ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Power Aware Scheduling on Multicore Systems

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΦΟΙΒΟΣ Κ. ΦΙΛΙΠΠΟΠΟΥΛΟΣ

Επιβλέπων : ΝεκτάριοςΚοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2013

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Power Aware Scheduling on Multicore Systems

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΦΟΙΒΟΣ Κ. ΦΙΛΙΠΠΟΠΟΥΛΟΣ

Επιβλέπων : ΝεκτάριοςΚοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από τηντριμελήεξεταστική επιτροπή την ….Ιουλίου 2013.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Νεκτάριος Κοζύρης	Δημήτριος Σούντρης	Δημήτριος Φωτάκης
Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.	Λέκτορας Ε.Μ.Π.

Αθήνα, Ιούλιος 2013

.....

Φοίβος Κ. Φιλιππόπουλος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Abstract

Energy consumption of modern computing devices is becoming an increasingly important topic, especially for battery-powered mobile devices that run on reserved power. As the progress in the field of battery capacity seems unable to follow the increase in processors power needs for performance, power aware scheduling problem has been a recent issue, as it could have a vital role on portable devices running life. Recent commodity processors support multiple operating points running under various supply voltage levels, giving programmers the ability to adjust their system power consumption level according to their current needs. Consequentially, the Dynamic Voltage-Frequency Scaling (DVFS) has become a popular technique and several scheduling algorithms have been developed. Those algorithms are aiming to propose ways to reduce power consumption by imposing appropriate frequency and voltage levels to the system, in order to avoid unnecessary energy expenses.

If the Operating System (OS) is aware of the power consumption on the various processes within the system, it can schedule processes based on the constrains derived by the thermal analysis and the remaining power of the system. In addition, OS can balance the resource allocation of each process to remain within a given power envelope. However, obtaining the processor and the system power consumption is a non-trivial task. Existing power meters generally report only the power consumption on the whole system and are unable to provide detailed information for each processor individually. As a result it is very hard to expose a task's runtime power consumption, if multiple tasks are running in the system at the same time. The estimation of the power consumption on the thread level for every running process is a crucial requirement in designing power efficient schedulers.

In this work we analyze the power consumption of the target system, running a Non-Uniform-Memory-Access (NUMA) processor, and formulate a single power consumption model. Then, we examine the relationship between application's scalability and its power consumption by running our benchmark suite with different thread counts. The importance of the frequency scaling (DVFS) techniques is explored by measuring the performance of each benchmark on all the available frequencies supported by the system. We use Energy Delay Product (EDP) and Energy Delay Squared Product (ED^2P) as metrics to evaluate our results and create Pareto graphs to reflect our benchmark's power profile. We choose a suite that includes benchmarks with different characteristics regarding their needs in memory and CPU and use them to compare different proposed scheduling policies. We attempt to reduce the power consumption of the benchmark applications by applying the previous results on them. A significant reduction on the power consumption is shown. Finally, we examine techniques to reduce the cache and background memory conflicts and propose a memory balancing power aware scheduling algorithm.

Keywords

Multicore Systems, NUMA architecture, scheduling, gang scheduling policies, applications/parallel applications, energy consumption, energy optimization.

Acknowledgments

For the fulfillment of the current work I would like to express my gratitude to professor N. Koziris for giving the great opportunity to work in CSLab. Also I would like to thank Dr. K.Nikas, Dr. G. Goumas and Dr. N. Anastopoulos for their continuous support and guidance throw my study and research and for their motivation, immense knowledge, and insightful comments during the whole study process. Moreover, I would like to thank all the members of the CSLab for their help and useful advices that helped me deal with problems and complete my research.

Table of Contents

1 Introduction

1.1 Problem definition

In the previous decade, computer architects were focused on designing simple uniprocessor systems. Performance improvements were obtained by increasing the operating frequency and by designing fat cores that could better exploit instruction-level parallelism (ILP). . However, power constraints and heat dissipation problems have caused a shift in the design paradigm. Nowadays the focus is on designing multicore systems that exploit thread level parallelism. In these systems cores are sharing resources, ranging from functional units to different levels of memory hierarchy, under the same power source. That enforced software developers to abandon serial programming and adapt parallel programming methods in order to obtain a better use of the currently available chip multi-processors (CMPs). Parallel execution systems gave system developers the opportunity to write multithreaded applications that benefit from the simultaneous thread execution and obtain better system usage and significant performance increase.

In order to deal with the problem of the allocation of scarce resources, a significant amount of research focuses on developing efficient OS task scheduling algorithms. The first approaches focused on sharing the available resources across the running processes for discrete time quanta. Although, as these time-sharing policies scheduled only one task for a time quantum, they left the system's cores underutilized for significant periods of time. So, many space-sharing methods have been suggested, that co-schedule different tasks in the same time quantum to fill the available resources. These approaches involve a wide range of performance criteria, such as execution time, to provide efficient schedules and increase the overall systems performance. The majority of these operating system schedulers treat the cores of a multiprocessor chip as distinct physical processors that have no knowledge about other cores running simultaneously under the same resources. Their goal is to take advantage of the benefits that multithreaded applications can gain, while running in multicore platforms, and produce efficient schedules.

Nowadays, the previous scheduling approaches that focus on the overall performance are no longer capable to fill the modern systems' needs. Many modern systems, such as portable devices that run under limited battery power or large server farms that run on a certain budget, need to execute under low power consumption rates, in order to extend their running life or lower their expenses. As a result, research now focuses on using scheduling in such ways as to ensure low power consumption, without of course totally sacrificing the performance of the system. Hence, managing the power consumption and the performance of processors became an important aspect of the chip design and consequentially power-aware scheduling became a hot topic of research.

The main objectives of power aware scheduling are:

- lowering processor's power consumption level

- maintaining the system within an allowable power envelope
- supporting hot-spot elimination and
- balancing the power consumption across processors.

Reducing the total energy consumption for a given series of multithreaded applications running on a CMP is the problem studied in this work. The main objective is to schedule a given list of processes to be executed in a power saving way without drastically affecting the overall performance of the system. To achieve our objective, this study creates power profiles for the selected benchmarks and classifies them according to their scaling ability and their memory requirements. Finally, this work deals with the problem of space sharing to reduce the conflicts due to the sharing of resources, such as the memory bus and the Last Level Caches (LLC).

1.2 Motivation

Since modern supercomputers became popular and widely used, scheduling has been a topic of interest in a large number of research papers. The main objective of a scheduling algorithm is to share a system's resources between different processes that currently run on it. As mentioned before, the first scheduling approaches aimed to produce simple time-sharing schedules for the given for execution tasks. The majority of the used systems contained simple uniprocessors and as a result the schedulers focused on sharing the whole resources across running tasks for certain time quanta. So, they only produced time-sharing schedules mainly based on the waiting time of each task, in order to decide which one would be able to use the available resources the next time quantum.

As multiprocessor systems arrived and became widely popular, time-sharing scheduling methodologies were unable to provide efficient solutions to the scheduling problem. Researchers focused on space-sharing techniques, along with time-sharing, in order to schedule more than one processes at a time and increase the overall system's performance. Several scheduling algorithms were created, proposing different ways to distribute the running tasks among the system's available CPUs and schedule them simultaneously in order to complete the execution of a given workload of tasks faster. Although these methods provided efficient schedulers, the power limits enforced computer architects to create processors that contain more than one cores that share certain resources (memory hierarchies) and run under the same power source (chip multiprocessors CMPs).

As a result several problems occurred to the existing scheduling policies. Schedulers, in order to run on the new machines, treated cores as distinct processors and apply on them the already used scheduling methods. This agnostic resource aware approach led co-scheduled tasks to compete for the shared resources, such as the last level caches and the memory bus, and in many cases had catastrophic effects on the whole execution. For that reason, scientific research turned on creating contention aware scheduling policies that use different performance criteria to ensure that all the co-scheduled processes run efficiently on the system. Over the past few years, research in performance optimization on CMPs has gone a long and accomplished great results.

On the other hand, research on the aspect of power consumption in modern CMPs has been mostly neglected. Power aware scheduling aims to reduce the energy consumption for the execution of a given workload of tasks, without significantly affect the overall systems performance. In most cases there is a trade-off between performance and energy consumption. When a system runs in high frequency values it would report great power consumption and small execution times, while a system running in low frequencies would result in less power consumption but high execution times that may increase the total energy consumption. Generally power aware scheduling policies' target is to provide the lowest possible energy consumption for executing a certain job, while staying above an average performance.

Nowadays, reducing energy consumption has become a necessity in many different cases and power aware scheduling has become a real issue. Power saving is necessary both for small, embedded devices and large computer clusters and servers. Embedded systems include lots of different, widely spread portable devices that run a multicore processor under reserved battery power and are in constant need to save power, in order to run as long as possible. On the large scale reducing energy consumptions also definitely beneficial for computer clusters or large server farms that consume significant amounts of energy, where power reduction could lead in lowering their running cost and prevent them from using expensive cooling equipment. In both domains the excessive power consumption could result in high thermal dissipation that could be harmful for the devices.

For all these reasons the industry started producing chips that support DVFS and thermal monitoring policies in order to adjust frequency and voltage level of the on-chip cores when they remain underutilized, or lower voltage levels when thermal monitor reports that system temperature exceed an undesired threshold. The current focus for the researchers is on combining performance with power aware scheduling strategies to balance their system between good performance and small energy expense.

1.3 Goal of this work - Contribution

In order to deal with the problem of power aware scheduling on multicore systems there are some issues that must be addressed. First of all, in order to obtain power measurements we use an Intel Sandy Bridge processor, which is based on the Non-Uniform Memory Access (NUMA) architecture, so it is necessary to obtain detailed information about the number and the organization of the available CPUs, along with the organization of the memory subsystem. Also, we need to deal with the memory allocation issues on a NUMA machine and study scalability of applications when the number of threads exceeds the available cores in a single package and placement issues due to interconnection between different packages. Moreover, we introduce a simple power consumption model for our system by studying the power consumption of the cores, the Dram and the memory controllers.

Furthermore, we use Energy Delay Product (EDP) and Energy Delay Squared Product (ED^2P), in order to study the effects of scaling techniques on reducing the energy expense to run a process and the influence of dynamic frequency scaling on reducing the total power consumption without sacrificing performance in the cases of memory-bound applications. Based on the previous, we determine a power profile for the NAS and Polybench benchmarks. Then we use these results on widely used scheduling policies, such as gang scheduling and bin-packing gang scheduling, and report a significant reduction on the energy consumption.

The rest of the thesis deals with contention issues on scheduling. We study the importance of co-scheduling applications with different characteristics in order to reduce the number of conflict on the shared caches and the memory bus. We divide our applications on categories according to their behavior and memory needs and we design a scheduler to co-schedule different applications in the same time quantum in order to produce a power and performance efficient scheduler and finally we compare different scheduler implementations according to their fairness and the overall system throughput.

2 Background

2.1 Scheduling – single processor system approaches

Since multitasking systems appeared, the schedulers have become a vital part of every modern operating system. In computer science, scheduling is defined as the method by which threads, processes or data flows are given access to system resources, such processor time, memory access or communications bandwidth [1]. This is usually done to distribute workloads across the system resources effectively or to achieve quality of service. The need for a scheduling algorithm in every operating system arises from the requirement for most modern systems to execute more than one process at a time and transmit multiple flows simultaneously (multiplexing).

A scheduler's main target may differ from one system to another. Although usually its major target is to achieve a combination of high system throughput, small system latency and fairness for the running processes. Throughput represents the total number of processes that the system is able to complete their execution in a certain time unit. Latency illustrates the system response time, which is the total time between a job submission and its completion and finally, fairness represents the total amount of time a ready for execution process would have to wait in a queue due to unavailable system resources to run. In practice, most of the times these goals conflict (for example higher throughput results in small latency or fairness), thus the scheduler should be able to take crucial decisions and implement suitable compromises among running tasks in order to fulfill the desired needs and objectives of the execution system.

Until recently, the excessive use of simple uniprocessor systems enforced OS schedulers to mainly focus on time multiplexing of tasks. As there was only one processing unit available in the system, every ready for execution process was added to a run-queue and the scheduler decided the amount of time to be make the system resources available to it, based on several different factors, such as waiting time and priority of each process. Linux earlier schedulers were implemented using an algorithm with O(n) complexity, in order to decide which task to be scheduled next [2]. In this type of scheduler, the time it takes to schedule a task is a function of the number of tasks in the system, so the more tasks (n) are active on the system, the longer it takes to schedule a task. As a result these schedules lacked scalability, because at very high workloads the processor could spend more time deciding which process to schedule next and less time to execute the processes themselves.

2.2 Scheduling in multicore systems

Uniprocessor systems were followed by symmetric multiprocessing systems (SMPs). SMP is a multiprocessor architecture that consists of multiple identical processors that connect to a common shared memory. Each processor of a SMP is independent from the others and the only contention point between them is the shared interconnection network to the memory. The Linux scheduler still used one run-queue for the SMP, which meant that every task could be scheduled on any processor of the system. This scheduling policy may have been effective for load balancing but it created problems regarding to caches. For example, if a task was executed on cpu-1 and allocated all its memory on that processor's cache, moving the task from cpu-1 to cpu-2 would require moving all its data to from one cache to the other. The prior scheduler also used a single run-queue lock so, in an SMP system, the act of choosing a task to execute locked out any other processors from manipulating the run-queues. This resulted on idle processors waiting a release of the queue lock and decreased efficiency.

Because of the importance of the task scheduling problem on multiprocessing systems lot of research was made by the computer science community and Linux was able to develop a completely O(1) algorithm for wakeup, context-switch, and timer interrupt scheduling decisions [3]. O(1) scheduler used run-queues consisting of priority lists for different priority processes and implemented interactivity heuristics to decide which process has the highest priority and should be scheduled next. So, the scheduler used a different run-queue for every processor of the SMP and a load-balancing algorithm to fairly distribute the load among the available CPUs. As a result this scheduler resolved the primary issues found on the O(n) scheduler.

Even though O(1) scheduler proved to be very successful it has been replaced by the Completely Fair Scheduler (CFS) as the scheduler of the Linux OS [4]. CFS uses a red-black tree to describe the "need" for cpu-time of every task in every processor's run-queue instead of keeping priority lists. The scheduler keeps for every running process its waiting time and decides which one will run for a time-quantum on the next available CPU according to the highest waiting time. That policy proved to be very efficient on improving the systems overall performance and its used by many modern computing systems.

In order to deal with the communication latency problem between different processors and reduce the chip's power needs computer architect turned from SMPs to chip multiprocessors (CMPs). Processors based on CMP architecture include multiple cores on a single chip, running under the same power supply and sharing the upper level of cache memory (usually the L3 cache). As a result this architecture is able to achieve faster on-chip communication between cores, but in addition to time-sharing cores among different running tasks it introduces the concept of space sharing, because multiple cores are trying to use the same shared resources (Last Level Cache, memory bus) at the same time.

Using the previous schedulers based on SMPs on the new architecture created a lot of new problems for the OS programmers to deal with. The existing scheduling

policies treated every core of a CMP as an independent processor, based on previous approaches, and space-sharing problem came to the surface. Different processes were competing for resources under the common shared memory and bus interconnection and affected each other even in catastrophic ways sometimes. As parallel programmers and OS schedulers has to deal with the resource contention problem. Depending on the mix of tasks that execute concurrently in the multiprocessor the level of contention can vary greatly. Mixing jobs that require heavy use of the memory resources could result in poor system performance. On the other hand choosing CPU intensive jobs with small memory needs to execute concurrently would result in great performance but also in an underutilized system, which is in most cases undesirable. So the scheduler has to make the right choices to avoid unpredictable performance behaviors of the running processes and ensure that the system would always perform over a significant performance baseline.

Research made on this important manner has shown that an efficient way to deal with the problem is execute threads of the same applications concurrently on the available cores, in order to increase the applications' throughput. Because multithreaded applications usually contain threads that communicate and share data, executing them simultaneously on the system cores could prevent a thread from waiting to send or receive messages from another sleeping thread and allow threads to use data pre-fetched by others of the same application.

However, modern OS schedulers, in order to achieve great responsive time for their users, lack the ability to treat threads efficiently. OS's threats every applications threads as independent running processes that run on the system and so they are unable to recognize them and force them to operate at the same time. Moreover schedulers used on modern operating systems does not use and available performance monitors to take scheduling decisions so they seem unable to apply resource aware policies that could be beneficial for the system.

In the last few years, researchers have made several approaches to resource aware co-scheduling techniques for CMPs and suggested many different policies. Their main goals are use the knowledge of their system's hardware to increase the overall throughput and to balance processes among the existing resources to produce power efficient schedules that would consume less power to get the desired job done.

2.3 Power Aware Scheduling – Current Approaches

Generally, modern CMPs contain power monitoring tools that report the actual energy usage of the chip, and also support DVFS mechanisms that could be used to change the total system's frequency, or in some cases the individual frequency of each running core. This gave researchers the opportunity to study the power consumption of their systems, as long as to design power aware scheduling strategies that could reduce the overall system's energy needs for executing a job. Research made on power aware scheduling usually aims at the following similar goals: The first

one is to estimate the execution power of different available thread counts of a parallel application based on performance metrics, such as cache misses and instructions retired, and use the best combination to fill the entire system in order to consume as less energy as possible to complete a requested job. The second goal is to balance the running processes among the system's cores to reduce conflicts in caches and the memory bus and run the system under a specified power envelope.

One of the first approaches on power-aware scheduling was made by Major Bhadauria and Sally McKee [5]. While creating a greedy scheduler for resource aware co-scheduling of applications, they introduced Normalized Thread Throughput per Watt as a metric ($\frac{NTT}{Watt} = \frac{Number\ of\ Instructions\ Retired}{Threads \times Execution\ Time \times Watts}$) and used it for extensive benchmark profiling, in order to find the thread count that maximizes the throughput per Watt in every used benchmark, and created a greedy bin-packing gang scheduler, that claimed to be very efficient for both performance and power. The scheduler is static as it creates gangs according to the power profiles of the applications that needs to be scheduled and does not make any changes during the execution, because it is unable to make real-time power estimations.

Estimating the power consumption for different number of threads is a very difficult manner, because the existing hardware only offers power monitors for the whole CMP socket and the DRAM. Cores share the same power planes and so it is not possible to measure the power consumption for each independent core. For that purpose Singh, Bhadauria and McKee implemented a power estimation algorithm [6] that could help the OS to estimate the real-time power consumption of every process without actually simulating it and make better real-time scheduling decisions. The main goal of the algorithm is to achieve accurate per-core estimates of multithreaded and multiprogrammed workloads on a CMP with shared resources (an L3 cache, memory controller, memory channel, and communication bushes) and a real-time power estimation, without the need for off-line benchmarking, in order to schedule task efficiently.

The proposed algorithm uses performance counters to capture the L2 cache misses (event1), the retired memory operations (event2), the retired instructions (event3) and the stalls (event4) on every CPU core and use the acquired information to predict the core and system power. It normalizes every event by dividing it with the total cycle count of the core ($r_i = {event_i}/{(cycle\ count)}$) and calculates the core power with the following power model: $P_{core} = p_0 + p_1 \times g_1(r_1) + \cdots + p_n \times g_n(r_n)$, where p_i are constants defined measured results of their experiments. They claim that model to achieve median errors of 5.8%, 3.9% and 7.2% for the NAS, SPEC-OMP, and the SPEC 2006 benchmark suites, respectively.

Moreover, they study the effects of temperature on the system power. Static power is a function of voltage, process technology and temperature, so increasing temperature leads to increasing leakage power and adds to total power. They monitored the temperature and power of the CMP and observe that they affect each other and not accounting for temperature could lead to increased error in power estimates. However, not all systems support temperature sensors on the die area, or

per core, so omitting information about temperature could be really hard to deal with in most cases.

Another important issue discussed in literature is balancing the available resources among all processes in order to minimize the system's power consumption rate and at the same time increase performance by avoiding unnecessary cache and bus conflicts. Balancing power consumption could be a critical design parameter for many modern data centers and enterprise environments as it is has a direct influence on the cost. On that scope Dhiman, Marchetti and Rosing introduced an algorithm for placing the running processes across the available resources in order to balance the overall machines power [7].

Their work is built on a virtualized environment called "vGreen", which is a multi-tiered software system to manage virtual machine (VM) scheduling across different physical machines (PMs) with the objective of managing the overall energy efficiency and performance. It is based on a client server model, where a central server perform the scheduling of VMs across the PMs. Every PM is referred as a virtual node and every VM contains several number of virtual CPUs.

From their experiments they indicate that co-scheduling VMs with similar characteristics is not beneficial from energy efficiency and power consumption point of view at high utilization rates. That's because when a PM is running similar processes it may result in undesirable cache conflicts and so it contributes to higher system energy consumption, since it runs longer. Then they use performance counters to count the following events: 1) Instructions Retired (INST), 2) Clock Cycles (CLK), 3) Memory accesses (MEM) and 4) CPU utilization (Util), and use them to estimate the MPC (MEM/CLK) and the IPC (INST/CK) for every VCPU and every VM. Finally, they propose an algorithm from balancing MPC, IPC and CPU utilization across their system.

For the purposes of their algorithm, in order to efficiently estimate the impact of the previous metrics on the VCPU power consumption and performance, they use weighted values of MPC and IPC: $wMPC = MPC \times CPUutil, wIPC = IPC \times CPUutil$, and then calculate the aggregate metrics for each VM by adding up the corresponding metrics of its consistent VCPUs. Also they specify thresholds for MPC, IPC and CPU utilization, which are representative of whether high values of these metrics are affecting the performance of the VMs. The algorithm runs at a constant time quantum to ensure that every VM runs in acceptable values for these metrics (under the threshold), which indicates that the MPC and IPC is balanced across all the virtual nodes in the system for better overall energy consumption and performance. If a VM exceeds the threshold the algorithm try to rearrange the VCPUs across the VMs to balance the overall system again.

A similar approach on the subject was made by Merkel and Bellosa[8]. In their research they also observed that in order to optimize a schedules runtime and expended energy, the main goal must me to avoid contention by combining tasks with different characteristics. Thus, co-scheduling memory-bound and CPU-bound applications together, is proven to be beneficial for the system, because applications do not waste time competing for system resources. In the extreme scenario that

nothing but memory-bound applications are available for scheduling they use frequency scaling policies to reduce the energy consumption without affecting the performance. So, they propose a policy for timeslice-based multitasking, multiprocessor scheduling, where frequency scaling is used only if contention cannot be avoided.

The policy is based on performance monitoring. Whenever the CPU executes a task for a timeslice the scheduler uses the processor's performance counters to determine the memory intention of the task by counting the number of memory transactions. Then the scheduler uses this characterization to sort the tasks in each processor's run-queue by the memory intension. The tasks of cores with even processor numbers are sorted descendingly, while the tasks of cores with odd processor numbers are sorted ascendingly, so that the scheduler is able to co-schedule tasks of different memory intensities at the same time quantum. To achieve this co-scheduling, the scheduler ensures that the cores process their run-queues synchronously. Moreover, to make sure that tasks of different memory intensity levels are available on each core, they employ a balancing mechanism that migrates tasks if needed.

Frequency scaling is used only in cases where contention cannot be avoided. On modern processors switching the frequency introduces delays in the order of microseconds, which is several orders of magnitude smaller than the granularity of scheduling, so the scheduler is allowed to select a suitable frequency on every task without introducing noticeable overhead. In their work they study the effects of frequency scaling on tasks with different memory bus utilization, by determining the values of Energy Delay Product (EDP) for different frequencies. They introduce a model where the EDP factor is calculated by the equation: $DP\ factor = 1.4 - 0.8x$, where x represents the bus utilization and check whether the average EDP factor of the tasks currently selected for execution on the cores is smaller than one. If so, the scheduler engages frequency scaling, else it disables it.

In conclusion, researchers that deal with power aware scheduling first of all focus on designing contention aware scheduling policies, which ensure that applications with different profiles and memory needs are co-scheduled together. Furthermore, they use DVFS methods to prevent the system from unnecessary energy expenses when dealing with memory intensive applications, and explore different ways to correlate performance metrics with the best execution frequencies.

3 Tools

3.1 Platform Characteristics – Analysis

3.1.1 Experimental Platform

The table below describes the characteristics of the platform on which we conducted our experiments. Our system contains an Intel Sandy Bridge family processor. It is a Non-Uniform Memory Access (NUMA) architecture processor that contains four sockets, and each one of them consisting of 8 cores. Every core has its own private L1 cache, two cores of the socket share a L2 cache and all core of the socket share the L3 cache. Every socket has its own memory node and the sockets use the QPI interconnection network to communicate with each other.

Platform Model: Intel(R) Xeon(R) CPU E5-4620	
name	Sandman
#Packages	4
#Cores/Package	8
#Threads/Core	2
CPU frequency	2.2GHz
CPU available frequencies	1.1-2.2GHz
L1i cache	32KB
L1d cache	32KB
L2 cache	256KB
L3 cache	16384KB
RAM	24 GB per package

Table 3.1.1-1 Experimental Platform

We executed our experiments on a Debian GNU/Linux system, with 3.7.10 kernel version. We also used gcc version 4.4.5 with O2 optimization level and OpenMp standard version 3.0 in order to compile Scaff..

3.1.2 Power Monitoring

For our experiments, Sandy Bridge offers 4 power monitors for every physical package, as shown in the figure below [14]. We can only measure package, core and DRAM power, as the graphics are not available on server parts.

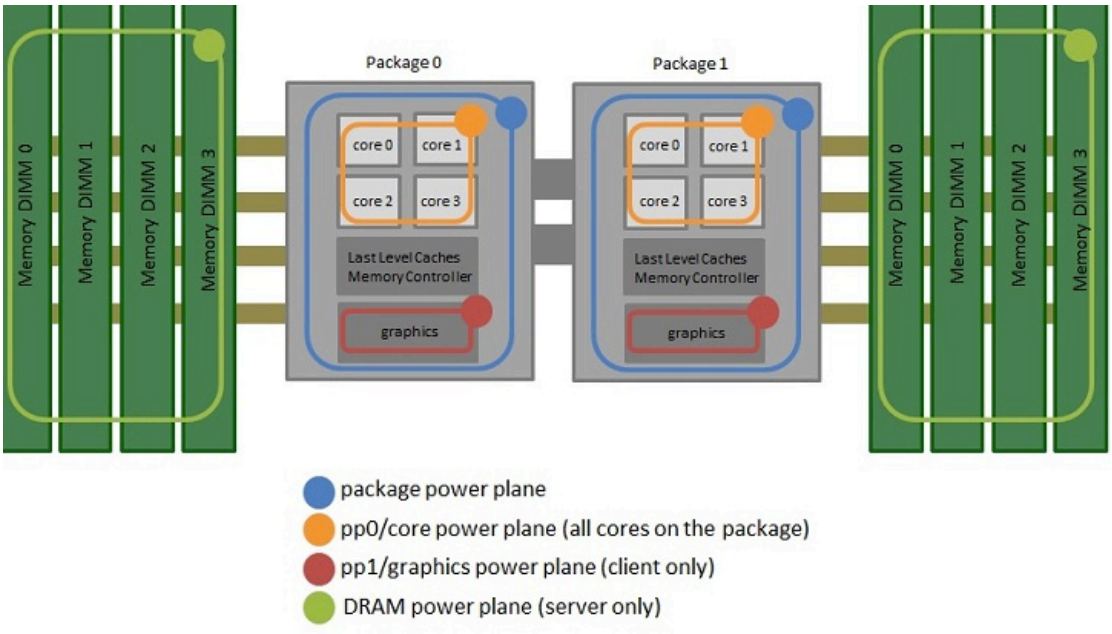


Figure 3.1.2-1 NUMA Power Monitoring

In our experiments we calculate the core, the uncore and the DRAM power. The core part contains package cores and L1 and L2 caches for each individual core in the package, the uncore part contains power consumed from Last Level Caches and Memory Controllers and the DRAM part contains power consumed only on the memory connected to the package.

For obtaining our measurements we read the information written on the following registers:

- MSR_PKG_ENERGY_STATUS: reports the measured actual energy usage of the whole package (core and uncore parts).
- MSR_PP0_ENERGY_STATUS: reports the actual energy usage on a power plane.
- MSR_DRAM_ENERGY_STATUS: reports measured actual energy usage on the DRAM.

Those are read-only registers provided by the Intel's chipset, which reports the actual energy usage for the package, the core and the DRAM domain. They are updated every 1msec and have a wraparound time of around 60 seconds when power consumption is high, so on our experiments we obtain samples every 30 seconds to avoid any data loss. The content of these registers is an unsigned long 64-bit integer.

To use the content of the above registers we also need related information stored in the MSR_RAPL_POWER_UNIT register. In this register there are exposed units for the power (expressed in Watts), energy (expressed in Joules) and time (expressed in seconds). Thus in order to calculate the actual core energy consumed

between 2 discrete times (t1, t2) we need to obtain the difference of the MSR_PP0_ENERGY_STATUS register's content between t2 and t1 and then multiply it with the energy units.

Finally we can use the MSR_PKG_POWER_INFO register to gain information about the average power usage limit of the package domain, the power limits of the package and the time window for our power limits.

