusmp | 0.58 | 0.56 | 0.55 | 0.55 | 0.55 | 0.55 | 0.55 |
| hmmer-sjeng-zeusmp | 0.61 | 0.60 | 0.60 | 0.60 | 0.60 | 0.60 | 0.60 |
| lbm-mcf-milc | 0.74 | 0.73 | 0.69 | 0.72 | 0.69 | 0.69 | 0.72 |
| lbm-mcf-namd | 0.62 | 0.61 | 0.61 | 0.61 | 0.61 | 0.61 | 0.61 |
| lbm-mcf-sjeng | 0.59 | 0.61 | 0.61 | 0.61 | 0.61 | 0.61 | 0.61 |
| lbm-mcf-zeusmp | 0.64 | 0.66 | 0.65 | 0.65 | 0.65 | 0.65 | 0.65 |
| lbm-milc-namd | 0.76 | 0.76 | 0.74 | 0.75 | 0.75 | 0.74 | 0.75 |
| lbm-milc-sjeng | 0.77 | 0.76 | 0.77 | 0.77 | 0.77 | 0.77 | 0.77 |
| lbm-milc-zeusmp | 0.78 | 0.78 | 0.79 | 0.79 | 0.79 | 0.79 | 0.79 |
| lbm-namd-sjeng | 0.65 | 0.63 | 0.65 | 0.65 | 0.65 | 0.65 | 0.65 |
| lbm-namd-zeusmp | 0.67 | 0.68 | 0.68 | 0.68 | 0.68 | 0.68 | 0.68 |
| lbm-sjeng-zeusmp | 0.71 | 0.72 | 0.73 | 0.72 | 0.72 | 0.73 | 0.72 |
| mcf-milc-namd | 0.58 | 0.53 | 0.44 | 0.46 | 0.47 | 0.44 | 0.47 |
| mcf-milc-sjeng | 0.59 | 0.54 | 0.47 | 0.49 | 0.49 | 0.50 | 0.50 |
| mcf-milc-zeusmp | 0.61 | 0.56 | 0.47 | 0.51 | 0.50 | 0.47 | 0.54 |
| mcf-namd-sjeng | 0.45 | 0.45 | 0.42 | 0.44 | 0.44 | 0.42 | 0.44 |
| mcf-namd-zeusmp | 0.48 | 0.48 | 0.47 | 0.47 | 0.47 | 0.47 | 0.47 |
| mcf-sjeng-zeusmp | 0.49 | 0.51 | 0.45 | 0.48 | 0.46 | 0.45 | 0.50 |
| milc-namd-sjeng | 0.63 | 0.59 | 0.62 | 0.60 | 0.60 | 0.61 | 0.59 |
| milc-namd-zeusmp | 0.64 | 0.58 | 0.54 | 0.56 | 0.57 | 0.54 | 0.57 |
| milc-sjeng-zeusmp | 0.68 | 0.63 | 0.68 | 0.67 | 0.66 | 0.68 | 0.64 |
| namd-sjeng-zeusmp | 0.56 | 0.55 | 0.49 | 0.53 | 0.54 | 0.49 | 0.54 |

