Exploring Cross-layer Power Management for PGAS Applications on the SCC Platform

Marc Gamell
marcgamell@uoc.edu
Advisor: Ivan Rodero

Open University of Catalonia (UOC)
2011-2012

Contents

List of Figures

List of Tables

Abstract

Technological limitations and power constraints are resulting in high-performance parallel computing architectures that are based on large numbers of high-core-count processors. Commercially available processors are now at 8 and 16 cores and experimental platforms, such as the many-core Intel Single-chip Cloud Computer (SCC) platform, provide much higher core counts. These trends are presenting new sets of challenges to HPC applications including programming complexity and the need for extreme energy efficiency.

In this study, we first investigate the power behavior of scientific PGAS application kernels on the SCC platform, and explore opportunities and challenges for power management within the PGAS framework. Results obtained via empirical evaluation of Unified Parallel C (UPC) applications on the SCC platform under different constraints, show that, for specific operations, the potential for energy savings in PGAS is large; and power/performance trade-offs can be effectively managed using a cross-layer approach. We investigate cross-layer power management using PGAS language extensions and runtime mechanisms that manipulate power/performance tradeoffs. Specifically, we present the design, implementation and evaluation of such a middleware for application-aware cross-layer power management of UPC applications on the SCC platform. Finally, based on our observations, we provide a set of recommendations and insights that can be used to support similar power management for PGAS applications on other many-core platforms.

Chapter 1

Introduction

1.1 Motivation

Technological limitations and overall power constraints are resulting in high-performance parallel computing architectures based on large numbers of high-core-count processors. Commercially available processors are now at 8 and 16 cores and experimental platforms, such as the many-core Intel Single-chip Cloud Computer (SCC) platform, provide much higher core counts. This architectural trend is a source of significant programming challenges for HPC application developers, as they have to manage extreme levels of concurrency and complex processor and memory structures [4].

Partitioned Global Address Space (PGAS) is emerging as a promising programming model for such large-scale systems and can help address some of these programming challenges, and recent research has focused on its performance and scalability. For example, existing PGAS research includes improvement of UPC collective operations [36], hybrid models to improve performance limitations [11] and the implementation of X10 for the Intel SCC [7] and UPC for Tilera's many core [37].

Another equally significant and immediate challenge is energy efficiency. The power demand of high-end HPC systems is increasing eight-fold every year [1]. Current HPC systems consume several megawatts of power, and power costs for these high-end systems routinely run into millions of dollars per year. Furthermore, increasing power consumption also impacts the overall reliability of these systems.

In fact, the trend towards many core architectures employing large numbers of simpler cores [5] is motivated by the fact that simpler cores are smaller in terms of their die-area, as per Pollack's Rule[1] have more attractive power/performance ratios. However, as we move towards sustained multi-petaflop and exaflop systems, processor/system level energy efficiency alone is no longer sufficient and energy efficiency must be addressed in a cross-layer and application-aware manner. While application-aware power management has been addressed in prior work, for example, for distributed memory parallel applications using message passing in previous work by exploiting CPU low power modes when a task is not in the critical path (i.e., it can be slowed without incurring overall execution delay) or is blocked in an communication call (i.e., slack) [35], these approaches do not directly translate to PGAS applications on many-core processors where, for example, such communication and coordination operations are implicit.

[1]See chapter 2

1.2 Objectives

This work explores application-aware cross-layer power management for PGAS applications on many-core platforms. To do so, we show the design, implementation and experimental evaluation of language level extensions and a runtime middleware framework for application-aware cross-layer power management of UPC applications on the SCC platform. Specifically,

- We first experimentally investigate the power behavior of scientific PGAS application kernels (i.e., the NAS Parallel Benchmarks) implemented in Unified Parallel C (UPC) on the experimental SCC platform under various constraints, and explore opportunities and challenges for power management within the PGAS framework.
- We then investigate application driven cross-layer power management specified using PGAS language extensions and supported by runtime mechanism that explore power/performance tradeoffs. These extensions are a set of user levels functions (e.g., `PM_PERFORMANCE()`) that provide hints to the runtime system (e.g., threshold values). Hints can define tradeoffs and constraints. Analogous to CPU governors for OS-level power management, we define a set of application level policies for maximizing application performance, maximizing power savings, or balancing power/performance tradeoffs. The runtime mechanisms effectively exploit dynamic frequency and voltage scaling of SCC frequency and voltage domains in regions of the program where cores are blocked due to either thread synchronization or a (remote) memory access. This is achieved using adaptations that adjust the power configuration based on a combination of static and dynamic thresholds at multiple power levels, and use asynchronous voltage and frequency (i.e., DVFS) or only frequency (i.e., DFS) scaling.

1.3 Contributions

Results obtained from experiments conducted on the SCC platform hosted by Intel[2] show that only certain PGAS operations need to be considered for power management, and our runtime power management approach results in energy savings of 7% with less than 3% increase in execution time. Furthermore, by using application level hints about acceptable power/performance tradeoffs, specified using the proposed language extensions, the energy savings can be significantly improved. In this case a 20% reduction of the energy delay product can be achieved.

The experiments also show that in the case of applications where application level power management does not provide any significant energy saving, a cross-layer approach can be used to achieve a wide range of energy and performance behaviors, and appropriate tradeoffs can be selected. These tradeoffs and the effectiveness of this approach are demonstrated using the Sobel edge detector application [27].

We also use a synthetic application (that generates different levels of load imbalance) to demonstrate that the adaptive runtime power management mechanism can handle different load imbalance scenarios and can provide significant energy savings. For example, when load imbalances are high, we can achieve up to 50% of available energy savings using DVFS and up to 25% using DFS without incurring a significant execution time penalty.

Our evaluation also reveals several power management limitations of the SCC platform that must be addressed in future architectures; for example, voltage scaling can be performed only on

[2] http://communities.intel.com/community/marc

domains of 8 cores. Finally, based on our observations, we provide a set of recommendations and insights that can be used to support similar power management for PGAS applications on other many-core platforms.

1.4 Planning

Figure 1.1 shows the gantt diagram that represents the planning done at the beginning of the work.

Jobs are distributed as described below:

- The first unavoidable phase is needed to study how the experimental SCC platform works, and the basis of UPC Runtime. There is also a need to prepare the environment and search the existing related work.
- A second phase is needed to approach application's behavior. In order to achieve this target we need to look for several UPC benchmarks which can run on the constrained SCC environment, and profile its behavior. Therefore, we need to port existing profiling tools to SCC or prepare a lightweight instrumentation platform.
- While extracting conclusions from the profiling phase, we can work on the power manager: a middleware which can take energy-efficiency related decisions.
- Finally, we will need to evaluate the designed and implemented algorithm by executing a big set of tests.

Note that, during the whole process (specially during periods in which the SCC is running tests), we will need to document the work, in order to avoid the final-term increased workload.

1.5 Organization

The rest of this report is organized as follows. Chapter 2 presents the architecture of the SCC processor and specific SCC platform used in our experiment, with special focus on power management aspects that are important for our evaluation. Chapter 3 introduces PGAS parallel programming model and focus on one of its incarnations: UPC language. Chapter 4 discusses relevant related work. Chapter 5 contains a study of power behaviors of PGAS applications based on application profiling, with the goal of identifying opportunities for power management. Chapter 6 presents the proposed programming extensions and power management system for UPC PGAS applications on SCC, while chapter 7 presents their evaluation. Finally, chapter 8 concludes the report and outlines directions for future work.

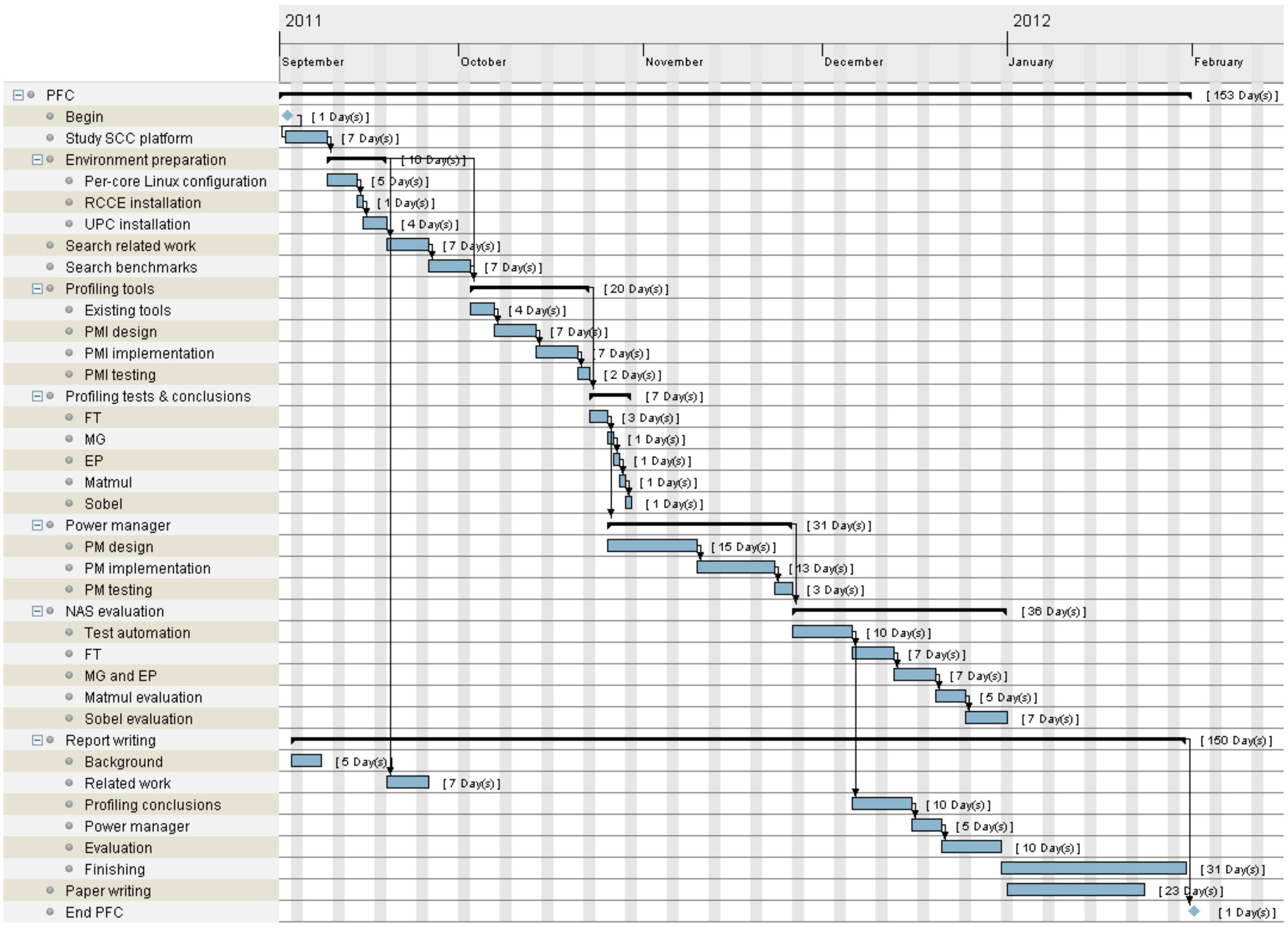


Figure 1.1: Planning: Gantt diagram.

Chapter 2

Single-chip Cloud Computer background

2.1 Many-core motivation

In the recent years, as discussed by Borkar [5], processors has been shifted towards multi-core architectures: the idea is to include in a processor more simpler cores, than less complex ones.

Note that the multi-core processor definition (a system with two or more independent processors packed together) can be applied to many-core systems too. We can find the main difference, however, in the number[1] and complexity of these cores. While a multi-core processor have got several complex cores, a many-core processor have got a large number of cores that are much simpler than multi-core ones.

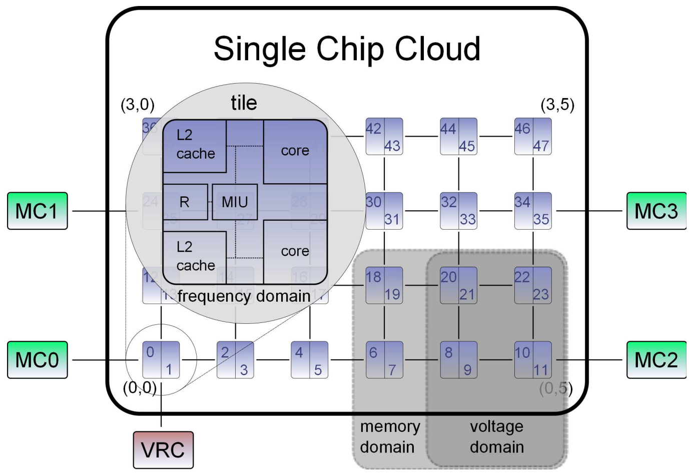
Complex cores are faster, but simpler cores are smaller, in terms of die-area. If we apply Pollack's Rule (performance increase is roughly proportional to square root of increase in complexity) inversely, performance of a smaller core reduces as square-root of the size, but power reduction is linear, resulting in smaller performance degradation with much larger power reduction.

Overall, the compute throughput of the system, on the other hand, increases linearly with the larger number of small cores. That is why we can suppose that future processors will be based, in some way, in many-core processors.

2.2 Intel SCC's architecture overview

Intel Labs has created an experimental many-core processor, inside the Intel's Tera-scale Computing Research Program, aimed to help accelerate many-core research and development. It is Single-chip Cloud Computer (SCC), and, as we can see in figure 2.1a, it consists of 48 x86 Pentium P54C cores, with increased L1 cache to 16 KB for data and another 16 KB for instructions and 256 KB of L2 cache per core. It is fabricated in a 45 nm process, and it's cache is non-coherent: libraries like RCCE (studied below), however, offers software-based cache coherence implementation. It's cores are grouped 2 by 2 in so-called tiles.

[1]Usually, *many-core* means 32 or more cores, while *multi-core* means fewer.



(a) Architecture overview

(b) Packed chip

Figure 2.1: Single-chip Cloud Computer

SCC features a fast (256 GB/s bisection bandwidth) 24-router on-die mesh network, with hardware support for message-passing, that communicates all tiles between them. This hardware message-passing support is helped by special per-tile 16 KB fast-r/w buffer, called message passing buffer (MPB). Therefore, every tile also includes a traffic generator (TG) used for ensure network reliability.

Note that the on-chip network also provides tile access to four dual-channel DDR3 memory controllers (MC) with typically 32 GB or maximum 64 GB of main memory for the entire chip. The memory controllers are attached to the routers of the tiles at coordinates (0,0), (0,5), (2,0) and (2,5).

As Single-chip Cloud Computer's (SCC) name reflects, it implements an on-die scalable cluster of computers such as you would find in a cloud datacenter. The network topology and the message-passing implementation is proven to scale to thousands of processors in existing cloud datacenters. Moreover, each core can run a separate OS (typically Linux) and software stack and act like an individual compute node that communicates with other compute nodes over a packet-based network.

SCC is intended to offer fine-grained power management, allowing to dynamically scale per-tile frequency, and dynamically scale group-of-eight-core's voltage. The power consumption can oscillate from 125 W to as low as 25W. This feature is what we are trying to exploit.

2.3 Other existing many-cores

As mentioned in [16], SCC has been influenced by previous architectures and research. Beginning on the Cell processor, which wasn't homogeneous and included 8 small SPE cores, SCC have been influenced by Polaris, a Intel 80-core experimental processor, which cores was simpler than x86's SCC cores.

On the other hand, there are other many-core architectures intended to be graphic processors. An example of this are the well known GPU architectures which must be programmed with special frameworks such as CUDA or OpenCL, as they are not x86. Another example is Intel's Larrabee, which, like SCC, features x86 cores, with added support to vector processing.

2.4 Memory

As mentioned above, SCC cores are regular x86 P54C cores. That means that it uses 32 bits to address the main memory. However, to access a main memory position we need to address one of four memory controllers, and a concrete position inside the 16 GB associated to that memory controller. As a result, we need 8 bits to point to the position of the MC in the mesh and 34 bits to address 16 GB. The difference between both addresses length is solved using a per-core lookup table (LUT), that translates a 32 bit core address to the corresponding system address. It is implemented as an area in the configuration block, and contains 256 22-bit positions. The 8 higher bits of a core address are used to point to a position of the LUT, which returns the 22 extra positions to fit the system address. This process is shown in figure 2.2, in which subdestination field points to a position inside a tile (i.e. core0, core1, MPB, east port, west port...) and a bit for MIU bypass.

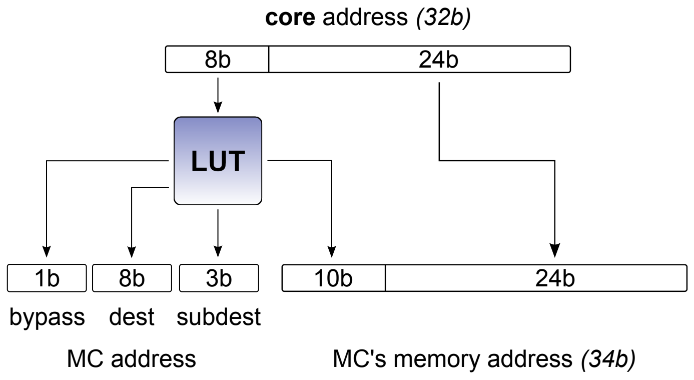


Figure 2.2: SCC LUT translation process

An SCC advantage is that LUT can be changed in run-time, meaning that we can redimension the core memory and/or the shared memory in order to adequate it to workload requirements.