Running a simple test on our system on a single 8-core package of our system we found that the values of the needed units are:

- *Power units = 0.125W*
- *Energy units = 0.00001526J*
- *Time units = 0.00097656s*

the package limitations are:

- *Package average power: 95.000W*
- *Package minimum power: 52.000W*
- *Package maximum power: 150.000W*
- *Package maximum time window: 0.046s*

and the power consumed from 1 core (core #0) sleeping 1 second was:

- *Package energy: 10.250595J consumed*
- *PowerPlane0 (core) for core 0: 1.681656J consumed*
- *DRAM energy: 2.230789J consumed*

3.1.3 NUMA Memory Allocation - Limitations

As mentioned before for the purposes of this work we use an Intel processor based on Intel's microarchitecture code name Sandy Bridge, in order to obtain information from the power monitoring registers included in such a processor. As the processor is built based on the Non-Uniform Memory Access (NUMA) design technique we need to study the NUMA characteristics NUMA is a computer memory design used in multiprocessing, where the memory access time depends on the memory location relative to a processor [10]. Under NUMA, a processor can access its own local memory faster than non-local memory (memory local to another processor or memory shared between processors). An example of NUMA memory architecture is shown in the figure below:

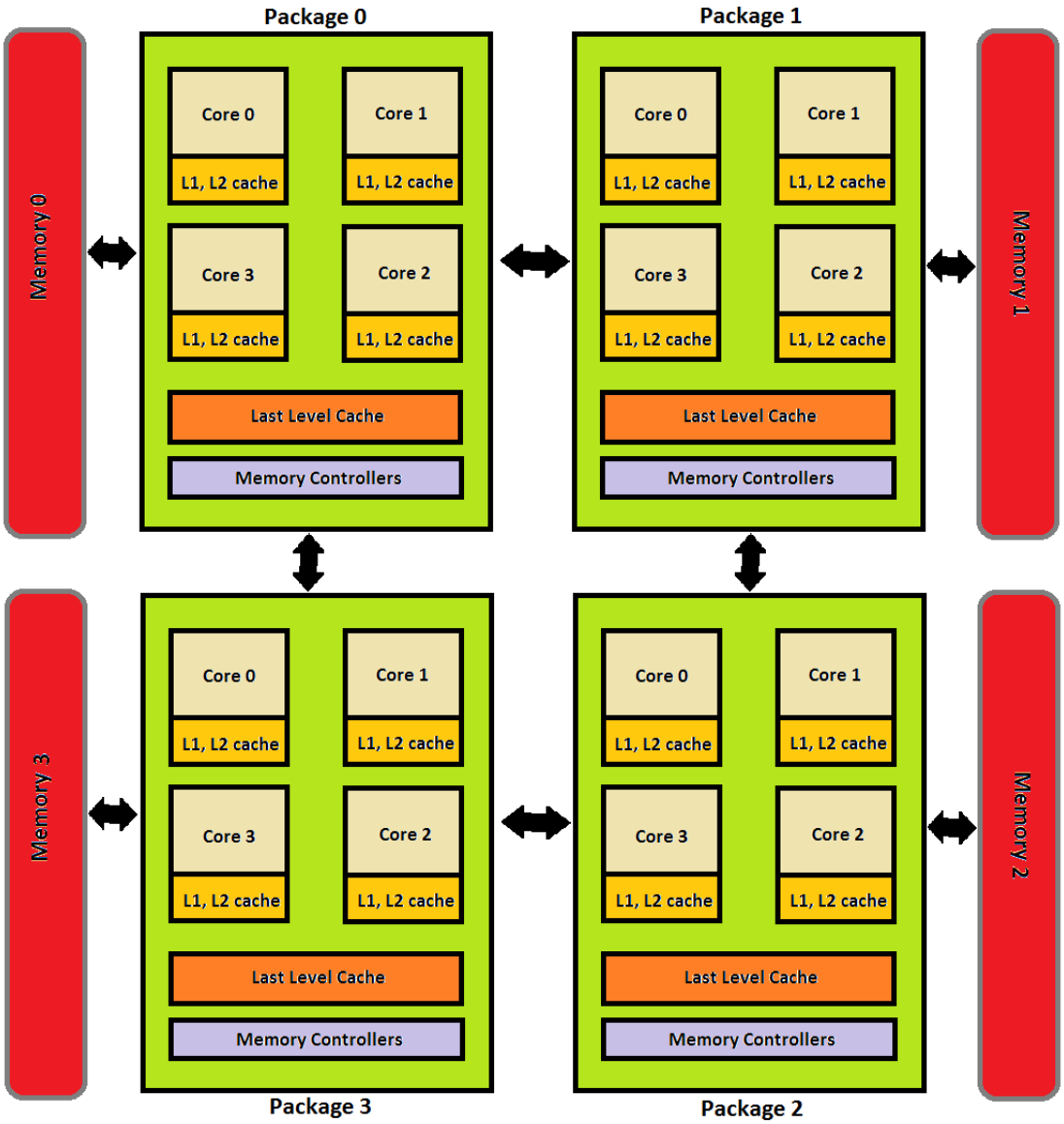


Figure 3.1.3 - NUMA Architecture

Modern CPUs operate considerably faster than the main memory they use and as a result, in many cases they may found themselves running on “idle” state as they have to stall while waiting for data to arrive from the memory. Multi-processor systems without NUMA make the problem considerably worse, as they frequently keep several processors in idle state waiting for data at the same time, notably because only one processor can access the computer's memory at a time. On the other hand, NUMA attempts to address this problem by providing separate memory for each processor, as shown in the figure above, in order to avoid the performance hit when several processors attempt to address the same memory node. Of course, not all data ends up confined to a single task, which means that more than one processor may require the same data. To handle these cases, NUMA systems include additional hardware interconnection to move data between memory banks of different packages. Thus, moving data between different NUMA packages slows down the processors of the involved packages and result in poor performance, so the overall speed increase

due to NUMA depends heavily on the nature of the running tasks and it's the software developers responsibility to develop and run parallel applications with NUMA-aware memory allocation in order to exploit the advantages of the architecture.

As mentioned before in a typical NUMA based multiprocessor system each package (or CPU) has its own memory, so maintaining cache coherence across the whole system's memory has a significant overhead. NUMA uses inter-processor communication between cache controllers to keep a consistent memory image when more than one cache stores the same memory location, allowing one package to transparently access memory connected to another package [11]. This mechanism solves the problem of cache coherence but it may result in poor performance when multiple processors attempt to access the same memory area.

In the NUMA architecture the memory access time is not constant. For example on the figure above, a core inside Package0 can access the local memory in bank 0 much faster than it can access memory connected to Package1 in bank 1, where only one hop over the interconnection is required. Moreover, accessing memory connected to Package 2 from the same core is even slower as it requires two hops over interconnection links. So it is obvious that the increased cost of accessing remote memory over local memory can affect the performance and the farther the requested data is stored the slower it would be available. In other words, software should try to allocate memory efficiently and increase the usage of local memory to result in better performance. For example, if we run a multithreaded application on packages 0 and 1 we must ensure that threads running on Package0 allocate and use data from Memory0 and threads running on Package1 use data from Memory1.

In order to deal with that, modern operating systems usually offer tools that help developers to control where each thread of their program will allocate its memory [12]. For example, Linux offers a library called "libnuma" that includes functions for allocating memory on specific NUMA-nodes. Well-designed NUMA-aware software carefully allocates memory and manages threads to maximize the local memory usage. However, it is a very difficult task to determine the topology of the system and allocate memory from specific memory banks to ensure that data is being manipulated by threads running only on the local package.

Moreover, another problematic case is running older software, not designed for NUMA machines, on a NUMA processor. Modern operating systems use virtual memory and give applications limited control over the mapping of virtual to physical memory. When an application allocates a memory block it is assigned a virtual memory region. The OS maps that virtual memory region to some physical memory location, but the OS typically retains a full control over when that happens or what physical memory range to use. Most of the existing operating systems use a "first touch" allocation policy, which means that when an application requests memory, that its virtual address is not mapped to any physical memory, the OS allocates a physical memory region and maps the virtual address to the physical range. The OS typically allocates physical memory from the same NUMA node as the core that executed the thread that first accessed the virtual memory block.

In order to study this policy let us consider 3 applications from the NAS parallel benchmarks with different characteristics. We choose bt.A (workload set class A), which is a block tri-diagonal solver, as a pseudo-application with significant memory needs, ep.A (embarrassingly parallel with small workload, class A) as a cpu-bounded application with small memory needs that could be meet by L1 and L2 caches, and is.C (Integer Sort, large workload class C) as an application that makes random memory accesses. We run each one alone with 4 threads and then change the placement of the running threads across the existing packages to report the differences in execution time for completing the job. We normalize the measured execution time over the single package's execution. Our NUMA processor contains 4 packages so every one has 2 neighbor packages and one distant, like the previous figure, so we obtain the following results:

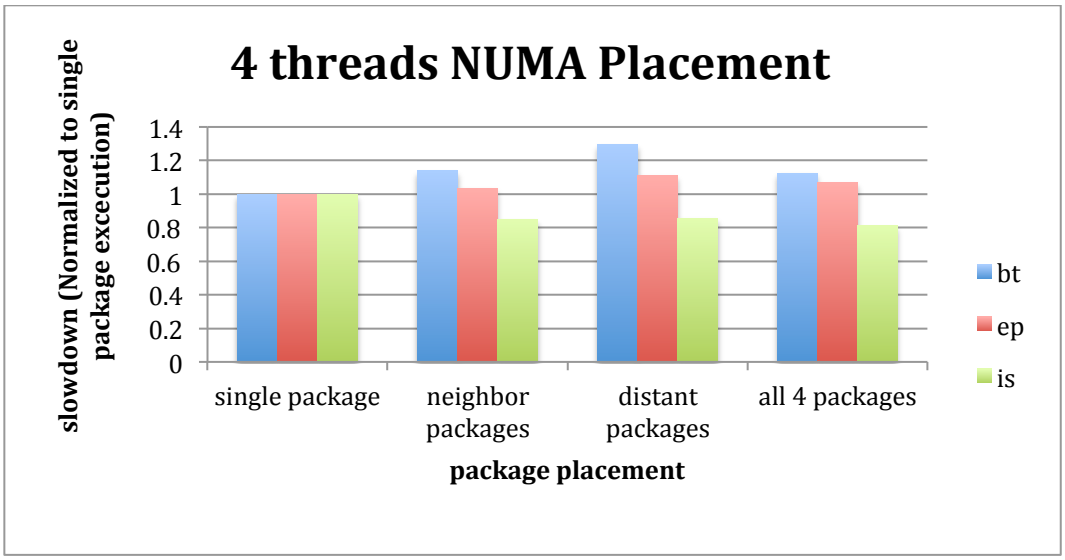


Chart 3.1.3-2 NUMA Placement Issues

Spreading the execution threads across packages is expected to lead in performance improvements, as each thread has more resources to run. For example placing 4 threads in 4 different packages instead of 1 would reduce cache conflicts between them, because each thread stores data in a different last level cache, and also reduce the completion among them for the memory bus, so the overall performance would be improved. However, in NUMA architectures that is not always the case. In the chart above we can observe that only is.C is able to gain from using more than one packages and report a speedup, while ep.A has almost the same execution time in every case and bt.A reports a significant slowdown when executed on multiple different packages.

That behavior is mainly a result of the NUMA-allocation policies. In an application like bt.A, where the first thread allocates a 3-dimensional array and the there is a parallel for loop that executes the tri-diagonal solver, all the useful data is "touched" at the beginning by the one thread and as a result of the Linux "first-touch" policy it is allocated on the memory node of this thread's package. So, as we can

observe from the above chart, running bt.A on neighbor packages or 1 thread on each package (which includes 2 neighbor and 1 distant package) results in a 1.1 slowdown rate, while running the same application on distant packages makes the execution more than 1.2 times slower. The same behavior but in a smaller amount is reported by ep.A, where the application's threads share less data and so the slowdown of the spreaded execution is below 1.1 in every case.

On the other hand, is.C, which allocates random data from the memory, succeeds in distributing the needed data between threads and results in performance improvements for "spreaded" execution. From the results above we observe that placing is.C threads in 2 packages, neighbor or distant, results in faster execution than placing all the treads inside the same package, while the best case scenario is using all 4 packages of our system in order to obtain the best performance for the application.

Another important problem is the way the NUMA memory organization and the dependencies between different threads of the same application affect the ability of a parallel application to scale efficiently. In order to deal with this issue we choose cg (with workload class B) from the NAS benchmarks, which is an application with great scaling ability on a Uniform Memory Access (UMA) processor system and execute it with 1 to 16 threads in both a UMA and a NUMA machine. We obtain the following results:

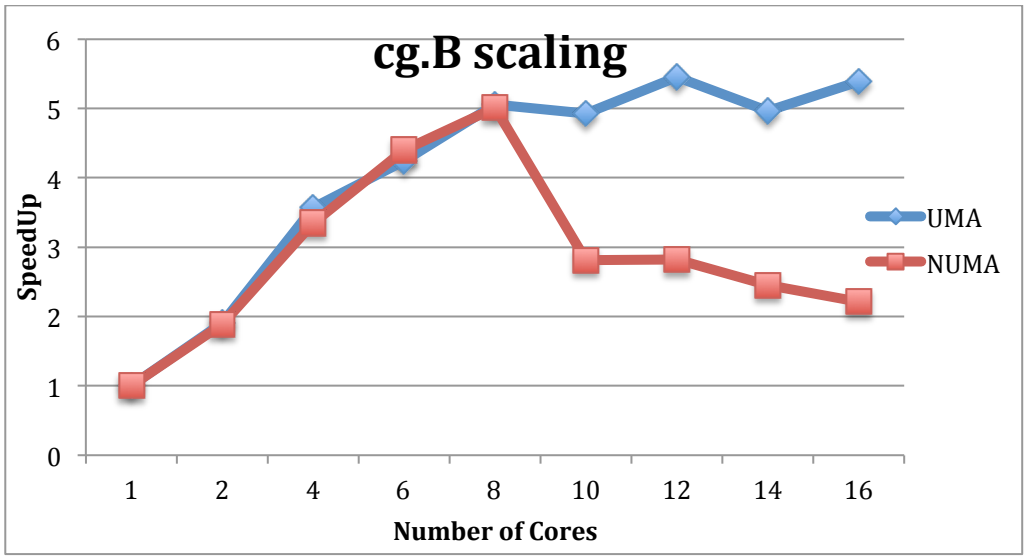


Chart 3.1.3-3 cg.B scaling between UMA and NUMA machines

In the chart above both UMA and NUMA contains 8 cores per package. Uniform Memory Access architectures contain multiple processors, with several cores each, but only one memory node. So every core has an almost constant memory access time based only on the competition for the shared memory bus. As we can see from the chart above that gives the chance to parallel applications with great scaling ability, like cg.B , to scale efficiently while using cores from multiple packages. On the other hand, NUMA allocation issues prevent the application from scaling when the number of threads exceed the available cores inside one single package (8 cores in

our case), because useful data is allocated only on one memory node and threads running outside that node's package waste a lot of time waiting for data.

In conclusion, NUMA architecture chips are designed to include individual memory nodes for every physical processor in order to reduce complexity and memory access time and provide a great boost to the overall performance of the system. Although, writing NUMA-aware code is a very difficult task for parallel software developers, because of the problems and limitations that may occur when running applications that are unaware of the system's topology, and random memory allocations of data may result in undesirable software performance slowdowns.

3.2 Scheduling Infrastructure - Scaff

For the purposes of this work we need handle processors performance monitoring, in order to read information from power and performance counters, and create different scheduler implementations to study their efficiency towards system's throughput and fairness. For that reason we modify and use Scaff runtime system provided by the CSLab.

3.2.1 Scaff Architecture

Scaff is a runtime system that orchestrates the execution of a workload of multithreaded programs. It operates at user-level, on top of Linux based systems, and its task is to let the user bypass the Linux scheduler and apply his own scheduling implementations on a workload of programs that need to be executed. Scaff consists of two main subsystems: the executor and the scheduler. The executor is responsible for handling events regarding the execution, such as creating or terminating a program and freezing or thawing programs in the commands of the scheduler, while the scheduler is responsible for making execution decisions about time and space sharing of applications among the existing resources.

The executor is always running on the system awaiting the appearance of various kinds of events, in order to trigger the appropriate function of the scheduler. These events are the creation and termination of a program, the completion of a time quantum and the scheduler decisions in order to execute a program or put it on "freeze" state to wait in the waiting list. On the other hand, the scheduler takes all the scheduling decisions about when a program will execute, on which cores and whether some programs will co-execute. On the following pages there are more information about the executor, the scheduler and the complete Scaff architecture, which is illustrated on the following figure:

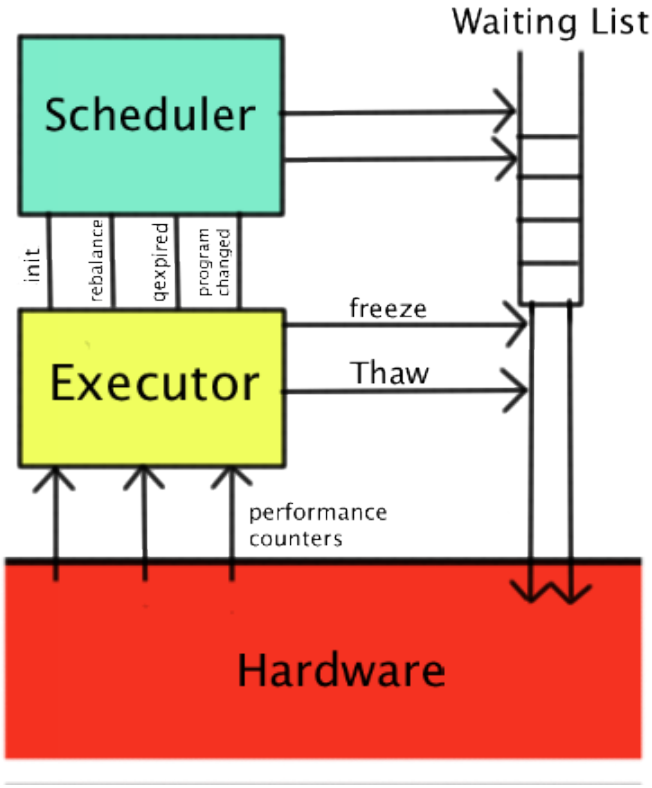


Figure 3.2.1-1 Scaff Architecture

Executor:

The executor keeps information about the programs of the workload and certain kinds of events that occur during the execution, in order to call the appropriate scheduler functions. Every program can be in one of the following states:

- WATING: waiting to arrive to the scheduler
- NEW: just arrived and is ready to start execution
- RUNNING: executing
- FINISHED: program has finished its execution

Programs of the workload as given as an argument to the executor, in a configuration text file, which also includes the time each program must start and the number of requested threads for its execution. Until their starting time comes, they are in WAITING state and then the NEW state begins in order to highlight to the scheduler that they are ready to be executed for the first time. A list (pnew_l) of programs in NEW state is given by the executor to the scheduler, so that it will begin their execution and add them to a list of handled by the scheduler programs. These programs that are executing and their execution is handled by the scheduler are in RUNNING state. Finally, a program enters the FINISHED state, either when it has finished its execution, or it got “killed” by another process. The executor adds all

programs in FINISHED state to a list (pfinished_l) and gives it to the scheduler, in order to remove finished programs from the execution schedule, while the executor cleans-up every structure that was created for handling and executing the finished programs.

Also, the executor keeps for every given program information about its cpuset, the number of cores that the program has requested for its execution, a shared memory area used for communication between the program and the scheduler, and the frequency requested for execution. The cpuset of a program is necessary, in order for the executor to decide on which cores the program will execute and which memory nodes is allowed to use for allocating memory pages, and the frequency tells the executor what frequency to apply on the execution cores, of course if the requested frequency is available for the system.

While a program is in RUNNING state, it can be either FROZEN, which means that it is stopped for the current time-quantum, or THAWED, which indicates that the program is currently running. So the executor needs to handle the two following events:

- EVNT_NEWPROG
- EVNT_QEXPIRED

EVNT_NEWPROG is created every time a program starts its execution and EVNT_QEXPIRED is created at the expiration of a time-quantum. Each event is associated with a timestamp, at which it must be processed by the executor. The executor uses a priority heap to keep truck of this events and during the execution it checks for events that their time has come to be processed and calls the appropriate scheduler function to handle them.

Scheduler:

A scheduler is used in order for Scaff to be able to manage the execution of the requested programs and handle the events that are created during the execution. The scheduler includes several functions to implement a specific scheduling policy and is responsible for sharing the available resources among the running programs, according to every programs demands and make important decisions about time and space sharing of running programs.

Every Scaff scheduler must implement the following functions:

- void *init(void): This function is called ate the beginning of the execution by the executor in order to initialize the scheduler. Its returned value is stored by the executor and used by the executor and the scheduler in later time intervals.
- void rebalance(void *sched_data): rebalance is called either when one or more programs are ready to be executed, so pnew_l is not empty and there are programs in NEW state, or, one or more programs are in the pfinished_l list

because the completed their execution and are now in FINISHED state. So rebalance is responsible for handling new programs, adding them in the execution schedule and giving them resources to run, and removing finished ones from the execution list and de-allocating their used resources and structures.

- voidqexpired(void *sched_data, structtimeval *now): this function is called whenever a time-quantum has finished. The scheduler must add aEVNT_QEXPIRED event in the priority heap, in order to implement time-sharing.
- intprog_changed(void *sched_data, aff_prog_t *prog, intnr_threads): this function is called whenever a program's requirements for resources have changed and it is responsible for handling the new information and make the necessary changes to the execution schedule.

3.2.2 Design and Operation of Scaff

The design of Scaff aims to assist a scheduler implementation that will interact with the workload of programs it is handling during execution time. It provides the means of communication between the two ends, and the necessary mechanisms to control the execution of programs.

In order to provide communication between the programs and the scheduler Scaff keeps information about every single program, the schedule and the system on which both are running on. The most significant piece of information that is kept is the number of cores that each program require for its execution as long as the number of cores that the scheduler allocated for it and the number of the available cores on the system. Moreover, Scaff must deal with the running programs' needs to send requests to the executor, as well as the need for synchronous communication. For that purpose it uses two pipes for every program, one for the program to make requests to the executor and another for the program to wait the executors response. So, whenever a program wants to make requests to the executor, it sends from the write-end of the pipe an identifier and waits for the executor to read the read-end of the pipe. This pipe is unique for all programs in order for the executor to keep a priority on the request arrived from different programs, and is stored in Scaff's structure describing the execution. Then the program is waiting on the read-end of another pipe for the executor to write some arbitrary value to the write-end, after it has processed the program's request. The second pipe is individual for every running program and is stored on the programs data.

When an execution begins, the first thing that Scaff must take care of is the initialization phase. This includes allocating the priority heap that keeps events during the execution, allocating a hash table, which maps the structures that describe

programs, with their process id's (pids), and creating the pnew_l and pfinished_l lists that are used to keep the new and the finished programs. Then Scaff must allocate an appropriate structure for every program of the workload that is to be executed. The workload is given as a command line argument in a configuration file, so Scaff parses the configuration file and allocates and initializes a structure (aff_prog_t) that is necessary in order to store all the useful data for every program. The main purpose of this structure is to keep stored a cpuset_t field, which is used as a handler for the program's cpuset and a pointer to the shared memory that will be used for communication with the executor for every program. It also contains information such as the requested cores and frequency that could be used by the scheduler during the execution.

After the new program's aff_prog_t structure is initialized, the executor will fork() a new process for it. The new process will then use execl()to begin the program's execution on a new shell. The executor will wait for the program to freeze itself, and then it attaches it to its new cpuset, which at first will contain all the cpus of the system. Finally it pushes on the heap an EVNT_NEWPROG event. The new program will remain in FROZEN state until it is time for this EVNT_NEWPROG to be handled. The scheduler will then take over responsibility for its execution.

After handling the programs include in the configuration file, Scaff needs to initialize a structure for the scheduler that will be used, which is given as an argument from the command line. Also it needs to install signal handlers to handle SIGCHLD and SIGTERM signals, for normal and unexpected program termination, respectively. A SIGCHLD would inform the scheduler for the normal termination of a running program and add the program in the pfinished_l list in order for the scheduler to deal with it and rearrange the existing execution schedule, and a SIGTERM signal will implement an execution error and will cause the execution to abort.

After the initialization phase is completed every program is in a WAITING state until its time comes to run for the first time, so it moves to the RUNNING state. Then the scheduler is responsible for deciding whether and whenever to "freeze" it or "thaw" it. While applications are executed by the scheduler, they are able to write to the write-end of Scaff's pipe, for programs communication, in order to make several requests to the executor. After a request the application remains blocked and waits for the executor to fulfill its request and write to its pipe an arbitrary value to unblock it and let it know that the request was filled. In order to relieve the applications from having to deal with the executor's specific implementation and data Scaff provides a function called affhook_region_notify(), which writes in the shared memory area and then sends the application's data structure (aff_prog_t) through the write-end of the executor's pipe.

Furthermore, during the execution Scaff's duties reduce to handling execution events, programs notifications and signals. The executor pops events from the priority heap until it becomes empty. Every event is associated with a timestamp that indicates the exact time quantum that must be handled by the executor. The executor selects the event with the highest priority and compares the current timestamp with the event's

timestamp and if its time to be processed has arrived the executor pops it from the heap and handles it. Handling these events requires different types of action to be taken depending on the nature of the event. For example handling an EVNT_NEWPROGevent requires adding a program to the pnew_l, while handling an EVNT_QEXPIRED event indicates that a time-quantum expired and requires calling the qexpired() function from the scheduler.

When no other events are to be handled for the current time being, if there are new programs, so that thepnew_l list is not empty, the scheduler's function rebalance() is called. In this procedure the scheduler handles scheduling issues that may appear by a new or a finished program and the returns the amount of time until the next event in the heap must be handle. Until that point of time the executor waits for program notifications. These notifications are in fact an aff_prog_t structure of the program that makes a request and a number of requested threads written on the shared memory. If implemented, the prog_changed() function is called, so that the scheduler takes into account the program's requests.

Finally, the execution stops when there are no programs left in any available state, so that Scaff exits. For communication between Scaff and the user before exiting Scaff uses functions from a stats.c file included in the implementation to report statistics about the whole execution in an exit test file that is given as a command line argument. Scaff is able to write data in this file during the execution time too in order to report the scheduler's decisions and information related to the programs statistics during the execution.

3.3 Benchmarks

For the purposes of this work we use benchmarks from the Polybench [16] and the NAS Parallel Benchmark [15] suites. In this section we present the two suites and the benchmarks we selected for our experiments.

PolyBench: is a collection of benchmarks containing static control parts. The purpose is to uniform the execution and monitoring of kernels. PolyBench features include:

- A single file, tunable at compile-time, used for the kernel instrumentation. It performs extra operations such as cache flushing before the kernel execution, and can set real-time scheduling to prevent OS interference.
- Non-null data initialization, and live-out data dump.
- Syntactic constructs to prevent any dead code elimination on the kernel.
- Parametric loop bounds in the kernels, for general-purpose implementation.
- Clear kernel marking, using #pragma scop and #pragma end scop delimiters.

The chosen benchmark applications explored in the current work are presented below:

1. Cholesky: In linear algebra, the Cholesky decomposition is a decomposition of a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose. It is an efficient tool for solving systems of linear equations.
2. atax: This application includes algorithms for matrix transpose and vector multiplication.
3. gemver: This application includes algorithms for vector multiplication and matrix addition.
4. syr2k: This application performs one of the symmetric rank 2k operations. It is given by the formula: C := alpha*A'*B + alpha*B'*A + beta*C where alpha and beta are scalars, C is an n by n symmetric matrix and A and B are n by k matrices in the first case and k by n matrices in the second case.
5. jacobi-1D: This application contains the 1-D Jacobi stencil computation algorithm. Stencil codes are a class of iterative kernels[1] which update array elements according to some fixed pattern, called stencil
6. mvt: This application includes algorithms for matrix vector product and transpose.

The Numerical Aerodynamic Simulation (NAS) benchmark suite is a set of benchmarks that has been developed for the performance evaluation of highly parallel super-comput-ers. These benchmarks consist of five "parallel kernel" benchmarks and three "simulated application" bench- marks. Together they mimic the computation and data movement characteristics of large-scale computational fluid dynamics applications.

The chosen benchmark kernels explored in the current work are presented below:

1. BT: Solution of multiple, independent systems of non diagonally dominant, block tri-diagonal equations with a (5 X 5) block size.
2. LU: A regular-sparse, block (5 x 5) lower and upper triangular system solution.. This problem represents the computations associated with the implicit- operator of a newer class of implicit CFD algorithms, typified at NASA Ames by the code "INS3D-LU".
3. CG: A conjugate gradient method is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. This kernel is typical of unstructured grid com-putations in that it tests irregular long distance communication, employing unstructured matrix vector multiplication.
4. IS: A large integer sort. This kernel performs a sorting operation that is important in "particle method" codes. It tests both integer computation speed and communication performance.
5. FT: A 3-D partial differential equation solution using FFTs. This kernel performs the essence of many "spectral" codes. It is a rigorous test of long-

distance communication performance.

6. EP: An "embarrassingly parallel" kernel, which evaluates- an integral by means of pseudorandom trials. This kernel, in contrast to others in the list, requires virtually no inter-processor communication.

3.4 Power Metrics

The metric of interest in power studies varies depending on the goals of the work and the type of platform being studied [13]. Well know and widely used on previous works metrics are:

- Energy (E): Energy, in joules, is often considered the most fundamental of the possible metrics. The value of this metric represents the total amount of joules consumed while executing a certain task.
- Power (P): Power, in watts (joules/sec), represents the rate of energy dissipation.
- Energy Delay Product (EDP): While low power often used to be viewed as synonymous with lower performance, that is no longer the case. In many cases, application runtime is of significant relevance even in energy- or power-constrained environments. With the dual goals of low energy and fast runtimes in mind, energy-delay product (EDP) was proposed as a useful metric. Its value is given by multiplying the energy consumed (joules) with the execution time (seconds). Some ways of computing EDP are listed below:

$$Delay = execution\ time$$
$$Energy = Watts * execution\ time$$
$$EDP = Watts * execution\ time * execution\ time$$
$$Execution\ time = Instruction\ Count / MIPS$$
$$EDP = Watts * (ICount / MIPS)^2$$
$$EDP = ICount^2 * 1/(MIPS^2/Watt)$$

If either energy or delay increases, the EDP will increase. Thus, lower EDP values are desirable.