Table B.3. Harmonic Mean of Weighted IPC results of three-threaded benchmarks

| | STATIC | HILL | ARPA | HH1 | HH2 | HH3 | HH4 |
|---|---|---|---|---|---|---|---|
| hmmer-lbm-mcf | 0.41 | 0.39 | 0.33 | 0.35 | 0.35 | 0.33 | 0.36 |
| hmmer-lbm-milc | 0.72 | 0.74 | 0.75 | 0.75 | 0.75 | 0.75 | 0.75 |
| hmmer-lbm-namd | 0.65 | 0.64 | 0.62 | 0.62 | 0.62 | 0.62 | 0.62 |
| hmmer-lbm-sjeng | 0.66 | 0.67 | 0.67 | 0.67 | 0.67 | 0.67 | 0.67 |
| hmmer-lbm-zeusmp | 0.69 | 0.68 | 0.68 | 0.68 | 0.68 | 0.68 | 0.68 |
| hmmer-mcf-milc | 0.44 | 0.33 | 0.30 | 0.30 | 0.28 | 0.32 | 0.31 |
| hmmer-mcf-namd | 0.43 | 0.28 | 0.25 | 0.26 | 0.27 | 0.25 | 0.26 |
| hmmer-mcf-sjeng | 0.41 | 0.31 | 0.25 | 0.27 | 0.26 | 0.27 | 0.27 |
| hmmer-mcf-zeusmp | 0.47 | 0.31 | 0.28 | 0.28 | 0.28 | 0.28 | 0.28 |
| hmmer-milc-namd | 0.61 | 0.58 | 0.56 | 0.57 | 0.57 | 0.56 | 0.57 |
| hmmer-milc-sjeng | 0.61 | 0.63 | 0.61 | 0.62 | 0.62 | 0.62 | 0.62 |
| hmmer-milc-zeusmp | 0.66 | 0.63 | 0.64 | 0.63 | 0.63 | 0.64 | 0.63 |
| hmmer-namd-sjeng | 0.52 | 0.52 | 0.49 | 0.50 | 0.51 | 0.50 | 0.51 |
| hmmer-namd-zeusmp | 0.57 | 0.54 | 0.52 | 0.52 | 0.52 | 0.52 | 0.53 |
| hmmer-sjeng-zeusmp | 0.61 | 0.56 | 0.55 | 0.55 | 0.56 | 0.55 | 0.56 |
| lbm-mcf-milc | 0.69 | 0.73 | 0.67 | 0.71 | 0.65 | 0.67 | 0.72 |
| lbm-mcf-namd | 0.58 | 0.58 | 0.59 | 0.59 | 0.59 | 0.59 | 0.59 |
| lbm-mcf-sjeng | 0.54 | 0.56 | 0.57 | 0.56 | 0.55 | 0.55 | 0.56 |
| lbm-mcf-zeusmp | 0.61 | 0.65 | 0.63 | 0.64 | 0.63 | 0.63 | 0.64 |
| lbm-milc-namd | 0.74 | 0.75 | 0.74 | 0.75 | 0.75 | 0.74 | 0.75 |
| lbm-milc-sjeng | 0.76 | 0.76 | 0.77 | 0.77 | 0.77 | 0.77 | 0.77 |
| lbm-milc-zeusmp | 0.76 | 0.78 | 0.79 | 0.79 | 0.79 | 0.79 | 0.79 |
| lbm-namd-sjeng | 0.62 | 0.61 | 0.61 | 0.62 | 0.62 | 0.61 | 0.62 |
| lbm-namd-zeusmp | 0.65 | 0.66 | 0.67 | 0.67 | 0.67 | 0.67 | 0.67 |
| lbm-sjeng-zeusmp | 0.70 | 0.71 | 0.72 | 0.72 | 0.72 | 0.72 | 0.72 |
| mcf-milc-namd | 0.53 | 0.52 | 0.33 | 0.41 | 0.43 | 0.33 | 0.43 |
| mcf-milc-sjeng | 0.54 | 0.50 | 0.37 | 0.42 | 0.41 | 0.45 | 0.45 |
| mcf-milc-zeusmp | 0.57 | 0.55 | 0.37 | 0.48 | 0.44 | 0.37 | 0.53 |
| mcf-namd-sjeng | 0.45 | 0.44 | 0.28 | 0.42 | 0.42 | 0.29 | 0.42 |
| mcf-namd-zeusmp | 0.47 | 0.46 | 0.26 | 0.41 | 0.41 | 0.34 | 0.43 |
| mcf-sjeng-zeusmp | 0.49 | 0.49 | 0.29 | 0.44 | 0.36 | 0.29 | 0.48 |
| milc-namd-sjeng | 0.59 | 0.58 | 0.57 | 0.58 | 0.58 | 0.57 | 0.58 |
| milc-namd-zeusmp | 0.61 | 0.56 | 0.50 | 0.54 | 0.55 | 0.50 | 0.54 |
| milc-sjeng-zeusmp | 0.66 | 0.62 | 0.68 | 0.67 | 0.66 | 0.68 | 0.63 |
| namd-sjeng-zeusmp | 0.55 | 0.54 | 0.36 | 0.52 | 0.53 | 0.36 | 0.53 |