The cores of the SCC are grouped into four memory domains, depending on which memory controller (MC) holds the core main memory. As we have seen talking about LUT, this table determines which MC is responsible for which core. That is why memory domains are not fixed. However, the standard configuration is to assign each core to the nearest MC. This results on a vertical and horizontal division exactly in the middle of SCC, leaving 6 tiles per memory controller and a maximum hop number of 3 to reach the corresponding memory controller's router.

2.5 Mesh

Each tile is provisioned with a four-port router, in charge of the tile-to-tile connections, and with a Mesh Interface Unit (MIU), in charge of pack, unpack and address decode, among other related functions. These routers, the MIUs, and their corresponding connections forms the so-called mesh, a 6x4 two-dimensional squared grid that connects each tile with surrounding tiles and other entities like memory controllers, voltage regulator or system interface.

This packet-switched network is governed by a simple deterministic (x,y) routing scheme. This means that data packets are routed first horizontally and finally vertically.

2.6 SCC's message passing library

SCC tools includes RCCE, a library that implements message passing functions specifically optimized to the SCC architecture. It takes profit of tile MPB's and the fast mesh. The library also supports high level abstractions that handles voltage or frequency scaling operations or simplifies shared memory allocation.

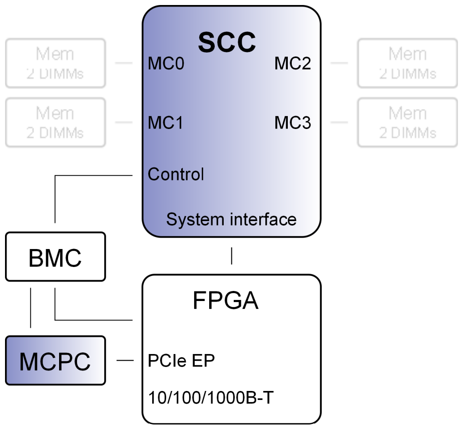
2.7 Rocky-Lake (SCC board) architecture

The experimental SCC chip needs a concrete environment to properly run, as it is not directly bootable. That is why a standard PC is used to control and manage SCC status, called management console PC (MCPC). The complete system's architecture used for the work in this study (rocky lake board) is shown in figure 2.3a. As we can see, the SCC chip is connected directly to 8 memory dimms, totaling 32 GB. On the other hand, an ARM processor called Board Management Controller (BMC) is connected to the SCC and can handle commands like platform initialization or power data collecting. Furthermore, it is connected, via the system interface, to a FPGA that controls it's external communications. Specifically, in addition to general purpose ports (22 I/O signals, SATA and PCIe interfaces) and a connection to the BMC, FPGA includes several ports (PCIe and Gigabit Ethernet) for connecting to the management PC.

2.8 Power management capabilities

As mentioned above, RCCE provides tools that abstracts power management details in SCC. In this section, however, we are trying to explain which are SCC power management capabilities and which limitations have it got.

Microprocessor's power management can be done over the whole processor, or in specific areas (for example, cores). SCC, compared to existing processors, allows fine-grained power management. SCC have got three components that works with different clock and power source: mesh, memory controllers and tiles. On the one hand, the frequency of the entire mesh can operate from 800 MHz to 1.6 GHz. On the other hand, memory controllers operates from 800 to 1066 MHz. However, as soon as mesh and memory frequency and voltage changes cannot be performed during run-time, in this study we are focusing only on tile and voltage-domain power management.

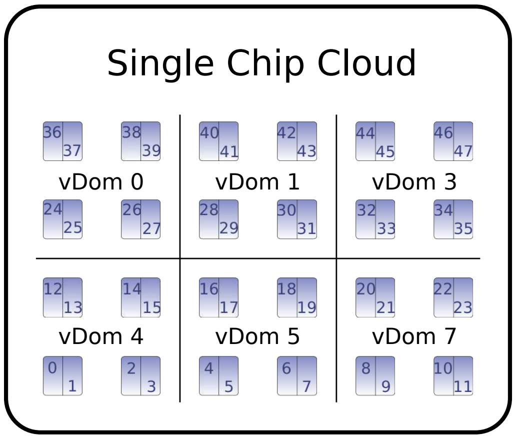


(a) Rocky Lake System architecture

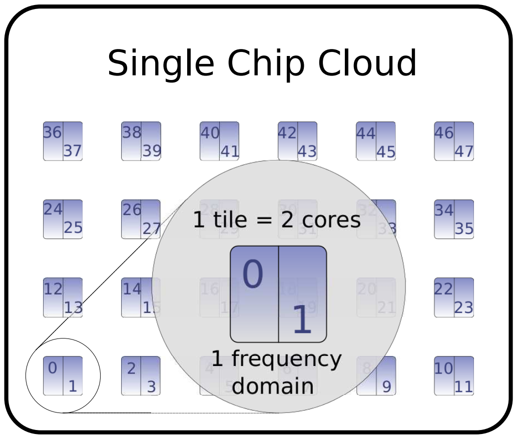
(b) Rocky Lake Board

Figure 2.3: Rocky Lake Board

SCC cores are grouped 8 by 8 in six voltage domains, as shown in figure 2.4a, and voltage can only be changed in the scope of a whole voltage domain. Similarly, as shown in figure 2.4b, so-called frequency domains corresponds to tiles, and, therefore, frequency can only be changed tile by tile.



(a) Voltage domains



(b) Frequency domains

Figure 2.4: Domains in the SCC

The part of the processor involved in the voltage is called the VRC (Voltage Regulator Controller), that works in a command-based strategy: cores sends messages with VRC destination, including the command. There is no limit in the source core of a command, which means that a core in a voltage domain can change the voltage of another domain, which can be very useful for sophisticated power management. The VRC is not distributed in every voltage domain, but it is a standalone part, reachable via the mesh by all cores. Therefore, the VRC only accepts one command at a time and, theoretically, *the state is not defined if two cores sends respectively commands simultaneously*. That is why a core can send a command three times in order to be sure the command have been finished. The first command will result in the actual change while the last two ones would assure the change process have been finished. As described in [2], VRC allows voltage requests with a granularity of 6.25 mV, between 0 and 1.3 V. However, we found that the lower subrange of this voltages cannot be used in practice, as cores both crashes or become unstable. To remain safe, the lowest voltage used (with their corresponding lowest frequency) was 0.65625 V. We experimentally determined that this process takes an average of 40.2 ms to complete, regardless the originating-tile frequency or voltage and the desired voltage.

Frequency scaling involved part, however, is distributed among tiles. Each tile contains a configuration register that is used to change the frequency divider, which can be any integer value between 2 and 16. As soon as global clock frequency is 1.6 GHz, resulting frequency oscillate between 800 to 100 MHz. When the process of writing the desired value in the register have been finished, the processor takes as little as 20 clock cycles to complete the actual frequency change.

Before changing the voltage, however, the frequency must be changed accordingly to the desired voltage level. Similarly, when changing frequency we must take care the current voltage in the corresponding voltage domain. In the RCCE source code and in the SCC Programmers Guide there is a table showing the maximum frequency allowed for a voltage level (see table 2.1). As SCC cores maximum frequency is 800 MHz, the useful part of the table, however, are the voltage levels 0, 1 and 4. Experimentally we found that the first two voltage levels seems not to be safe, as many times the voltage domain became unstable, as described in [16]. The frequencies that worked, in our case, are shown in table 2.2.

Voltage Level	**Voltage (volts)**	**Max freq. (MHz)**
0	0.7	460
1	0.8	598
2	0.9	644
3	1.0	748
4	1.1	875
5	1.2	1024
6	1.3	1198

Table 2.1: SCCProgrammersGuide, version 0.75, Table 9: Voltage and Frequency values

Voltage Level	**Voltage**	**Max freq.(MHz)**	**Tested freq. (MHz)**
0	**0.75**	460	400
1	**0.85**	598	533
4	1.1	875	800

Table 2.2: Safer Voltage and Frequency values, experimentally determined.

To sum up, despite dynamic voltage scaling is difficult to exploit due to large voltage domains, its energy reduction is far better than frequency scaling which, unlike happens with voltage, a change in frequency only has a linear impact on the energy consumption. However, this technique is far faster than voltage scaling, and more flexible due to the smaller domain size, and, therefore, it can be applied in more variety of scenarios.

2.8.1 Power monitoring

SCC chip does not provides monitoring tools itself. In order to collect the measured voltage and current consumption we can send, from the management PC, a query command to the BMC (see figure 2.3a). Thanks to this features we can automatically collect the measured power consumption, obtaining a sampling frequency of about 6,5 measures per second.

Figure 2.5 shows us the SCC's consumed power, according to the workload and the technique used. Note that DVFS allows exponential power reduction regardless the workload, and DFS technique only allows linear power reduction. Note too that 400 MHz (which corresponds to lowest voltage allowed (0.75 V) is a good DVFS target in order to save energy, as lower levels provide only little-saving with huge frequency reduction.

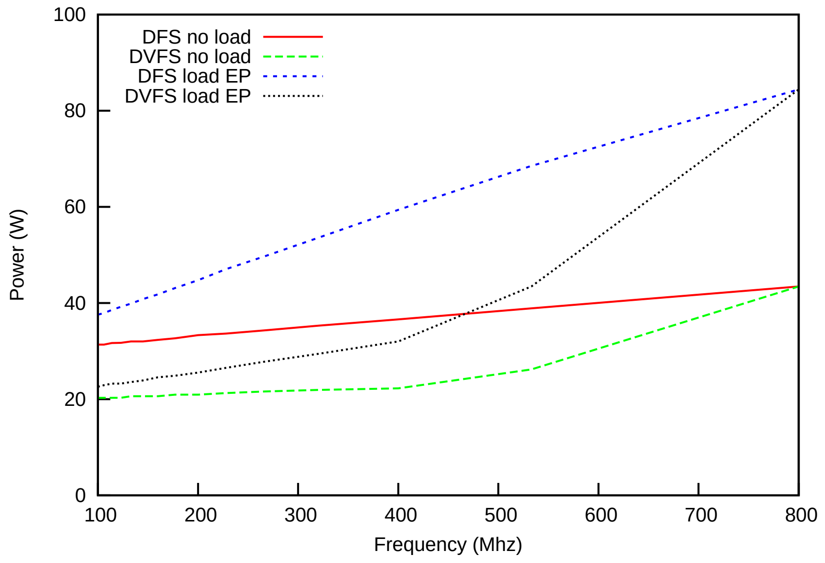


Figure 2.5: Consumed power by SCC according to the workload and the technique (DFS / DVFS). The data have been collected in the 15 possible frequencies available.

Chapter 3

Unified Parallel C background

3.1 PGAS

In order to help the development stage of parallel and distributed applications, in the last few decades there have been a lot of research about parallel design languages and techniques. Partitioned global address space (PGAS) paradigm is a relatively new model that proposes a easy-of-use solution to the parallelization problem. It assumes a global, shared memory space that is logically partitioned among all threads and, therefore, each portion of the memory is local to one of the processors.

As each thread is aware of which data is local, the application can improve *performance* by exploiting *data locality*. The performance, however, is not the only PGAS focus. Another important goal is to enhance *user productivity* significantly by abstract details like thread synchronization or implement implicitly the message passing.

The PGAS model is the basis of Unified Parallel C (UPC), Co-array Fortran (CAF), Titanium, Chapel and X10.

3.2 UPC overview

Unified Parallel C is an ISO C 99 extension that supports explicit parallelization and the PGAS model. It tries to get the best points of several previous C parallel extensions (like PCP, AC or Split-C), hence the name.

It uses a Single Program Multiple Data (SPMD) model of computation in which the same code runs independently on different threads, in parallel.

Apart from the private variables (which can only be seen inside the process), the user can define shared variables, usually vectors, which UPC distributes following a default pattern or a specified one. All threads can read or write a defined shared variable, but each memory position is physically assigned to one processor. In case the R/W operation is local, the UPC implementation should write it directly. If it is not local, however, the implementation is the responsible of mapping the operation to the corresponding R/W message and send it to the processor with affinity to that memory position.

Aside from the easy-of-use of PGAS paradigm, UPC offers a high level control over data distribution among cores. Furthermore, it extends the standard C pointers allowing it to point to an arbitrary position of the shared space, regardless it have local affinity or not. Shared space allows both static and dynamic memory allocations, as standard C does for the corresponding private addresses.

Moreover, we should pay attention to the ability of incremental performance improvements that UPC paradigm offers. The programmer can begin by designing the application as plain sequential C code, converting it, later, to a simple shared-memory implementation, sharing some vectors. The programmer can then improve the performance by tuning data locality layout. Finally, for critical applications, the programmer can go deeply and tune it by making the memory management and one-sided communications explicit.

The UPC extensions to C are as simple as an explicitly parallel execution model (global constants `THREADS` and `MYTHREAD`), shared variables and pointers (`shared` token), synchronization primitives (such as `barrier` or `lock`) and memory management primitives (such as bulk memory copy `memget` operation).

As C is a well-known language inside the HPC user community (as for example scientists), the learning curve for UPC is easy to achieve.

3.3 Berkeley UPC

In our study we have been using **Berkeley UPC**, which is an open-source UPC implementation. It is composed basically of:

- The Berkeley UPC Translator. This module compiles the UPC source code to ANSI-compliant C code which can be linked with the *abstract-machine* described by the UPC Runtime.
- The Berkeley UPC Runtime, which includes platform-independent job/thread control, shared memory access (put/get operations and bulk transfer operations), shared pointer manipulation and arithmetic, shared memory management, UPC barriers and UPC locks.
- GASNET is the layer below UPC Runtime, which is a portable high-performance low-level networking layer and, among other things, is the responsible of implementing communication algorithms such as remote memory access. It runs over a wide variety of high-performance networks such as well-known MPI, Infiniband, Myrinet GM, Cray Gemini, or even UDP.

Chapter 4

Related work

Our work is mainly centered in three topics: layered power management, many-core power management and PGAS power management. In the following sections we are describing the existing and ongoing research on these three topics.

4.1 Layered power management research

Existing and ongoing research in power efficiency and power management has addressed the problem at different levels such as processor and other subsystems level, runtime/OS level and application level.

Processor level
Since processors dominate the system power consumption in HPC systems [25], processor level power management is the most addressed aspect at server level. The most commonly used technique for CPU power management is Dynamic Voltage and Frequency Scaling (DVFS), which is a technique to reduce power dissipation by lowering processor clock speed and supply voltage [17,18].

Operating system level
OS-level CPU power management involves controlling the sleep states or the C-states [28] and the P-states of the processor when the processor is idle [29] [30]. The Advanced Configuration and Power Interface (ACPI) specification provides the policies and mechanisms to control the C-states and P-states of the processor when they are idle [38].

Workload level
Some of the most successful approaches for workload-level CPU power management were based on overlapping computation with communication in MPI programs, using historical data and heuristics [14, 15, 20, 22, 34], based on application profiles [6, 32], scheduling mechanisms [8] or exploiting low power modes when a task is not in the critical path [35].
Another result is that we can take profit of slack CPU periods slowing it down, and, therefore, saving energy. The problem have been to determine which periods are the appropriate. Here, the policies varies in complexity. Scheduled communication techniques take profit of the big difference between network and CPU speeds, and, while the application asks for a barrier, a synchronous send, or a synchronous receive and the processor is waiting for the transmission to finish, the runtime slow it down (by applying DVFS techniques) to save energy. Examples of this techniques

can be found in work by Liu et al. [24] or work by Lim et al. [23].

Compiler level
Another targeted layer for power management have been the compiler. Wu et al. [42] introduced a dynamic-compiler-driven control for energy efficiency. A dynamic compiler (HP Dynamo, IBM DAISY or Intel IA32EL) is a runtime software system that compiles, modifies or optimizes a program as it runs.

Application level
Aforementioned techniques are transparent to the application. This means that the programmer does not need to modify its application. However, existing work has also addressed power efficiency and power management at the application level. For example, Eon [39] is a coordination language for power-aware computing that enables developers to adapt their algorithms to different energy contexts.

Overall, however, any of the existing research have been addressed the power management in a cross-layer approach.

4.2 Many-core research

Power management
Existing power management research has also addressed many-core systems. For example, Majzoub et al. [26] introduced a chip-design approach to voltage-island formation, for the energy optimization of many core architectures. Alonso et al. [3] proposed extending the power-aware techniques of Dense Linear Algebra algorithms to SCC.