- *Energy-delay-squared and beyond:* Following on the original EDP proposal, other work has suggested alternative metrics, such as energy-delay-squared product (ED^2P) or energy- delay-cubed product (ED^3P). These alternatives aim to highlight the importance of keeping the performance over a baseline in our try to make an energy saving model.
- NTT/Watt : Normalized per Thread Throughput per Watt is defined as :

$$\frac{NTT}{Watt} = \frac{Number\ of\ Instruction\ Retired}{Threads * Execution\ Time * Watts}$$

The problem with this metric that it should be used only on a fully utilized system, where no cores are left idle, because otherwise higher thread count leads in higher Watt consumption and lower NTT/Watt values. As a result this metric is better used from a scheduler on a system with no idle cores left or an execution with a fixed thread count.

In this work we consider Energy Delay Product to be the most useful metric when we need to highlight the energy of executing a workload or study benchmarks behaviors on different thread counting and DVFS (Dynamic Voltage and Frequency Scaling) and Energy Delay Squared Product when we want to give performance priority against energy savings.

4 Power Consumption

In this section we use the previous mentioned power monitoring tools to obtain power measurements and introduce a simple power consumption model for our system. Then we study the power consumption rate of applications according to the available resources and placement and deal with the influence of scaling on the EDP of an application. We also study the importance of dynamic voltage and frequency scaling (DVFS) when dealing with programs with large datasets, and finally we study the power profile of the NAS and the Polybench benchmarks and create pareto chards for their EDP.

4.1 Power Model

Running different applications from the NAS parallel benchmarks demonstrated that every application has a different rate of power consumption, based on its cpu intensity and memory needs of the application. Each application based on its behavior has different needs in core, uncore and DRAM power. For example consider the table below, which contains measured core, uncore and DRAM power for the applications bt.A, cg.B, ep.A and lu.A (included in the NAS parallel benchmarks) running on 1 core and the power consumption when our system remains idle:

application	idle	bt.A	cg.B	ep.A	lu.A
Pcore(Watt)	0.43	9.72	9.49	8.26	9.38
Puncore(Watt)	13.5	14.07	13.77	13.5	13.6
Pdram(Watt)	3.2	6.56	6.36	4.41	5.04
P (Watt)	17.13	30.35	29.62	26.17	28.02

Table 4,1-1 Single Threaded Power Measurements

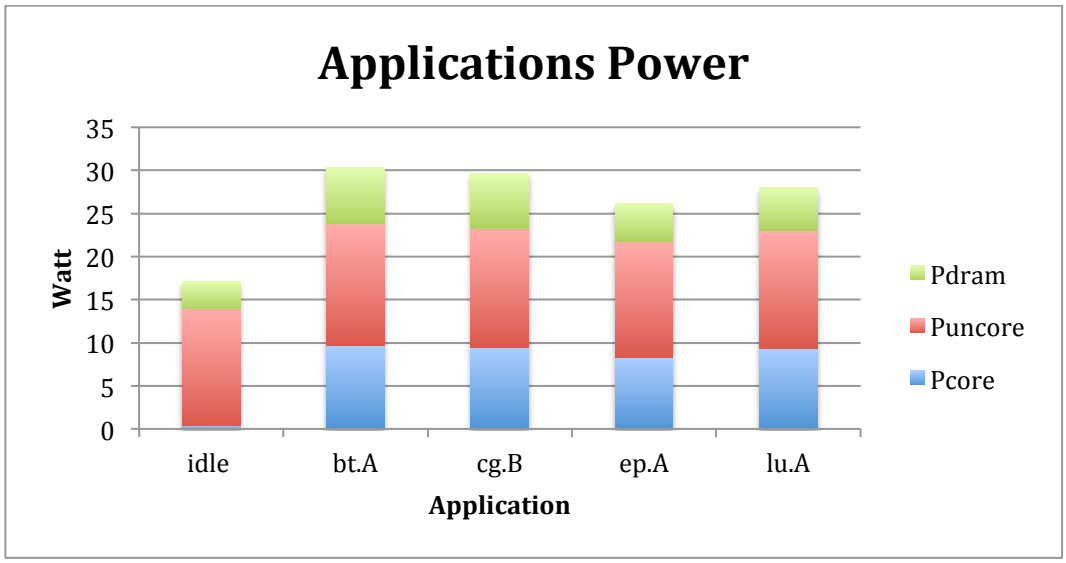


Chart 4.1-1 Single Threaded Power Measurements

When our system is empty each one of the idle cores has a power consumption of 0.43W and Dram need 3.2W while the uncore parts of our package need 13.5 W, a significant higher amount from core and Dram. That's because memory controllers and LLC are never switched off or run on an idle state and they are in constant need for power. Then running different application results in different power behavior for each one. For instance bt.A and cg.B, which are memory bound, have a higher need for DRAM power and even core power from ep.A and lu.A, which exploit higher level of parallelism and less need for memory. As a result we can characterize bt.A to be the more power "hungry" of our 4 benchmarks needing 30.35W to execute on a single core, while our system needs 17.13W when it is empty. Finally the uncore parts need 13.5-14W for the controllers to run regardless the current running application.

From the above measurement we can conclude that every application has a different power/energy profile based on memory or the cpu intensity while executing. For that purpose we build 2 corner case applications using the STREAM benchmark [18]. This is a pseudo-benchmark designed to make streaming memory access without any cache reuse, in order to constantly transfer data from memory to LLC and define the limits of the system's memory bandwidth. Our System reported a top transaction rate around 15000 MBs/second for every physical package. We make the following adjustments to the tuned_STREAM_Triad() function to add arithmetic operations to the computational kernel:

```
voidtuned_STREAM_Triad(STREAM_TYPE scalar)
{
        ssize_tj,k;
        #pragmaomp parallel for shared (val)
        for (j=0; j<STREAM_ARRAY_SIZE; j++){
                a[j] = b[j]+scalar*c[j];
                for (k=o; k<ARITHMETIC_OPS; k++){
                        val = val + scalar;
                }
        }
}
```

Code 4.1.1-1 Stream Software Triad Function

We need the first one to be memory-bounded so it is designed to use the whole memory bandwidth of one package in our system, and the second to be cpu intensive, so it is designed to perform 1000 arithmetic operations between every memory transaction. We first run each one on a single core and then on 4 and 8 cores to fill at first half and then the entire package and we obtain the following results:

	1 core		4 cores		8 cores	
application	Cpu intensive	Memory bound	Cpu intensive	Memory bound	Cpu intensive	Memory bound
Pcore(Watt)	7,98	11,61	18,95	24,17	33,89	40,79
Puncore(Watt)	13,45	15,28	13,47	15,96	13,54	16,05
Pdram(Watt)	4,76	9,2	4,79	11,02	4,84	11,51
P (Watt)	26,19	36,09	37,21	51,15	52,27	68,35

Table 4.1-2 Power Consumption of the Stream Benchmarks

From the above results we conclude that running an 8-core package may require power consumption of 8~12Watts per core (making a total 64~96W for all 8 cores), 3.2~12 Watts for DRAM purposes and 13~17 Watts for the needs of the Last Level Caches and the Memory Controllers. That indicates that for every package of our NUMA-processor we can assume a simple power model as shown in the chart below:

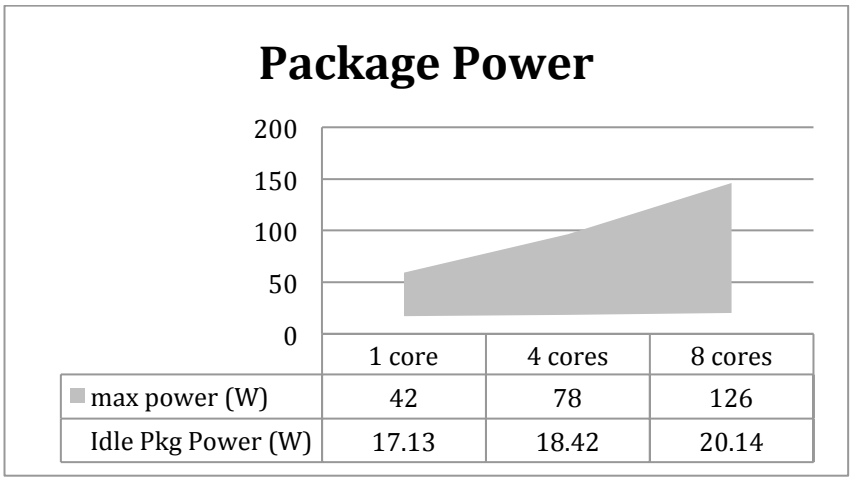


Chart 4.1-2 Power Model

. Where for every execution our package power consumption rate should be inside the grey area of the chart.

As we are interested in the total energy consumption for a job execution we need to consider our system's power rate along its performance to get the total energy consumption. For example running bt.A on 1 core requires our system to use 33.7Watts(12.73W core power, 14.07W uncore and 6.57W DRAM) and has a total execution time of 147.84 seconds, consuming a total amount of 33.7*147.84 = 4982.21Joules, while running on 2 cores requires our system to use 39.88Watts (17.25W core power, 14.64W uncore and 7.99W DRAM) and has an execution time of 78.62 seconds, which lead us in a total energy consumption of 39.88 * 78.62 = 3135.37 Joules. In that case the scaling ability of our application is very crucial for our scheduling decision because letting the application to run on 2 cores instead of one leads to an increase of 7.18 Watts in the total power consumption but also it almost doubles the performance and result in consuming almost 37% less energy (in Joules) to perform the requested job.

So it is quite obvious that it is not always necessary for a power efficient scheduler to try keeping the average package power on the left side of the grey area in our chart above, which contains lower values of package power consumption rates, but it should consider a lot of other factors that could lead to a more power friendly execution, such as scalability of our applications and, as we will mention bellow memory transactions, stalls, cache misses and more.

4.2 Thread Scheduling - Scaling

There are many studies about different ways to obtain a power friendly thread scheduling and many different scheduling policies suggested. Most of them aim to make a power estimation of different available thread counts for running an application, based on certain performance counters, and use the best combination to fill the available resources of a system. The main purposes of these scheduling

policies are to run a list of requested jobs under a certain power envelope or try to achieve as less energy consumption as possible.

Running an application on different thread counts results in different performance and power behavior. For example, giving more cores to an application with good scalability will result in better performance, but higher power consumption too. Also running a 2-threaded application on 2 cores in 1 package may perform worse compared to execution in 2 packages (1 core on each package), because more packages offer more resources such as larger cache capacity, which results in fewer last level cache misses, but it would be harmful from a power wasting point of view, because 2 packages will double the use of memory, Last Level Caches and memory controllers on a NUMA processor.

First of all it is an obvious fact that giving more resources to an application to run would result in higher power consumption rates. On the chart below we illustrate the power of ft.B, from the NAS Parallel Benchmarks, while running on 1 to 8 cores:

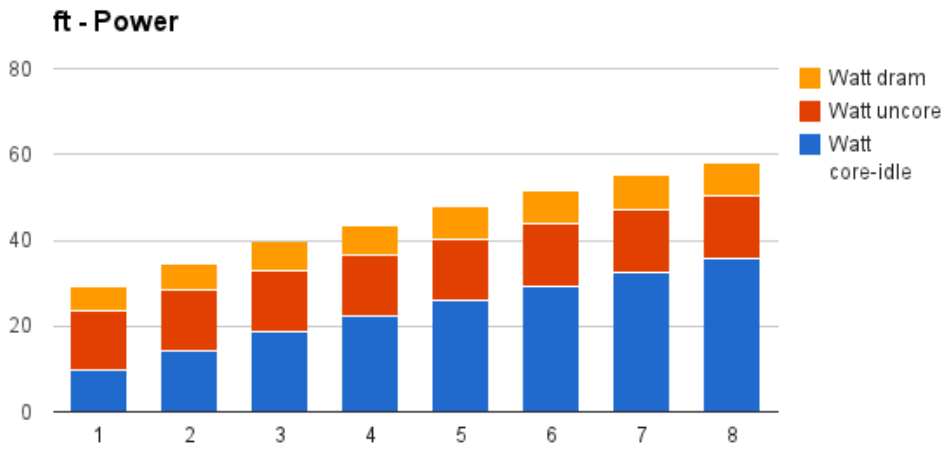


Chart 4.2-1 ft.B Power consumption with increasing thread count

From the chart we can see that only the uncore part of our package seem to consume power in an almost steady rate, while DRAM power slightly increases as we use more cores and as expected core power increases proportionally to the number of cores. But what's important here is to decide which thread count is better for execution in a power saving manner. So we measure the execution time and the total energy consumption until ft.A exits and calculate the Energy Delay Product (EDP) of every execution:

cores	1	2	3	4	5	6	7	8
exec time	80.8083	44.2010	32.8540	28.4949	23.9559	21.1219	20.0673	21.0738
energy until execution	2352.26	1528.45	1301.77	1241.17	1144.55	1088.20	1110.99	1226.19
EDP	190082.3	67559.2	42768.5	35367.1	27418.8	22984.9	22294.6	25840.6

Table 4.2-1 EDP for ft.B

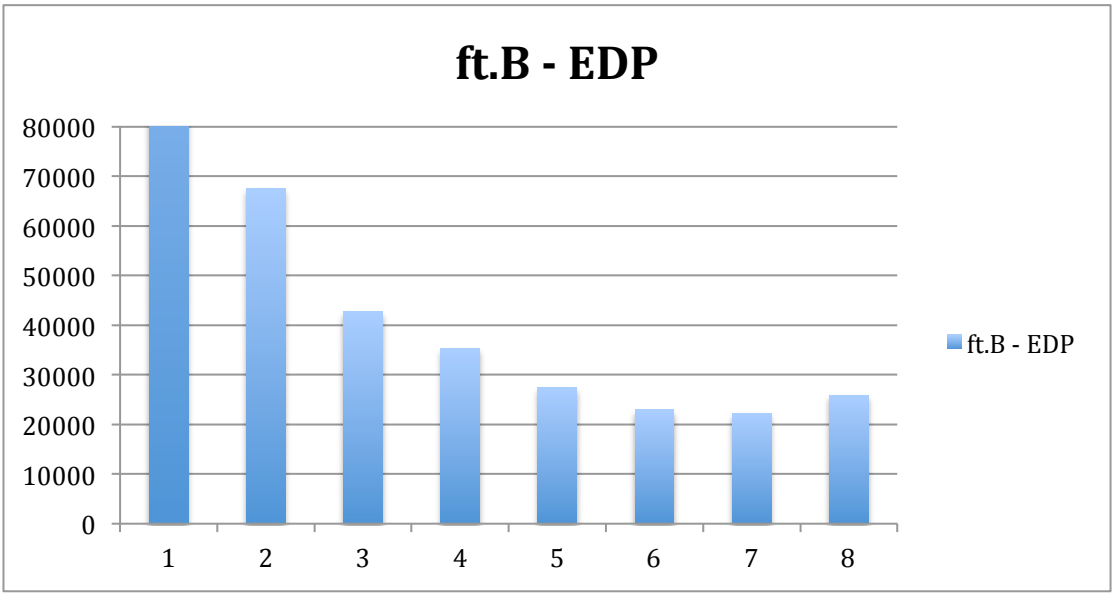


Chart 4.2-2 ft.B EDP with increasing thread count

From the above chart we observe that the best Energy Delay Product values are given by running ft.B on 6-7 cores (EDP lower values are better, as mentioned before). Running ft.B on 1 core gives us the highest EDP value and then as we increase the number of cores on which ft.B is executed the value of EDP is decreasing until it reaches a minimum for running our benchmark on 6 or 7 cores. Also from the execution time row of our table we observe a corresponding behavior on the time the benchmark needs to complete its execution. Giving the application more cores to run results in smaller execution times until the core count reaches 6 cores. Then ft.B seems to be unable to benefit from larger number of available cores to run and result in execution times close to 20-21 seconds. As a result we conclude that scalability may be a very important part of a power aware scheduling model.

When a parallel application has good scalability it means that it is capable to benefit from using more resources and execute in significant smaller time. In that case giving more cores may require higher power consumption rate (Watts), but the time reduction will result in less power consumption until completing the execution and of course lower Energy Delay Product values. On the other hand when our application exhibits low levels of scalability it means that giving more resources to run on may have small effect on reducing the execution time but a great impact on energy consumption, as our system requires more power to run, and results in higher energy consumption to complete our job.

Another important fact is that, from a power saving point of view, we can sometimes benefit from letting some cores to run idle for a while, in contrast to performance aware scheduling policies, which always try to obtain a fully utilized system. For example when we have to run an application, with good scalability between 1 and 4 threads and then stops, alone on an 8-core processor a power aware policy would be to use only 4 cores and leave the remaining idle so that we pay a small performance loss but also achieve a smaller energy consumption than using the whole system.

To obtain a better image about the issues discussed above we run all the NAS [15] and Polybench[16] benchmarks on 1 to 8 cores, in order to stay inside a single package and avoid unexpected scaling behaviors of the benchmarks due to the NUMA architecture characteristics, we measure the speedup (scalability) and the EDP. The charts below contain the acquired results:

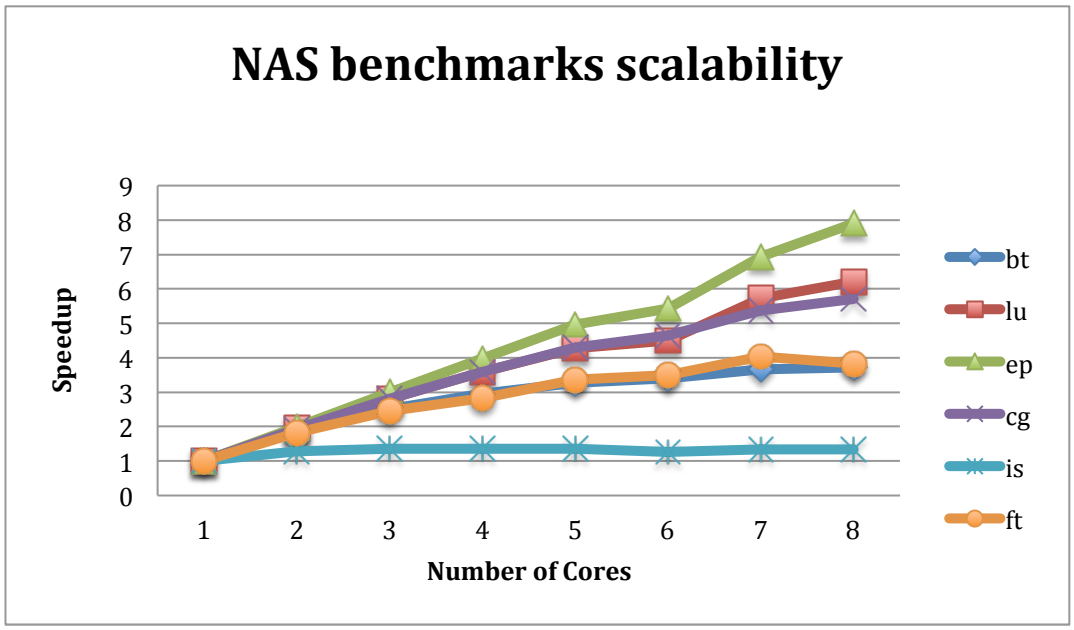


Chart 4.2-3 NAS Benchmarks Scalability

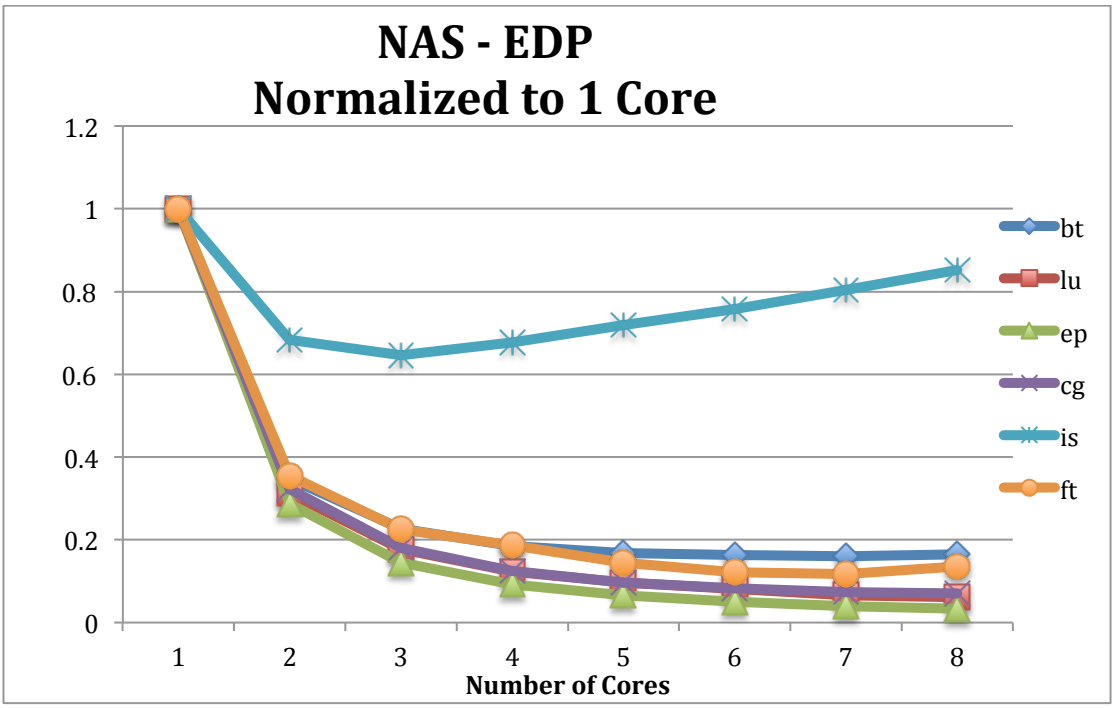


Chart 4.2-4 NAS Benchmarks EDP

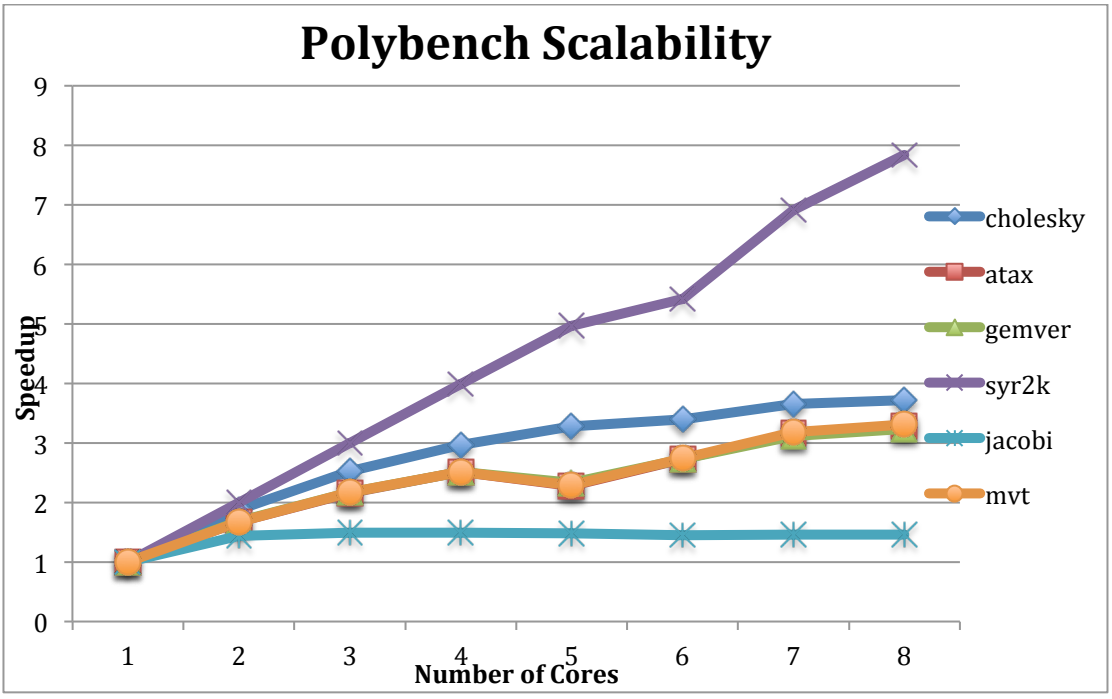


Chart 4.2-5 Polybench Scalability

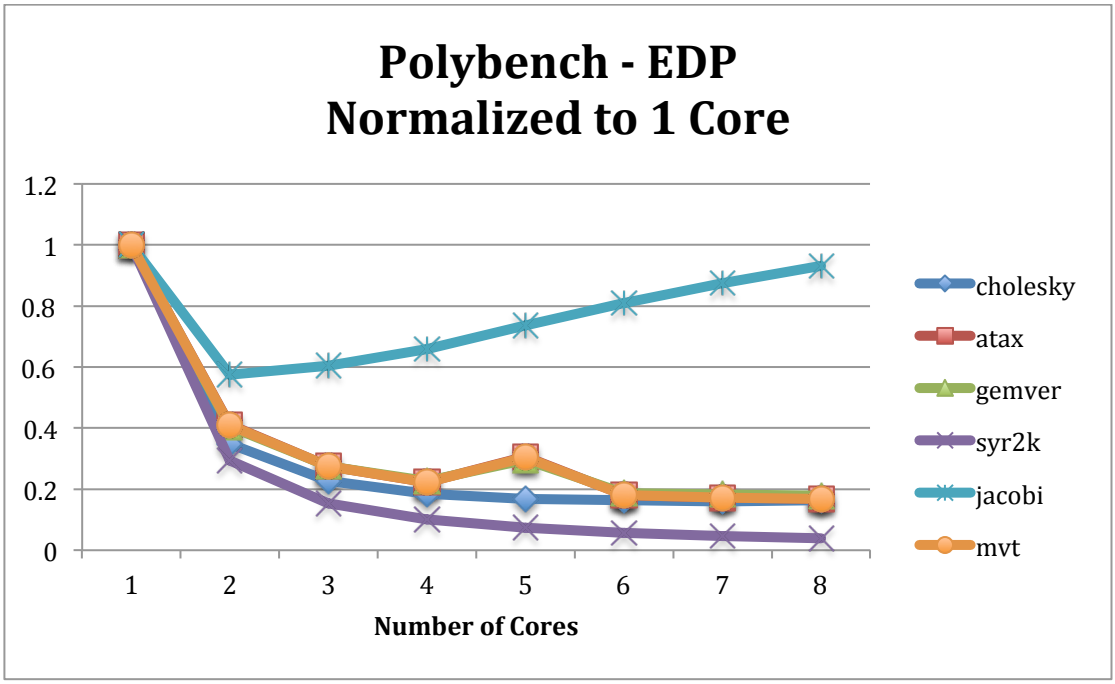


Chart 4.2-6 Polybench EDP

From the above charts we observe that allocating more cores for an application is beneficial for EDP only if the application exhibits good scalability. For example, in NAS benchmarks only *ep, lu* and *cg,* the three application with the greater scaling ability on 6 to 8 cores, would result in a more power efficient execution when they allocate 7 or 8 cores, while the other applications, which fail to scale efficiently above 6 cores would result in wasting power if executed on such threadcounts. So the most power-efficient execution would include allocating 6 cores for both bt and ft. Finally, is, which fails to scale above 2 or 3 threads, should be executed only on 2 or 3 cores in order to avoid unnecessary waste of energy.

Similar results are obtained with the Polybench suite. As we can see, *jacobi*, which fails to scale efficiently, requires only 2 cores in order to consume less energy as possible for its execution, while on the other hand, applications like syr2k would benefit by allocating more cores and this would result in better performance and EDP as well. The rest of the benchmarks scalability characteristics lies between jacobi and syr2k and their best EDP values could be obtained by executing them on 6 to 8 cores.

In conclusion, changing the placement and the available cores of an application in the system will result, as expected, in different power consumption rates as well as different performance. So it is very important to know the scaling ability of an application in order to make decisions that would lead to more power efficient executions without negatively affecting the overall execution performance. Moreover, in most of the cases the most power efficient choice is included on a list of choices that result in better performance and it is the choice that allocates as less cores as possible.

4.3 Dynamic Voltage and Frequency Scaling

As mentioned before, modern CPUs operate considerably faster than the main memory they use. That gap between memory and CPU speed often forces the CPU to stall and wait for requested data to come, and results in significant performance decrease. Even though a scheduler is not able to deal with this problem from a performance point of view, an energy efficient scheduler should try to identify such cases and use dynamic voltage and frequency scaling techniques in order to reduce the systems total energy consumption.

This problem could be enlarged when different applications compete for the same resources, like the LLC, and result in cache conflicts that enforce them to transfer the same data blocks from memory multiple times. There are many reasons that could cause memory contention in the execution of a workload. Most of them are related with the competition between running applications for shared resources, which on our system would be the L2 and L3 caches and of course the memory bus. As L2 cache is shared by only two cores, its contribution on the overall memory contention is small and for the purposes of this work could be ignored. But competition for the L3 cache and the memory bus may be harmful and sometimes catastrophic for the running applications. Moreover, memory contention might be a problem for applications running alone on the system if their dataset is too large. Applications with large dataset usually experience problems because even if the bus bandwidth can fulfill their requests for data fast enough, caches may not be large enough to service the processor's needs and the system is forced to load the same data from main memory to the last level caches over and over again.