# APPENDIX C:  RESULTS OF FOUR-THREADED BENCHMARKS

Table C.1. IPC results of four-threaded benchmarks

| | STATIC | HILL | ARPA | HH1 | HH2 | HH3 | HH4 |
|---|---|---|---|---|---|---|---|
| hmmer-lbm-mcf-milc | 3.98 | 4.63 | 4.69 | 4.68 | 4.64 | 4.68 | 4.66 |
| hmmer-lbm-mcf-namd | 4.22 | 4.59 | 4.66 | 4.63 | 4.61 | 4.65 | 4.62 |
| hmmer-lbm-mcf-sjeng | 3.67 | 4.17 | 4.62 | 4.44 | 4.38 | 4.37 | 4.37 |
| hmmer-lbm-mcf-zeusmp | 4.34 | 4.84 | 4.91 | 4.89 | 4.89 | 4.91 | 4.88 |
| hmmer-lbm-milc-namd | 4.03 | 3.99 | 4.14 | 4.10 | 4.09 | 4.11 | 4.09 |
| hmmer-lbm-milc-sjeng | 3.55 | 3.58 | 3.71 | 3.68 | 3.67 | 3.70 | 3.67 |
| hmmer-lbm-milc-zeusmp | 4.11 | 4.15 | 4.32 | 4.27 | 4.25 | 4.30 | 4.27 |
| hmmer-lbm-namd-sjeng | 3.90 | 3.84 | 3.99 | 3.94 | 3.93 | 3.96 | 3.93 |
| hmmer-lbm-namd-zeusmp | 4.45 | 4.60 | 4.70 | 4.68 | 4.67 | 4.69 | 4.67 |
| hmmer-lbm-sjeng-zeusmp | 4.12 | 4.25 | 4.34 | 4.30 | 4.29 | 4.33 | 4.31 |
| hmmer-mcf-milc-namd | 4.31 | 4.56 | 4.41 | 4.54 | 4.50 | 4.52 | 4.55 |
| hmmer-mcf-milc-sjeng | 3.84 | 4.05 | 4.45 | 4.47 | 4.46 | 4.46 | 4.52 |
| hmmer-mcf-milc-zeusmp | 4.56 | 4.84 | 4.67 | 4.83 | 4.78 | 4.78 | 4.81 |
| hmmer-mcf-namd-sjeng | 3.88 | 4.00 | 4.56 | 4.56 | 4.55 | 4.56 | 4.56 |
| hmmer-mcf-namd-zeusmp | 4.63 | 4.83 | 4.85 | 4.83 | 4.82 | 4.84 | 4.83 |
| hmmer-mcf-sjeng-zeusmp | 4.22 | 4.47 | 4.65 | 4.77 | 4.75 | 4.76 | 4.78 |