Performance improvements
Other approaches have considered the SCC platform but mainly from the performance perspective. For example, Rotta [33] discussed how to efficiently design and implement the different strategies for message passing on SCC. Pankratius [31] introduced an application-level automatic performance tuning approach on the SCC. Urena et al. [40] implemented an MPI runtime optimized for the SCC message passing capabilities, RCKMPI. Clauss et al. [9,10] improved message passing performance on SCC by adding a non-blocking communication extension to RCCE library. Van Tol et al. [41] introduced an efficient memcpy implementation.

4.3 PGAS research

Existing PGAS research has focused mainly on its performance and scalability. Salama et al. [36] proposed a potential improvement of collective operations in UPC. Dinan et al. [11] proposed an hybrid programming paradigm based upon MPI and UPC models that try to improve performance limitations. Chen et al. [19] compare the performance of benchmarks compiled with their own optimizations in BUPC with that of HP UPC compiler. Tarek et al. [12,13] benchmarks UPC and propose compiler optimizations. Kuchera et al. [21] study the UPC memory model and memory consistency issues.

Existing PGAS research on many-core systems is not very large. Chapman et al. [7] implemented the X10 programming language on the Intel SCC (using RCCE library) and performed a comparative study versus MPI using different benchmark applications. Serres et al. [37] ported Berkeley UPC to Tilera's many-core Tile64.

Overall, these approaches do not directly translate to PGAS applications on many-core processors and, to the best of our knowledge, power management in the PGAS framework has not been addressed yet.

Chapter 5

Profiling of UPC applications

In this chapter we are trying to show the behavior of some UPC applications suitable to work on SCC processor. We show an analysis of some applications from the UPC implementation of the NAS Parallel Benchmarks specification (FT, MG and EP kernels), a matmul-based synthetic imbalanced application and an application for edge detection (Sobel) from the UPC official test suite.

5.1 Methodology

In order to profile the UPC applications we need to have got some instrumentation that tells us which UPC runtime operations an application uses. Existing profiling tools are designed to be executed in traditional clusters with big amounts of memory. If we use it in the SCC constrained environment, our profiling would be biased due to the overhead of the profile process.

For this reason we decided to implement a low-footprint instrumentation system (that we called `pmi`, for Power Management Instrumentation), currently working on UPC, but extensible to other PGAS runtimes. Our design tries to minimize the runtime part, and do the main work as a post data processing. This is represented in the figure 5.1 flow. As seen, we achieve low overhead by writing lightweight intermediate files containing data in raw binary format. Then, the application parses the stored data (timestamps of the corresponding begin and end for each call) and, automatically, generates logfiles and relevant plots.

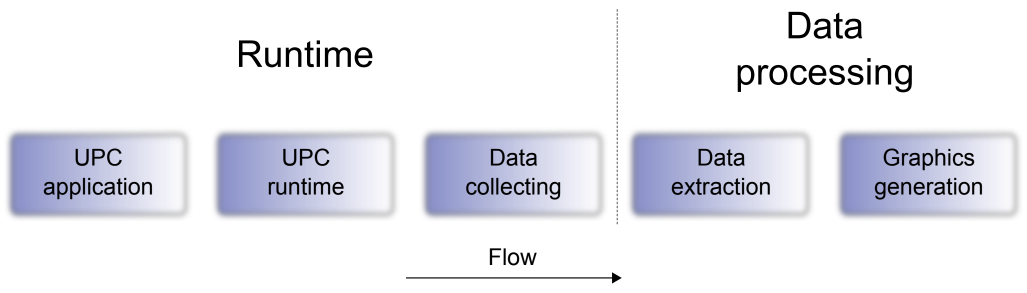


Figure 5.1: Global architecture of the profiling platform.

5.2 NAS Parallel Benchmarks

NPB are a well-known benchmark collection for parallel computing, standardized by the NASA Advanced Supercomputing. The suite comprises a set of kernels and pseudoapplications that reflect different kinds of relevant computation and communication patterns used by a wide range of applications.

Each kernel can be configured with several different versions. These versions, called classes, does not differ in the problem nature, the difference is mainly in the size of the data.

Apart from the MPI version of NPB, distributed and maintained by the NASA, there are several more implementations in different languages. One of these is NPB in UPC language, that have been developed and distributed by the George Washington University. Note that this distribution does not implement the whole set of tests, only a part. Kernels have been manually optimized through techniques that mature UPC compilers should handle in the future. Therefore, researchers can choose the level of optimization they wants to run.

As mentioned above, the evaluated kernels are FT (Fast Fourier Transforms), EP (embarrassingly parallel cpu-intensive code) and MG (Multi Grid). We executed each benchmark on the whole SCC (involving all 48 available cores) because we want to stress the whole platform to obtain the most real potential of power management in many-core environments.

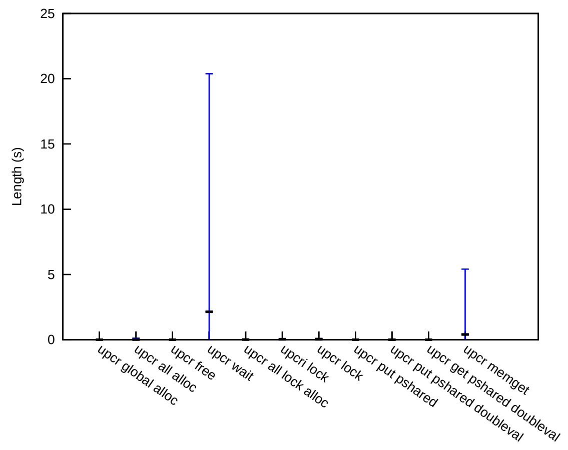
BT and SP benchmarks, which require the number of processors be a perfect square, doesn't use all the SCC platform potential. Other NAS benchmarks, like CG, that needs powers of two number of threads, does not stress the whole 48-core SCC platform neither. That is why we discarded these algorithms.

5.2.1 NAS FT kernel

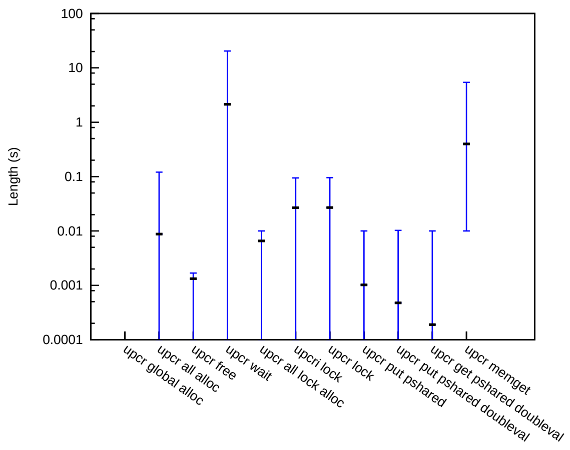
FT is a kernel that calculates three 1D Fast Fourier Transforms, one for each dimension of a 3D partial differential equation. It is floating-point computation intensive and requires *long range communications*.

We can run several versions of the same NPB kernel. For our experiments, we have been using *FT class C*, optimization O1, that is the largest problem that runs on the SCC platform and, therefore, it is the one that takes more time to finish.

After instrumentation phase, we have collected data and processed it as mentioned in the section above, obtaining results in a set of graphics. The most representatives are shown here.



(a) Linear scale.

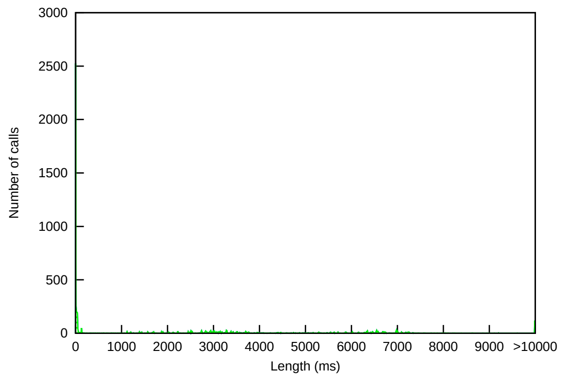


(b) Logarithmic scale.

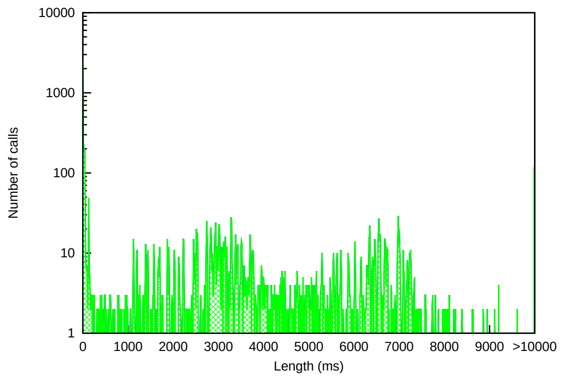
Figure 5.2: Length of FT calls.

If we take attention to figure 5.2a, we can see that it represents the length of each UPC call. Note that the candles shows the maximum, minimum and average duration (in seconds) of the corresponding call. As we can see in the mentioned figure (and the more detailed logarithmic version, figure 5.2b), FT uses only 11 UPC runtime operations, but only `memget` and `wait` seems to be long enough for being the focus of energy savings strategies. There have been 1056 memgets and 136 barriers per core, so we can analyze how many of these are short and how many are long. In figures 5.3a and 5.3c (and the corresponding figures 5.3b and 5.3d, in logarithmic scale) we analyze the number of calls per delay that FT performed for `wait` and `memget`, respectively. The figures corresponding `wait`, shows us that, apart from the 2500 calls of almost-null slack periods, there are lots of calls that needs 1-4 seconds, 6-7 seconds and more than 10 seconds to finish. `memget`, however, only uses operations of almost-zero and around 3 seconds delay.

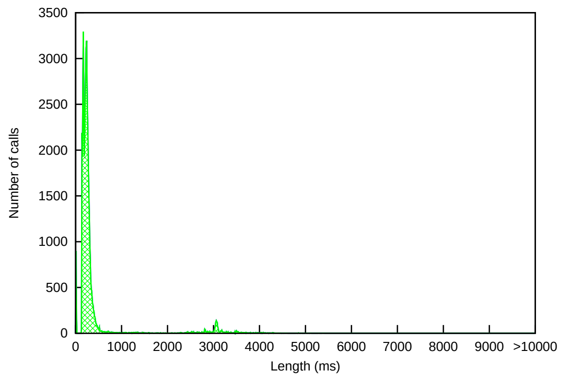
The main conclusion of these tests is that *FT have a communication-bound profile.*



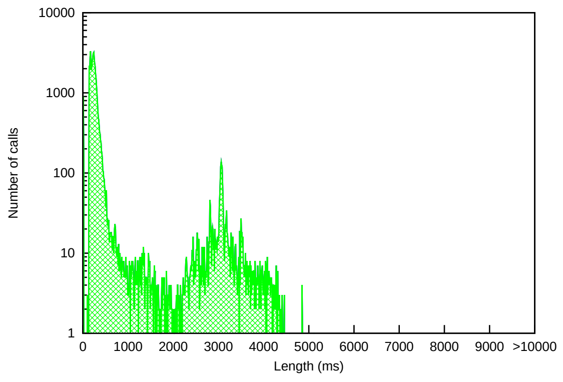
(a) Wait calls. Linear scale.



(b) Wait calls. Logarithmic scale.

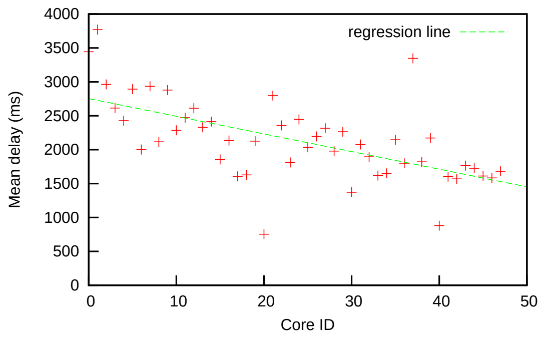


(c) Memget calls. Linear scale.

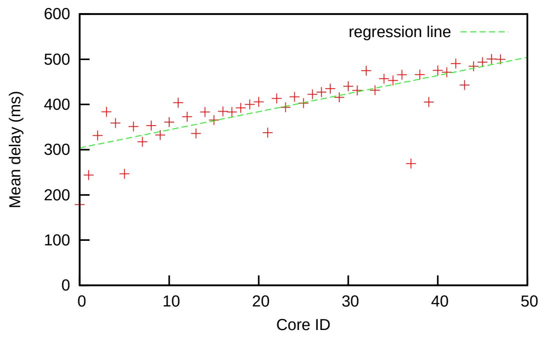


(d) Memget calls. Logarithmic scale.

Figure 5.3: Number of calls per delay.



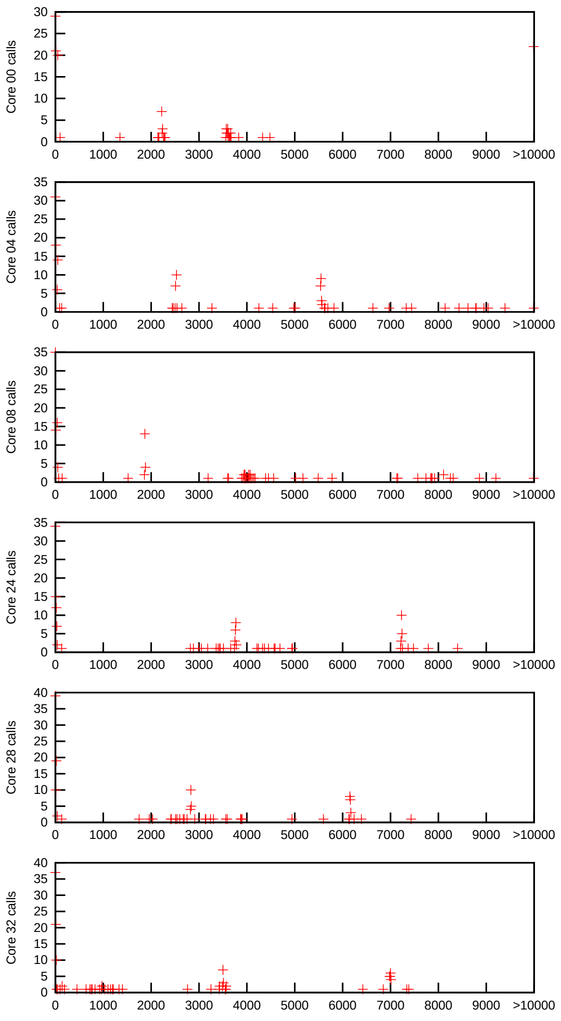
(a) Average barrier slack period (wait).



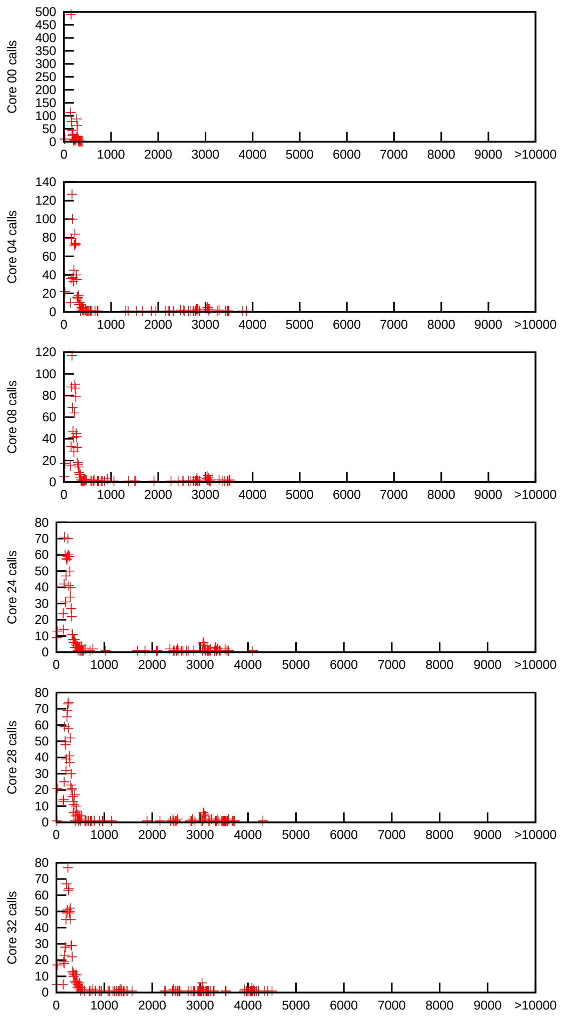
(b) Average memget delay.

Figure 5.4: Average delay of the call, per core.

Note that in all these graphics we have been looking at the behavior of all cores together. In addition, the per-core behavior is notable. As we can see in figure 5.5a for waits, and figure 5.5b for memgets, each core have a different profile. Figure 5.6 shows the execution point in which each `wait` or `memget` call have been done, and their corresponding delays (in seconds). In these figures we can suspect that the average length is linearly dependent to the number of the core. If we look at the average of each call (figures 5.4a and 5.4b), per core, we will see that this suspect is not unfounded: there is a trend.