In order to study this issue let us consider again the STREAM benchmark mentioned before. For our purposes we now create 5 different instances of the STREAM application, each one of them using approximately 0%, 30%, 60%, 80%, 100% of the total memory bandwidth, respectively. We run each application with 8 threads, in order to fill an entire package and avoid measuring power from unused cores, for all the available system frequencies, from 1.2GHz to 2.2GHz with step 0.1GHz, and calculate the EDP for every different run. We made the previous choices of bandwidth because our applications reported gain by DVFS on EDP for bandwidth 60% and higher. The charts below contain all the EDP values for every different instance of STREAM created:

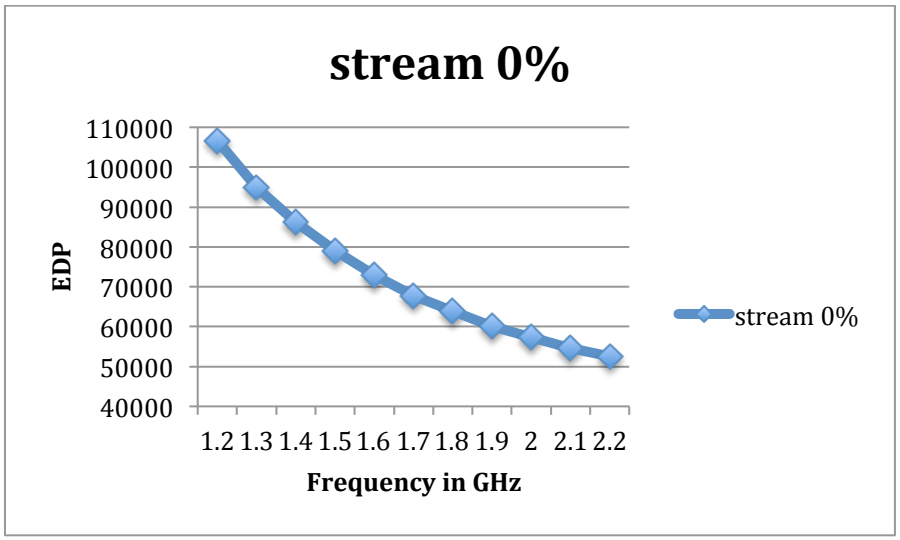


Chart 4.3-1 Stream 0% EDP

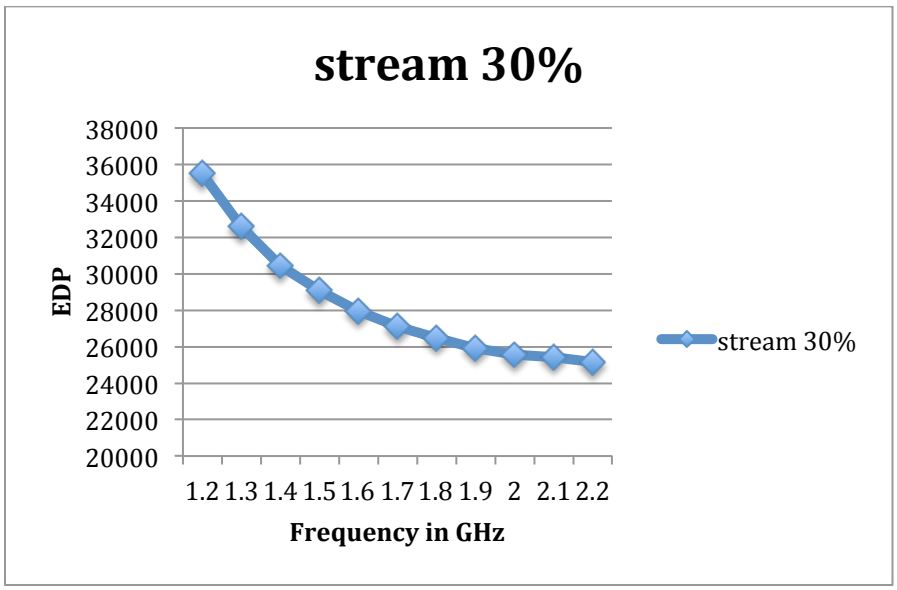


Chart 4.3-2 Stream 30% EDP

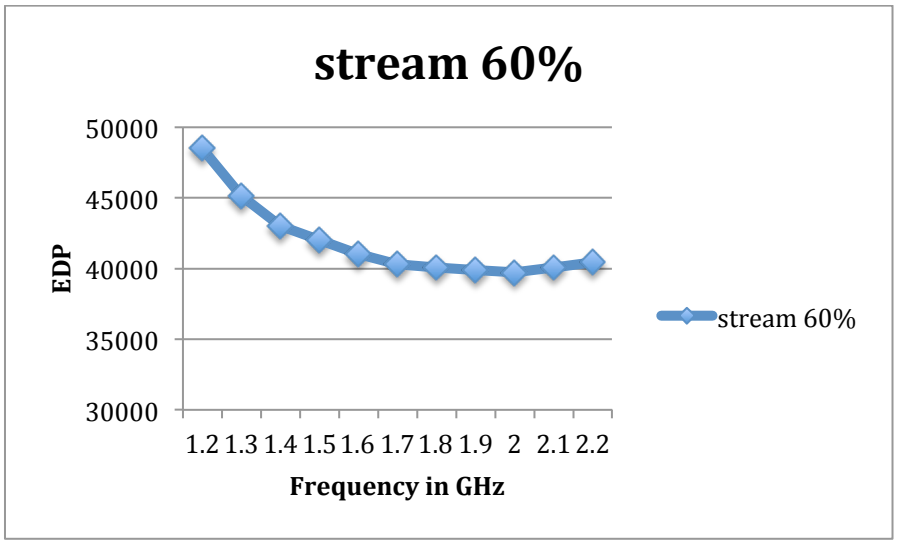


Chart 4.3-3 Stream 60% EDP

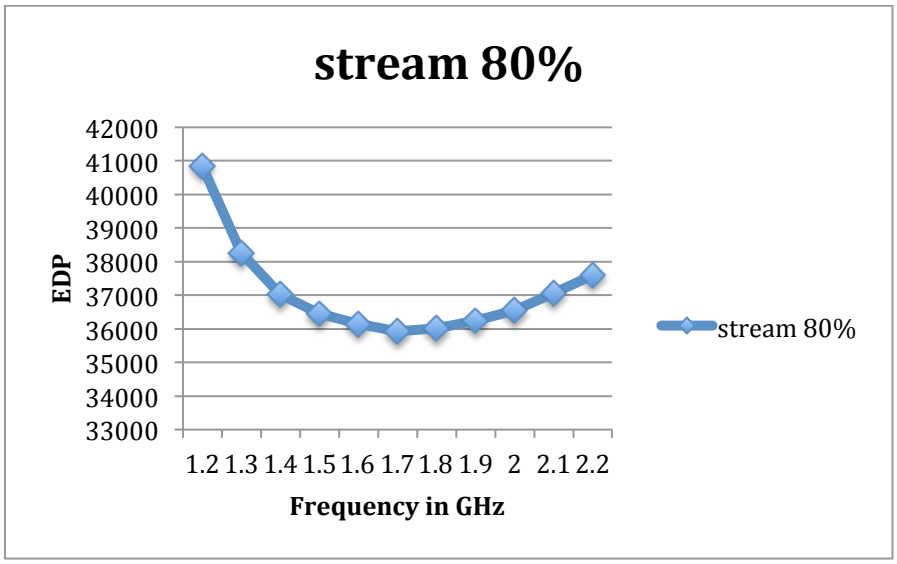


Chart 4.3-4 Stream 80% EDP

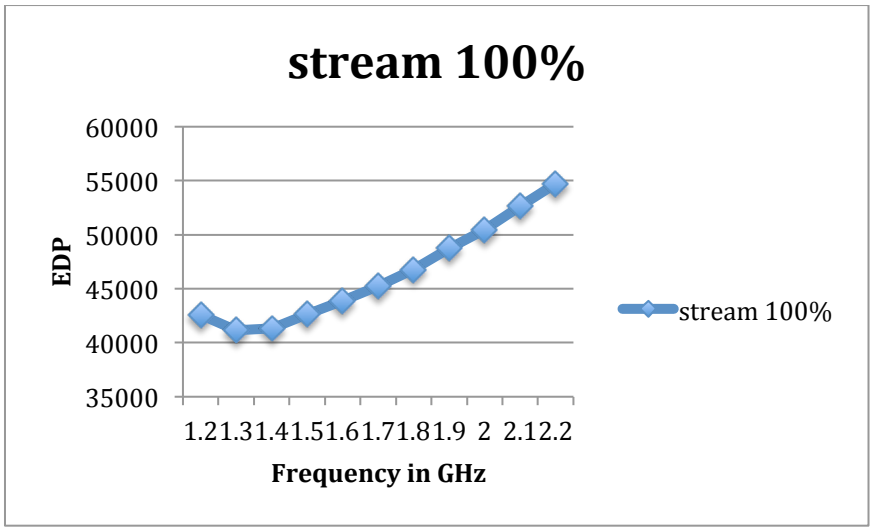


Chart 4.3-5 Stream 100% EDP

From the results we observe that for the extremely cpu-bound applications EDP report an almost linear drop for increasing frequencies, as shown by the STREAM-0%, because in such applications the processors need significant small amount of data, that could be filled in many cases by the first level caches, and spend most of the time inside the core rather than waiting for data. As a result changes in the cores frequency have a great effect in the overall system performance and thus to the final EDP. As we increase the memory bandwidth usage we observe this almost linear behavior to change into a curve and for bandwidth usage of 60% and more the EDP line reports a lower bound curve value.

This lower bound implies that the best power-aware execution is obtained by running our cores in lower frequency values than the system's available max frequency, and that such an execution would result in lower power consumption without significantly affecting the performance. Another interesting fact is that the lower bound values on our curves move from right to left, from high to low frequencies, as the memory bandwidth usage increase. For example, the lowest EDP value for STREAM-60% is reported for execution in 2.0 GHz, while STREAM-80% reports a low at 1.6-1.7GHz and finally, the original STREAM application, which is designed to use the total package's memory bandwidth report the best EDP value when running at 1.3GHz. Moreover, in the last case we observe that running our application with the highest available frequency is significantly worse that running it with the lowest frequency available, from a power saving point of view.

As a result we see that knowing the memory needs of an application is very important when using DVFS techniques on our execution. In [8] the authors propose an equation for calculating the effects of the total memory bandwidth usage in an execution. The equation for calculation an EDP factor is:

$$EDP\ factor = 1.4 - 0.8 * x$$

. Where x is the total memory usage bandwidth rate of the execution. Applying this on our experiments we calculate that for 60% of bandwidth the equation gives us an EDP factor = 0.92, which indicate a frequency of 2.024 GHz, for 80% of bandwidth EDP factor = 0.76, indicating an execution frequency of 1.672GHz, and for 100% of bandwidth EDP factor = 0.6 and the proposed running frequency is 1.32GHz. So we observe that this equation gives us an almost accurate assessment of the spot where the lower bound of the EDP curve lies, and as a result the execution frequency to obtain the lowest value of EDP.

To obtain a better view of the importance of memory bandwidth usage on the overall execution's EDP, we normalize the measured EDP values to the measure for the lowest frequency and produce the following chart:

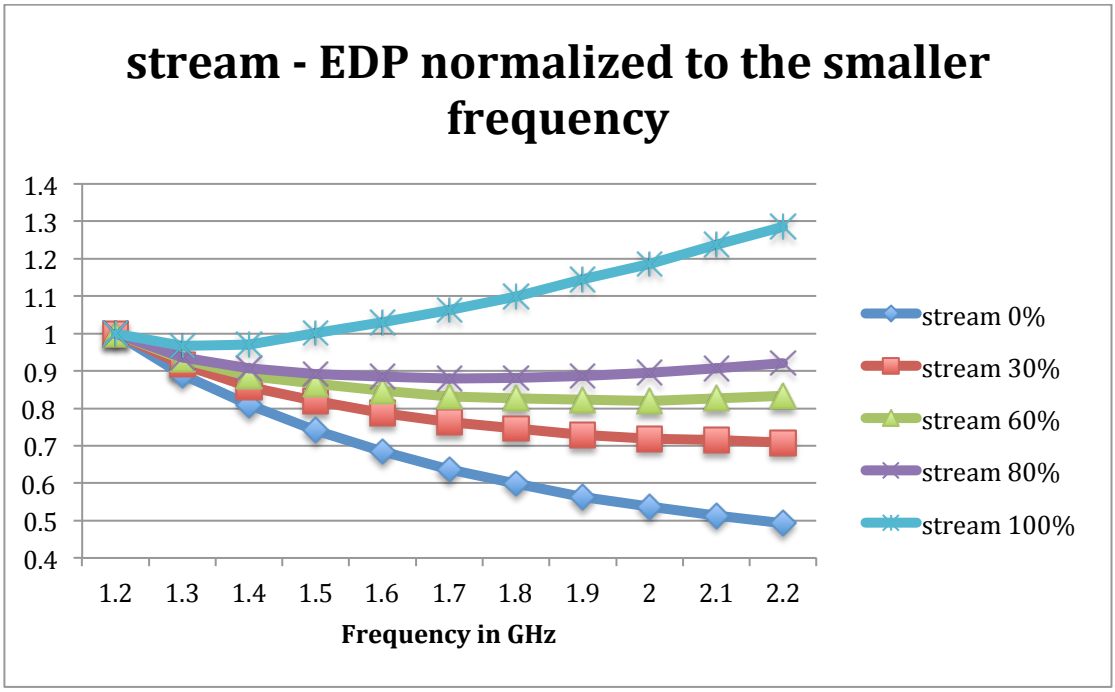


Chart 4.3-6 Stream EDP Normalized to lowest frequency values

From this chart we report that the EDP line starts with an almost linear decreasing form, for STREAM-0%, and as the usage of the total memory bandwidth increases the line moves higher and finally, for full memory bandwidth usage results in an almost reverse form than the first one.

In conclusion, for many reasons related to the memory access and transfer speed, the knowledge of memory needs is necessary to produce a power efficient scheduler, which includes frequency scaling techniques to balance the consumed power. This knowledge would help the scheduler make crucial decisions based on the currently executed workload and apply different frequencies to the used system cores to prevent unnecessary energy expenses.

4.4 Placement Issues

Another issue that needs to be studied is the importance of placement in performance as well as power consumption. As mentioned earlier there is not a big difference between a performance and a power aware scheduler, because when a scheduler lacks performance efficiency, and make selections that may increase the contention between running applications, in most cases results in long executions that are harmful for both performance and power.

For that purpose lets consider once more the STREAM application and create three instances: one cpu intensive with very little memory usage, one with medium memory usage and one that uses the whole memory bandwidth capacity. To highlight the importance of placement we run each one with 4 threads and co-schedule it with itself and each other inside the same package, and then run both of them in 2 different packages. In order to obtain more valid results we design each one of them to run for approximately 27-28 seconds when running with 4 threads at the maximum available frequency, and we report the stand-alone energy consumption of every application, as well as the energy consumption and the execution time of co-scheduling compared to the same metrics for simultaneous execution in two different packages. We assume that when we run applications on only one package, the others are powered off and have no contribution to the system's overall energy consumption. So we obtain the following results:

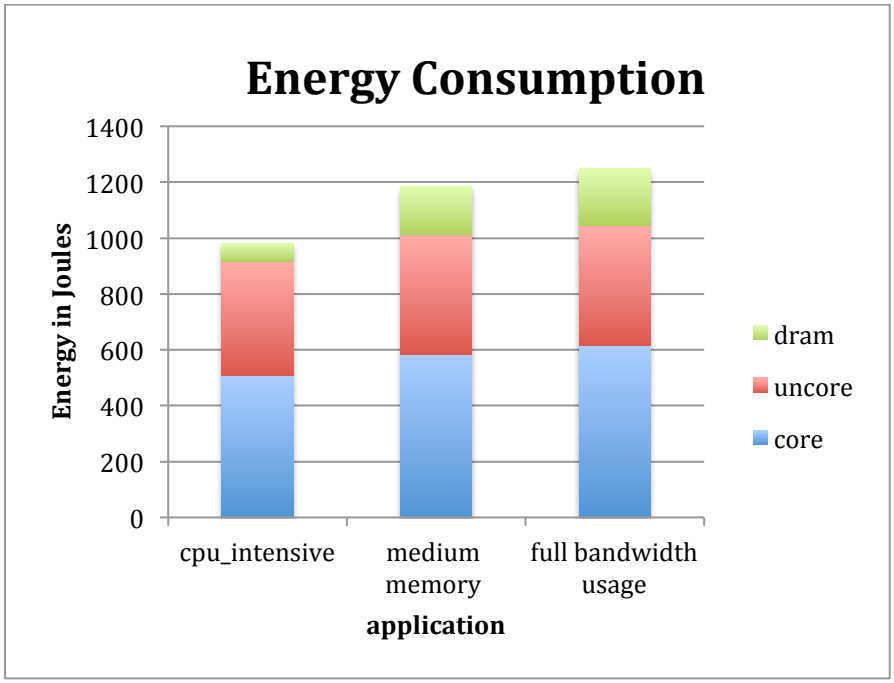


Chart 4.4-1 Stream versions energy usage 4 threads for each application and approx. 40 secs execution time

As it was expected we see that the more memory bound an application is the more DRAM and uncore energy consumes, as the uncore parts include the last level cache and the memory controllers. The interesting fact from the above chart is that, due to the L1 and L2 caches contained inside the measured core parts of the processor, the core energy consumption is lower in our cpu intensive application than the other two, even though it has the more arithmetic operations to perform and so the higher computational ratio.

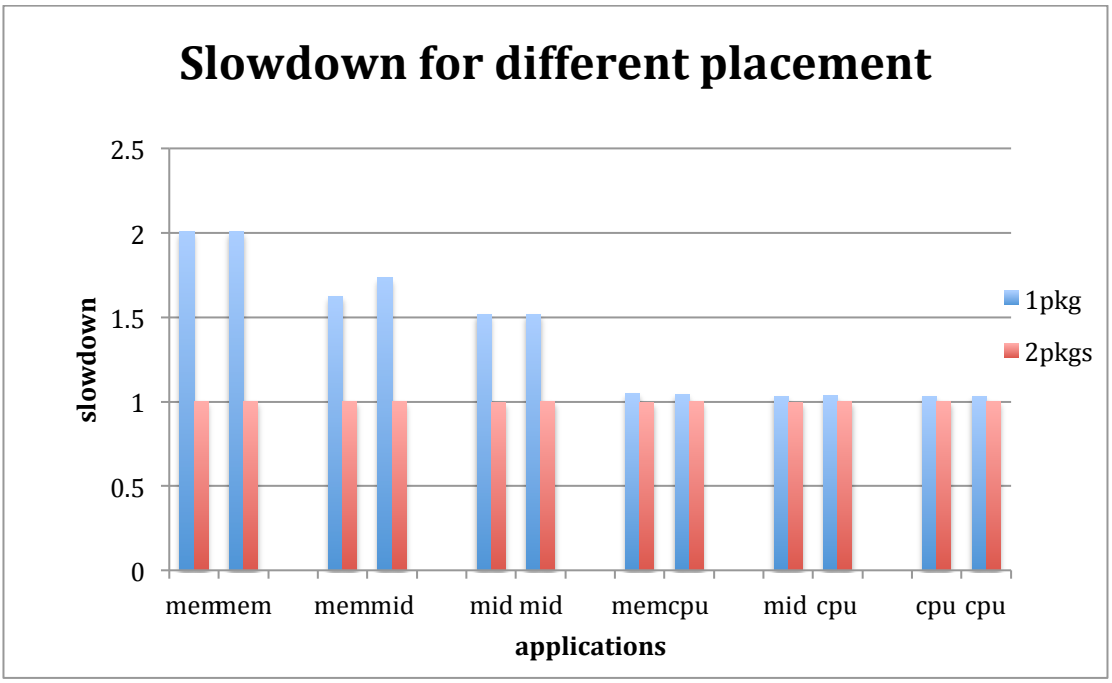


Chart 4.4-2 Slowdown According to Placement

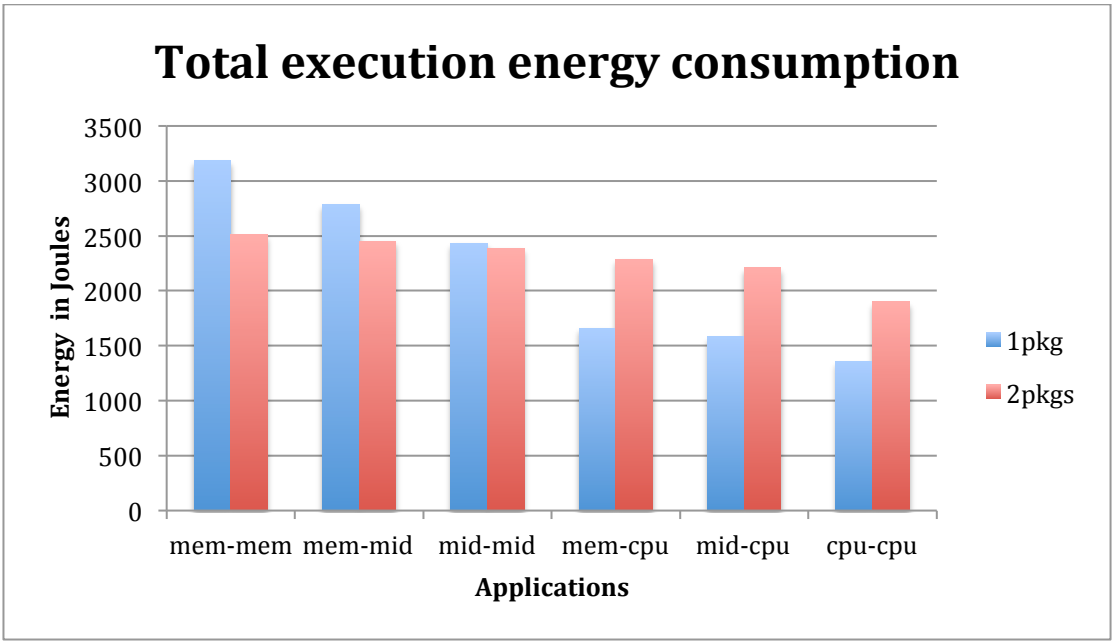


Chart 4.4-3 Total energy consumption of different placements

The above charts highlight the importance of co-scheduling applications with different memory characteristics. We observe that co-scheduling applications with high memory needs together on the same package, results in significant slowdown, which has a great influence on the total energy consumption for completing our

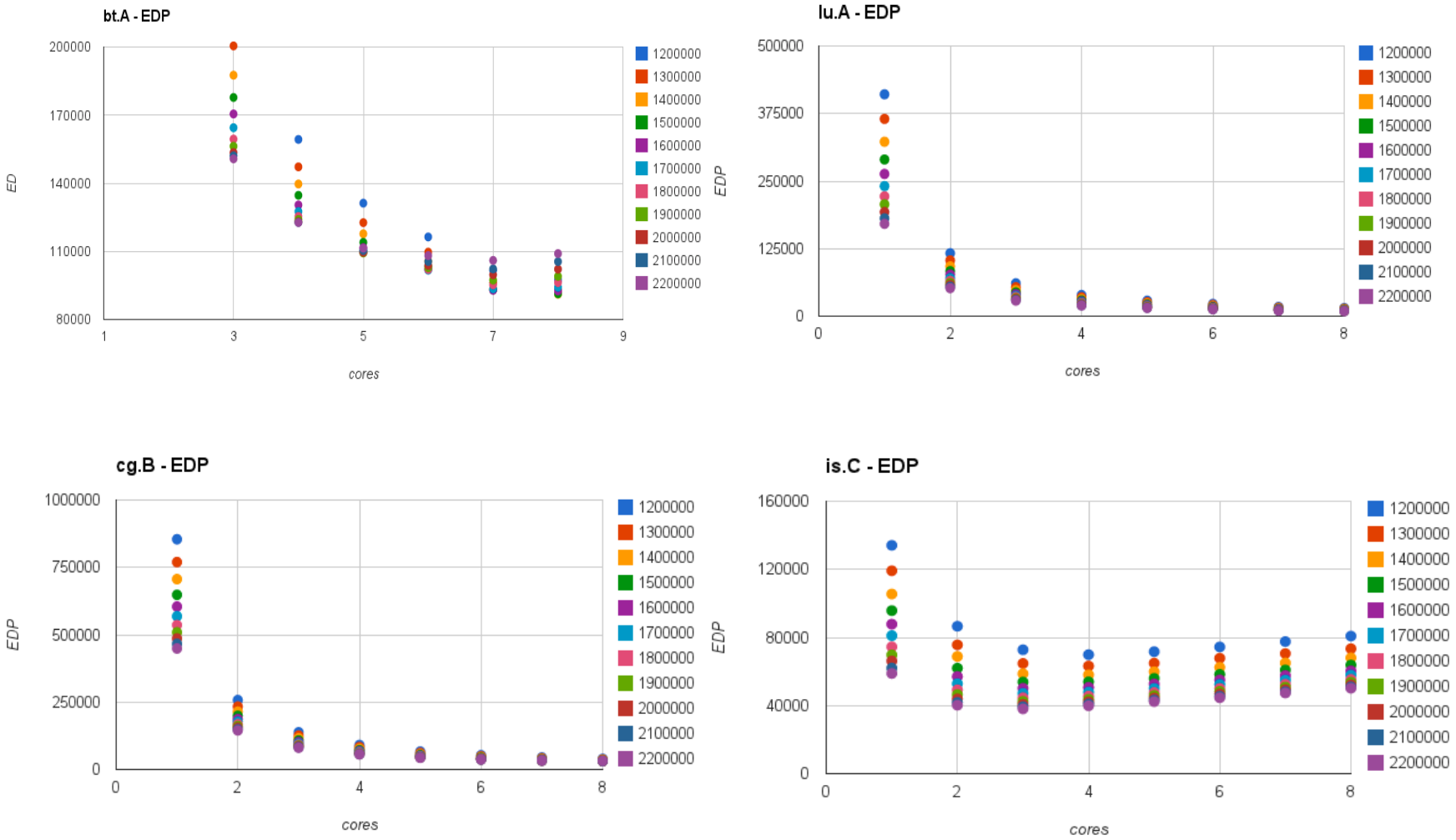
execution. So in cases like that it would be beneficial for both energy and performance to schedule the applications on different packages, as shown on the previous page. Even when we co-schedule 2 instances of the medium memory application, we observe a slowdown of 1.5 for each one, and a slightly greater energy consumption that running the two applications on two separated packages. On the other hand, co-scheduling applications with different characteristics inside the same package is beneficial for the total energy consumption, while it reports significant low values of slowdown, so the performance is slightly affected. For example, running the memory bounded and the cpu-intensive application inside the same package, reports a slowdown less than 1.1 for each application but a great reduction on the consumed energy.

Actually, the cpu-intensive application that we created seems to be rather "friendly" to every other application, as all the experiments that included this application, reported very low values of slowdown and benefited from co-scheduling in the same package because of the smaller energy consumption until the completion of the job. The main reason for this behavior is that our cpu-intensive allocation has a very small usage of the shared system resources, such as the L3 cache and the memory bus, and mainly its execution is bounded inside the allocated cores and their private L1 and L2 caches, so it does not affect any other application that is running under the shared resources of the used package.

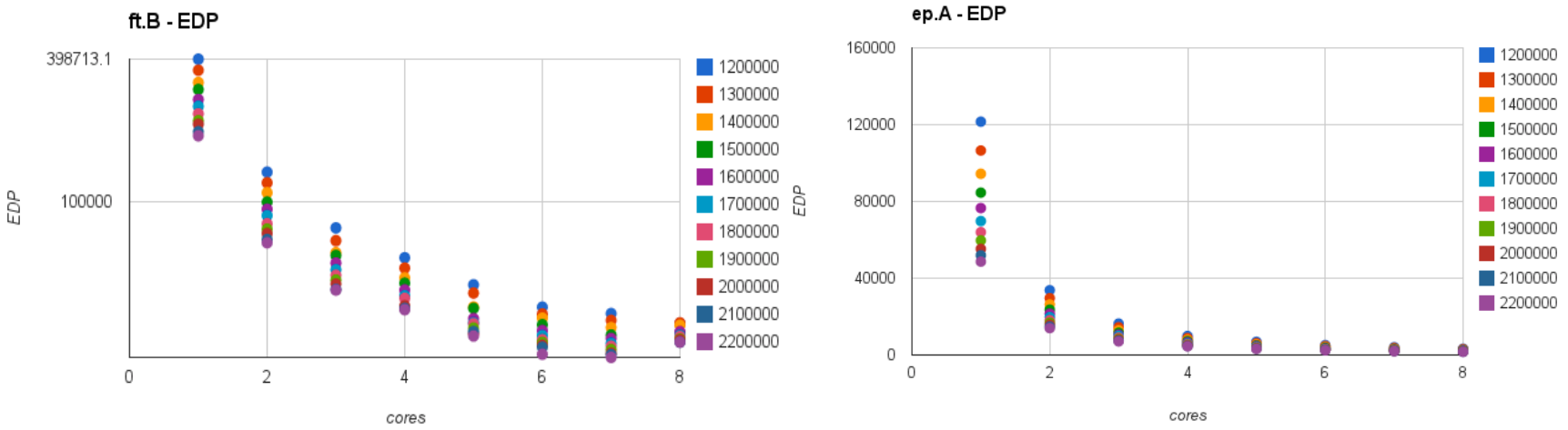
In conclusion, co-scheduling applications with different characteristics inside the same NUMA package reduces the contention level between them and results in a significant decrease in the system's energy consumption, while not really affecting the overall performance. So a useful power-aware scheduling policy would be to separate the applications according to their memory profiles and use this information to combine applications, in order to reduce the memory contention and as a result the total energy consumption.