| hmmer-milc-namd-sjeng | 3.92 | 3.89 | 3.99 | 3.98 | 3.96 | 3.97 | 3.97 |
| hmmer-milc-namd-zeusmp | 4.56 | 4.67 | 4.67 | 4.66 | 4.66 | 4.66 | 4.66 |
| hmmer-milc-sjeng-zeusmp | 4.20 | 4.33 | 4.36 | 4.35 | 4.33 | 4.35 | 4.33 |
| hmmer-namd-sjeng-zeusmp | 4.37 | 4.56 | 4.53 | 4.53 | 4.52 | 4.52 | 4.53 |
| lbm-mcf-milc-namd | 4.69 | 4.69 | 4.67 | 4.73 | 4.72 | 4.71 | 4.72 |
| lbm-mcf-milc-sjeng | 4.01 | 4.19 | 4.60 | 4.62 | 4.62 | 4.61 | 4.65 |
| lbm-mcf-milc-zeusmp | 4.74 | 4.95 | 4.80 | 4.91 | 4.93 | 4.87 | 4.94 |
| lbm-mcf-namd-sjeng | 4.14 | 4.18 | 4.78 | 4.54 | 4.75 | 4.52 | 4.48 |
| lbm-mcf-namd-zeusmp | 4.98 | 5.00 | 5.05 | 5.02 | 5.01 | 5.04 | 5.02 |
| lbm-mcf-sjeng-zeusmp | 4.31 | 4.63 | 4.78 | 4.87 | 4.87 | 4.82 | 4.76 |
| lbm-milc-namd-sjeng | 4.02 | 3.81 | 4.06 | 4.04 | 4.01 | 4.03 | 4.02 |
| lbm-milc-namd-zeusmp | 4.73 | 4.66 | 4.78 | 4.75 | 4.75 | 4.76 | 4.75 |
| lbm-milc-sjeng-zeusmp | 4.18 | 4.30 | 4.32 | 4.31 | 4.30 | 4.32 | 4.31 |
| lbm-namd-sjeng-zeusmp | 4.38 | 4.54 | 4.61 | 4.59 | 4.57 | 4.60 | 4.58 |
| mcf-milc-namd-sjeng | 4.54 | 4.59 | 4.49 | 4.61 | 4.58 | 4.55 | 4.57 |
| mcf-milc-namd-zeusmp | 5.03 | 5.06 | 4.83 | 5.02 | 5.02 | 4.95 | 5.03 |
| mcf-milc-sjeng-zeusmp | 4.70 | 4.87 | 4.73 | 4.83 | 4.78 | 4.81 | 4.83 |
| mcf-namd-sjeng-zeusmp | 4.91 | 4.97 | 4.89 | 4.99 | 5.00 | 4.96 | 4.98 |
| milc-namd-sjeng-zeusmp | 4.67 | 4.81 | 4.70 | 4.75 | 4.74 | 4.74 | 4.75 |