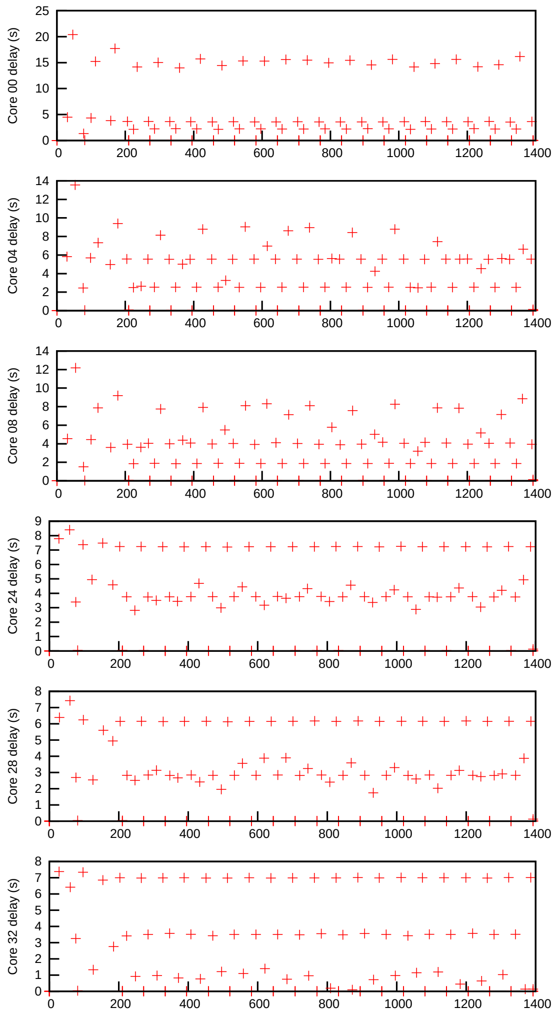


(a) Length of wait operation (ms).

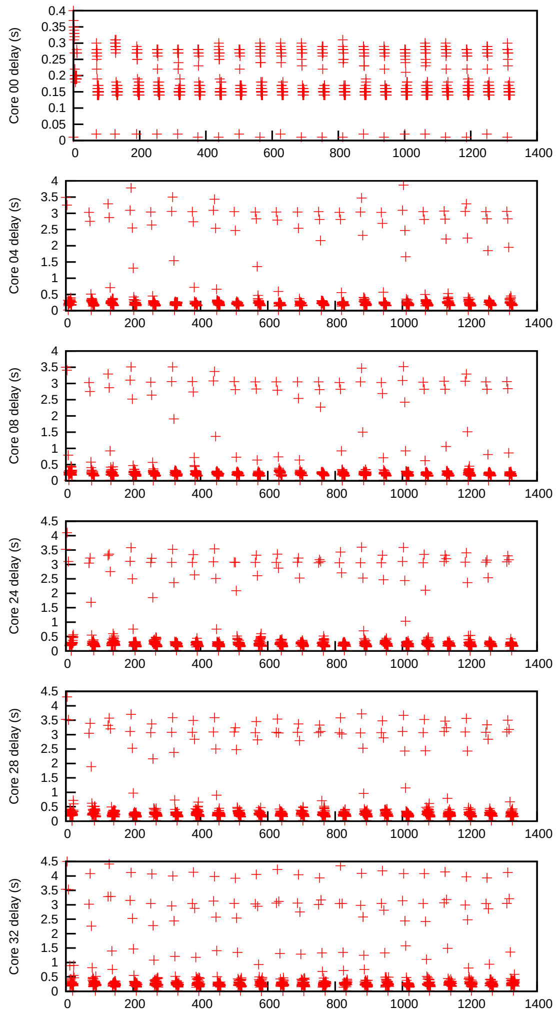


(b) Length of memget operation (ms).

Figure 5.5: Number of operation calls per length (in ms). Each subplot represents a power domain controller.



(a) Wait.



(b) Memget.

Figure 5.6: Operation calls and corresponding delays (in seconds) according to execution point (in seconds). Each subplot represents a power domain controller.

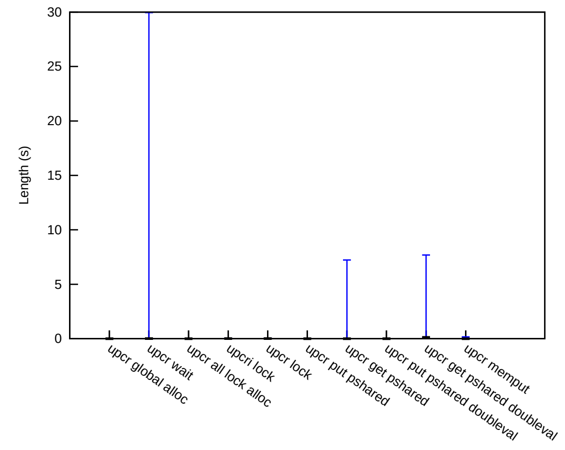
The conclusion of this part is that, although FT is well-balanced, some cores suffer from big slack periods. Therefore, FT can be a good candidate in order to study UPC power management techniques.

5.2.2 NAS MG kernel

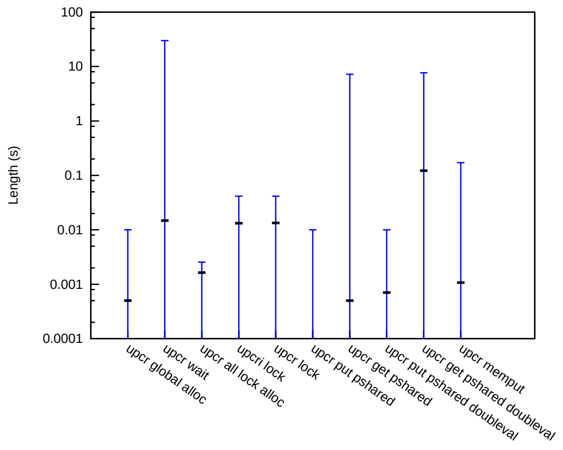
MG, that stands for Multi Grid, is a benchmark that solves a 3D scalar Poisson equation. It performs both structured *short and long range communications*.

In our profiling process we have been using MG class C, optimization O3, because it is the maximum that supports SCC constrains.

As we can see in figure 5.7, MG features mainly `wait` and two types of `get_pshared` calls.



(a) Linear scale.



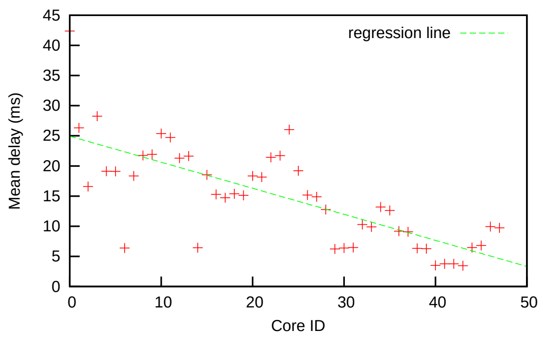
(b) Logarithmic scale.

Figure 5.7: Length of MG calls.

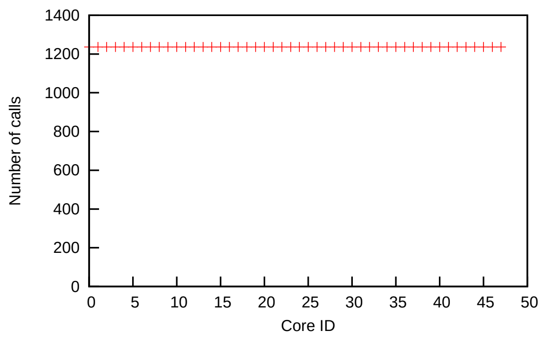
Let's analyze first `wait` behavior. In figures 5.8a and 5.8b we can observe that MG performs lots of barriers (about 1200 per core), and, on average, the corresponding waits are short (about 5 to 30 ms). However, figure 5.8c shows that only a little quantity of calls are very long (10 seconds or more), while the bast majority are less than 0.25 seconds. Finally, with the help of figure 5.8d we can figure out this behavior: there is only one imbalanced barrier at the beginning of the execution (in the initialization phase), and the rest of the MG execution is well-balanced.

The conclusions obtained from the `wait` analysis can be applied to `get_pshared` (figure 5.9): there are lots of small calls (25000 calls per core, 0.5 ms on average), and only one huge call, on the initialization phase. The same conclusions can be drawn for `get_pshared_doubleval` operation (see figure 5.10).

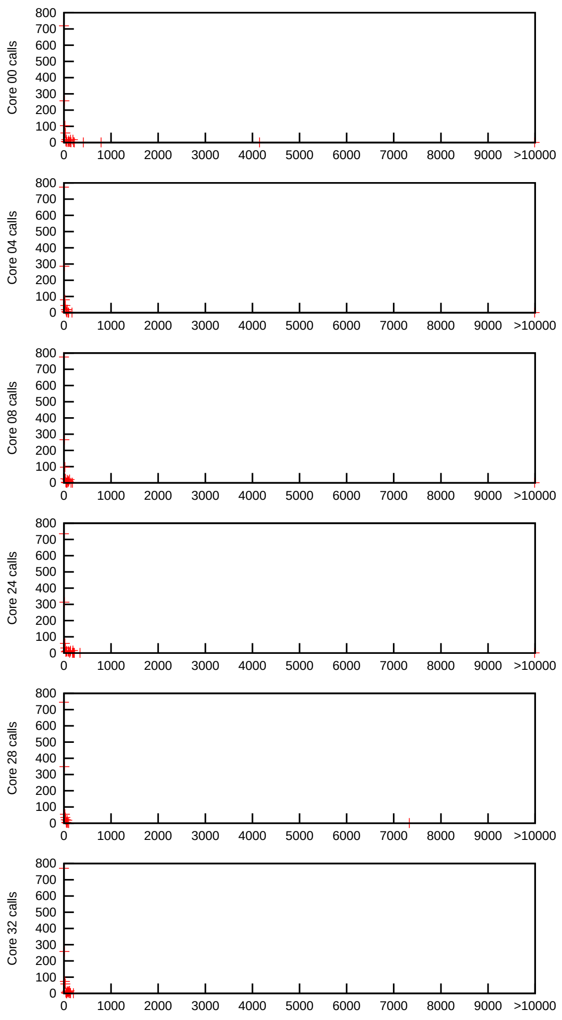
With all this conclusions in mind, we see why MG does not show energy saving opportunities.



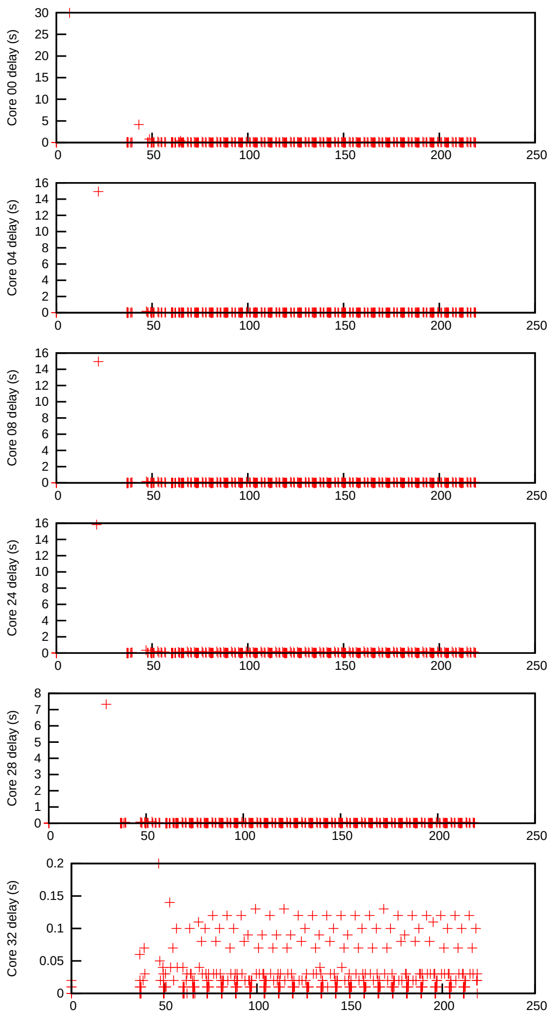
(a) Mean delay per core



(b) Total calls per core

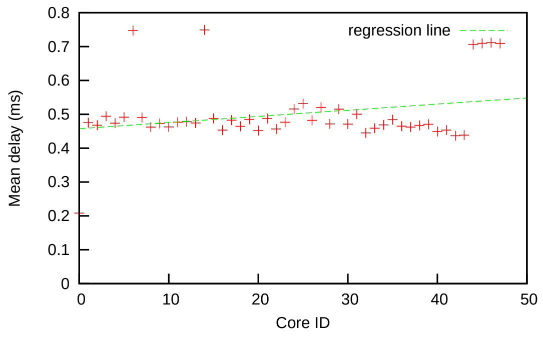


(c) Number of operation calls per length (in ms).

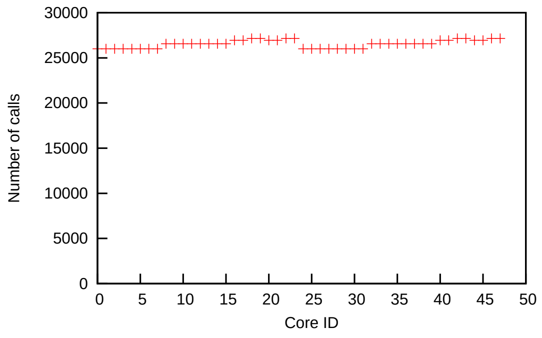


(d) Calls and corresponding delays (in seconds) according to execution point (in seconds).

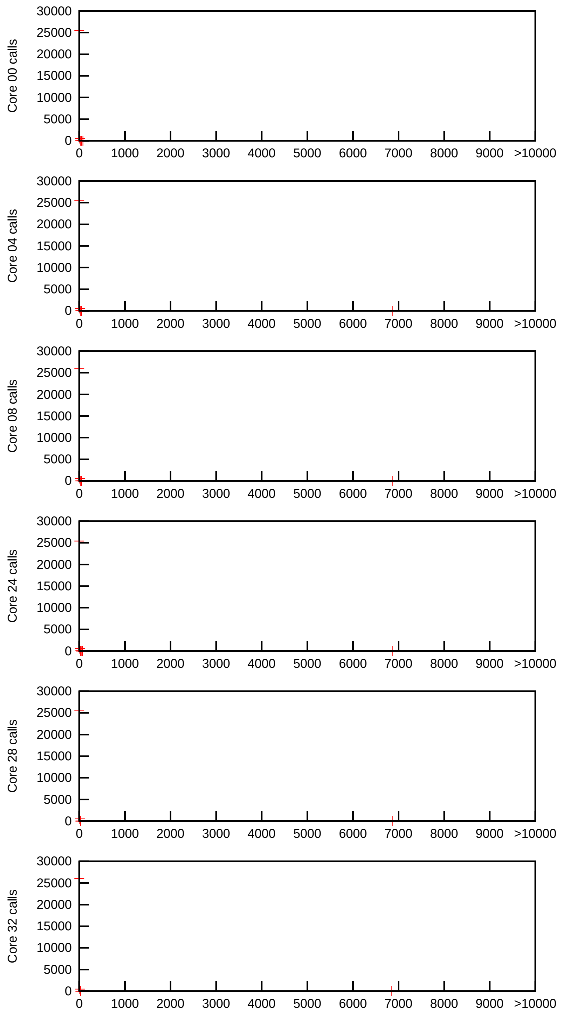
Figure 5.8: MG wait behavior.



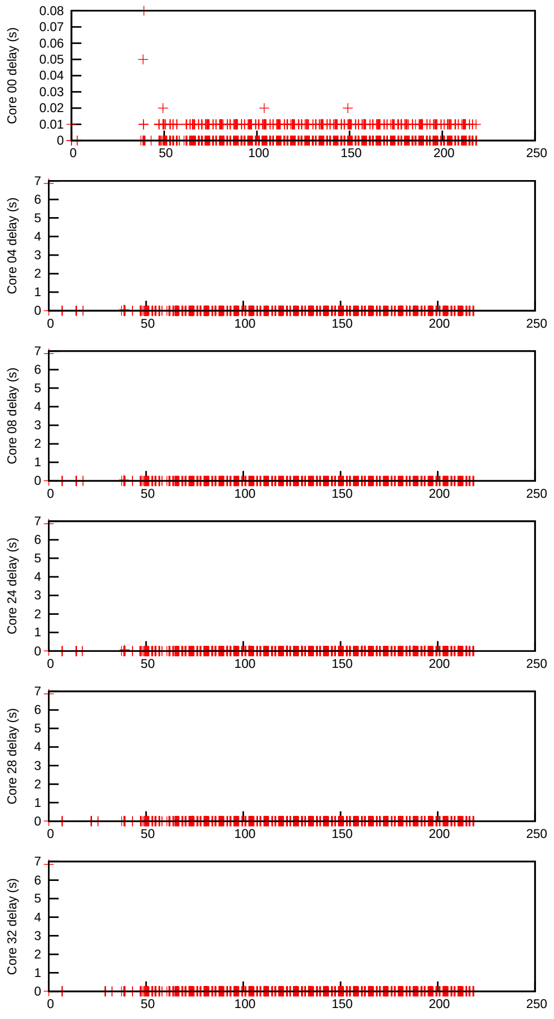
(a) Mean delay per core



(b) Total calls per core

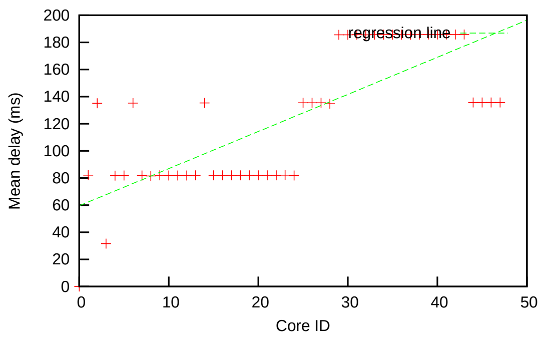


(c) Number of operation calls per length (in ms).