4.5 Power Profiling

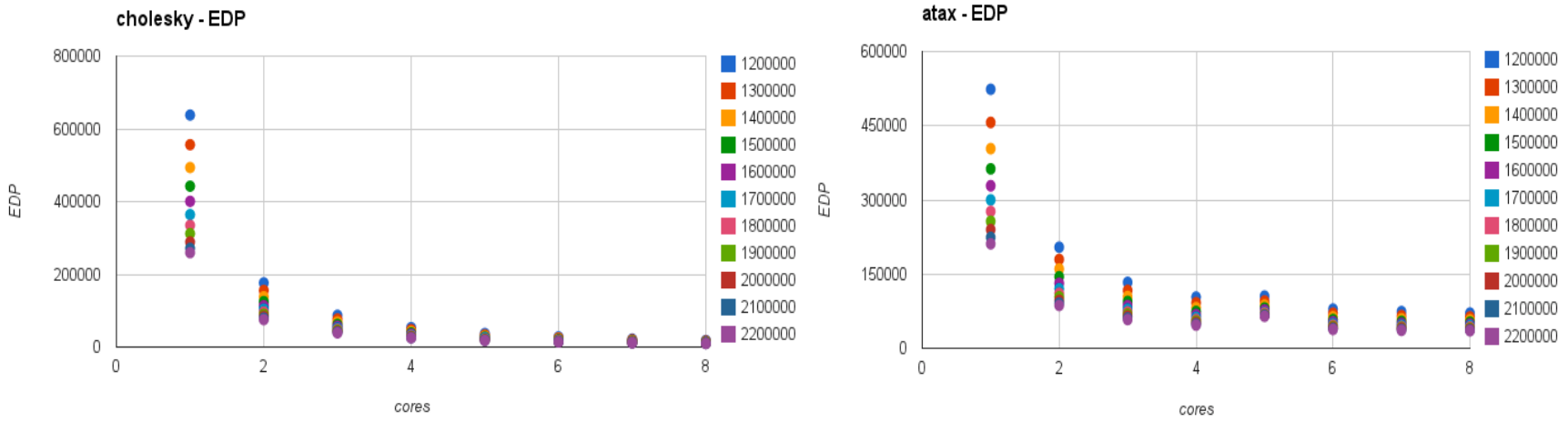
Based on the previous sections of this chapter we create a power profile for each one of the Polybench and the NAS benchmarks we use in this work. Also we study the profiles of 2 artificial applications made with the STREAM benchmark implementation, one with almost 0% and one with almost 100% usage of the total memory bandwidth, and the energy profile of the Floyd-Warshall solver application, which proved to be very useful for our experiments. For that purpose we run each benchmark for 1 to 8 threads, so that we do not exceed the number of cores inside a single package, with all the available system frequencies and calculate the EDP for each run. We use the obtained information to produce a pareto chart for each benchmark, where the lowest points of each graph represent the best combination of threads and frequency, for each application, that reports the lower values of EDP.



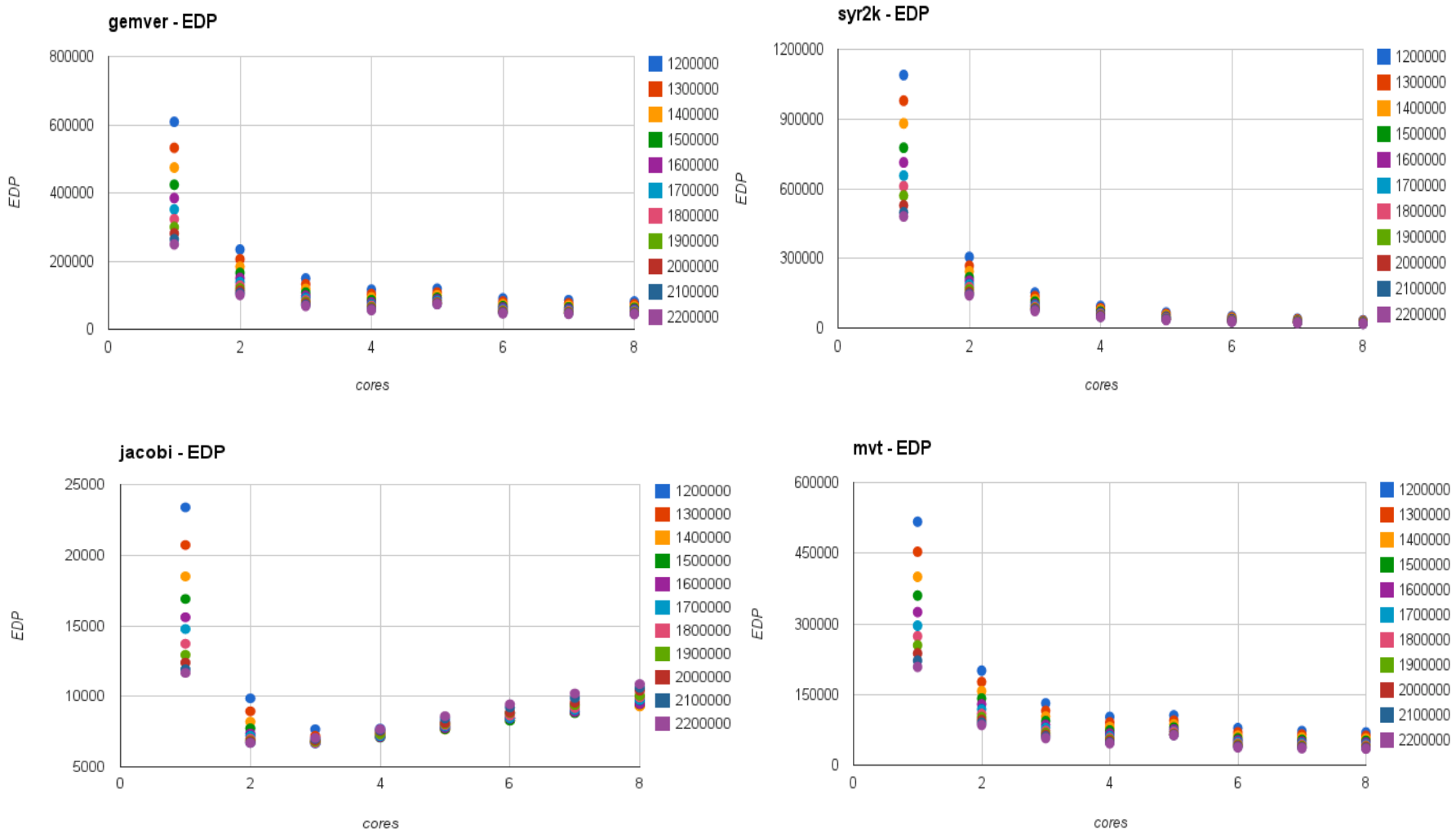
Charts 4.5-1 NAS Benchmarks Power Profiles (1)



Charts 4.4-2 NAS Benchmarks Power Profiles (2)



Charts 4.5-3 Polybench Suite Power Profiles (1)



Charts 4.5-4 Polybench Suite Power Profiles (2)

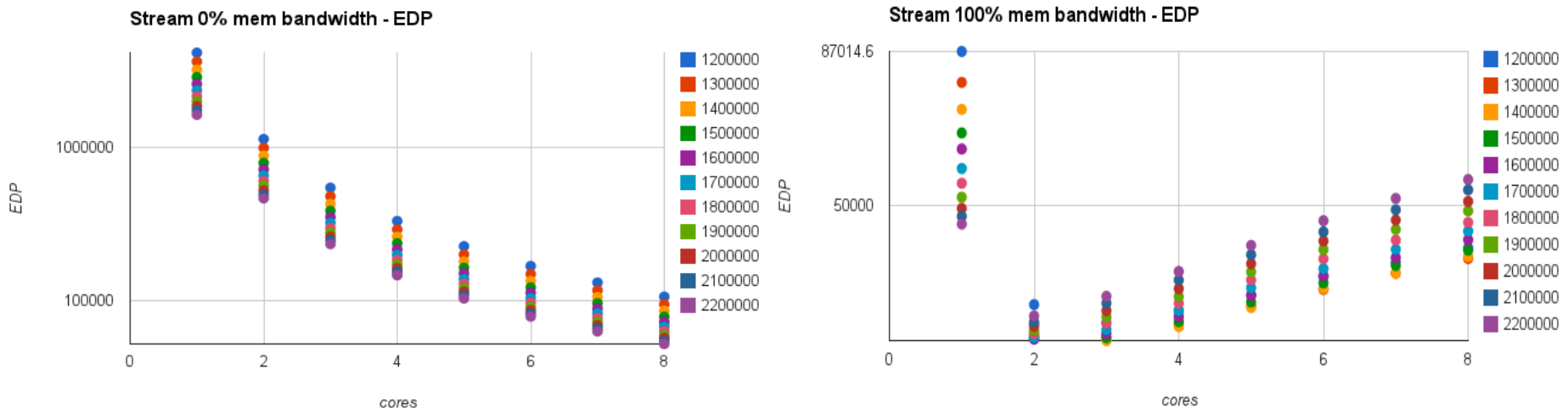


Chart 4.5-5 Stream "Corner Cases" Power Profiles

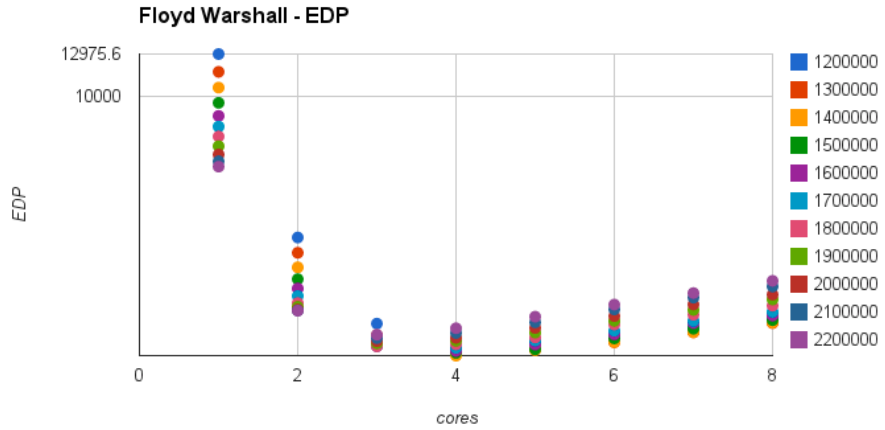


Chart 4.5-6 Floyd-Warshall Power Profile

In conclusion, we observe that every application according to its scaling ability and memory needs have a different power profile, as illustrated on the above charts. For example, Floyd-Warshall Jacobi and bt.A report their lower, and thus better, EDP values for frequencies lower than the highest system's value, while ep.A, lu.A and cholesky, which are cpu-intensive applications benefit from higher frequencies and threadcounts and report almost the same EDP pareto graphs. In the following chapters, where we explore and evaluate different scheduling policies, we will create workloads of applications that request thread numbers and frequencies according to their best EDP values on this section's charts.

5 Scheduling Policies

In this chapter we explore different scheduling methodologies that could be useful for our problem. First of all we analyze the gang scheduling methodology and study the positive effects and advantages that offers compared to the current Linux scheduling implementation. Then we explore two static state-of-the-art scheduling policies, one thread aware policy and one contention aware, and create a dynamic contention aware scheduler, based on the applications' miss rate. Finally, we classify our applications in four different categories and create a greedy application aware scheduling policy, which aims to maximize the overall system throughput and apply DVFS methods in order to prevent from unnecessary energy expenses. Because on modern processors like the Sandy Bridge, switching the frequency introduces delays in the order of microseconds, which is several orders of magnitude smaller than the used time quantum, selecting a suitable frequency on every task does not introduce a noticeable overhead

All these scheduling policies are based on gang scheduling. So for each scheduler we create gangs of applications and run each one for a time quantum, to complete a whole round. In every scheduling implementation we define and use 1 second as a time quantum. It is very important to highlight that due to NUMA-memory allocation issues, we limit our problem in scheduling inside one single NUMA package, which contains 8 cores sharing the last level cache and using a single memory node. This is important in order to discuss scheduling policies for CMPs generally and not focus on implementations to provide memory allocation only for systems using sandy bridge processors.

5.1 Gang Scheduling (GANG)

Gang scheduling is a scheduling algorithm for parallel systems that schedules related threads or processes to run simultaneously on different processors [17]. In most cases these will be threads all belonging to the same process, but they may also be from different processes, for example when the processes have a producer-consumer relationship, or when they all come from the same Message Passing Interface (MPI) program. Gang scheduling is used so that if two or more threads or processes communicate with each other, they will be ready to communicate at the

same time. If they were not gang-scheduled, then one could wait to send or receive a message to another while it is sleeping, and vice-versa. When processors are over-subscribed and gang scheduling is not used within a group of processes or threads, which communicate with each other, it can lead to situations where each communication event suffers the overhead of a context switch.

Our GANG scheduler does not implement space sharing, which means that every application is scheduled alone on the system for one time quantum and then wait for all the other applications to run once before it is scheduled again in a round robin fashion. We divide GANG scheduling into the three following scenarios:

- Full threads – Full frequency: on that case the scheduler has absolutely no information for the applications to be scheduled, so it creates gangs and fit one application on each gang that will run with 8 cores in the higher system frequency for its time quantum.
- Best threads – Full frequency: on that case every application arrives to the scheduler requesting the optimal number of threads according to its scaling ability. So the scheduler creates gangs that contain only one application again, but in this case each application is granted the requested cores and not all the available system cores.
- Best threads – Best Frequency: this is almost like the previous case with the difference that every application, except from threads, requests the optimal frequency to run too, so the scheduler enforces the system to run on the requested frequency when executing each application.

For example lets consider a given workload containing four different applications that request 8,6,4 and 2 threads, respectively. On the first scenario the gang scheduler will create the schedule illustrated on Figure 5.1-1, while the second scenario is depicted on the Figure 5.1-2.The second and the third implementations of Gang scheduling are, of course, more efficient for power and performance, as they exploit the provided information for an application's scalability and frequency scaling, but one the other hand, they require profiling for every application before it is submitted to the scheduler for execution.

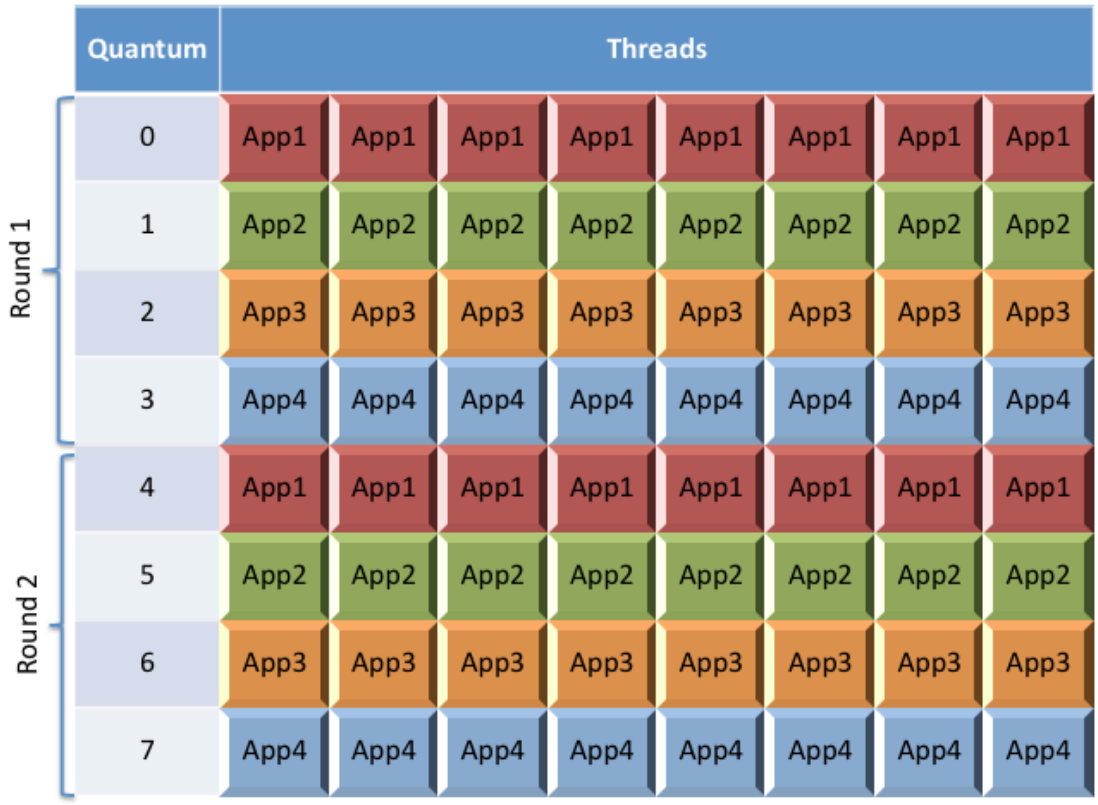


Figure 5.1-1 Gang Scheduling: Full Threads – Full Frequency

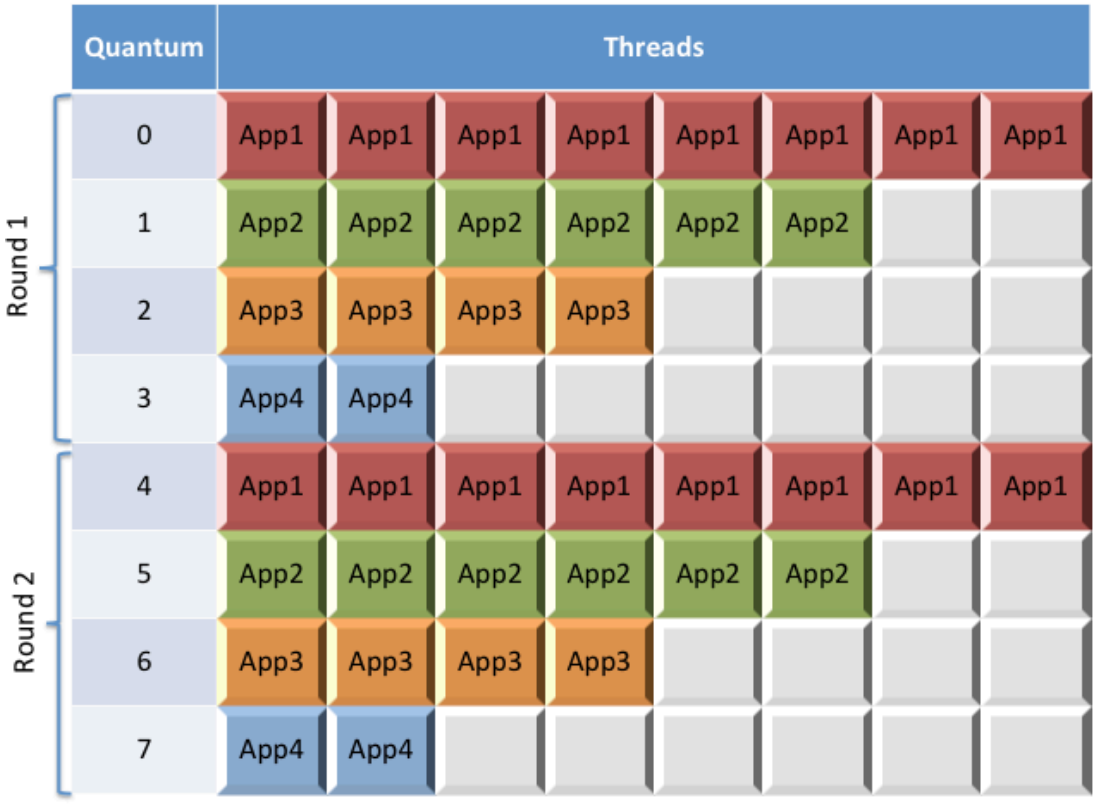


Figure 5.1-2 Gang Scheduling: Best Threads – Full Frequency

5.1.1 GANG versus Linux Scheduler

Modern scheduler implementations, like the Linux CFS scheduler [4], are designed to treat the threads of an application as single separate entities and distribute them across the available cpus, in order to provide a balanced execution. In this way the Linux scheduler avoids leaving any cores idle for a certain time-quantum. Although this scheduling policy achieves very high cpu utilization, it also results in threads of the same application being scheduled in different time-quanta, which in many cases undermines the progression of the application. On the other hand, Gang scheduling requires that threads of the same application must be scheduled in the same time-quantum. Therefore, applications gain from the benefits of simultaneously scheduling threads, such as avoiding large waiting periods in synchronization events (barriers), and locks better exploitation of data locality under shared-cache configurations, etc.

To see the difference between the Linux and the GANG scheduler we choose a representative workload of applications from our benchmarks and the ones we created with the STREAM software, according to their energy profile in section 4.5, that includes at least one application for every different power behavior.

workload	
application	requested threads
stream_cpu	8
ft.B	6
floyd-warshal	4
ep.A	8
is.C	3
stream_mem	2
jacobi	3
bt.A	8

Table 5.1.1-1 Random Workload

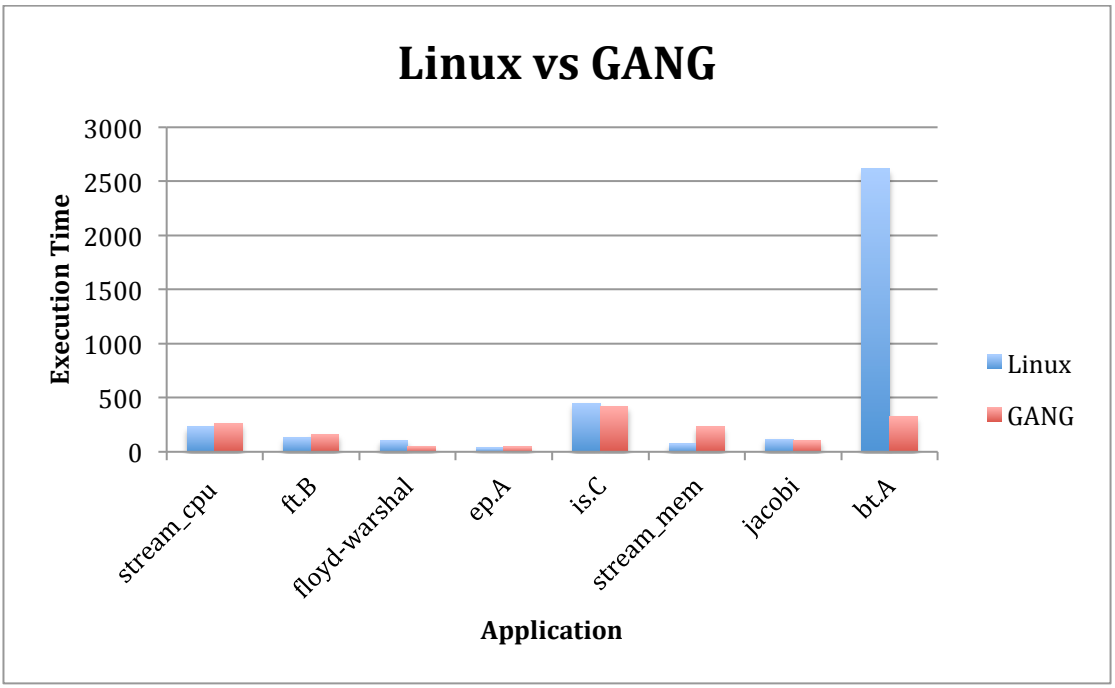


Chart 5.1.1-1 Linux vs Gang scheduling performance

In the above experiment we re-spawn every application that completes its execution to keep the system fully loaded all the time. As we observe not many applications of the workload benefit significantly from the execution with the gang scheduler. However, applications like bt.A result in extremely poor performance, which indicates a poor overall throughput for large executions. In contrast, gang scheduling seems to manage our applications more efficiently and reports small slowdown for some applications, while it helps others, like bt.A and is.C, to perform significantly faster. As a result, the total execution time to get our job done in a fully loaded system is about 5 times faster with gang scheduling than using the Linux scheduler.

Even though the gang scheduling methodology has been proved to provide higher throughput for multithreaded applications, it requires information about the applications' scaling ability and leaves many cores idle for long time, which decreases the total system utilization. So, we need to study gang-scheduling implementations that include space-sharing methods in order to fill our system's resources and increase the cpu utilization.

5.2 Proposed State-of-the-Art Scheduling Policies

In this section we explore two different proposed scheduling methodologies. The first on is thread aware scheduler that tries to achieve the highest possible system utilization, and the second one is a contention aware scheduler, which aims to co-schedule applications with different characteristics, in order to gain from the advantages of co-scheduling such applications.

5.2.1 Greedy Thread Scheduler (Static)

This is a scheduler suggested by McKee and Bhadauria [5] that tries to employ space-sharing in gang scheduling in order to utilize the available cores at the maximum degree. For that purpose, it implements a greedy bin-packing placement of applications into gangs, so that the percentage of unutilized cores in one round is minimized. The greedy thread scheduling algorithm takes the following steps:

1. It chooses the best-scaling program (the one that requests the most threads for its execution) from a set of sampled programs, and schedules it on the available system cores. If there are no remaining idle cores, it cannot find any more programs that could run in the current time quantum, so it runs the currently scheduled ones and repeats Step 1 for the next program on the list, and time quantum. If there are remaining cores, it proceeds to the next step.
2. If the set of unscheduled applications is empty, scheduling is finished. Otherwise, the scheduler chooses the next best scaling program from the set whose minimum processor requirement s met by the available idle cores on the system, and schedules it concurrently.
3. If there are empty cores remaining, Step 2 repeats, otherwise co-scheduling is finished for this set. If there are insufficiently many idle cores for any application, then thread counts of the currently scheduled programs are increased. The best scaling program's thread count is increased until performance stops to improve. The scheduler chooses the best scaling program since it has higher throughput with increasing number of threads and is less likely to overtly consume shared resources.

Figure 5.2.1-1 is illustrates the way that the greedy thread scheduler would handle the applications of the previous example. The scheduler chooses to co-schedule application 2 with application 4 and so it needs one time quantum less than the gang scheduler in order to execute every application for one time. If application 2 and application 4 run efficiently when co-scheduled together this schedule could give us an increased performance by 25% compared to the gang scheduler.

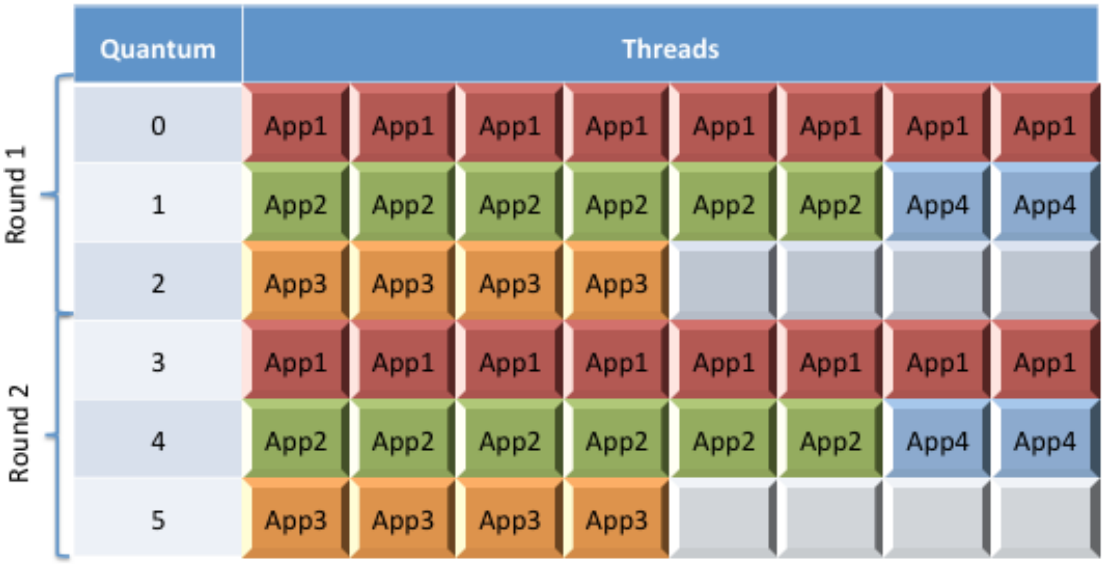


Figure 5.2.1-1 Thread Aware Scheduling Example

Even though this scheduler provides high utilization of the system's cores, its decisions are completely unaware of the contention the co-scheduling creates, since its decisions are based only on the level of parallelism for every application. As a result, this scheduler may experience significant performance problems when co-scheduling memory-bound applications together. So we need to study ways to efficiently co-schedule applications to reduce contention and make sure that the applied scheduling policy would improve the overall execution's performance.

5.2.2 Miss Rate Balance Scheduler (Static)

This scheduler tries to enforce a contention aware co-scheduling methodology, based on the profiles of the applications that need to be executed. It uses the last level cache (LLC) misses per thousand instructions ratio as a metric to evaluate the applications and create efficient gangs, which contain applications with different characteristics. The scheduler chooses the LLC miss ratio as an indicator of the contention an application causes, because LLC is the last on-chip shared resource, so misses in the LLC reflect the contention caused in LLC as well as in every off-chip subsystem, such as the memory bus and the DRAM controller. In that way it tries to separate applications into memory-bound (the ones with high last level miss rate) and compute-bound (the ones that report low last level miss rate because they rarely use

memory outside their private L2 cache) and combine them, so that they slightly affect each other during execution.

This scheduler is static, as it create its gangs at the beginning of the execution and never changes them during the execution, and it is completely based on previous profiling of the applications that was made before they enter the scheduling phase. So every application should be able to provide to the scheduler information about its miss rate. The scheduler's first step is to sort the applications according to their miss ratio, from higher to lower values. Then it chooses the application with the higher ratio and creates a gang to schedule it, and searches in the bottom of the sorted list of applications to find the application with the lowest ratio that could fit this gang, in order to co-schedule it with the already selected one. These two applications are removed from the list of applications that are waiting to be scheduled and the scheduler repeats the previous step until there are no applications left in the list. The following algorithm shows how the scheduler creates its gangs at the beginning of the execution.

```
Program List progs;      //List of programs to be executed
                         //contains all the programs
Gang List gangs;  //List of gangs
                         //initially empty

progs.quicksort();       //sort the programs list according
                         //to their LLC misses/thousand instructions ratio

while ( !progs.empty() ) do
      Program prog = progs.head();       // Get the head of the program's list
      Gang gang = Gang.create();   // Create a new gang
      gang.add(prog);                    // Add program to the gang
      gang->cores_allocated = prog->cores_needed; // update the
                                         // gang's allocated cores
      progs.remove(prog);                // remove the selected program
                                         // from the program's list

      //search for the suitable program with the lowest possible miss rate
      for_each_entry_bottom_up(Program temp : progs ):
            if ( temp->cores_needed + gang->cores_allocated<= system_cores) then
                  gang.add(temp); // add it to the created gang
                  progs.remove(temp);          // remove it from the programs list
                  break;
end if

done;
```

Code 5.2.2-1 Miss rate balance scheduling algorithm

For example, lets consider the scenario where we have to schedule 6 applications with increasing LLC miss rates, from application 1 to application 6. Every application requires 4 cores to run on, and our system provides 8 cores. The above scheduler would decide to co-schedule application 6 together with application 1, application 5 together with application 2, and application 4 together with application 3, creating that way 3 gangs for each round as illustrated on Figure 5.2.2-1.