Table C.2. Average Weighted IPC results of four-threaded benchmarks

| | STATIC | HILL | ARPA | HH1 | HH2 | HH3 | HH4 |
|---|---|---|---|---|---|---|---|
| hmmer-lbm-mcf-milc | 0.58 | 0.55 | 0.52 | 0.52 | 0.51 | 0.52 | 0.53 |
| hmmer-lbm-mcf-namd | 0.48 | 0.46 | 0.46 | 0.46 | 0.46 | 0.46 | 0.46 |
| hmmer-lbm-mcf-sjeng | 0.48 | 0.48 | 0.47 | 0.48 | 0.48 | 0.47 | 0.48 |
| hmmer-lbm-mcf-zeusmp | 0.51 | 0.50 | 0.50 | 0.50 | 0.50 | 0.50 | 0.50 |
| hmmer-lbm-milc-namd | 0.62 | 0.60 | 0.60 | 0.60 | 0.60 | 0.60 | 0.60 |
| hmmer-lbm-milc-sjeng | 0.63 | 0.63 | 0.65 | 0.64 | 0.64 | 0.65 | 0.64 |
| hmmer-lbm-milc-zeusmp | 0.65 | 0.60 | 0.64 | 0.63 | 0.62 | 0.64 | 0.63 |
| hmmer-lbm-namd-sjeng | 0.54 | 0.53 | 0.53 | 0.53 | 0.53 | 0.53 | 0.53 |
| hmmer-lbm-namd-zeusmp | 0.55 | 0.54 | 0.54 | 0.54 | 0.54 | 0.54 | 0.54 |
| hmmer-lbm-sjeng-zeusmp | 0.58 | 0.57 | 0.58 | 0.58 | 0.57 | 0.58 | 0.58 |
| hmmer-mcf-milc-namd | 0.43 | 0.41 | 0.33 | 0.35 | 0.34 | 0.33 | 0.36 |
| hmmer-mcf-milc-sjeng | 0.44 | 0.40 | 0.35 | 0.37 | 0.36 | 0.35 | 0.40 |
| hmmer-mcf-milc-zeusmp | 0.47 | 0.43 | 0.36 | 0.41 | 0.38 | 0.36 | 0.40 |
| hmmer-mcf-namd-sjeng | 0.34 | 0.32 | 0.32 | 0.33 | 0.33 | 0.32 | 0.34 |
| hmmer-mcf-namd-zeusmp | 0.37 | 0.36 | 0.35 | 0.35 | 0.35 | 0.35 | 0.35 |
| hmmer-mcf-sjeng-zeusmp | 0.38 | 0.37 | 0.34 | 0.36 | 0.36 | 0.34 | 0.38 |
| hmmer-milc-namd-sjeng | 0.48 | 0.48 | 0.48 | 0.48 | 0.48 | 0.49 | 0.48 |
| hmmer-milc-namd-zeusmp | 0.51 | 0.46 | 0.47 | 0.47 | 0.47 | 0.47 | 0.47 |
| hmmer-milc-sjeng-zeusmp | 0.53 | 0.50 | 0.52 | 0.51 | 0.51 | 0.52 | 0.51 |
| hmmer-namd-sjeng-zeusmp | 0.43 | 0.43 | 0.42 | 0.42 | 0.42 | 0.42 | 0.42 |
| lbm-mcf-milc-namd | 0.57 | 0.54 | 0.48 | 0.50 | 0.50 | 0.48 | 0.51 |
| lbm-mcf-milc-sjeng | 0.56 | 0.53 | 0.48 | 0.50 | 0.50 | 0.49 | 0.52 |
| lbm-mcf-milc-zeusmp | 0.60 | 0.56 | 0.50 | 0.53 | 0.54 | 0.50 | 0.54 |
| lbm-mcf-namd-sjeng | 0.47 | 0.46 | 0.47 | 0.48 | 0.48 | 0.47 | 0.48 |
| lbm-mcf-namd-zeusmp | 0.50 | 0.50 | 0.50 | 0.50 | 0.50 | 0.50 | 0.50 |
| lbm-mcf-sjeng-zeusmp | 0.50 | 0.51 | 0.48 | 0.50 | 0.50 | 0.48 | 0.51 |
| lbm-milc-namd-sjeng | 0.58 | 0.55 | 0.56 | 0.57 | 0.57 | 0.56 | 0.57 |
| lbm-milc-namd-zeusmp | 0.60 | 0.57 | 0.56 | 0.56 | 0.56 | 0.56 | 0.56 |
| lbm-milc-sjeng-zeusmp | 0.62 | 0.59 | 0.61 | 0.60 | 0.60 | 0.61 | 0.60 |
| lbm-namd-sjeng-zeusmp | 0.52 | 0.53 | 0.53 | 0.53 | 0.53 | 0.53 | 0.53 |
| mcf-milc-namd-sjeng | 0.45 | 0.42 | 0.33 | 0.35 | 0.35 | 0.33 | 0.39 |
| mcf-milc-namd-zeusmp | 0.45 | 0.42 | 0.36 | 0.40 | 0.40 | 0.36 | 0.40 |
| mcf-milc-sjeng-zeusmp | 0.48 | 0.45 | 0.36 | 0.41 | 0.39 | 0.36 | 0.42 |
| mcf-namd-sjeng-zeusmp | 0.38 | 0.39 | 0.34 | 0.38 | 0.38 | 0.34 | 0.38 |
| milc-namd-sjeng-zeusmp | 0.49 | 0.46 | 0.46 | 0.46 | 0.46 | 0.46 | 0.46 |