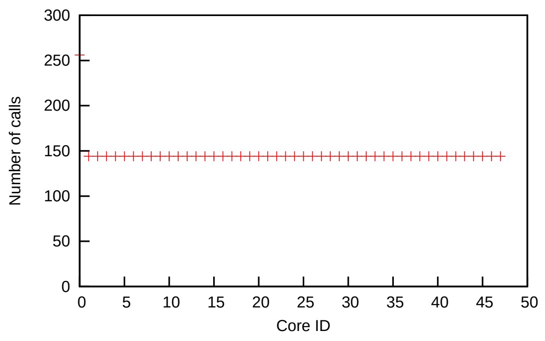


(d) Calls and corresponding delays (in seconds) according to execution point (in seconds).

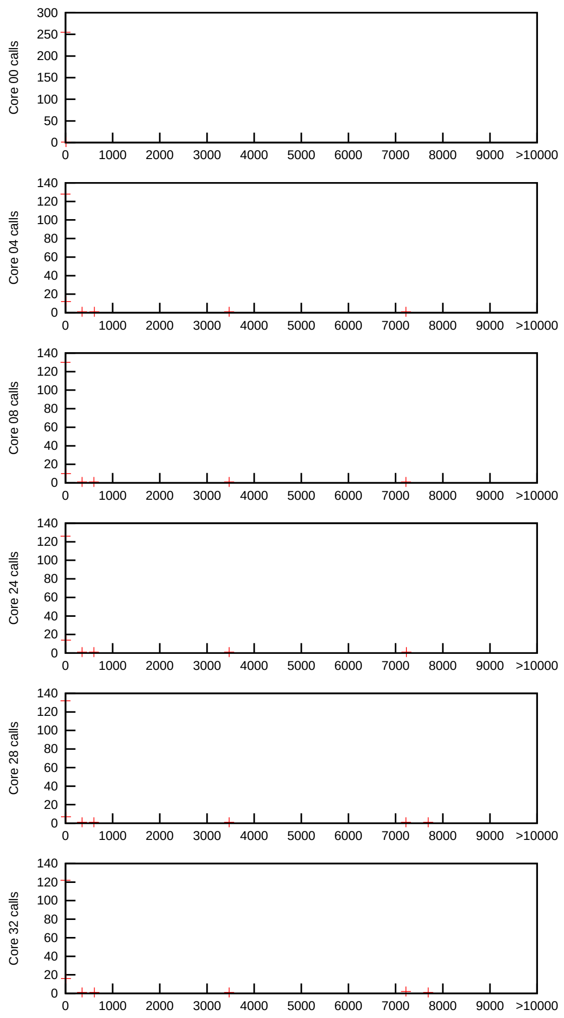
Figure 5.9: MG get_pshared behavior.



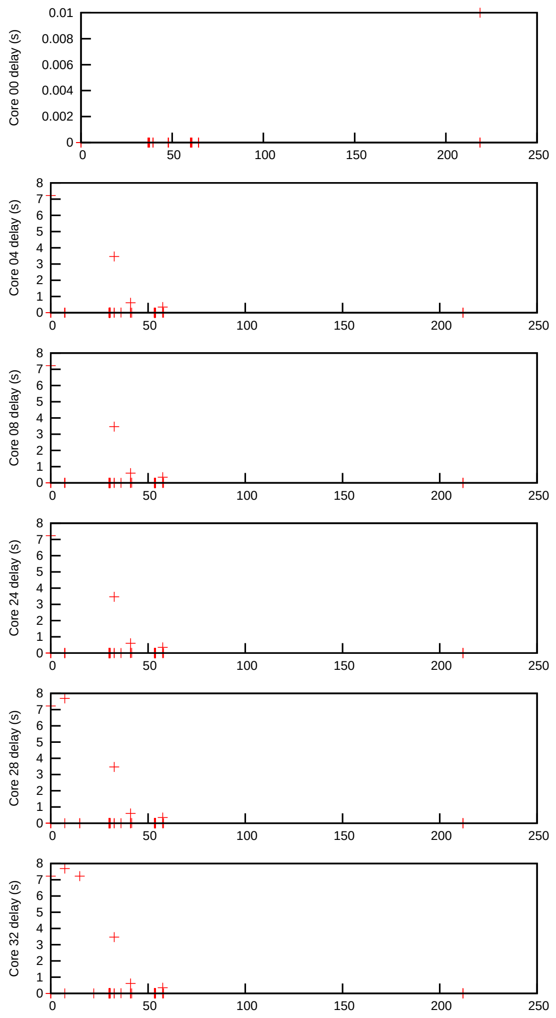
(a) Mean delay per core



(b) Total calls per core



(c) Number of operation calls per length (in ms).



(d) Calls and corresponding delays (in seconds) according to execution point (in seconds).

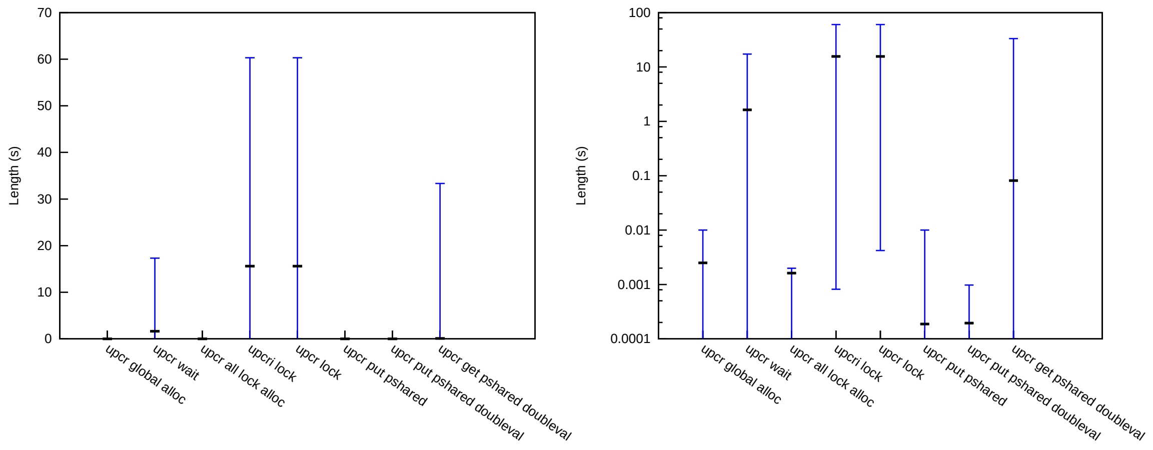
Figure 5.10: MG get_pshared_doubleval behavior.

5.2.3 NAS EP kernel

EP, the last NAS benchmark that we will use, is an embarrassingly parallel application that performs floating point operations with almost no-communication.

In our study we have been running EP class D, without optimization, because it is the maximum configuration that supports SCC constrains.

Beginning with figure 5.11, we can see that the long operations that EP uses are `wait`, `upcr_lock`, `upcri_lock` and `get_pshared_doubleval`. Note that both lock operations are equivalent, because one always calls the other in UPC Runtime implementation.



(a) Linear scale.

(b) Logarithmic scale.

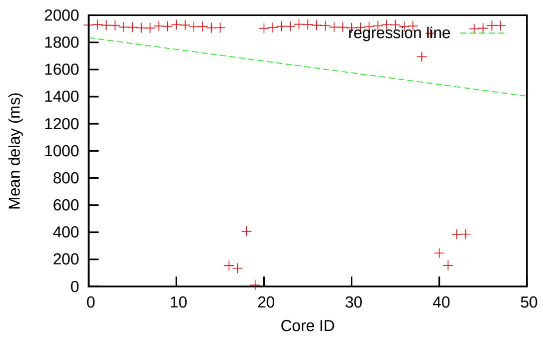
Figure 5.11: Length of EP calls.

As in the previous analysis, we begin studying `wait` operation profile on EP kernel. The first conclusion drawn from figure 5.12 is that there are only 9 calls per core and 8 have got insignificant delays. The ninth one corresponds to the barrier at the end of execution, and is large (18 seconds) because it accumulates the short imbalance through all the execution.

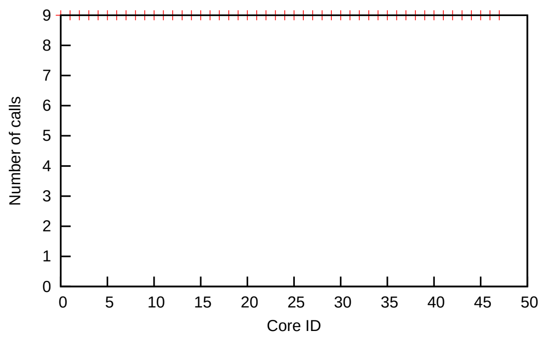
In figure 5.13b we can observe that there is only one `lock` call in the whole execution. This makes `lock` a bad candidate for energy savings opportunities.

Finally, as we can see in the set of figures 5.14, `get_pshared_doubleval` operation, that initially seemed interesting, is not interesting from energy savings point of view, because there are only few and short calls in 47 cores, and only in the first core the delay is, sometimes, large.

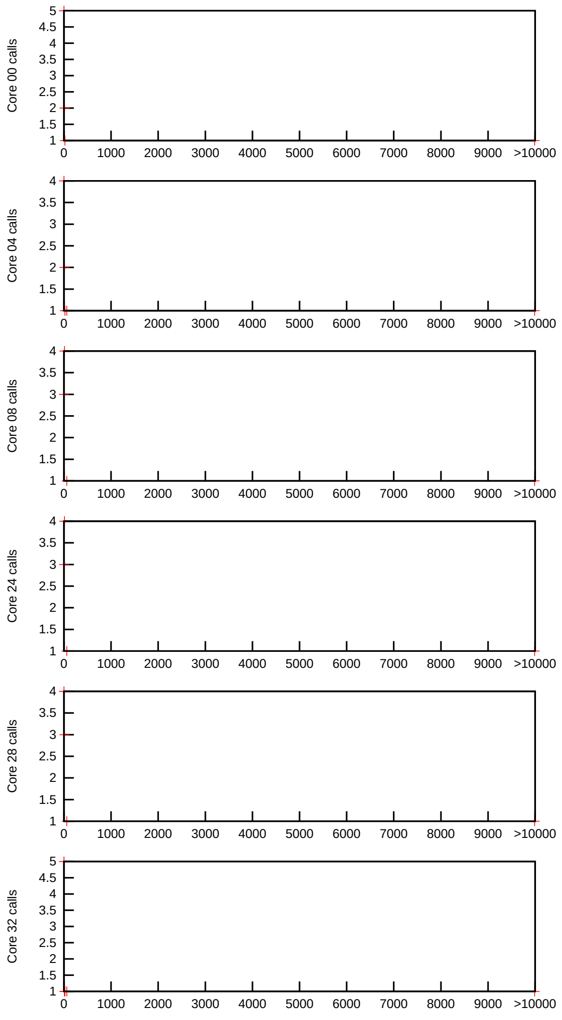
Although common sense tells us that CPU-intensive applications are not suitable for energy savings, this study shows us that EP does not give many energy savings opportunities, because EP's profile is not communication-bound nor memory-bound.



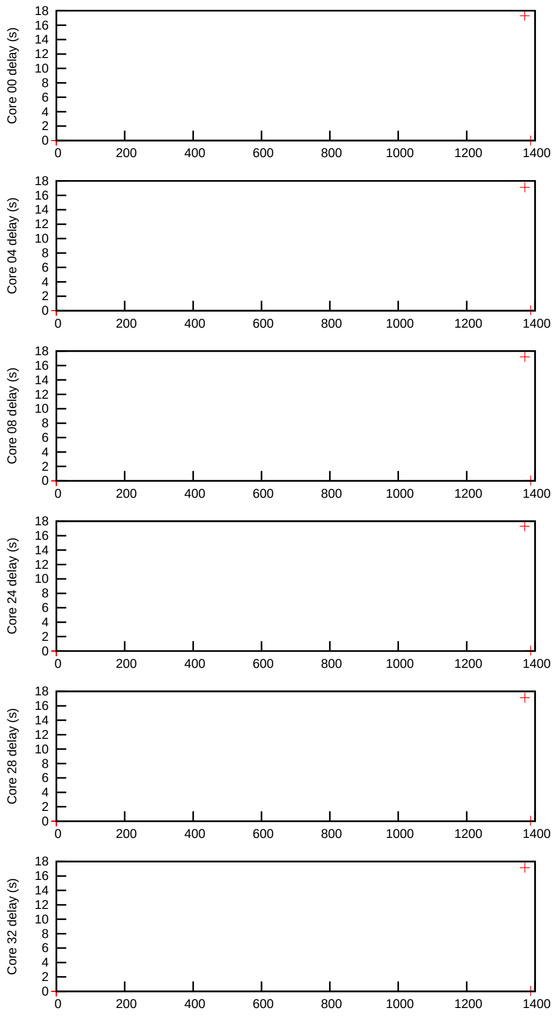
(a) Mean delay per core



(b) Total calls per core

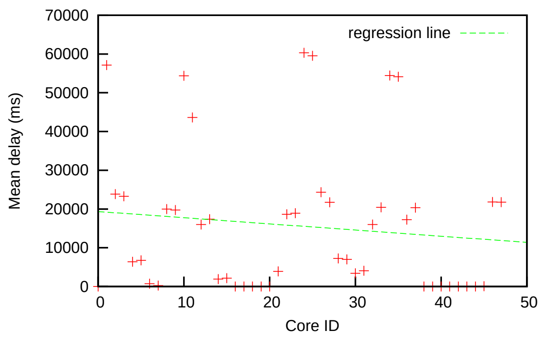


(c) Number of operation calls per length (in ms).

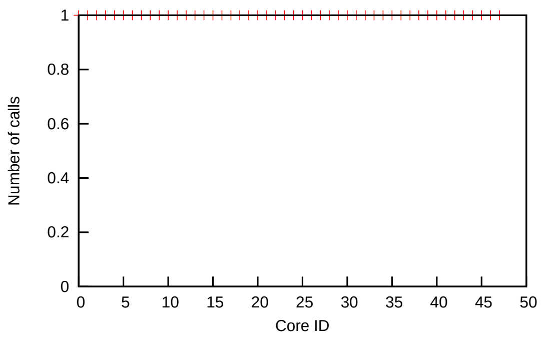


(d) Calls and corresponding delays (in seconds) according to execution point (in seconds).

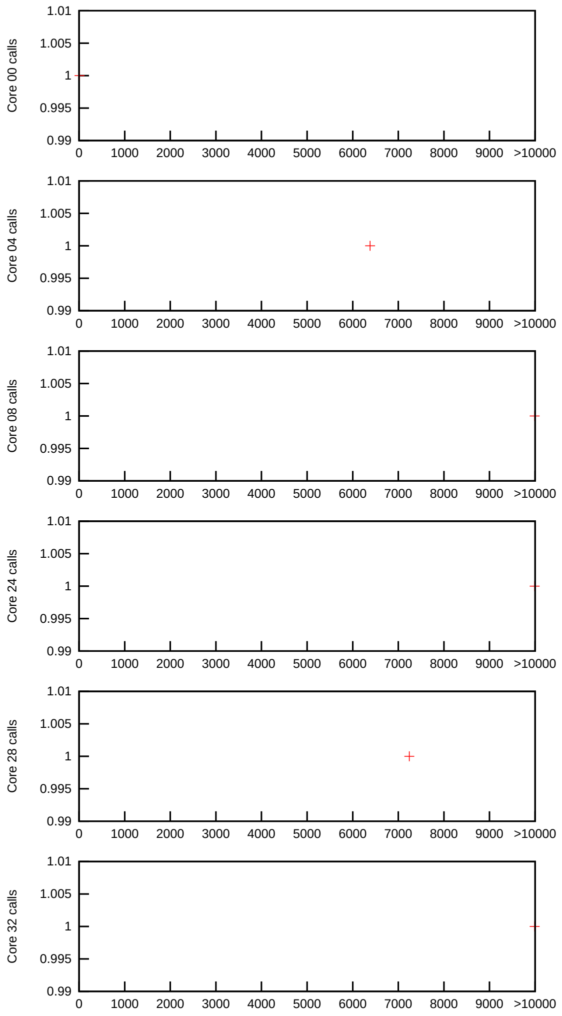
Figure 5.12: EP wait behavior.