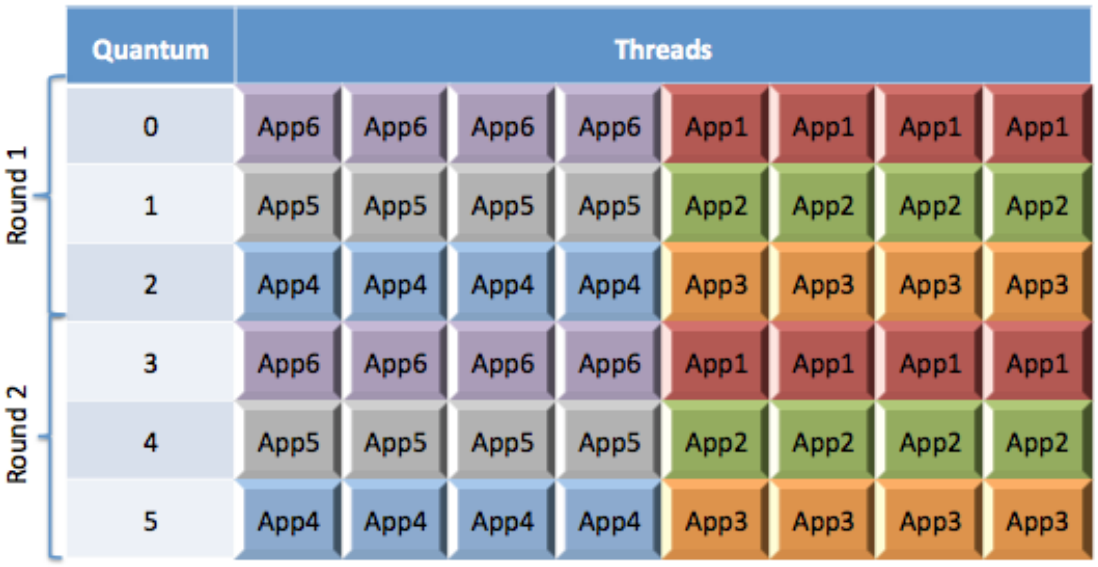


Figure 5.2.2-1 Miss Rate Balance Scheduling Example

The above policy is proven to be very effective because it exploits the advantages of co-scheduling applications with different characteristics and memory needs. Nevertheless, it allows only two applications per gang, which in cases of low threaded applications may result in leaving the system underutilized.

5.3 Miss Rate Bound Scheduler (Dynamic)

This is a dynamic contention aware gang scheduler that uses the LLC misses per thousand instructions rate (MPI), in order to detect contention based problems and deal with them during the execution. The main concept of this scheduling implementation is that it keeps a total LLC miss rate for every gang, which is measured as the sum of the individual rates of the applications of the gang, and ensures that this rate is below a defined threshold. This threshold is representative of whether high MPI values in a gang are affecting the performance of the applications running in it.

In order to define an appropriate threshold for our scheduler we create 5 different versions of the STREAM benchmark, and create gangs with increasing number of instances for every version. The different STREAM versions are designed to use 20%, 40%, 60%, 80% and 100% of the total memory bandwidth, respectively. Each instance is defined as the used application running with one thread on one single core in our system. We run the created gangs alone in the system and measure the execution time and the total gangs MPI for every execution. Then we report the slowdown of every execution according to the stand-alone execution time of one instance compared to the total gang's MPI. We assume that slowdown values greater than 2 are catastrophic because in such cases splitting the gang into separated ones would result in a better overall execution time. The following charts show the result we obtained:

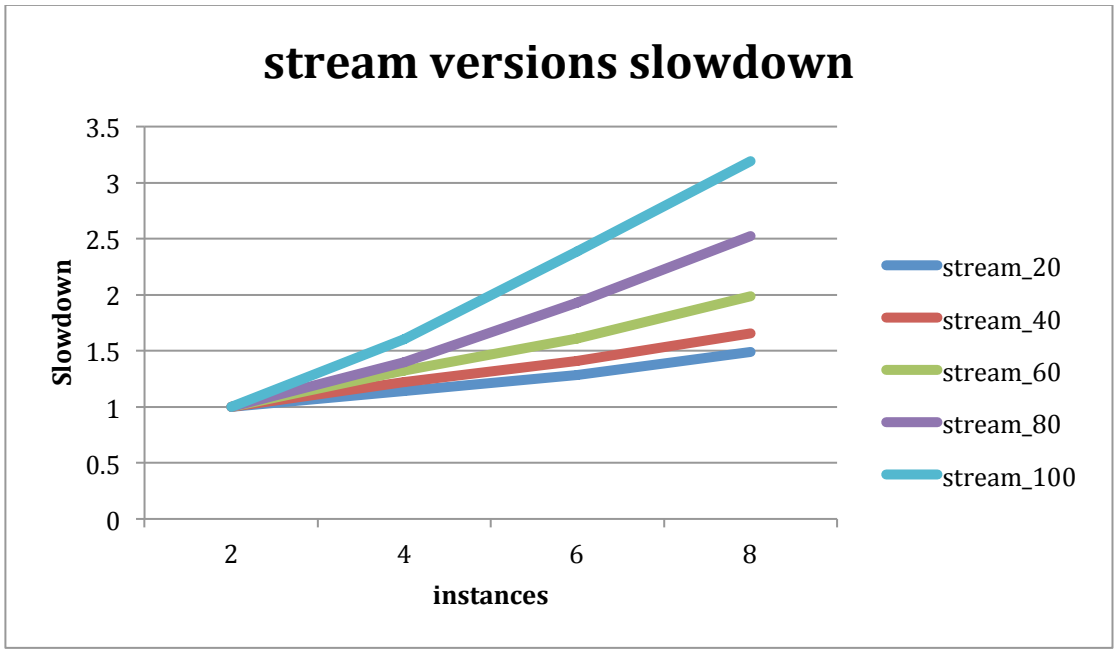


Chart 5.3-1 Stream Versions Slowdown

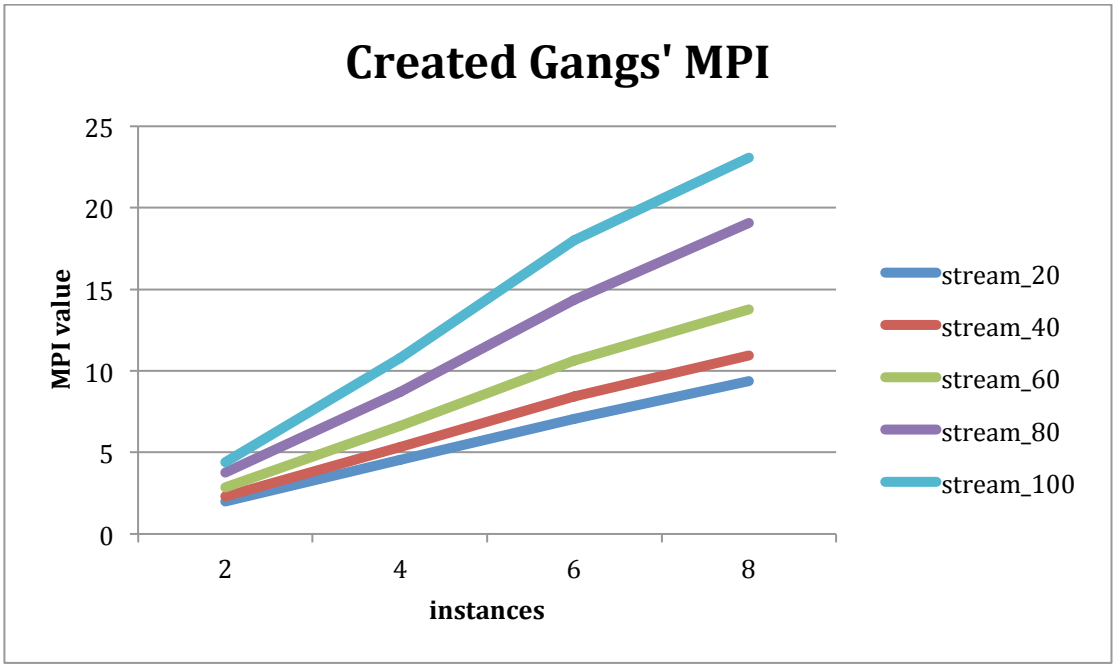


Chart 5.3-2 Miss per Instruction rates for gangs created with stream instances

From the results we assume values of MPI between 12 and 15 to be a representative threshold for cases that the MPI affects catastrophically the overall execution. Experiments with the NAS and the Polybench benchmark suites showed us that the best values for our threshold are actually a bit lower and lie between 10-12.5, according to every different workload. In our experiments we use 12.0 as the defined threshold for our scheduler.

The most interesting fact of the Miss Rate Bound scheduler is that when the execution begins, the only information needed is the requested threads by each application of the given workload. The first step of the scheduler is to randomly create gangs with zero gang MPI values and start its execution. While the applications are executing into gangs, the scheduler tracks information about every applications MPI value for every time quantum, in order to update the gangs MPI value. This value is obtained by the following equations:

$$gang.MPI_{current} = \sum_{program \in gang} program.MPI$$

$$gang.MPI = a * gang.MPI_{current} - (1 - a) * gang.MPI_{previous}$$

, where the value of the gang's MPI is calculated as an exponential average of the previous MPI value and the gang's MPI value for the current time quantum. The factor a determines the weight of the current value and the history. In our implementation we use $a = {}^{1}/_{2}$, in order to give equal weight to both.

When the MPI value of a gang exceeds the defined threshold the scheduler chooses the gang's application with the lowest MPI value that its removal would

cause the gang to run under the defined threshold, and search for another existing gang that the selected application could fit in. If that is not possible the scheduler chooses the highest MPI application and allocates a new gang for it. Finally, when the scheduler find gangs that contain only 1 application, it searches for an appropriate gang to schedule it, in order to de-allocate that gang and reduce the total gangs number. The previous are illustrated in the Algorithms 5.3-1 and 5.3-2:

```
program_fits_gang(gang, prog)
        if ((gang->cores_allocated + prog->cores_needed) && (gang->MPI + prog->MPI
<= threshold)) then
                        return true;
        else
                        return false;
        end if
```

Code 5.3-1 Checks if a program could fit into a gang

```
Gang List gangs;          //List of gangs
                          //initially empty

for_each( Gang gangs : g):
// if the gang contains only 1 program
// search for a another gang that could be fitted in
        if (g->progs_nr == 1) then
                for_each( Gang gangs - {g} : temp) do
                if ( program_fits_gang(temp,prog_min) then
                        g.remove(prog_min);
                        temp.add(prog_min);
                        return;
                end if
                done
        end if
        // if the gang's MPI is below the threshold then do bothing
        if (g->MPI < threshold) then
                return;
        end if
        // if it is higher then
        // find the lowest MPI program in the gang and removing it would
        // cause the gang to run below threshold
        prog_min = g.programs->head();
        for_each( Program g.programs : p) do
                if ((p->MPI <prog_min->MPI) && (g->MPI - p->MPI < threshold)) then
                        prog_min = p;
                end if
```

```
done
// and fit an appropriate gang to mere it into
for_each( Gang gangs - {g} : temp) do
        if ( program_fits_gang(temp,prog_min) then
                g.remove(prog_min);
                temp.add(prog_min);
                return;
        end if
done
// if the previous couldn't be done
// find the highest MPI program
prog_max = g.programs->head();
for_each( Program g.programs : p) do
        if (p->MPI >prog_max->MPI) then
                prog_max = p
        end if
done
// and create a new gang for it
Gang gang = Gang.create();   // Create a new gang
gang.add(prog_max);                  // Add program to the gang
gang->cores_allocated = prog_max->cores_needed; // update the
                                    // gang's allocated cores
gang->MPI = prog_max->MPI; // update the gang's MPI
g.remove(prog_max);          // remove the selected program
                            // from the program's list
```

Code 5.3-2 MPI gang balance algorithm

The above algorithms run at the end of every round of gangs in order to keep the execution gangs balanced. The main advantages of this scheduling implementation are that it is a completely dynamic scheduler that does not require much information about the given workload and that it can schedule more than 2 applications in a gang at the same time quantum. Moreover it can capture the behavior of the applications dynamically and take critical decisions to help the execution. However, this scheduling implementation may exhibit interior performance compared to the previous static one, because it could spend lot of time adjusting the gangs until it results in an efficient schedule.

5.4 Application-Aware Scheduler

In this section we develop a greedy application-aware scheduler based on all the previous observations. First of all, we divide our applications into 4 different categories, according to their memory behavior, and study the effects f co-scheduling different applications at the same time quantum. Then we create a scheduling policy based on the slowdown every different application causes to each other and finally, we study the effects of frequency scaling on the available scheduling decisions.

5.4.1 *Categories of Applications*

Every application has a different computational and memory profile and thus, different needs for the system's resources. In gang scheduling policies the most important characteristic a scheduler should be aware of for every application is its usage on shared resources, such as the last level caches and the memory bus. In this section we divide applications in four different categories according to their memory and computational behavior, in order to study the results of co-scheduling different categories together. For every one of the following categories we create an artificial application that represents the category and use it to study the contention between applications from different categories and create a greedy application–aware scheduler.

According to its needs we assume that every application could be classified in one of the following categories:

- CPU-Intensive: applications with high computational needs and low memory needs that could be satisfied by the first and second level caches. Applications included in this category often show great scaling abilities because they lack memory dependencies between threads. We choose as a representative application for this category ep (embarrassingly parallel) from the NAS benchmarks suite with the smallest available working set (class = A).
- Limited Memory Usage: applications with memory requirements that exceed the size of the low level caches and usually allocate small amounts of space in the system's last level cache (LLC). As a representative we create an application using the STREAM benchmark that uses about 25% of the system's memory bandwidth and performs 10 arithmetic operations for every memory transaction. This means that our application needs to transfer around 4 MBs from main memory to caches every second, which results in exceeding the capacity of the available L1 and L2 caches and stores a small amount of data in the LLC too.
- Memory-Intensive: applications with high memory needs that could even exceed the size of the LLC. As a representative we create an application using the stream software that uses about 80% of the total system's memory

bandwidth and performs only 1 arithmetic operation for every memory transaction. This application requires more than half of the LLC's capacity to store its data.

- Random Memory Access: applications with random memory access patterns. Applications in this category differ from the previous because they are unable to use the processor's prefetcher efficiently, and every cache miss results in large penalty times. As a representative for this category we create an application that allocates an array of integers with size that equals half the LLC size and create a random pattern to access every element of the array 1000 times. When this application runs alone on our system the first access would results in a "miss" for every element with a high penalty and every one of the next accesses would result in a "hit" and no penalty time. So it is vital for this application to reuse the data it stores in the LLC, because increasing its miss rates would result in dramatically increasing its execution time.

Based on the above categorization we run each one of the representative applications alone in our system, as long as together with each other one, in order to report the effects of co-scheduling applications from different categories. In the Chart 5.4.1-1 we report the slowdown of every execution compared to the stand-alone time for every application.

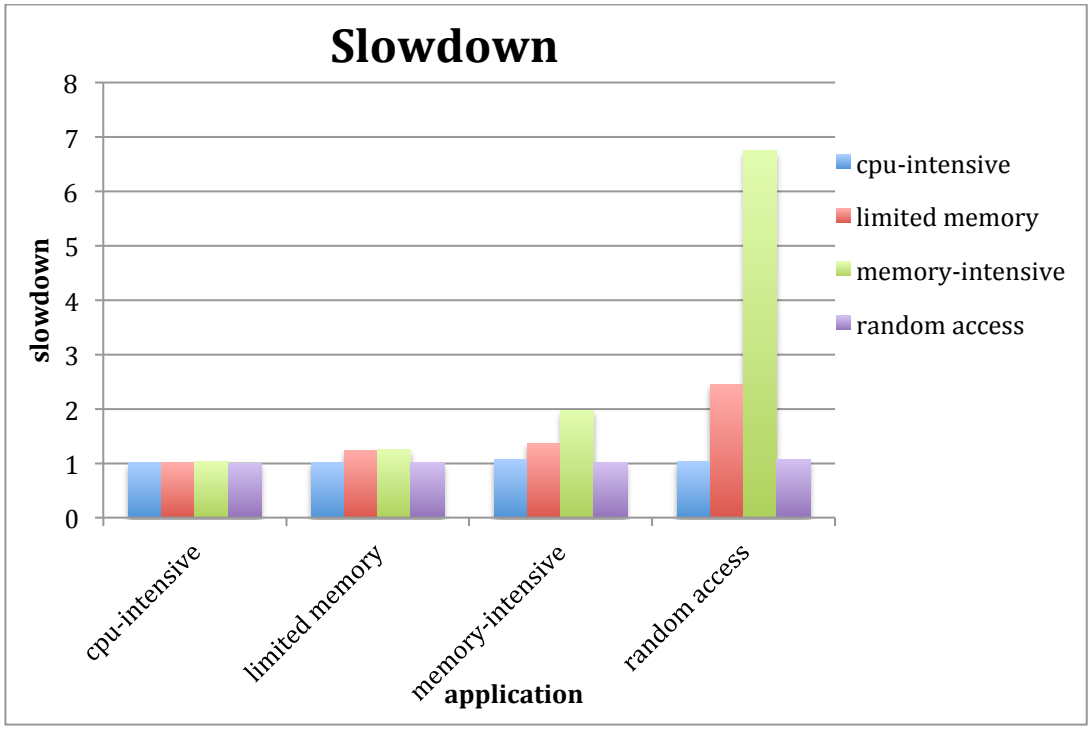


Chart 5.4.1-1 Artificial Applications Slowdown

As shown on the chart every category has a different behavior. For example, cpu-intensive applications run efficiently with an almost zero slowdown with every other application, while limited-memory-usage applications slowdown by a factor of

1.25 when running with memory intensive applications or applications from their own category. On the other hand, running together memory-intensive applications at the same time reports an almost 2 times slower execution, and running random memory access applications with limited-memory or memory-intensive ones could be catastrophic for their overall performance. As a result, it would be reasonable to assume that every one of our categories has certain co-scheduling preferences that are unique for every different category.

5.4.2 A Greedy Application-Aware Scheduler

Based on the above analysis we create a table that contains each applications preferences for co-scheduling. In the table lower values represent higher preference rates and dashes mark the undesirable co-scheduling combinations:

matching	cpu-intensive	limited memory	memory-intensive	random access
cpu-intensive	1	1	1	1
limited memory	1	2	3	-
memory-intensive	1	2	-	-
random access	2	-	-	1

Table 5.4.2-1 Application Aware Preferences

Based on this table we create a greedy bin packing scheduler that tries to leverage this co-scheduling information, in order to create gangs that could lead to increased execution throughput. For that purpose the algorithm divides programs into 4 lists according to their category and gives priority to the random memory access applications, as they are more likely to report performance problems. Then it handles the memory intensive applications by allocating a new gang for each one and schedules it there alone. After that, the scheduler accesses the list of cpu-intensive applications in order to fill the previously created gangs. If all the existing gangs have 2 applications and there are still programs left in the cpu-intensive list the algorithm creates new gangs for them. Finally, the scheduler deals with the limited-memory-usage programs list the same way it did with the cpu-intensive list. The exact way the scheduler schedules its programs into gang is presented on the Code 5.4.2-1.

```
Program List cpu-intensive;   // List of cpu intensive programs
Program List mem-intensive;         // List of memory intensive programs
Program List mem-limited;    // List of limited memory access programs
Program List random-access;// List of random access programs

Gang List gangs;  // List of gangs - initially empty

// First of all we handle the random-access applications
while (not_empty.random-access()) do
        // For every program check if there is an existing gang
        // necessarily with random-access applications because
        // they are the first ones we handle
        Program p = extract_head.random-access();
        flag = false;
        for_each(Gang gangs : g):
                ifprogram_fits_gang(p,g) then
                        remove.random-access(p);
                        g.add(p);             // and add the application to this gang if fits;
                        flag = true;
                end if
        if (!flag) then               // else create a new gang
                Gang g = Gang.create();
                g.add(p);             // and place the application
        end if
done

// Next handle the mem-intensive programs
// and create a new gang for each one
while (not_empty.mem-intensive()) do
        Program p = extract_head.mem-intensive();
        Gang g = Gang.create();
        remove.mem-intensive(p);
        g.add(p);
done

// Next handle the cpu-intensive programs and
// search for existing gangs they could fit in
while (not_empty.cpu-intensive()) do
        Program p = extract_head.cpu-intensive();
        for_each(Gang gangs : g):
                ifprogram_fits_gang(p,g) then
                        remove.cpu-intensive(p);
                        g.add(p);             // and add the application to this gang if fits;
                        flag = true;
                end if
        if (!flag) then               // else create a new gang for our application
```

```
            Gang g = Gang.create();
            g.add(p);           // and place tit
        end if
done

// Finally repeat the same process for the limited
// memory access programs list
while (not_empty.mem-limited()) do
        Program p = extract_head.mem-limited();
        for_each(Gang gangs : g):
            ifprogram_fits_gang(p,g) then
                remove.mem-limited(p);
                g.add(p);           // and add the application to this gang if fits;
                flag = true;
            end if
        if (!flag) then             // else create a new gang for our application
            Gang g = Gang.create();
            g.add(p);           // and place tit
        end if
done
```

Code 5.4.2-1 Greedy Application-Aware Scheduler's Algorithm for Creating Gangs

For example, lets assume that we need to execute a workload with 12 programs, 3 from each category, each one of them requiring 4 cores to run. Our system offers 8 cores under the same memory node for the execution. Figure 5.4.2-1 illustrates the way our scheduler will separate them into gangs, where rand stands for the random-memory access, mem for the memory-intensive, lim for the limited memory access, and cpu for the cpu-intensive programs' type.

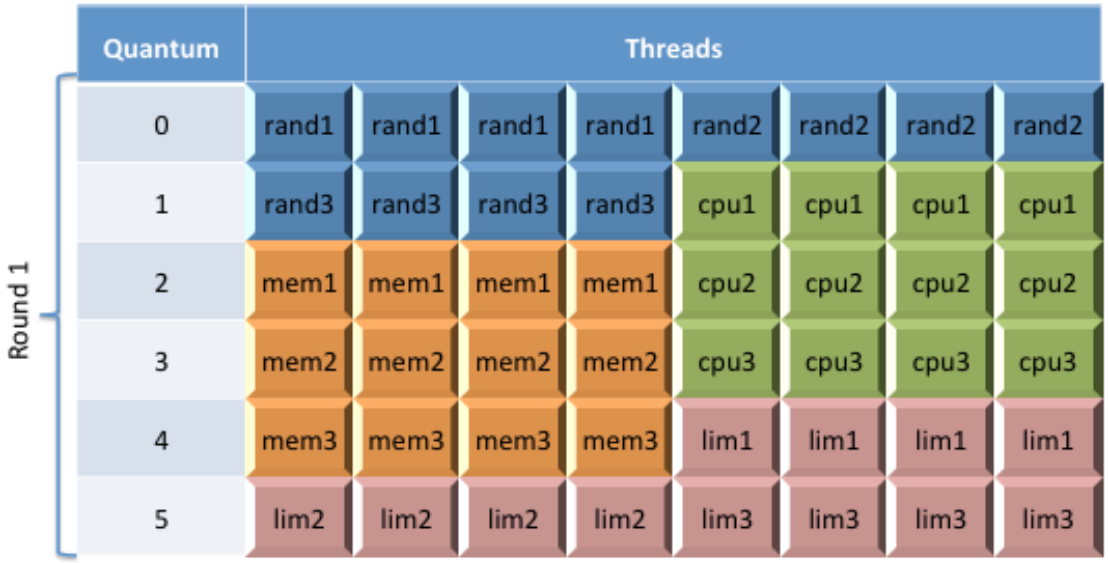


Figure 5.4.2-1 Greedy Application Aware Scheduling

5.4.3 Frequency Scaling

In this section we study the effects of dynamic frequency scaling on the gangs, which are possibly created from the above scheduler. As mentioned in section 4.3 the only cases that need to be studied are those where the executed programs use 60% or more of the memory bandwidth. Thus, in our implementation, because we use artificial applications created by us, the only cases where frequency scaling needs to be studied are: 1) co-scheduling random access programs together, 2) co-scheduling memory intensive programs with memory limited ones, and 3) co-scheduling memory intensive and cpu intensive programs. For that purpose we run each one of these three cases separately for all the available processor's frequencies and report the following results:

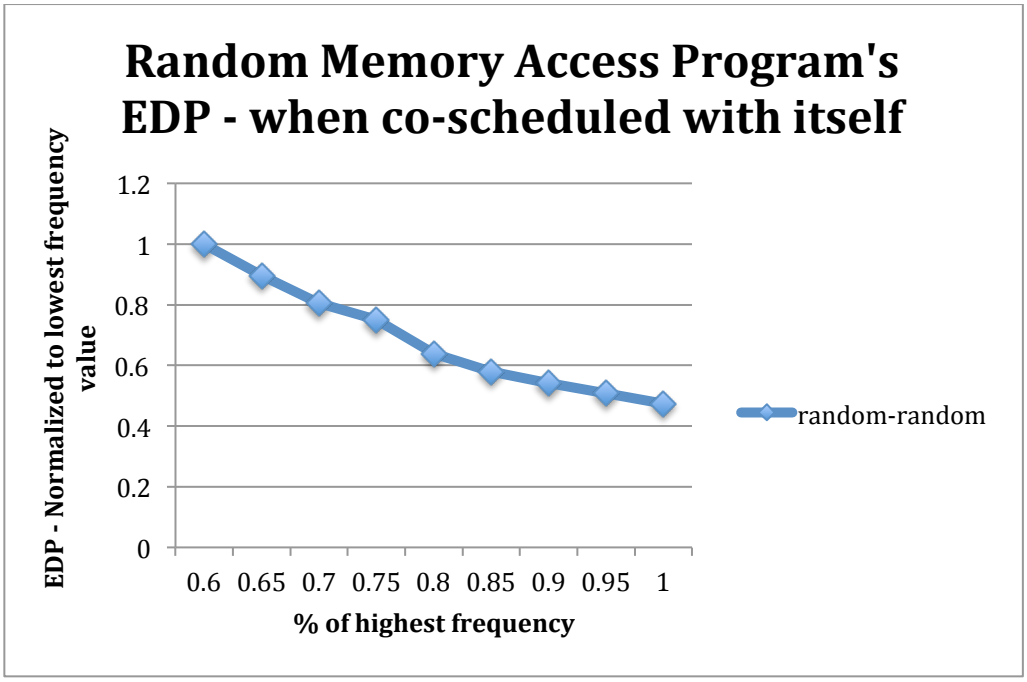


Chart 5.4.3-1 Random Memory Access Gang's EDP

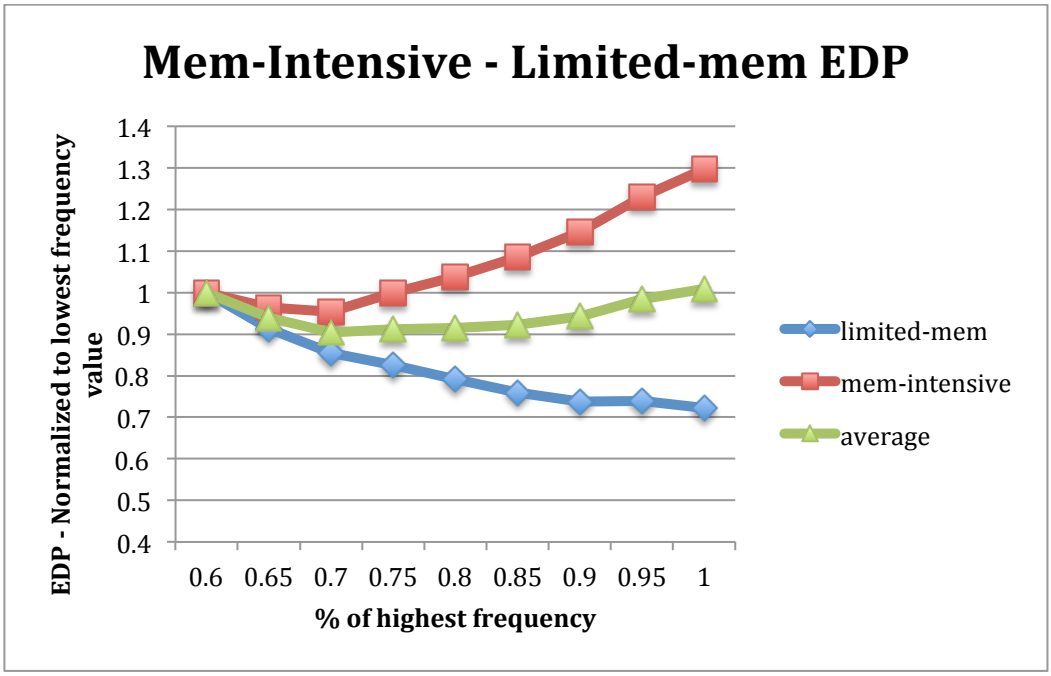


Chart 5.4.3-2 Memory intensive – Limited Memory Access Gang's EDP

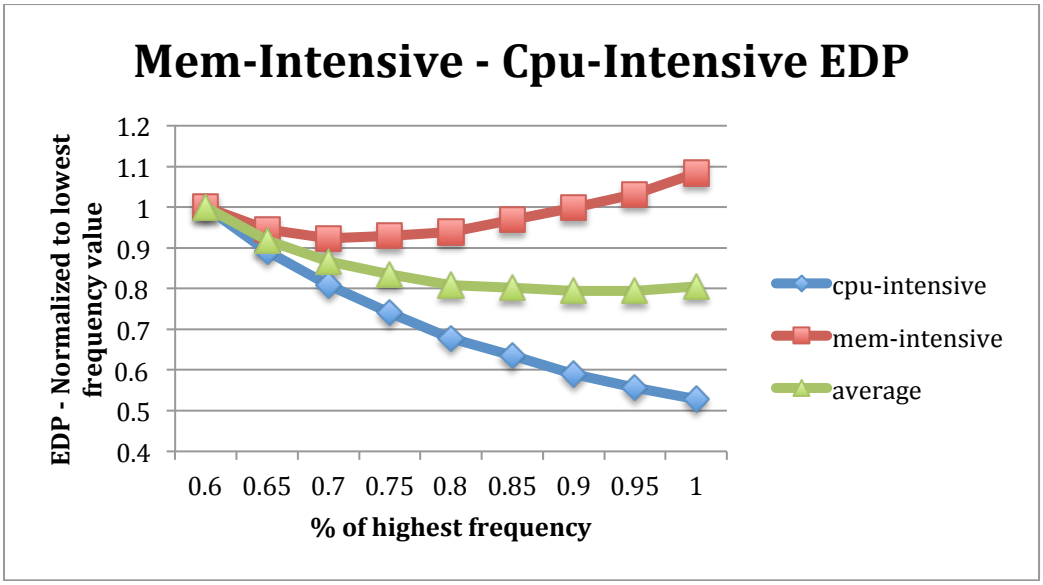


Chart 5.4.3-2 Memory intensive –Cpu intensive Gang's EDP

From the results we deduce that our random memory access application benefits from higher frequencies, because it stores all the needed data in the system's caches and leverages the system's speed to process it faster. On the other hand, gang's where our memory intensive is involved in seem to gain from lower frequency values, because the higher the processor's frequency is the more cycles they would have to pay as "miss penalty" each time the application stalls and waits for useful data to come. Although, in both such cases above, only the memory intensive application is benefited from frequency scaling and reports lower EDP values for 0.7 of the highest system frequency. So frequency scaling should probably be used only on the cores that our memory intensive program runs on, in order to achieve better results.