Table C.3. Harmonic Mean of Weighted IPC results of four-threaded benchmarks

| | STATIC | HILL | ARPA | HH1 | HH2 | HH3 | HH4 |
|---|---|---|---|---|---|---|---|
| hmmer-lbm-mcf-milc | 0.47 | 0.34 | 0.29 | 0.30 | 0.29 | 0.29 | 0.31 |
| hmmer-lbm-mcf-namd | 0.43 | 0.27 | 0.28 | 0.28 | 0.28 | 0.28 | 0.28 |
| hmmer-lbm-mcf-sjeng | 0.42 | 0.32 | 0.25 | 0.29 | 0.29 | 0.25 | 0.29 |
| hmmer-lbm-mcf-zeusmp | 0.46 | 0.33 | 0.30 | 0.31 | 0.31 | 0.30 | 0.31 |
| hmmer-lbm-milc-namd | 0.56 | 0.53 | 0.52 | 0.52 | 0.52 | 0.52 | 0.52 |
| hmmer-lbm-milc-sjeng | 0.58 | 0.57 | 0.58 | 0.58 | 0.58 | 0.58 | 0.58 |
| hmmer-lbm-milc-zeusmp | 0.61 | 0.56 | 0.58 | 0.58 | 0.58 | 0.58 | 0.58 |
| hmmer-lbm-namd-sjeng | 0.50 | 0.48 | 0.48 | 0.48 | 0.48 | 0.48 | 0.48 |
| hmmer-lbm-namd-zeusmp | 0.52 | 0.48 | 0.48 | 0.48 | 0.48 | 0.48 | 0.48 |
| hmmer-lbm-sjeng-zeusmp | 0.56 | 0.51 | 0.52 | 0.52 | 0.52 | 0.52 | 0.52 |
| hmmer-mcf-milc-namd | 0.38 | 0.26 | 0.18 | 0.21 | 0.20 | 0.18 | 0.22 |
| hmmer-mcf-milc-sjeng | 0.38 | 0.31 | 0.21 | 0.24 | 0.22 | 0.21 | 0.26 |
| hmmer-mcf-milc-zeusmp | 0.42 | 0.30 | 0.22 | 0.27 | 0.25 | 0.22 | 0.27 |
| hmmer-mcf-namd-sjeng | 0.33 | 0.26 | 0.19 | 0.22 | 0.22 | 0.19 | 0.23 |
| hmmer-mcf-namd-zeusmp | 0.37 | 0.25 | 0.25 | 0.26 | 0.26 | 0.25 | 0.26 |
| hmmer-mcf-sjeng-zeusmp | 0.37 | 0.29 | 0.20 | 0.24 | 0.25 | 0.20 | 0.26 |
| hmmer-milc-namd-sjeng | 0.44 | 0.43 | 0.42 | 0.42 | 0.42 | 0.42 | 0.42 |
| hmmer-milc-namd-zeusmp | 0.48 | 0.43 | 0.45 | 0.45 | 0.44 | 0.45 | 0.44 |
| hmmer-milc-sjeng-zeusmp | 0.51 | 0.47 | 0.49 | 0.48 | 0.48 | 0.49 | 0.48 |
| hmmer-namd-sjeng-zeusmp | 0.42 | 0.39 | 0.40 | 0.40 | 0.40 | 0.40 | 0.40 |
| lbm-mcf-milc-namd | 0.51 | 0.46 | 0.29 | 0.41 | 0.40 | 0.30 | 0.44 |
| lbm-mcf-milc-sjeng | 0.51 | 0.49 | 0.28 | 0.34 | 0.32 | 0.32 | 0.39 |
| lbm-mcf-milc-zeusmp | 0.54 | 0.54 | 0.37 | 0.47 | 0.49 | 0.37 | 0.51 |
| lbm-mcf-namd-sjeng | 0.41 | 0.40 | 0.35 | 0.43 | 0.39 | 0.35 | 0.43 |
| lbm-mcf-namd-zeusmp | 0.45 | 0.44 | 0.46 | 0.45 | 0.45 | 0.46 | 0.45 |
| lbm-mcf-sjeng-zeusmp | 0.45 | 0.47 | 0.30 | 0.39 | 0.43 | 0.30 | 0.47 |
| lbm-milc-namd-sjeng | 0.53 | 0.50 | 0.53 | 0.53 | 0.53 | 0.53 | 0.53 |
| lbm-milc-namd-zeusmp | 0.56 | 0.53 | 0.54 | 0.54 | 0.53 | 0.54 | 0.54 |
| lbm-milc-sjeng-zeusmp | 0.59 | 0.57 | 0.59 | 0.59 | 0.59 | 0.59 | 0.59 |
| lbm-namd-sjeng-zeusmp | 0.47 | 0.48 | 0.50 | 0.49 | 0.49 | 0.50 | 0.49 |
| mcf-milc-namd-sjeng | 0.41 | 0.41 | 0.19 | 0.27 | 0.24 | 0.19 | 0.36 |
| mcf-milc-namd-zeusmp | 0.42 | 0.40 | 0.25 | 0.39 | 0.40 | 0.25 | 0.40 |
| mcf-milc-sjeng-zeusmp | 0.45 | 0.43 | 0.25 | 0.36 | 0.32 | 0.25 | 0.41 |
| mcf-namd-sjeng-zeusmp | 0.38 | 0.36 | 0.27 | 0.37 | 0.37 | 0.27 | 0.37 |
| milc-namd-sjeng-zeusmp | 0.46 | 0.44 | 0.46 | 0.46 | 0.45 | 0.46 | 0.45 |