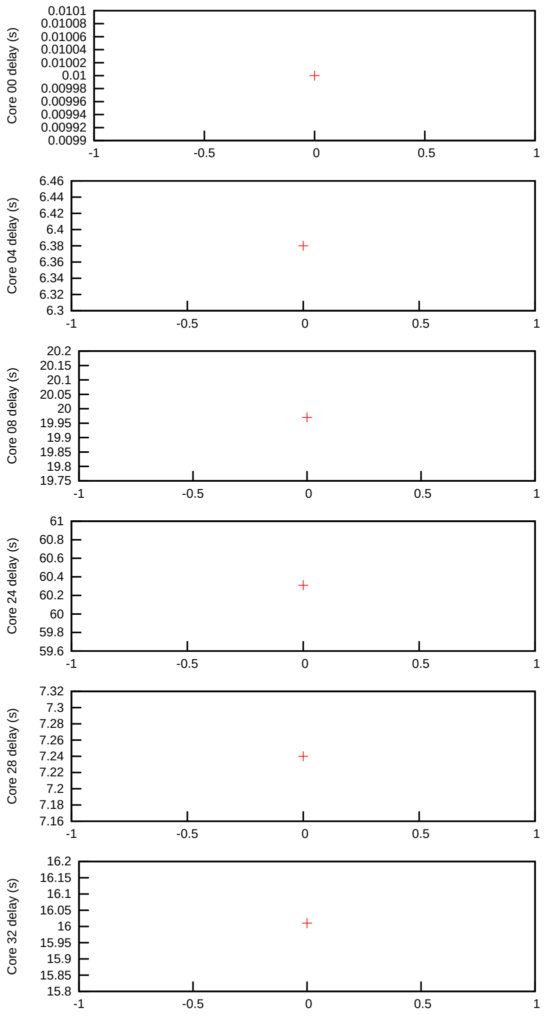
(a) Mean delay per core



(b) Total calls per core

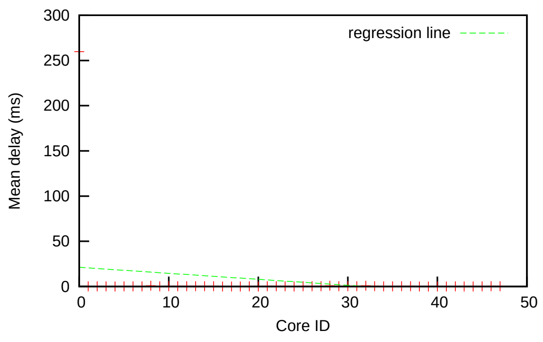


(c) Number of operation calls per length (in ms).

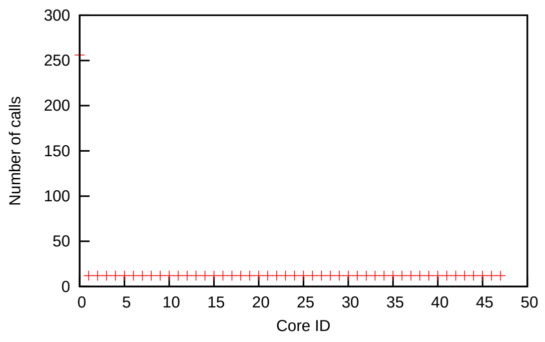


(d) Calls and corresponding delays (in seconds) according to execution point (in seconds).

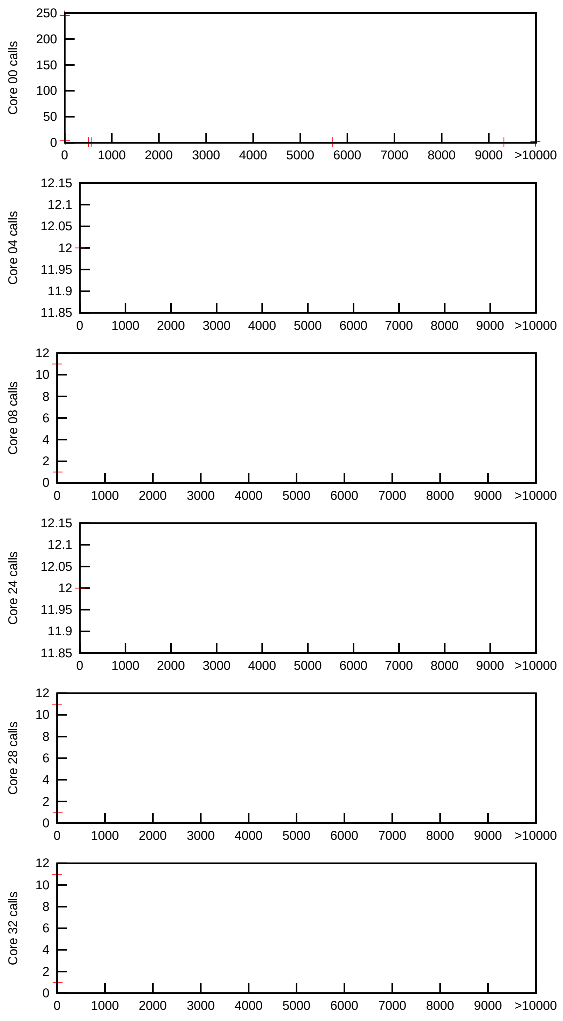
Figure 5.13: EP lock behavior.



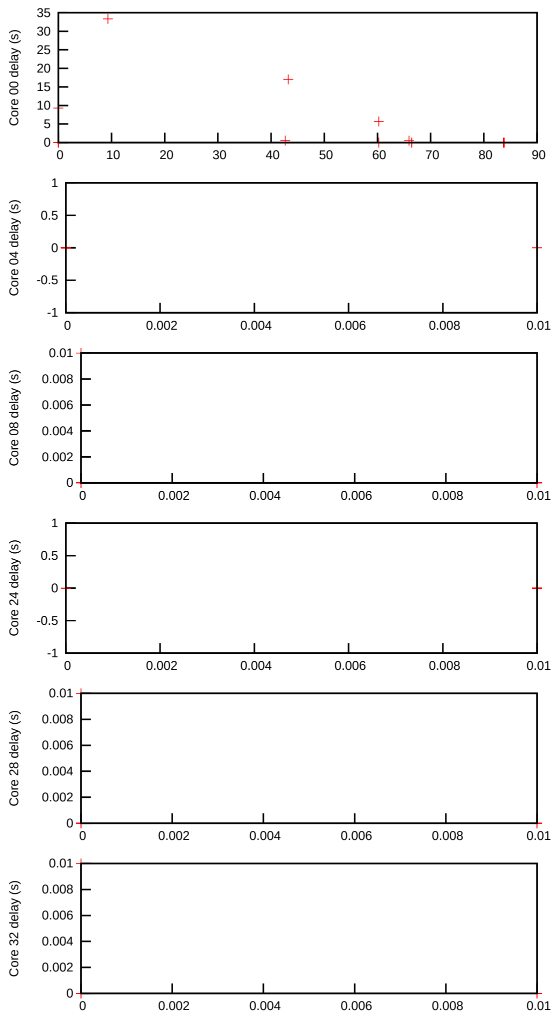
(a) Mean delay per core



(b) Total calls per core



(c) Number of operation calls per length (in ms).



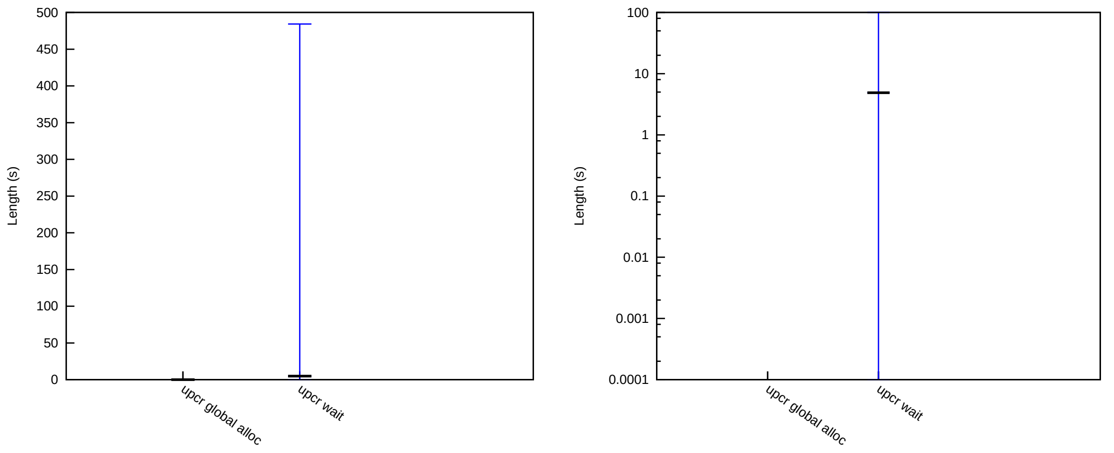
(d) Calls and corresponding delays (in seconds) according to execution point (in seconds).

Figure 5.14: EP get_pshared_doubleval behavior.

5.3 Sobel

Sobel is an edge detection application, a kind of applications useful in several fields such as computer vision. The parallelized version of this algorithm partitions the image among the cores, performs calculations locally and, when it needs to shift the data through the last row of a thread data, it access to the elements of the next row (allocated in the next contiguous core).

The first step to analyze the Sobel application is to determine which UPC Runtime operations it uses. In figure 5.15 we can see that it uses `wait` and `global_alloc`. Note that during the instrumentation phase we disabled the `upcr_get_pshared` operation logging, because we observed that each core called it about 90 millions of times (89786623 calls); this huge amount of calls produced the execution time during instrumentation increase a lot, and that is why we decided to disable it to run the execution more realistic.



(a) Linear scale. (b) Logarithmic scale.

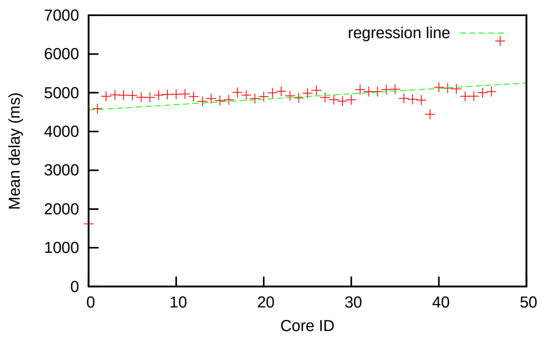
Figure 5.15: Length of Sobel calls.

Like NAS benchmarks studied above, the next step is to observe the behavior of the main calls; in this case, only `wait`. We don't show the results of `upcr_get_pshared` because all the calls are negligible.

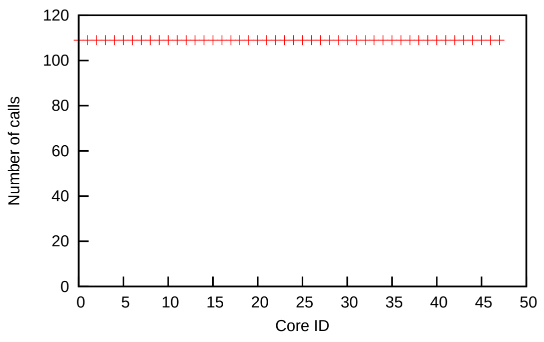
As we observe in figure 5.16, we should distinguish the behavior of the core 0 and the rest of the cores. One reason for this is due to the initialization phase, made only by core 0. Note that there is a barrier in the beginning of the execution, that delays the non-0 cores about 500 seconds, and this barrier is finished only when core 0 finishes the init phase and calls the corresponding barrier operation (see the lonely 0-delay point around position $x = 480$, in figure 5.16d).

Once the initialization is done, the benchmark repeats $N = 100$ times the Sobel algorithm, synchronizing with a barrier at the end of each iteration. As we can see in figure 5.16c, the slack period corresponding to the wait operation is about 0.5 seconds on all cores except cores 0 and 47 (core 47 log is not shown in the figure because of a space matter), which are about 2 seconds long.

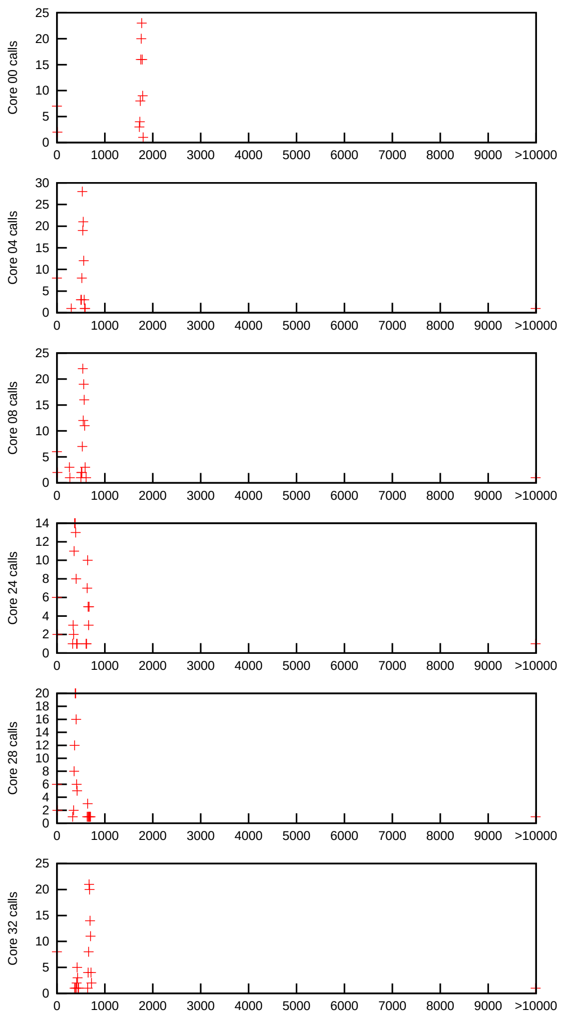
Although this application does not seems very interesting from the point of view of the iterations phase (due to little imbalance), it can be useful to study the core-0 driven initialization phase.



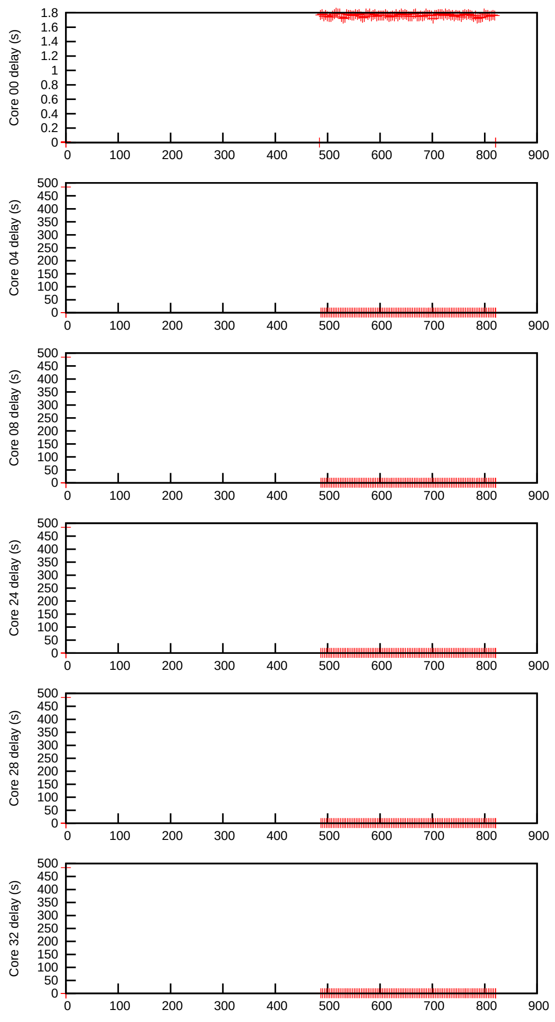
(a) Mean delay per core



(b) Total calls per core



(c) Number of operation calls per length (in ms).



(d) Calls and corresponding delays (in seconds) according to execution point (in seconds).

Figure 5.16: Sobel wait behavior.

5.4 Matmul-based synthetic imbalanced application

In the profile of the UPC version of the NAS applications we have seen that `wait` is an usual operation. We know that this UPC Runtime instruction is used to implement barriers. That's why we can suppose that the more *imbalanced* an application is, the more energy savings we will achieve. A parallel or distributed application is balanced if the slack period during a barrier is almost null. It is imbalanced if there are threads that spend lots of time waiting, in barriers (big slack periods), compared to other threads, which slack periods are almost null.

The main goal of using the synthetic `matmul` is to study the potential of voltage scaling for different levels of load imbalance caused by barriers (`wait` operation).