6 Experimental Evaluation

In this chapter we introduce some performance metrics and use them to compare the scheduling implementations that were presented on the previous chapter. Also we evaluate the importance of frequency scaling and measure the total energy that every scheduler consumes in a given time window.

6.1 Evaluation Metrics

In this section we define some metrics, which will help us compare schedulers and evaluate their characteristics. The metrics aim to highlight the overall throughput for the whole workload and the fairness towards all programs that are executed. Moreover, we consider the responsiveness of the system, by measuring the time a program waits for available cores and finally we compare the total energy consumption of every different scheduling policy. The used metrics for these purposes are the following:

1) **Execution Time**: we define as execution time for a program the time that the program needs in order to complete its execution in a fully loaded system. This include the time the program actually runs on some cores, and the time that it spends waiting for cores to become available.

$$t_{exec}(i) = t_{running}(i) + t_{waiting}(i)$$

2) **Throughput**: we define throughput as the number of times that a program completed its execution during a certain window of time. On our experiments we chose time windows big enough that allow every application to complete its execution at least 10 times, in order to obtain a better picture of the overall scheduler's throughput.

3) **Fairness**: we define fairness as the time that a program actually runs until it completes its execution in a scheduling implementation, compared to the time that the program needs in order to finish its execution alone in the system. So fairness is given by the following equation:

$$fairness(i) = \frac{t_{running}(i)}{t_{stand-alone}(i)}$$

, where lower values (near 1) are better.

4) **Average Waiting Time**: we define as waiting time the time that a program spends waiting for system cores to run on.

5) **Energy Consumption**: we define energy consumption as the total energy in Joules that was consumed in order to finish a requested job. This metric contains the energy consumed by the cores, the caches, the DRAM, and the memory controllers during the execution.

6.2 Importance of DVFS – Preliminary evaluation

In this section we use a workload that contains applications that could benefit from frequency scaling in order to report the reduction in energy consumption when applying DVFS during the execution. All the experiments below were made in the evaluation platform described in section 3.1.

6.2.1 Evaluation Workload

We create our workload by selecting applications based on their power profiling made in section 4.5. We chose 3 benchmarks that benefit by running on lower frequencies and 4 benchmarks that require the maximum available frequency. Moreover, we measure the LLC misses per thousand instructions (MPI), which is needed by the miss rate balance scheduler, and also categorize the selected applications based on their behavior when running along the categories representative applications mention in section 5.4.1. The table below presents all the needed information about our workload:

Application	Requested Threads	Requested frequency (MHz)	MPI	Category
Floyd Warshall	4	1400	2.32	Memory intensive
Bt.A	8	1500	4.94	Memory Intensive
Jacobi	3	1600	5.07	Memory Intensive
Lu.A	4	2200	1.43	CPU Intensive
Ep.A	5	2200	0.01	CPU Intensive
Is.C	3	2200	37.46	Random Memory Access
Atax_parallel	4	2200	5.81	Limited Memory Access

Table 6.2.1-1 Evaluation Workload

The requested threads by every application in the workload are based on the lower EDP values of the charts in section 4.5, and moreover, in benchmarks with great scaling ability, such as the *ep* and the *lu*, have been alternations from the best values in order to help our scheduling implementations to achieve high cpu utilization. At this point we mainly want to highlight the power consumption to solution, so these changes does not affect our results.

6.2.2 Frequency Scaling - Power Evaluation

In order to test the effects of frequency scaling on the needed energy to complete a given job, we use the workload in Table 6.2.1-1 and submit it to each scheduler twice. The first time we use the default system's frequency, which is the highest available, and the second we alternate our schedulers in order to enforce the system run every application at its requested frequency. As mentioned before the processors overhead to switch frequencies is of the order of milliseconds and our scheduling implementations use 1 second as a defined time quantum. So this overhead would not significantly affect our experiments. Moreover, we run the applications in the workload until each one finishes its execution at least one time, and re-spawn every application that finishes before that time in order to keep the system fully loaded all the time. Thus, we measure the energy consumption of each scheduling policy with and without frequency scaling, until the completion of a requested job in a fully loaded system.

First of we lets consider Chart 6.2.2-1 and Chart 6.2.2-2, where "ff" stands for full frequency and "bf" stands for best frequency that equals the frequency column on Table 6.2.1-1.

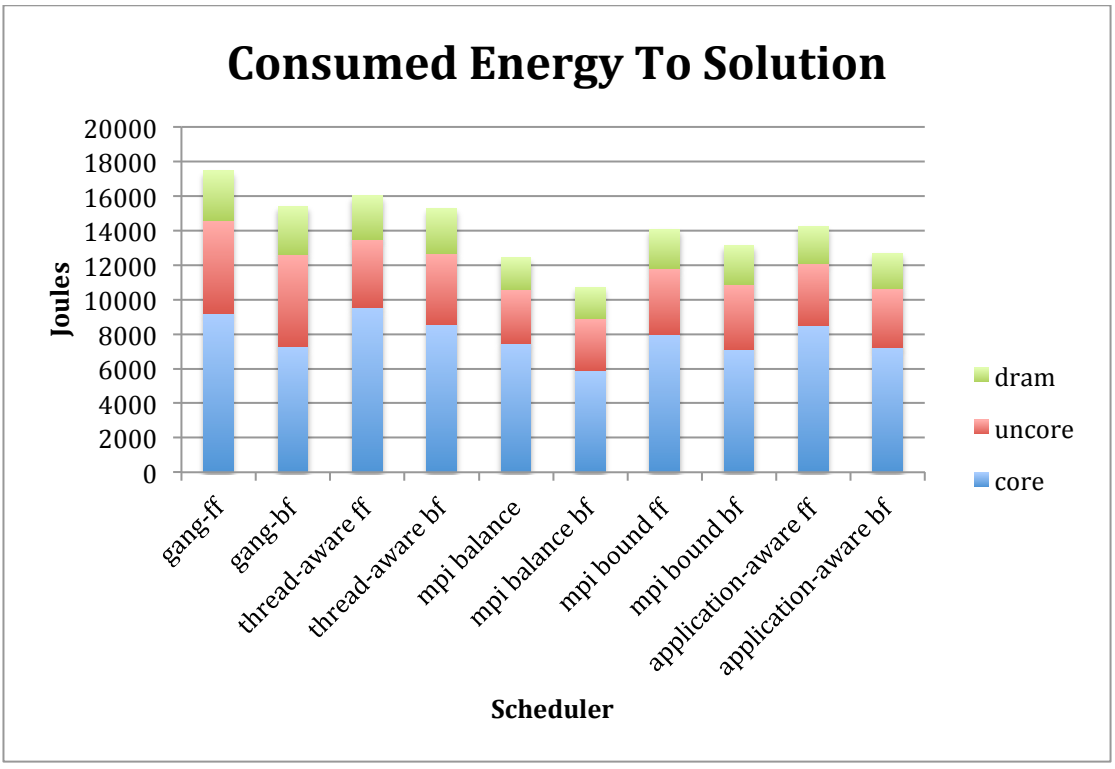


Chart 6.2.2-1 Energy to solution for each scheduler

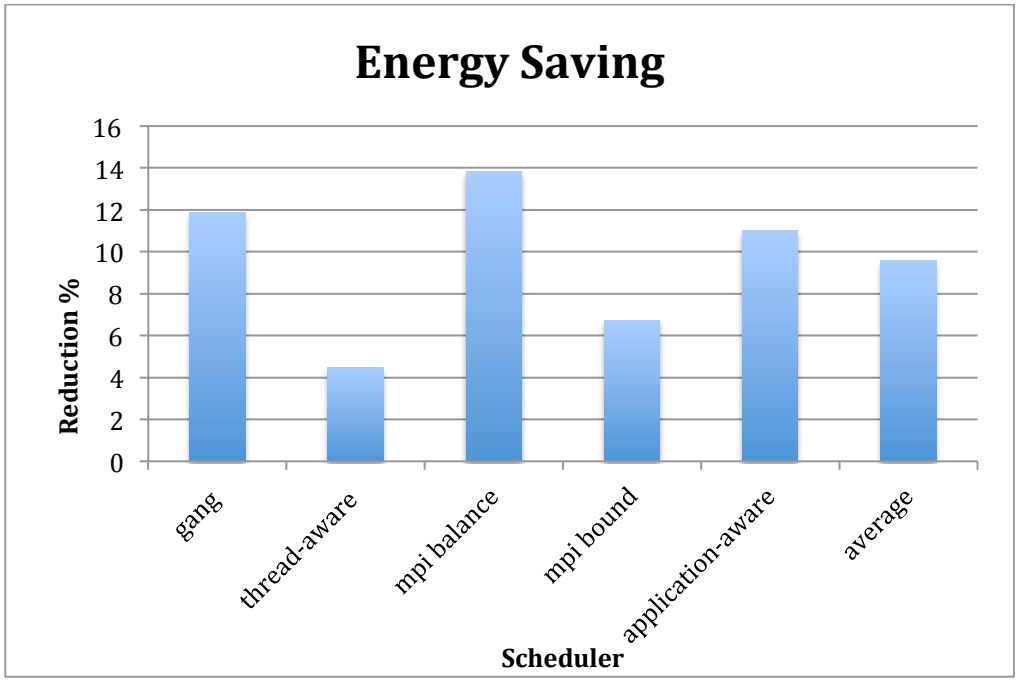


Chart 6.2.2-2 % Reduction of frequency scaling for each scheduler

We observe that applying dynamic frequency scaling to our execution results in energy savings that differ from 4% to 14%, and reports and average energy reduction close to 9% for our schedulers. Moreover, the miss rate balance and the application aware schedulers seem to gain more from the lower frequencies, as they report the greatest reduction percentage. Also, miss rate balance scheduler reports the lowest energy to completion, but that does not necessarily implies a better power balance across gangs. As mentioned before performance has also a vital role on the total energy consumption of a scheduler.

Chart 6.2.2-3 contains the execution times for every application of our workload, using each one of our schedulers, until it finishes for the first time. We can observe that the lower, and thus, better values are reported for the miss rate balance scheduler and the application-aware scheduler. Moreover, the miss rate balance scheduler finishes all the applications for at least one time each first (199 seconds), which explains the lower values of energy reported on Chart 6.2.2-1. The next one to exit was our implementation for the application-aware scheduler (230 seconds) 31 seconds later, which led us in consuming approximately 1900 Joules more for the job.

Finally, Chart 6.2.2-4 illustrates the average reduction between full frequency and best frequency values for each scheduler. As we see the reduction values lie between 0.4% and 3.5%, and more important between 0.4% and 1.2% for the contention aware scheduling approaches, which means that the performance is only slightly affected.

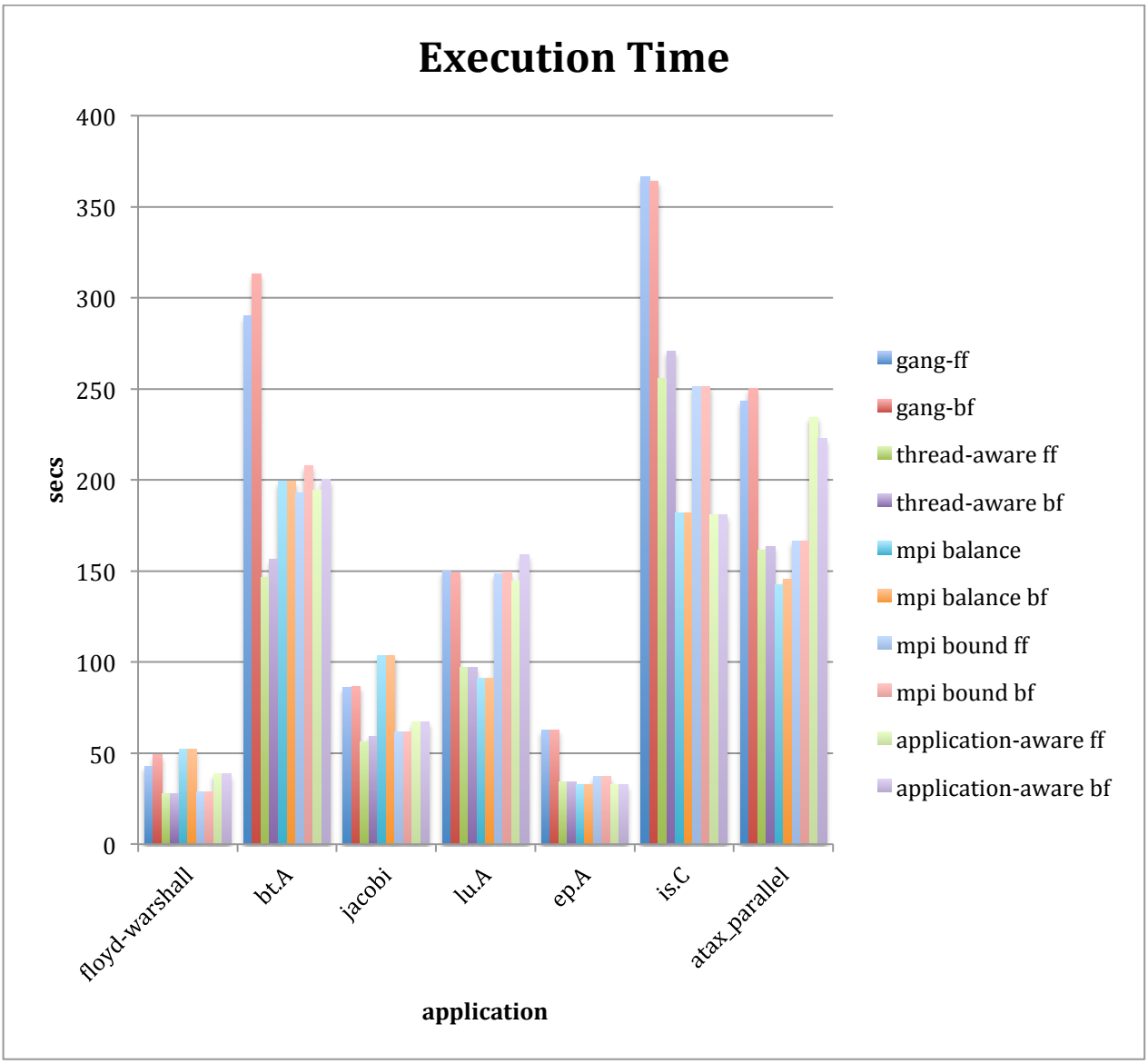


Chart 6.2.2-3 Execution time of every application for each scheduler

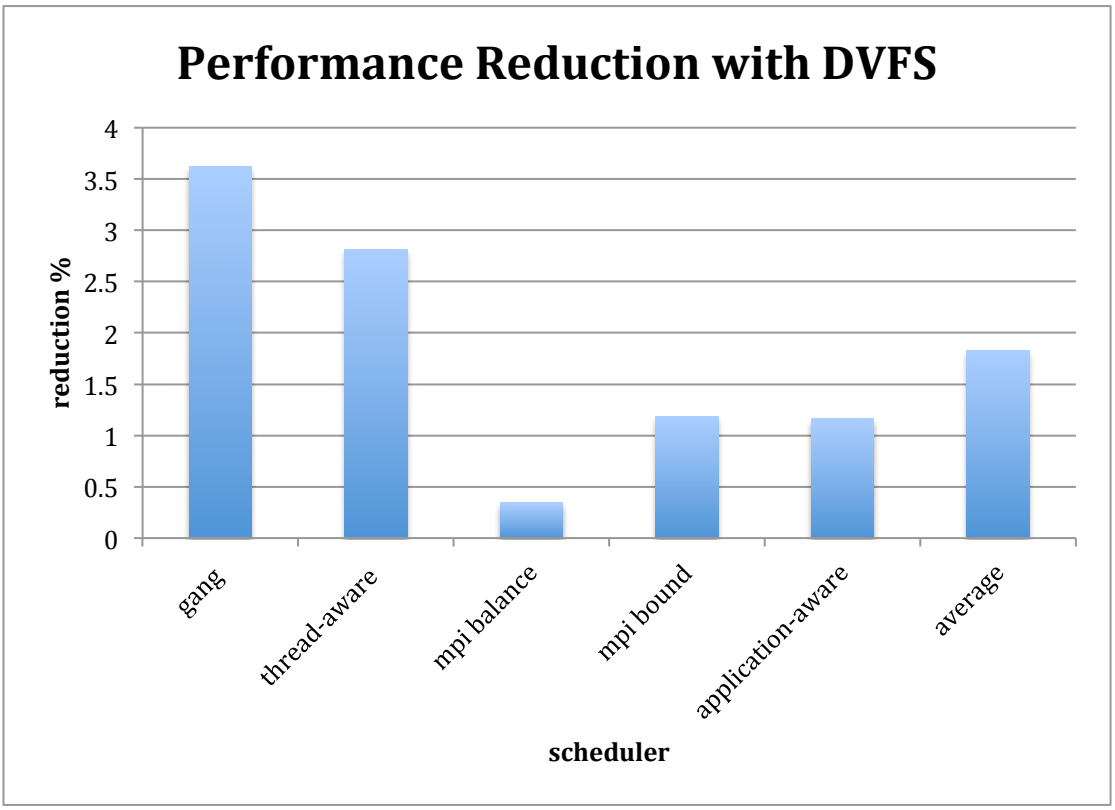


Chart 6.2.2-4 Performance reduction caused by applying frequency scaling on each scheduler

In conclusion, applying dynamic frequency scaling is proven to be very efficient for an energy saving scheduling policy without affecting the overall systems performance. Furthermore, in contention aware scheduling the energy consumption could be reduced by more than 10% with sacrificing performance less than 1.5%. So, knowing the best frequency values for every application and dynamically adjusting the cores frequency to it is critical for power-aware scheduling and its usage could have very important results.

6.3 Evaluation of Scheduling Policies

In this section we evaluate the scheduling policies discussed in chapter 5. First of all we create an artificial workload that contains the applications created in section 5.4.1, in order to highlight the performance of our application-aware scheduling model towards the other schedulers. Then we select applications from the

Polybenchand the NAS benchmarks suites to create a new workload and use it to re-evaluate our scheduling policies.

6.3.1 Artificial Workload

In order to highlight the overall performance of our scheduling implementation we create a workload based on the artificial applications in section 5.4.1. The workload contains 3 different versions of each one of the categories representatives. These versions differ only in the amount of threads that each applications requests, in order for our schedulers to create different gangs during their execution. Table 6.3.1-1 contains the evaluation workload.

Application	Requested Threads	MPI	Requested Frequency (MHz)
Cpu-intensive1	4	0.01	2200
Cpu-intensive2	5	0.01	2200
Cpu-intensive3	6	0.01	2200
Limited-Memory1	4	1.66	2200
Limited-Memory2	5	1.78	2200
Limited-Memory3	6	1.99	2200
Memory-Intensive1	3	5.02	1500
Memory-Intensive2	4	5.38	1500
Memory-Intensive3	5	5.57	1500
Random-Access1	2	0.01	2200
Random-Access2	3	0.01	2200
Random-Access3	4	0.01	2200

Table 6.3.1-1 Artificial Workload

The random memory access application has an almost zero stand-alone MPI value because it allocates a random access array and then access it 1000 times. So, running alone would result in only 1/1000 miss rate. Thus, it hides the information that running along a memory intensive application could be catastrophic for its execution. As a result, we expect that this workload would highlight the weaknesses of the state-of-art schedulers that use threads or miss rate as a metric to create their gangs.

6.3.2 Evaluation – Artificial Workload

We use the workload above to evaluate our schedulers. We make a configuration file that contains the workload and alternate the schedulers to re-spawn every application that finishes its execution, in order to keep our system fully loaded

for the whole execution time. We run every scheduler for two hours to ensure that every application of the workload completes its execution at least ten times. To evaluate the scheduler we use throughput, fairness, waiting time, and energy consumed as metrics.

- Throughput: In order to measure the throughput for every program we keep track of the times that it completed its execution and re-spawned. The scheduler's throughput is the average throughput value of the programs it executed. Chart 6.3.2-1 contains the throughput values for every application for every different scheduling implementation, as long as the average value that represents the overall scheduler's throughput. The results shown in the chart are normalized to the values obtained by executing the workload with the gang scheduler. Higher throughput values are better.

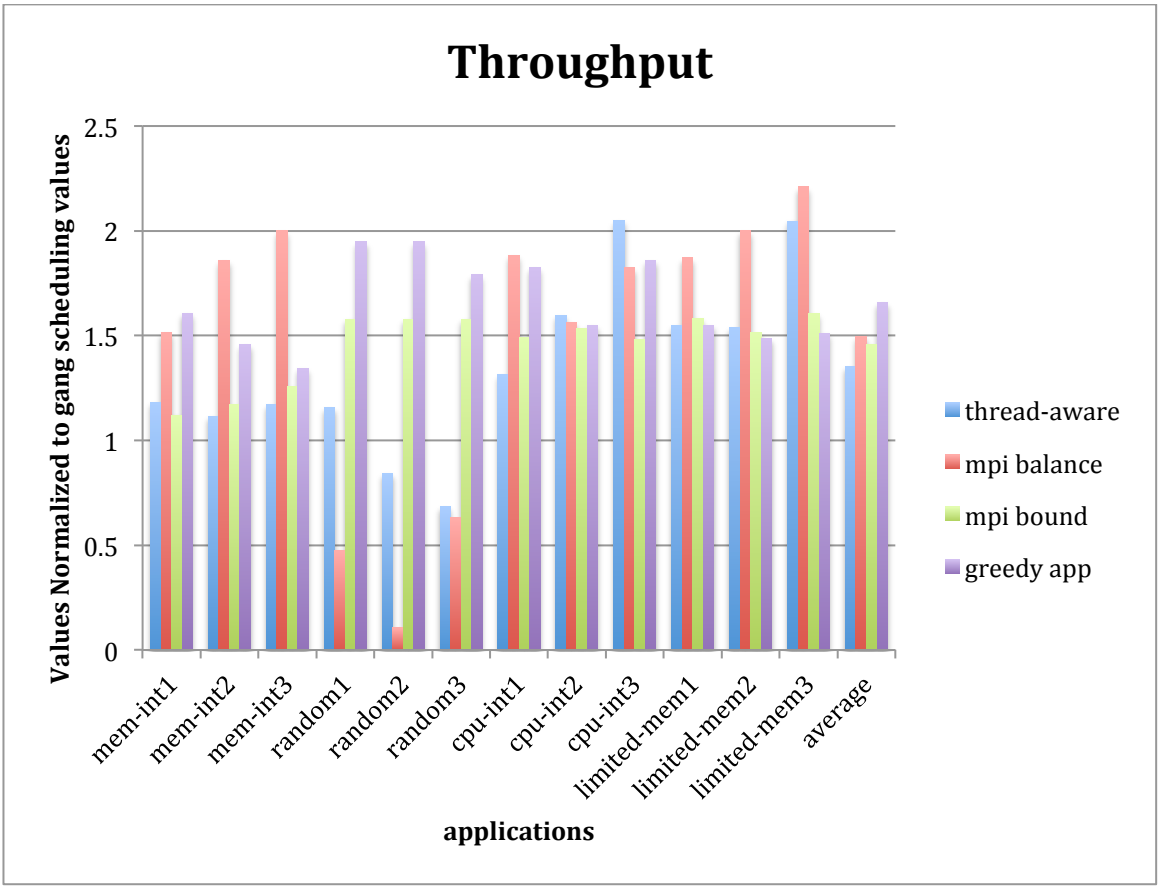


Chart 6.3.2-1 Throughput Normalized to Gang Scheduling Values

As we observe from the above chart, miss rate balance scheduler handles efficiently the memory intensive applications, but fails to handle the random access ones. Furthermore, in some cases, such as the second random access application, its decisions are catastrophic for the application's execution. The main reason for that is the random-access application's MPI that hides its actual behavior and leads the miss

rate balance scheduler to co-schedule it with a memory intensive application. On the other hand, the dynamic MPI bound scheduler captures the dynamic behavior of this application and re-schedules it alone in a new gang in order help its progress. As a result, the dynamic MPI bound scheduler reports throughput values over 1 and provides and efficient scheduling policy for the given workload. Finally, the application-aware scheduler produces the more efficient schedule for our applications and results in the higher throughput among the scheduling implementations with 1.65 more throughput than the gang scheduler.

- Fairness: In order to measure fairness for every application we keep track of its running time until its execution is completed. Moreover, because every application is re-spawned many times during the execution we store the average running time for each application. The fairness values are given by dividing the running time for each application, according to the current running scheduling implementation, with the running time reported when running with gang scheduler. We expect gang scheduler to give us lower running times because every application is scheduled alone for a time quantum in our system, and higher waiting times because of the more gangs it creates (equal to the number of applications).

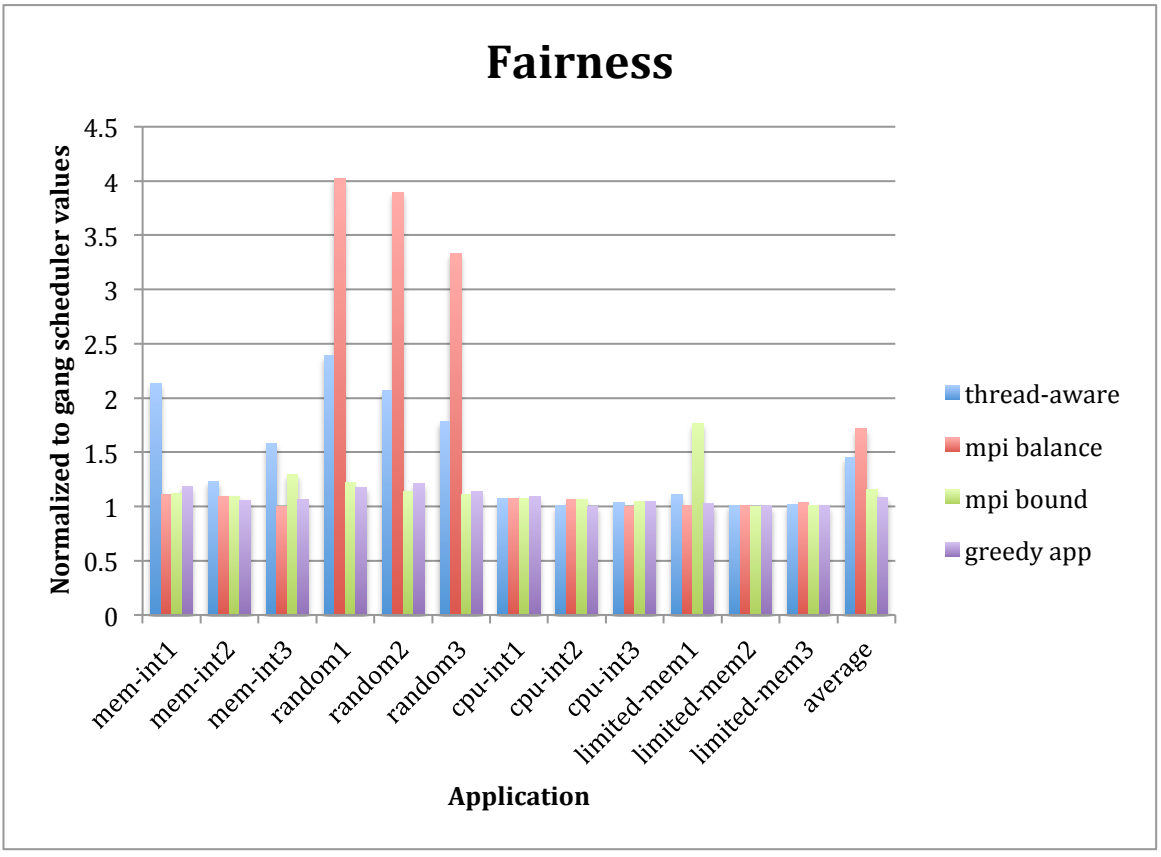


Chart 6.3.2-2 Fairness Normalized to Gang Scheduling Values

Chart 6.3.2-2 shows that MPI balance scheduler treats our random memory access applications unfairly, as their running time until completion is from 3.35 to 4 times higher than the gang scheduler's values. As a result it reports highest, and thus the worst fairness values. On the other hand we observe that our dynamic MPI bound scheduler handles applications efficiently and results in only 1.16 fairness values. Although, on that case this may mean that the scheduler chooses to create a lot of gangs in order to stay under the desired MPI threshold. Finally, the application aware scheduler seems to handle application well and create the most efficient gangs among our schedulers, which leads in an average fairness value below 1.1. Moreover, all the individual applications report fairness below 1.2 when running with this scheduler.