# REFERENCES

1. Tullsen, D. M., S. J. Eggers and H. M. Levy, “Simultaneous Multithreading: Maximizing On-chip Parallelism”, in 25 *Years of the International Symposia on Computer Architecture (Selected Papers)*, ISCA '98, pp. 533-544, ACM, New York, NY, USA, 1998, ISBN 1-58113-058-9.

2. Tullsen, D. M., S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo and R. L. Stamm, “Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor”, *SIGARCH Comput. Archit. News*, vol. 24, no. 2, pp. 191-202, May 1996.

3. Tullsen, D. M. and J. A. Brown, “Handling Long-latency Loads in a Simultaneous Multithreading Processor”, in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pp. 318-327, IEEE Computer Society, Washington, DC, USA, 2001, ISBN0-7695-1369-7.

4. Cazorla, F. J., A. Ramirez, M. Valero and E. Fernandez, “Dynamically Controlled Resource Allocation in SMT Processors”, in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pp. 171-182, IEEE Computer Society, Washington, DC, USA, 2004, ISBN0-7695-2126-6.

5. Choi, S. and D. Yeung, “Hill-climbing SMT Processor Resource Distribution”, *ACM Trans. Comput. Syst.*, vol. 27, no. 1, pp. 1:1-1:47, Feb. 2009.

6. Wang, H., I. Koren and C. M. Krishna, “Utilization-Based Resource Partitioning for Power-Performance Efficiency in SMT Processors”, *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 7, pp. 1150-1163, Jul. 2011.

7. Burke, E. K., M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Özcan and R. Qu, “Hyper-heuristics: A Survey of the State of the Art”, *Journal of the Operational Research Society*, vol. 64, no. 12, pp. 1695-1724, 2013.

8. Kiraz, B., A. S. Etaner-Uyar and E. Özcan, "Selection hyper-heuristics in dynamic environments", *Journal of the Operational Research Society*, vol. 64, no. 12, pp. 1753-1769, 2013.

9. Uludağ, G., B. Kiraz, A. c. Etaner-Uyar and E. Özcan, "A hybrid multi-population framework for dynamic environments combining online and offline learning", *Soft Computing*, vol. 17, no. 12, pp. 2327-2348, 2013.

10. Prakash, T. K. and L. Peng, "Performance Characterization of SPEC CPU2006 Benchmarks on Intel Core 2 Duo Processor", *ISAST Transactions on Computers and Software Engineering*, vol. 2, no. 2, pp. 36-41, 2008.

11. Eyerman, S. and L. Eeckhout, "Memory-level Parallelism Aware Fetch Policies for Simultaneous Multithreading Processors", *ACM Trans. Archit. Code Optim.*, vol. 6, no. 1, pp. 3:1-3:33, Apr. 2009.

12. Limousin, C., J. Sebot, A. Vartanian and N. Drach-Temam, "Improving 3D Geometry Transformations on a Simultaneous Multithreaded SIMD Processor", in *Proceedings of the 15th International Conference on Supercomputing*, ICS '01, pp. 236-245, ACM, New York, NY, USA, 2001, ISBN1-58113-410-X.