5.4.1 Algorithm

Basically, the main algorithm is:

```
for (i=0 ; i<N ; i++) {
  perform A matrix multiplications, distributed in 48 cores;

  if(MYTHREAD is in Voltage Domain X)
    perform B matrix multiplications;

  upc_barrier;
}
```

5.4.2 Parameters

The parameters to control the behavior of the aforementioned algorithm are:

N The number of overall iterations

A The number of all 6 voltage domain distributed matrix multiplications

B The number of only one voltage domain matrix multiplications

The first set of tests performed with this benchmark was the regular execution (i.e. no power management), called in this document *original tests* or *base tests*. The parameters used in this set of tests are shown in the table 5.1, which relates the level of imbalanced produced by the parameters, and that is calculated experimentally, with the data collected during the execution.

Another significant results from the collected data are the execution time, that was about 20 minutes per test and the consumed energy, about 105 KJ.

N	A	B	Imbalance %
36	29	500	3%
36	28	1000	7 %
36	25	2500	17 %
36	22	4000	27 %
36	20	5000	33 %
36	18	6000	42 %
36	15	7500	51 %
36	13	8500	58 %
36	10	10000	67 %
36	7	11500	76 %
36	5	12500	84 %
36	2	14000	93 %
36	1	14500	97 %

Table 5.1: Parameters used in the matmul *original* (or base) test.

5.4.3 Conclusions

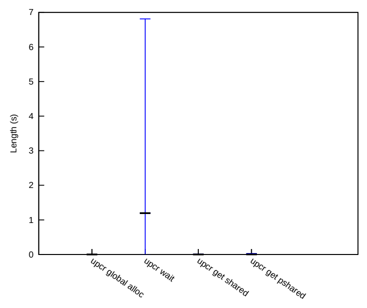
We ran `matmul` with load imbalances ranging from 3% to 97%. Figure 5.17 shows that `wait` calls are the largest ones (i.e. most energy-savings capable). The histograms and the mean delay per core plots of the `wait` calls are shown by figures 5.18 and 5.19, respectively.

Note that in tests with very imbalanced applications (Figures 5.18b, 5.18e,5.18c and 5.18f) many calls are longer than 10 seconds.

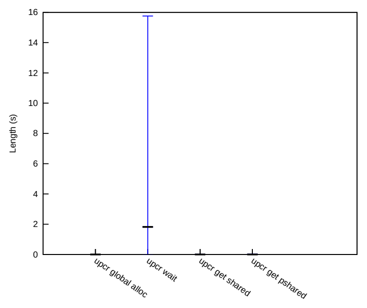
Comparing parameter B in table 5.1 and resulting `wait` calls in figure 5.20, we can see that the number of calls is directly proportional to the B parameter.

Figure 5.21 shows the histogram of the `wait` operation length, while figure 5.22 shows the execution point in which each `wait` call have been done, and their corresponding delays (in seconds).

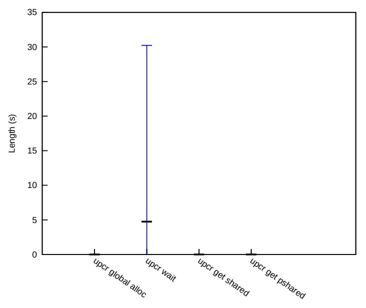
The results show very different behaviors depending of the percentage of load imbalance. Specifically, longer `wait` calls correspond to larger load imbalance percentages.



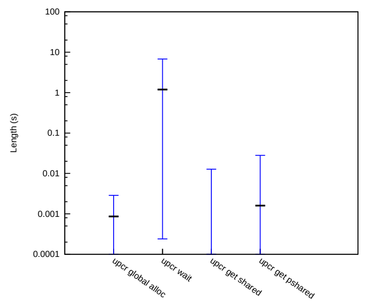
(a) 3% imbalance in linear scale



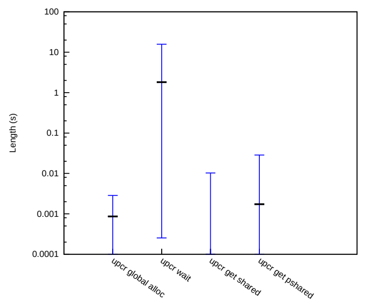
(b) 51% imbalance in linear scale



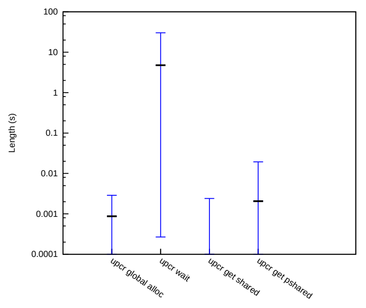
(c) 97% imbalance in linear scale



(d) 3% imbalance in log scale

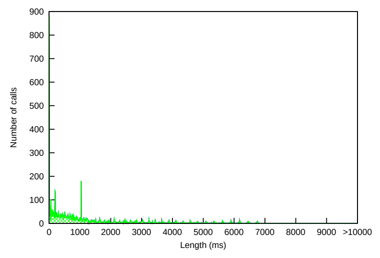


(e) 51% imbalance in log scale

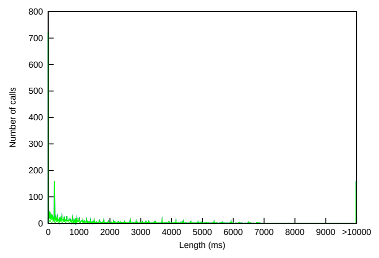


(f) 97% imbalance in log scale

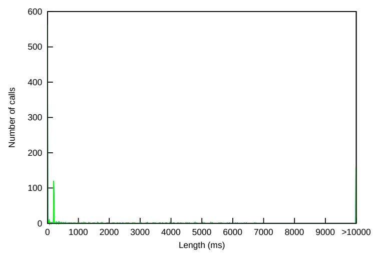
Figure 5.17: Matmul per-core length of calls in linear and logarithmic scales.



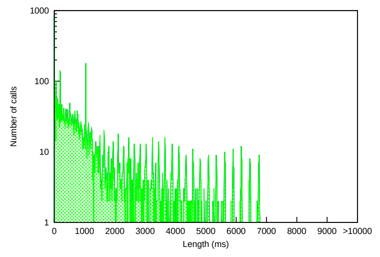
(a) 3% imbalance in linear scale



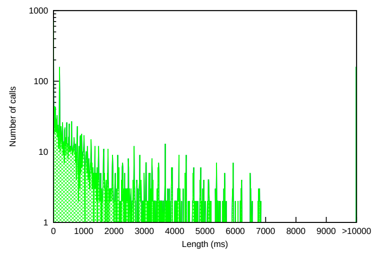
(b) 51% imbalance in linear scale



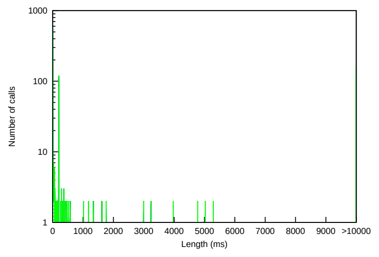
(c) 97% imbalance in linear scale



(d) 3% imbalance in log scale

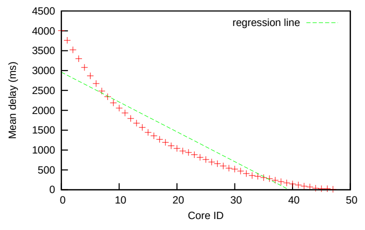


(e) 51% imbalance in log scale

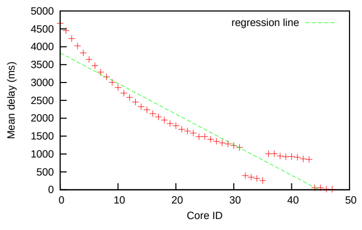


(f) 97% imbalance in log scale

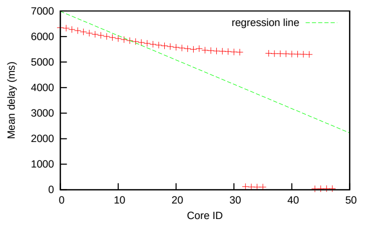
Figure 5.18: Matmul per-core num of calls in linear and logarithmic scales.



(a) 3% imbalance

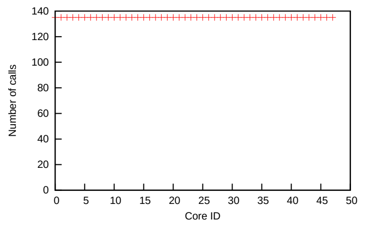


(b) 51% imbalance

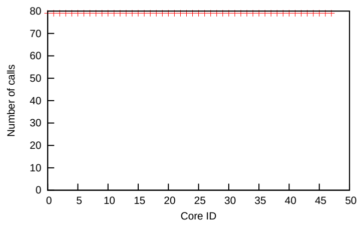


(c) 97% imbalance

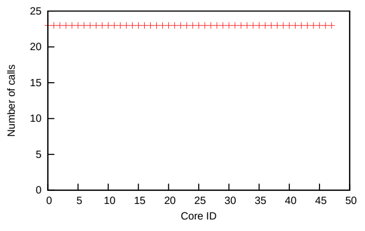
Figure 5.19: Matmul mean delay per core.



(a) 3% imbalance

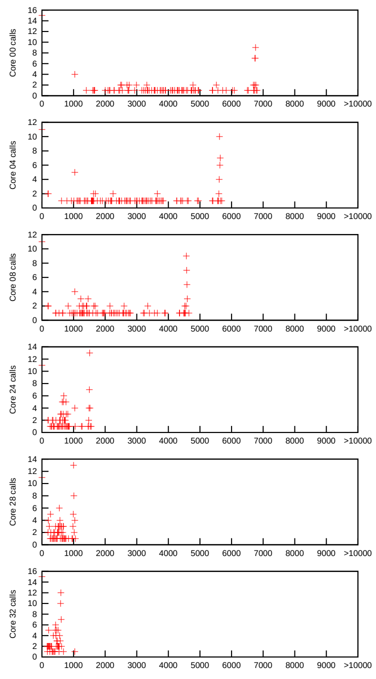


(b) 51% imbalance

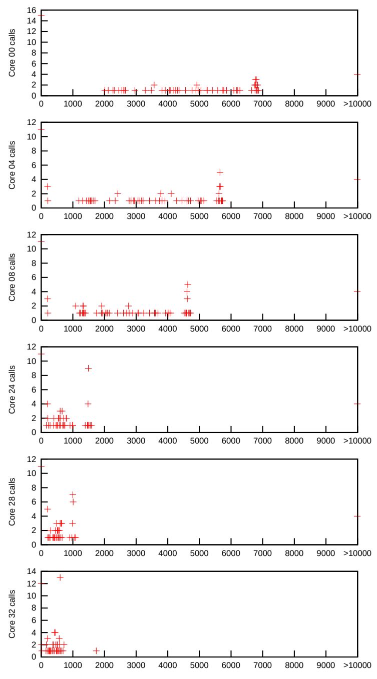


(c) 97% imbalance

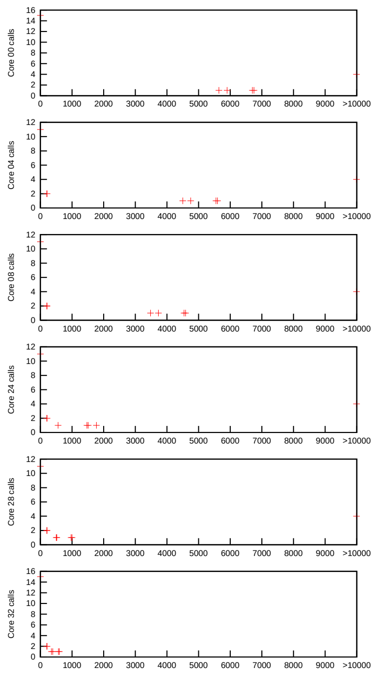
Figure 5.20: Matmul total calls per core.



(a) 3% imbalance.

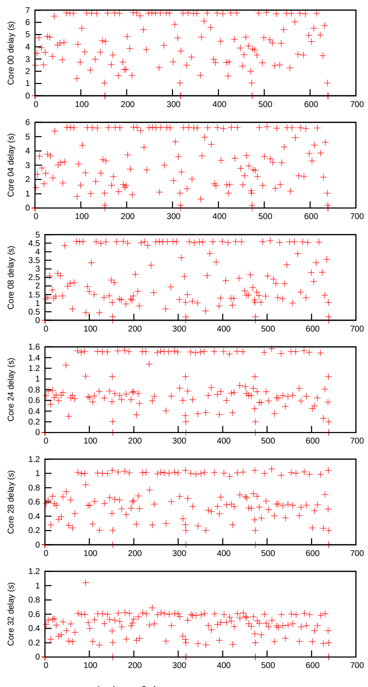


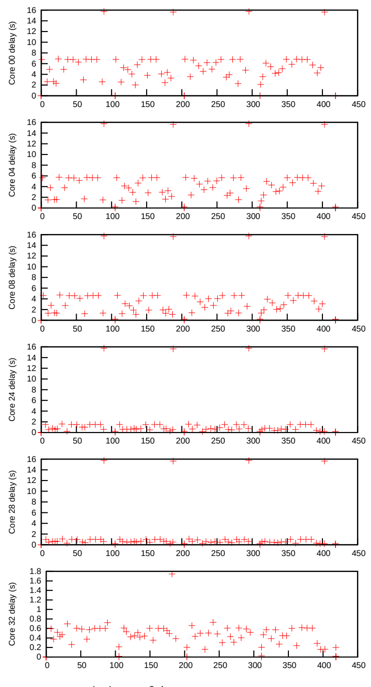
(b) 51% imbalance.



(c) 97% imbalance.

Figure 5.21: Matmul number of wait operations per call length (in ms).


(a) 3% imbalance.


(b) 51% imbalance.

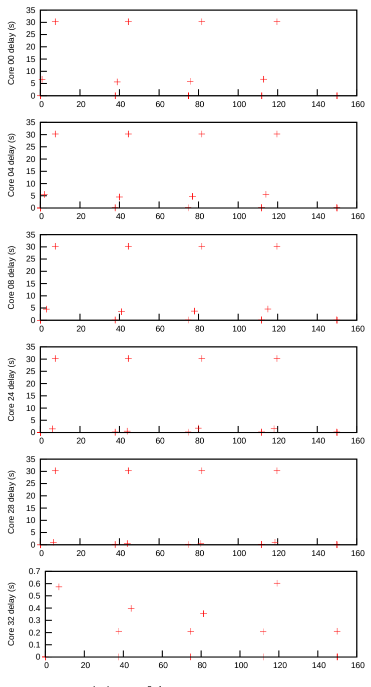

(c) 97% imbalance.

Figure 5.22: Matmul calls and their corresponding delays (in seconds) according to execution point (in seconds).

5.5 Conclusions

With NAS profiling we concluded that memget and wait are the most energy-saving-capable operations:

- `wait`: this operation, executed when user application requests a barrier, stops the application execution until the other cores reaches the barrier too. That's why during this period of time we can slow down the processor (i.e. drop the frequency or voltage down): there is a slack period while running this `wait` operation. In this sense, imbalanced applications (distributed applications with nodes that must wait for long periods) seems more suitable to be power managed.
- `memget`: this operation, that is called frequently and is relatively long, is intended to copy bulk data from the local memory of one node to the local memory of another node. In usual systems, where the CPU is far faster than the network connecting nodes, during the execution of the operation we can save energy by decreasing the processor power. In the SCC, however, the speed of transference is related to the speed (i.e. frequency) of the core (because the origin or destination of the data is the L2 cache). Therefore, at first sight, a decrease in the power of a core during `memget` call would end up to a repercussion on the data transfer delay.

Another conclusion is that we may use Sobel in order to study the power and flexibility of cross-layer application-level tools.

Finally, the main goal of the synthetic `matmul` application is studying the potential energy savings and delay penalty (upper bounds) of both runtime and application-aware power management techniques for different levels of load imbalance.

Chapter 6

Power management middleware

6.1 Motivation

Our aim is very clear:

- We want to reduce the energy consumption by taking profit of the slack periods of an application. As we have found thanks to the profiling of several applications on the studied platform, the main operation that makes the processor relax is `wait` and, sometimes, `memget`.
- This process should be independent of the PGAS implementation (UPC, CAF...).
- Additionally, we want to provide the tools to check which periods are critical and which are more flexible, in order to allow the programmer the possibility of easy-tune their application's energy performance. This goal can be achieved by allowing the programmer include some 'hints' in its code.

We have been developing a middleware dedicated to facilitate power management (PM) with the aforementioned features. This middleware is based upon requests, regardless they come both from the user or the PGAS run-time. We consider a *request* or a *notification* a message addressed to the middleware, dedicated to help it decide when the processing unit can be slowed down, when it may run at maximum performance... Usually notifications can be of three types: indicating the beginning of a call, notifying the corresponding end or pointing out a user hint. We have designed it as generic as possible, in order to allow all kinds of requests. Despite this, after the results of the profiling phase with SCC and UPC, we decided to allow by default only wait and memget operations.

6.2 Architecture overview

The *Application* column of figure 6.1 represents a standard execution model of a UPC application on the SCC. As we can see, there are several layers in the application column, each one communicating to their surrounding layers, vertically. Note that, in order to improve communication, we have been using RCKMPI, the MPI implementation for SCC (see [40]) that uses the SCC's on-die message passing buffers and mesh as a physical resource.