- Waiting Time: Our next step is to consider the average time each application stayed in our system waiting for cores to become available. A user-friendly scheduler should always report low levels of waiting time in order to respond to the user's needs. Chart 6.3.2-3 shows the average waiting time in seconds for each program, as long as the average waiting time for every scheduling implementation.

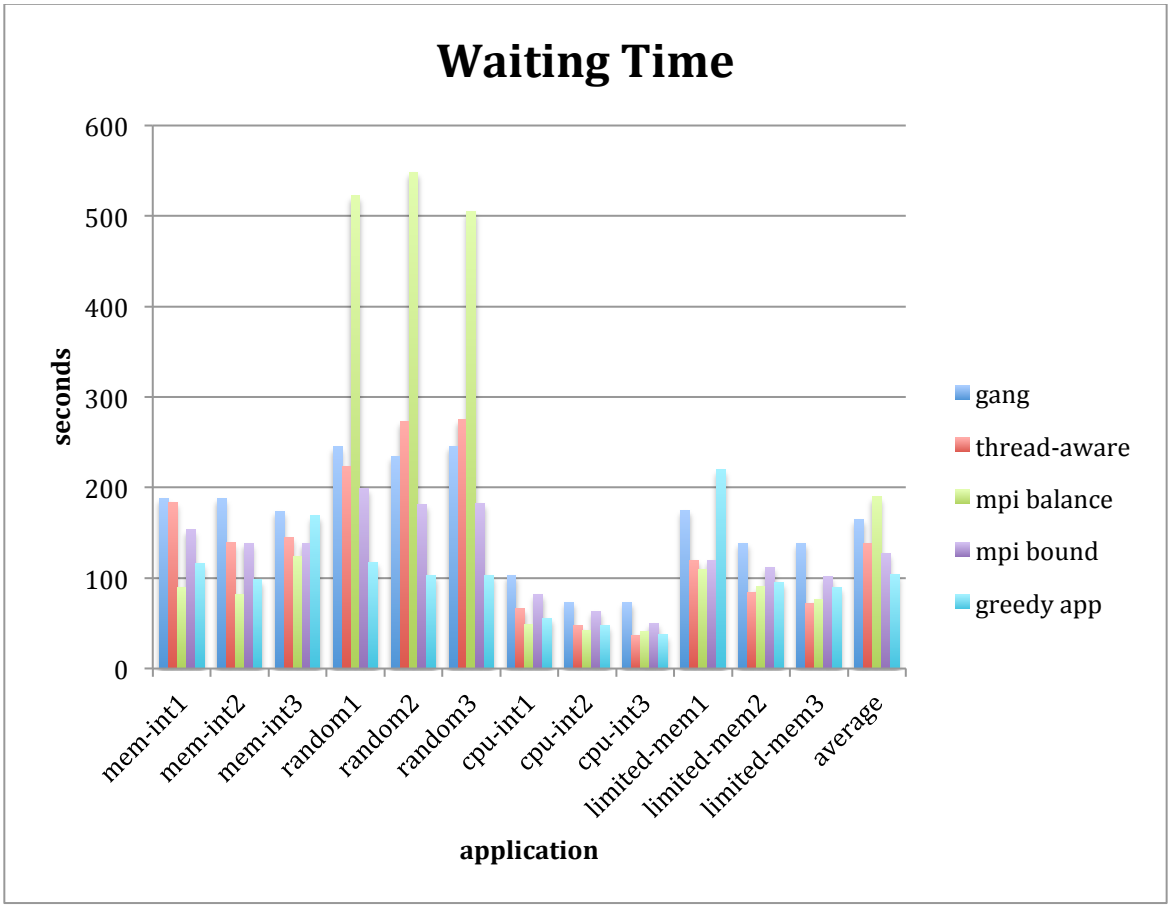


Chart 6.3.2-3 Average Waiting Time

As expected application aware scheduler reports the lower waiting times because it has the best throughput and fairness values. The highest average waiting values are reported from the MPI balance scheduler, because of the high random memory access applications' waiting times. Due to bad placement these applications need more running time to complete their execution and as a result more time quanta and more elapsed rounds, which increases the time they wait idle in the system for resources to become available.

- Energy Consumption: Finally, we report the total energy that every scheduler consumed for the whole execution, which means for running 2 hours. Chart 6.3.2-4 shows the reported values for the energy consumed in the core, the uncore, and the DRAM parts of the system:

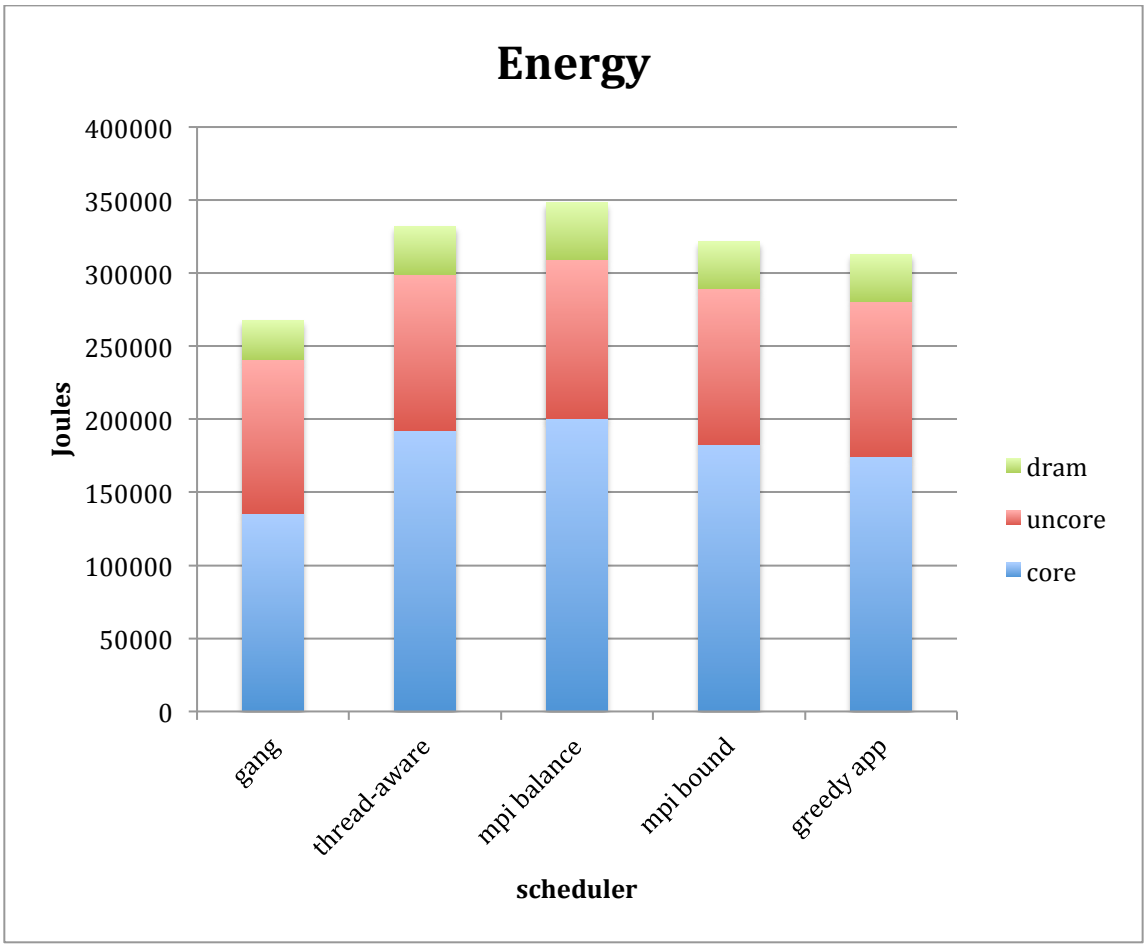


Chart 6.3.2-4 Total Energy Consumption

Even though gang scheduler reports the lowest energy consumption, it left many cores unutilized every time quantum. So the lower energy values reflect that a great part of the system is left unutilized and less work is done inside the given time window. On the other hand we observe application aware scheduler and miss rate bound scheduler to work efficiently and result in consuming less energy than the others. The main reason for this behavior is that they handle applications efficiently and produce gangs with low contention levels. By that way they prevent unnecessary LLC misses which cause the execution to halt and go back to the memory to regain useful data and thus, the system lowers the usage of DRAM and its memory controllers.

In conclusion, our implementation of the application aware scheduler reports the best values in all the previous evaluation metrics. In the defined 2 hours time window it does more job than the others (greater throughput), it is fair to every application and it reports the lowest average waiting time. Moreover, it reports the lowest energy for the whole 2 hours execution. Also the dynamic MPI bound scheduler seems to handle applications efficiently, given the fact that it is a dynamic scheduler with no knowledge for the applications before the execution. Because of its nature it captures the dynamic behavior of the random memory access applications and rearrange the produced gangs to place the with applications with low MPI rates, which are often cpu-intensive ones, or let them run alone inside a gang.

On the other hand, the static implementation of the miss rate balance scheduler seems to handle our random memory access application rather inefficiently. Because its stand-alone MPI values hide useful information it choses to co-schedule applications from that category with applications that need much LLC space to allocate for their data. As a result, MPI values dramatically increase and the applications spend time transferring the same data from memory to the LLC over and over again. So, the scheduler's decision force our applications to starve for data and reallocate the same data many times, which results in lower performance, as long as higher energy consumption.

6.3.3 Benchmark's Workload

In order to obtain a better picture of our schedulers we create a workload that contains applications from the Polybench and the NAS benchmarks suites. The categorization of the applications is based on their behavior when they are co-scheduled with each one of the categories representatives. We use is.C in three different versions (that request different number of threads) because it is the only random memory access application in both benchmark suites, and we want our workload to contain more than one application from each category. Table 6.3.3-1 contains the evaluation workload:

Application	Requested Threads	MPI	Requested frequency (MHz)	Category
bt.A	5	4.76	1900	Memory Intensive
Floyd-Warshall	3	2.28	1400	Memory Intensive
ep.A	6	0.01	2200	CPU Intensive
lu.A	5	1.43	2200	CPU Intensive
cg.B	5	2.99	2200	Limited Memory
is.C.2	2	36.43	2200	Random Access
is.C.3	3	37.46	2200	Random Access
is.C.4	4	38.28	2200	Random Access
atax_parallel	4	5.81	2200	Limited Memory
jacobi	3	5.07	1700	Memory Intensive

Table 6.3.3-1 Evaluation Workload

6.3.4 Evaluation – Benchmark's Workload

In this section we evaluate the above workload just like we did in section 6.3.2. We report the total throughput, the average running time and the average waiting time for each application, as long as the average values for each scheduler. Also we report the total energy consumption for the execution. We keep the same time window for each one of our executions, which is 2 hours and we apply frequency scaling as requested by every application in table 6.3.3-1.

- Throughput: Chart 6.3.4-1 contains the measured throughput, which is defined as the number of times each program finished its execution normalized to the same metric reported by gang scheduling. The results shows that each scheduling implementation favors different types of applications and all the static scheduling policies finally report close throughput values. The application aware scheduler is the only one that handles efficiently is.C versions and ends up with an average 1.5 throughput for the whole execution, which is the highest reported on the chart.

Moreover, we observe that the dynamic MPI bound scheduler lacks the ability to create a large number of efficient gangs and ends up with the lowest average throughput, only 1.22 times higher that the gang scheduling.

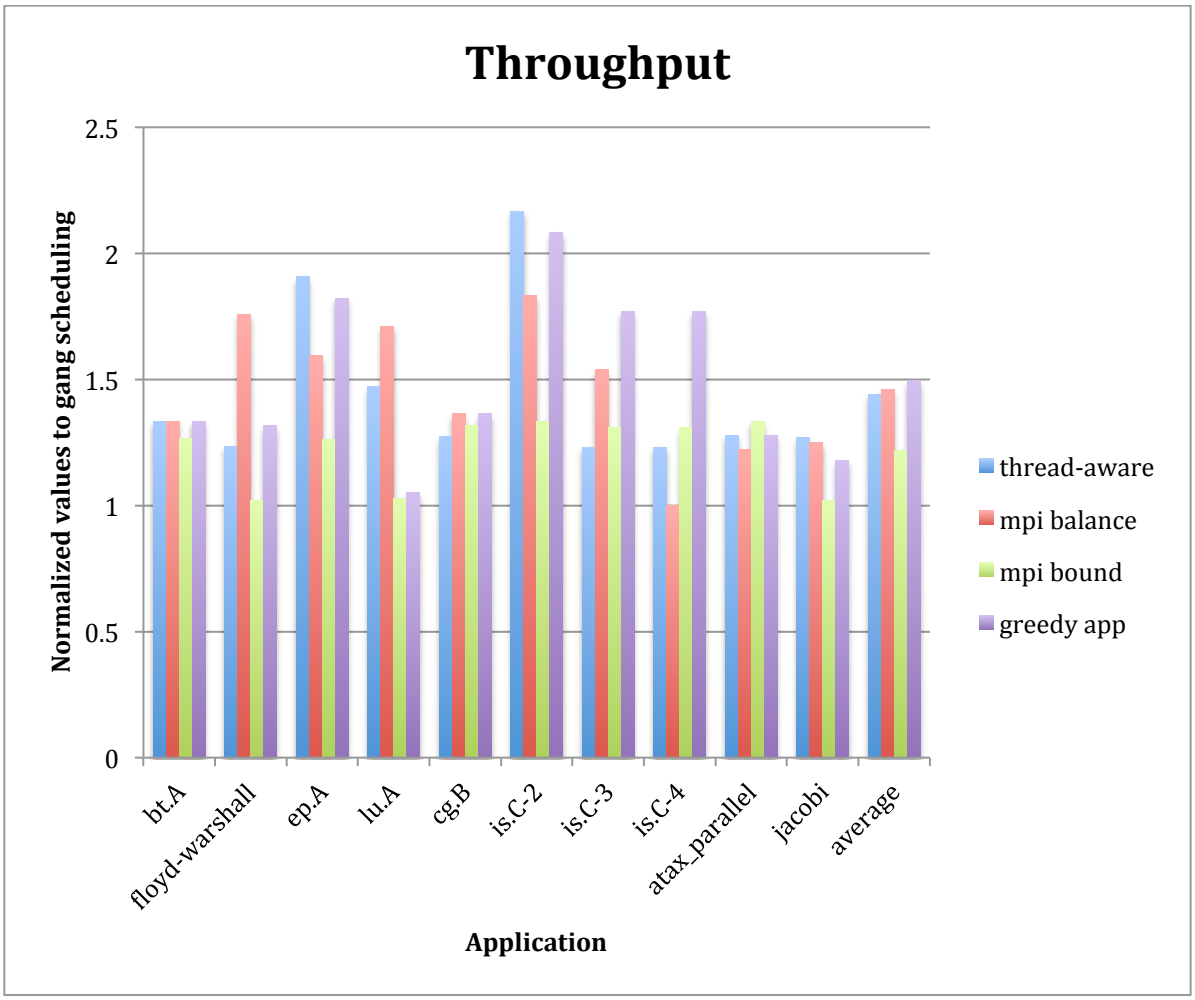


Chart 6.3.4-1 Throughput Normalized to Gang Scheduling Values

- Fairness: Like in section 6.3.2 we report fairness as the time an application was executing on cores of the system divided to the same time reported by gang scheduler. We expect gang scheduling to give us smaller running time and higher waiting times for our applications, as it runs every application alone (so there is no competition for resources), but more gangs to run in every round. Chart 6.3.4-2 illustrates the fairness values reported for every application with every different scheduling policy, and the average fairness values for each scheduler.

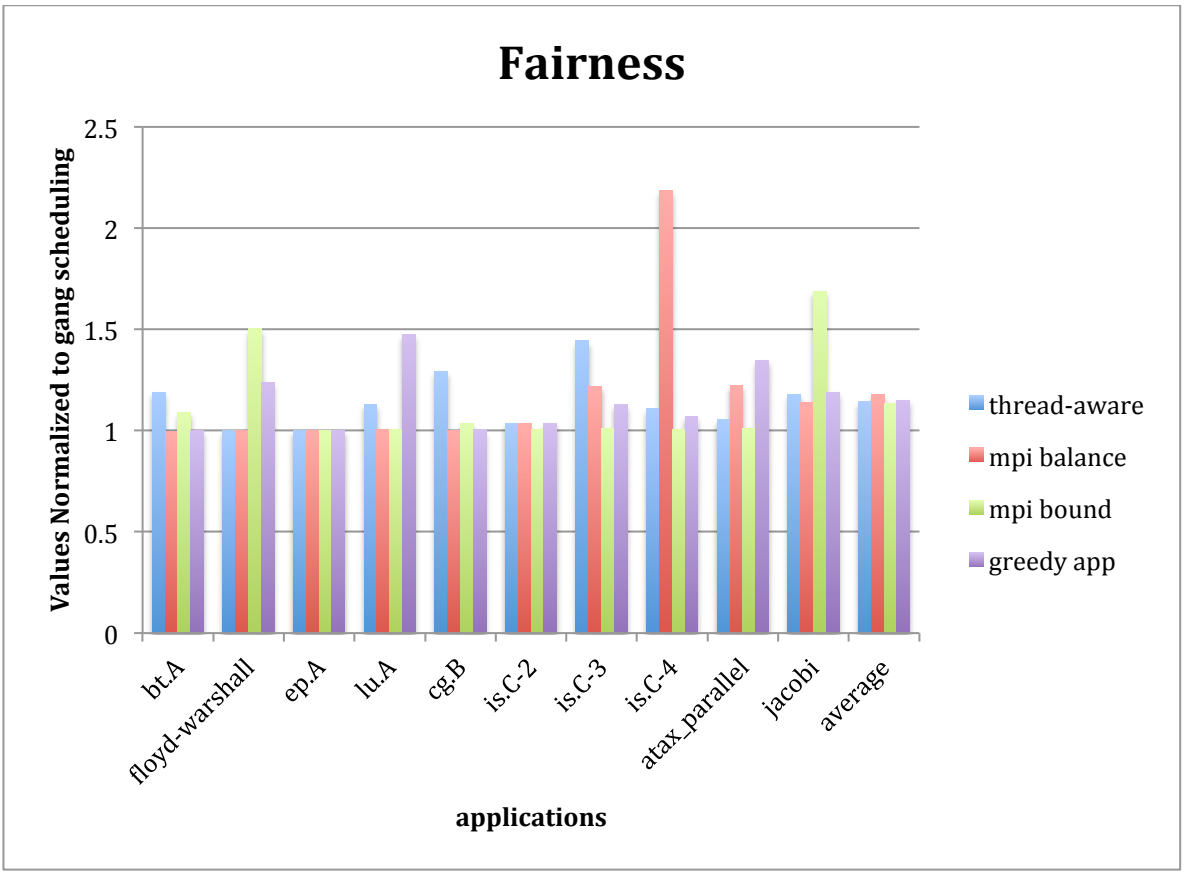


Chart 6.3.4-2 Fairness Normalized to Gang Scheduling Values

Even though MPI bound scheduler achieve the lower values for the majority of the application, this indicates that it results in creating lots of gang with just one application. Thus, we cannot assume that MPI bound scheduler is more efficient without being aware of the throughput values reported above. Moreover, we observe that all the scheduler finally report almost the same average fairness values, so it is hard to pick one of these schedulers based on that metric.

- Waiting time: Moreover, we measure the average time each application had to wait until cores became available for its execution. As we see from Chart 6.4.3-3 scheduling policies that create less gangs, such ass the thread aware, the MPI balance, and the application aware scheduler, report lower waiting times than the others. Moreover, application aware scheduler reports the lowest average waiting time, which may indicate that it creates the most efficient gangs among our schedulers that let applications complete their execution faster.

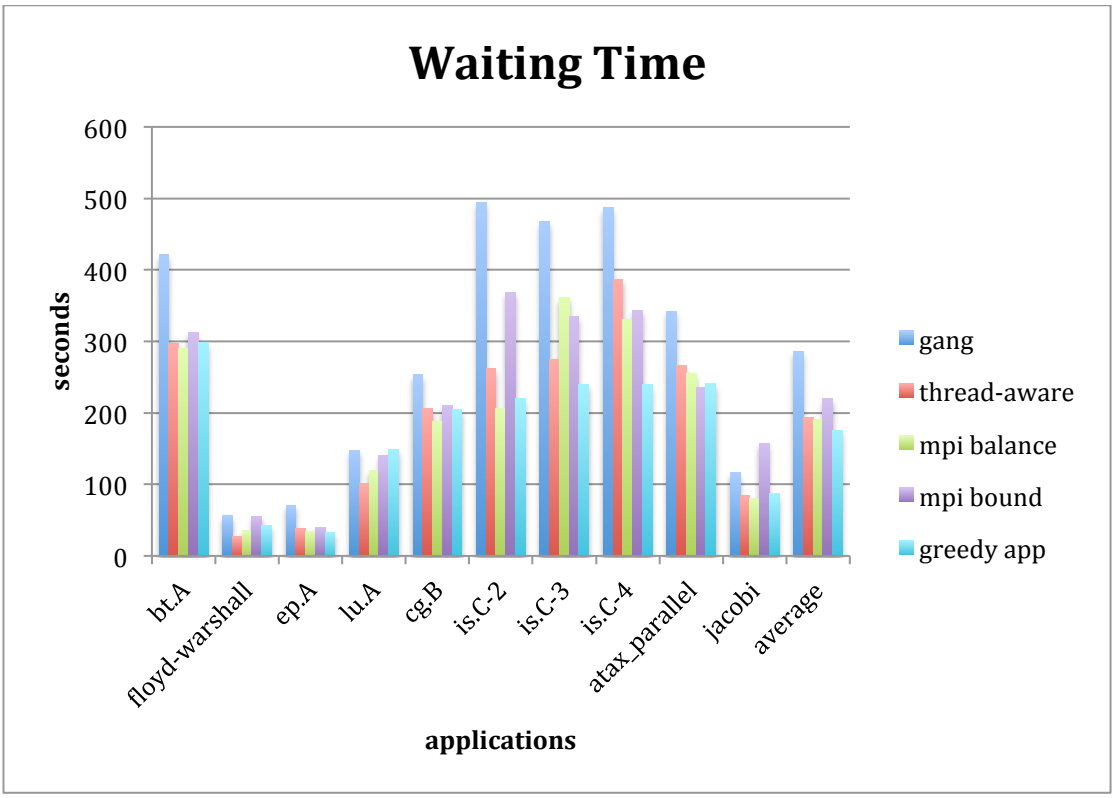


Chart 6.3.4-3 Average Waiting Time

- Energy Consumption: Finally we report the total energy consumption, in Joules, for the whole 2 hours execution time in Chart 6.3.4-4. We observe that gang scheduler and MPI bound scheduler report the lowest energy consumption, but that's due to the fact that they created more gangs than the others and left our system less utilized. The other three scheduling approaches report almost the same levels of energy needs for their execution.

In conclusion, the benchmark's workload led our schedulers into handling less corner cases, than the artificial one, and thus, lowered the differences between scheduling policies. Even thought, applications aware scheduler still managed to produce the most efficient schedule and report the higher throughput and the lower waiting time average values. On the other hand, our dynamic MPI bound scheduler lost a lot of time rearranging its gangs and resulted in higher gangs number than the static schedulers and as a result it

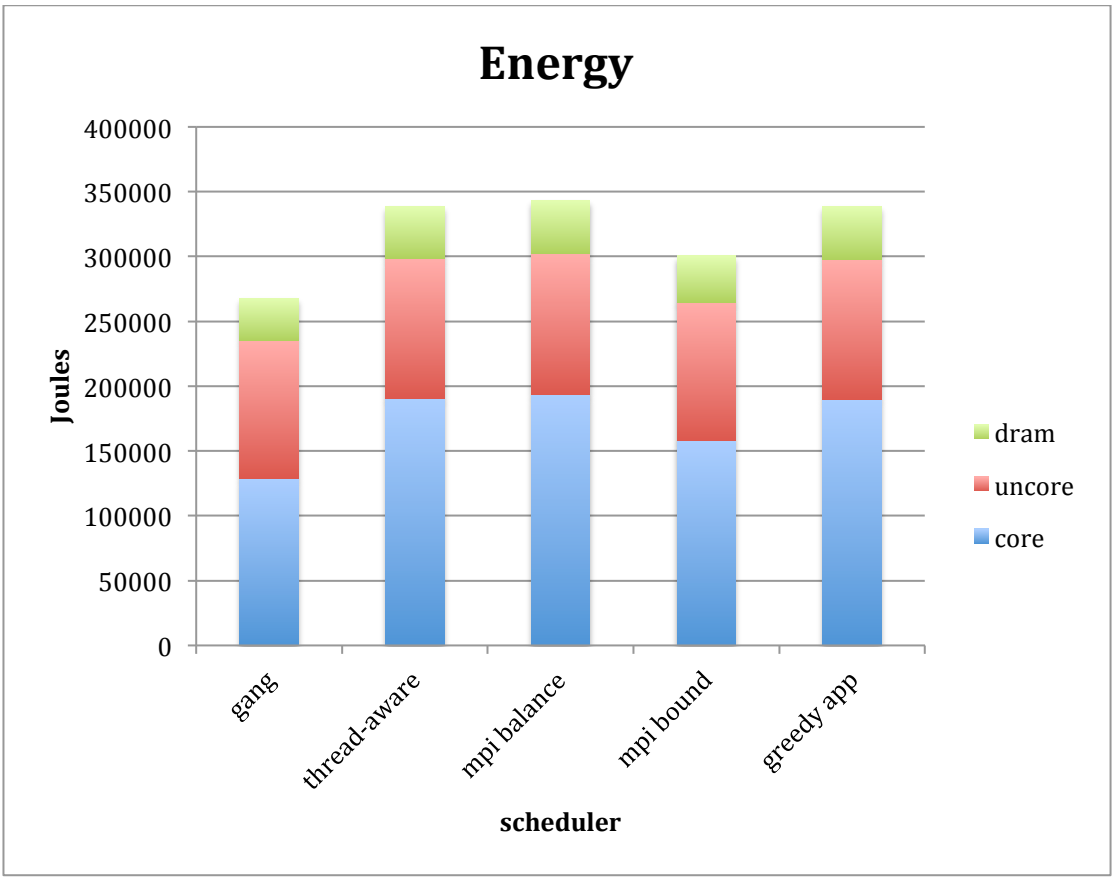


Chart 6.3.4-4 Total Energy Consumption

7 Conclusion and Future Work

As the number of used portable devices that run under reserved power is constantly increased and more servers are necessary to run continuously to support the needs of today's huge networks, power aware scheduling is an emerging issue. In this work we explored scenarios where energy aware decisions could be made in order to prevent a system from unnecessary energy expenses and highlighted that the knowledge for an application's scalability and memory needs could be very important for energy aware decisions. As modern processors contain multiple cores and support dynamic frequency and voltage scaling we found that applications could report significant differences in performance and energy consumption according to the allocated cores and the frequencies applied on them. Moreover, we explored the advantages of gang scheduling methodologies and studied ways to provide efficient schedules for execution. So, we categorized our applications and created a theoretical application aware scheduler, which proven to be efficient even in cases of non-artificial application workloads compared to state-of-the-art policies. Finally, we made an approach to a dynamic miss rate balance gang scheduling methodology that tries to create efficient gangs by keeping their overall miss rate values below a defined threshold.

As an expansion of this work we could study ways to dynamically identify different categories of applications and create a dynamic expansion of our application aware scheduler. Moreover, we could try to expand the defined applications' categories and alternate our scheduling policy, in order to deal with more cases that could appear during an execution and created more efficient schedules.

REFERENCES

[1] http://en.wikipedia.org/wiki/Scheduling_algorithm Scheduling (Computing)

[2] **http://www.ibm.com/developerworks/linux/library/l-scheduler/** Inside the Linux scheduler

[3] http://en.wikipedia.org/wiki/O(n)_schedulerO(n) Linux scheduler

[4] https://www.kernel.org/doc/Documentation/scheduler/sched-designCFS.txt

[5] M. Bhadauria, S. A. McKee. 2010. An Approach to Resource-Aware Co-scheduling for CMPs. In ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing.

[6] K. Singh, M. Bhadauria, S. A. McKee. 2009.Real time power estimation and thread scheduling via performance counters. InACM SIGARCH Computer Architecture News Volume 37 Issue 2, May 2009, Pages 46-55

[7] G. Dhiman, G. Marchetti, T. Rosing. 2009. vGreen: a system for energy efficient computing in virtualized environments. InISPLED '09 Proceedings of the 14th ACM/IEEE international symposium on Low power electronics and design, Pages 243-248

[8] A. Merkel, F. Bellosa. 2008. Memory-aware Scheduling for Energy Efficiency on Multicore Processors. In HotPower'08 Proceedings of the 2008 conference on Power aware computing and systems

[9] W. Stallings. Operating Systems: Internals And Design Principles. 7th Edition. Published 2009.

[10]http://en.wikipedia.org/wiki/Non-Uniform_Memory_Access Non-Uniform Memory Access

[11] P. Kaminski. “NUMA aware heap memory manager”. 2009. 9 Advanced Micro Devices, Inc.

[12] C. Mccurdy, J. Vetter. 2010. Memphis: Finding and fixing numa-related performance problems on multi-core platforms. In Proceedings of ISPASS.

[13] S. Kaxiras, M. Martonosi. Computer Architecture Techniques for Power-efficiency. Published 2008

[14]http://software.intel.com/en-us/articles/intel-power-governor Intel Power Governor

[15]http://www.nas.nasa.gov/publications/npb.html NAS Parallel Benchmark Suite

[16] http://www.cse.ohio-state.edu/Polybench Benchmark Suite

[17] http://en.wikipedia.org/wiki/Gang_scheduling Gang Scheduling

[18] http://www.cs.virginia.edu/stream/ STREAM Benchmark