13. Vandierendonck, H. and A. Seznec, "Managing SMT Resource Usage Through Speculative Instruction Window Weighting", *ACM Trans. Archit. Code Optim.*, vol.8, no. 3, pp. 12:1-12:20, Oct. 2011.

14. Küçük, G., G. Uslu and C. Yeşil, "History-Based Predictive Instruction Window Weighting for SMT Processors", in *International Supercomputing Conference (ISC)*, ISC '2014, 2014.

15. Fisher, H. and G. L. Thompson, "Probabilistic Learning combinations of local job-shop scheduling rules", in J. F. Muth and G. L. Thompson (editors), *Industrial Scheduling*, pp. 225-251, Prentice-Hall, Inc, New Jersey, 1963.

16. Burke, E. K., M. Hyde, G. Kendall, G. Ochoa, E. Özcan and J. R. Woodward, “A Classification of Hyper-heuristics Approaches”, in M. Gendreau and J.-Y. Potvin (editors), *Handbook of Metaheuristics*, vol. 57 of *International Series in Operations Research & Management Science*, chap. 15, pp. 449-468, Springer, 2nd edn., 2010.

17. Özcan, E., B. Bilgin and E. E. Korkmaz, “A comprehensive analysis of hyper-heuristics”, *Intelligent Data Analysis*, vol. 12, no. 1, pp. 3-23, 2008.

18. Cowling, P., G. Kendall and E. Soubeiga, “A Hyperheuristic Approach to Scheduling a Sales Summit”, in E. Burke and W. Erben (editors), *Practice and Theory of Automated Timetabling III*, vol. 2079 of *Lecture Notes in Computer Science*, pp. 176-190, Springer Berlin Heidelberg, 2001, ISBN 978-3-540-42421-5.

19. Nareyek, A. ,”Choosing Search Heuristics by Non-Stationary Reinforcement Learning”, in M. G. C. Resende and J. P. de Sousa (editors), *Metaheuristics: Computer Decision-Making*, chap. 9, pp. 523-544, Kluwer, 2003.

20. Özcan, E., M. Misir, G. Ochoa and E. K. Burke, “A Reinforcement Learning - Great-Deluge Hyper-heuristic for Examination Timetabling”, *International Journal of Applied Metaheuristic Computing*, vol. 1, no. 1, pp. 39-59, 2010.

21. Bai, R., J. Blazewicz, E. K. Burke, G. Kendall and B. McCollum, “A simulated annealing hyper-heuristic methodology for flexible decision support”, 4OR, vol. 10, no. 1, pp. 43-66, 2012.

22. Lehre, P. K. and E. Özcan, “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*, FOGA XII '13, pp. 97-104, ACM, New York, NY, USA, 2013, ISBN 978-1-4503-1990-4.

23. Kheiri, A., E. Özcan and A. J. Parkes, "A Stochastic Local Search Algorithm with Adaptive Acceptance for High-school Timetabling", *Annals of Operations Research*, under review.

24. Asta, S., D. Karapetyan, A. Kheiri, E. Özcan and A. J. Parkes, "Combining Monte-Carlo and Hyper-heuristic methods for the Multi-mode Resource-constrained Multi-project Scheduling Problem", *Journal of Scheduling*, underreview.

25. Cowling, P., G. Kendall and E. Soubeiga, "Hyperheuristics: A Tool for Rapid Prototyping in Scheduling and Optimisation", in S. Cagnoni, J. Gottlieb, E. Hart, M. Middendorf and G. Raidl (editors), *Applications of Evolutionary Computing: Proceedings of Evo Workshops 2002*, vol. 2279 of *Lecture Notes in Computer Science*, pp. 269-287, Springer-Verlag, Kinsale, Ireland, 3-4 April, 2002.

26. Ross, P., "Hyper-heuristics", in E. K. Burke and G. Kendall (editors), *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, chap. 17, pp. 529-556, Springer, 2005.

27. Sharkey, J. J., D. Ponomarev and K. Ghose, "M-SIM: A Flexible, Multithreaded Architectural Simulation Environment", Tech. Rep. CS-TR-05-DP01, Department of Computer Science, State University of New York at Binghamton, 2005.