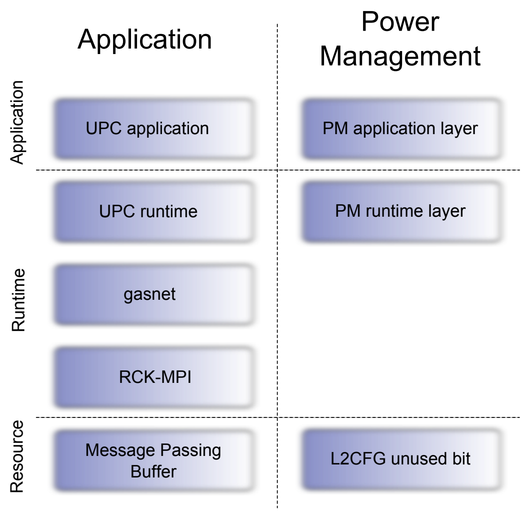


Figure 6.1: External architecture overview of the power manager.

Our middleware can be considered as the left column of figure 6.1, as it adds extra capabilities and the corresponding intra-layer communication between both columns.

The internal architecture of the power manager middleware is shown in the figure 6.2. As we can see, the main input of our middleware is the communications subsystem, which is responsible for receiving the requests and redirect it to the requests handler subsystem. This module will distinguish which notifications needs to be filtered, which requests are critical enough to be redirected directly to the power adjuster, or which changes the middleware state (application hints). Moreover, we give the programmer the capability of tune some power manager parameters, in order to adequate the power manager to each concrete application. Thereby, the power manager loads the configuration from a specific configuration file.

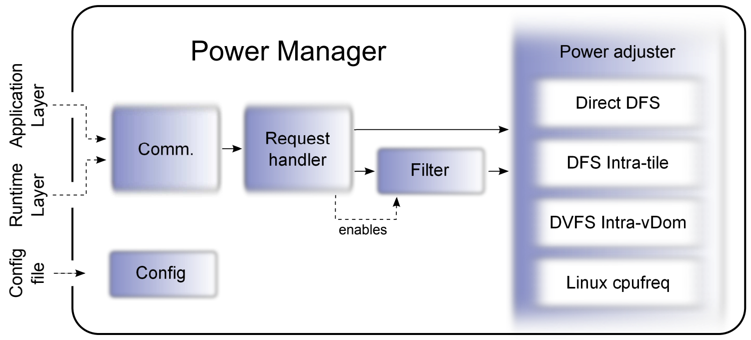


Figure 6.2: Internal architecture overview of the power manager.

The power manager have been implemented using the C programming language, as it allows, among several other advantages, easy and efficient methods to access to processor registers. Note that all modules have been designed to be thread safe, protecting critical parts with mutexes.

In the following sections we will describe the main features of each of this modules. Let's take a look first, however, on how the cross-layer power management is implemented.

6.3 Cross layer

We mentioned that both application layer and runtime layer are able to send requests to the power manager. As stated before, application notifications will only appear when the programmer wants to tune the application performance. For this reason, the power manager must process this requests with higher priority in front of runtime layer requests.

This means that an application layer hint will determine the behavior of the power manager in front of the rest of notifications received from runtime layer. There are several hints, which one with its own effect:

- The `PERFORMANCE` hint indicates the power manager to discard all notifications (until the next hint) and to work on full voltage and frequency, ensuring full performance.
- Regarding `CONSERVATIVE` hint, power manager will work in order to find optimal situations in which energy savings can be profitable (although reduced) with little or no time penalty. This situations are usually the slack period produced by a long memget or wait call.
- Contrary to what happens to `PERFORMANCE`, `SAVE_ENERGY` hint tries to save energy without taking into account the possible time-penalty. This hint is mapped to a continuous voltage reduction, regardless the slack periods of the application. This voltage reduction tries to balance the time penalty and energy savings (i.e. tries to decrease energy delay product, regardless the time-penalty).
- Finally, power manager have the same behavior with the `AGGRESSIVE_SAVE_ENERGY` hint. In this case, however, the voltage reduction is maximum, which can be useful on some applications. We should be careful, however, because potentially it can produce the opposite effect: because of the big frequency reduction, the execution time will increase that much that the energy consumed would overpass the non-power managed consumed energy.

6.4 Communication subsystem

The designed system can work in two modes: *integrated* in the PGAS runtime, or as a *standalone* application. Both modes are possible due to the fact that the power manager have been implemented as a standard C library and to the modularity of the design. In case of having the power manager code integrated in the PGAS runtime, communication subsystem does not make sense, as this is done via function calls. Actually, both logical applications (PGAS runtime and power manager) would run in separated execution threads and requests are sent through a queue buffer in order to reduce the footprint in the main thread.

However, to ensure the independence of the power manager to the PGAS runtime implementation (UPC, CAF...), we added a communication subsystem that allowed us to isolate the power management modules from UPC Runtime. In addition to this isolation, we try to reduce the intrusiveness in the PGAS (UPC) runtime (i.e. the application stack have been modified as few as possible). Therefore, all the modifications required to allow power management are simple calls to a C function, which will notify the power management system that a certain condition happened (a barrier's wait that just begins -in the runtime layer-, a critical zone is about to begin -in the app layer-...). This application-sided C function lazy connects to the power manager (on the first call of each execution) and is implemented in a separated header file.

The requests arrives to the power manager by standard unix sockets. Therefore, this module acts as a server. A brief outline of the process is:

- It listens to a socket file.
- For each connection established, it creates a new thread that handles the power management notifications.
- For each request received, the communication module redirects it to the requests handler subsystem, in order to do an efficient pre-processing.

With both system implementations (integrated PM, or as a standalone application) we could test the footprint introduced by the extra communication module. With the help of several tests we determined that this module introduces about 1 % of penalty in both time and energy.

We decided to use it, however, to increase the usability, portability and maintainability of the system, and keeping in mind that it is used as an experimentation platform, not designed for production.

6.5 Request handler subsystem

The request handler subsystem is the next module in the information flow. It receives the requests both redirected from the communications module (if in stand-alone mode) or directly from the PGAS runtime (if in integrated mode) and pre-process them. This preprocessing carries out several tasks:

- Distinguish operation notifications in front of programmer hints.
- Implement the established layer priorities: application layer requests may have more priority in front of runtime layer ones, because we must give more control over power decisions to the programmer.
- Distinguish begin and end notifications and check their coherence. This can be easily understood with an example, if a `OPERATION_1_BEGIN` was received from the runtime layer, the corresponding `OPERATION_1_END` must be the next command from the runtime layer (as it cannot be nested calls).

This module is in charge of controlling the filter module, disabling it if necessary, depending on the request received. For example, due to priorities, if a runtime layer request arrives while an application layer's `AGGRESSIVE_SAVE_ENERGY` hint is active, the request handler subsystem may want to discard the runtime request, regardless its intention.

We decided to allow all requests come from any layer, allowing application layer the full control and, therefore, giving the programmer the possibility of more fine-grained tuning than only hints, if desired. Therefore we don't control this situation.

To sum up, this module is the responsible of storing the state of the power manager, and act differently depending on it. It can discard requests, it can send it to the filter module or even it can transform it to a direct frequency or voltage change.

6.6 Filter module

The power manager will be notified when an external request is received. Then, it is pre-filtered by the request handler module. Finally, it may redirect the notification to the filter module. This module, as the name indicates, is the responsible of determining or predicting which redirected requests are susceptible to be profitable from an energy-savings point of view, and which aren't.

The input of this subsystem is, therefore, a begin or end request, and the corresponding desired frequency or voltage level. The first thing the module does is to check the desired level against the current level. In case both levels match, the request is discarded and, otherwise, the actual filtering process begin.

Ideally, the filtering process would have to know the length of notified situation and directly discard the requests that may have short delay. Predicting this, however, is not algorithmic nor computationally easy. After studying several filtering approaches and other prediction models, we decided to design a simple one, as it must run in a performance-constrained environment, like a SCC core is. The prediction algorithm must have, therefore, low footprint (i.e. small memory and processor requirements) to avoid performance loose.

As we know from the profiling phase, among the potential useful operations (basically, wait and memget), only the longest ones are susceptible to allow energy savings. The shorter a call is, the less benefit we will achieve, since changing the frequency, and even more the voltage, consumes time and energy. This ends up to a threshold delay, in which, from the energy savings point of view, shorter operations are adverse

The methods that our system currently supports follow the aforementioned simplistic approach, and all three are based on the same principle: discard the shortest calls by waiting a given amount of time before accepting requests:

- Fixed time *threshold*. When a begin request is received, the filter waits a specified time indicated by the *threshold*'s value. If this delay is reached and no end notification have been received, then the request subsists and, therefore, is traduced to a corresponding frequency or voltage change (i.e. the request is redirected to the power adjuster subsystem). Otherwise, if in this period the corresponding end notification reaches the filter, the original request will be discarded, i.e. filtered. This technique tries to exploit the fact that we have seen during the profiling phase: if we look at the delay distribution of the calls, we would be able to create call clusters. In other words, a big percentage of calls are very short, while the remainder are generally spread through bigger time-bands (i.e. are longer). To sum up, this technique tries to eliminate the shortest calls (i.e. the zero-length cluster).

- Variable time threshold. This approach is based upon the previous one. The only difference is that, in order to obtain the threshold, we calculate the product of the call length *moving average* and a given constant (for example, 10 %), customizable as a fixed parameter. The average is recalculated on each request, based on the history of a call.

- *Mixed* solution. This policy is based upon the fixed and variable threshold and is aimed to accommodate the variations according to the execution region. The second algorithm have got the drawback that is very influenced by individual calls (for example, a lonely big call would increase the moving average a lot). This motivates the creation of this mixed solution: within a fixed range, regulate the threshold using the moving average of the recent history.

Note that all three solutions can be designed to have a linear processing cost, depending only on the number of processed requests (O(n) cost) and that, even on the moving average approach, it still guarantees the same computational cost.

There are applications which call's length is homogeneously spread among time spectrum (i.e. the length of the call follows a random pattern), and therefore, calls cannot be clustered. In this cases, the aforementioned approach is not ideal. Remember that the initial purpose of the filter was to eliminate short slack periods to avoid performance reduction problems due to the DFS/DVFS technique overhead. Therefore, calls with a length similar to $threshold + a, a \rightarrow 0$, will end up causing the same problem. In other words, although the smallest calls have been filtered, the remainder medium calls would cause the same effect as the smallest calls used to cause, as immediately after passing through the filter, the corresponding end would be received. Therefore, the penalty induced by the power change would be bigger.

To minimize these situations, a prominent feature in the filtering algorithm has been implemented. This new feature is the capability of including a medium level of frequency or voltage. It allows, in certain circumstances, to distinguish between short, medium and long calls. Shorter calls are discarded by waiting a short amount of time before accepting the request. When this point of time is reached, the algorithm can slow down the frequency or voltage[1] to a middle step, in order to reduce the penalty of DFS or DVFS method.

With this approach, smallest calls are discarded, medium calls doesn't cause a big time penalty, and long calls are most profitable, because in the beginning of the slack period they run in medium frequency, and for the remaining slack period, the processor is in low frequency.

The policy and its parameters can be individually tuned according to the call profile. To set an example, memget's have got different profile than wait's and, therefore, they should be predicted using different parameters and/or policies.

6.7 Power adjuster subsystem

The main target of this module is to apply the frequency or voltage+frequency scaling to achieve a reduction of the energy footprint. This is done at the end of the power manager path: when a request have been considered critical or longer enough to allow a profitable energy reduction, the filter or request handler subsystems may send a request to the power adjuster indicating the desired frequency or voltage to set. This module will adjust the system power level to the desired, if this condition is not being fulfilled.

In a typical cluster, this module would be the simplest, as we could use the tools that the operating system offers (i.e. the cpu-freq module in Linux systems). To guarantee the middleware portability, we have implemented the cpu-freq based power adjuster. In order to work, however, the userspace governor must be enabled.

The experimental Linux ported to the SCC does not support SCC-specific power management tools yet. That is why the cpu-freq module is not useful on the SCC, and, therefore, we implemented this feature directly managing the hardware. However, SCC library (RCCE) offers power management functions, which allows to change the voltage or the frequency in the scope of so-called power domains (at practical effects, a power domain corresponds to a voltage domain). Apart from not allowing fine-grained frequency management, RCCE library uses the Message Passing Buffer in order to synchronize all cores on a power domain. This is a problem, as

[1]Although middle scaling of voltage have been implemented, the really useful case is frequency, because the cost of changing voltage is very high.

RCKMPI, which uses RCCE too, needs MPB to work. As we want to take profit of fine-grained power management capabilities with minimal performance penalty, we decided to re-design power management, accessing directly to the hardware.

As described in the SCC power management tools, we can apply two techniques to save energy: frequency or voltage scaling. As stated before, frequency can be changed as a per-tile basis, and voltage must be changed in the voltage domain scope.

Taking this into account, the power adjustments can be done in several ways, each one have been implemented on a separated submodule:

- If we want to change the frequency in a core, we must take in consideration the fact that it will slow down not only the involved core, but the other core in the tile.
  - If the user decides to change the frequency regardless this situation, the time performance may decrease, as the other core may be using the CPU intensively. The *Direct DFS* submodule implements this case.
  - The other option, however, is to synchronize both cores in a tile and apply the frequency reduction only when both agree: when they are in a slack period at the same time. The *DFS Intra-tile* covers this assumption and, therefore, this option may prevent the loss of performance.
- We may want to adjust the voltage. In this case, the eight cores included in a voltage domain (vDom) must be synchronized in order to avoid a performance loose that would end in an energy increment. Note, however, that to guarantee stability is imperative to adjust the frequency of all tiles in a voltage domain before adjusting voltage. This feature have been implemented by the *DVFS intra-vDom* submodule.
- As stated before, a module that uses the cpu-freq interface have been implemented too in the *Linux cpu-freq* submodule, to guarantee the aforementioned portability.

In the following sections we are detailing some aspects of each SCC-related power adjuster submodule.

6.7.1 Direct DFS

The main goal of this module is to implement frequency adjusting tools. As we explained in the SCC's Power management capabilities section (see 2.8), the mechanism to change the frequency is simply writing a concrete value (corresponding in a register). This value corresponds to the desired frequency divider level, which must be in the 2 to 16 range.

An application of this method may be using only one core per tile (totaling 24 cores in SCC), and, thus, not introducing time penalty due to non-slack-synchronization periods. This allows working at the maximum fine-grained power control, although cannot be used in production environments, because of the obvious power resources loss.

6.7.2 DFS intra-tile synchronization

To achieve this core-to-core synchronization we need some kind of communication mechanisms. The SCC mechanism to send messages between cores, is, as we stated before, the Message Passing Buffer. As said, this buffer is being used for the MPI SCC implementation, so the power manager can't use it without a reduction on performance. On the other hand, another possibility communication is via external memory. This method, however, would be not as fast as we desire.

Therefore, we have been studying extra communication methods that does not require to use external SCC resources. The result of this study is that we can implement the synchronization using an unused bit in the hardware-implemented L2CFG configuration register. This register is present twice in a tile, so we can use 1 bit per core to communicate each other.

The intratile synchronization algorithm to communicates cores have been designed to be distributed, fast, and to waste very few resources.

This algorithm can work in two modes: using only high and low frequencies, or introducing a third component, the medium frequency.

- In case of two frequencies, when a core wants to drop the frequency, it checks their mate bit. If it is set, the frequency is dropped. If it is unset, the core sets to 1 it's own bit. When one core wants to rise the frequency, it simply does it and unset both bits. Note that to change the frequency we use the functionality implemented in the direct DFS module.
- On the other hand, we can use a third component: the medium frequency. In such a case, the algorithm is completely different. One bit is dedicated to encode the lower frequency desire, while the other encodes the medium frequency desire. In the initial state, both bits are low (0) and the highest frequency is set. When a core wants to change to medium or low frequency, it checks if the corresponding bit is set (1). If it is set, it indicates that the mate core changed it, so it wanted to adjust the frequency. On this occasion, the core will change the frequency to the desired level. On the other hand, if the corresponding bit is not set (0), the core will set it up in order to inform the mate core the desire to change the frequency level. A different thing happens when some core needs the frequency to be high. In this case, the core immediately rises it and unsets both bits (0).

Note that this module allows power management in applications that uses the whole SCC (48 cores) avoiding non-synchronization penalties.

We had been thinking on several more techniques in order to implement this synchronization like using MPI or UPC gasnet layer. However, although this techniques would guarantee portability (a property that, by the way, we don't need, as this module is only useful on SCC), this would limit performance and would be restricted to UPC PGAS implementation.

6.7.3 DVFS intra-vDom synchronization

To coordinate all eight cores in a voltage domain, we have used the same mechanism as in inter-tile communication case: the unused bit in the L2CFG register.

In this case, however, the algorithm is more complex, and is centralized. There is a core called the *controller*, and the other seven are considered *clients*. The controller core is the responsible of running the synchronization algorithm (in a separate thread, for performance purposes) and, when it detects that the different cores in a power domain wants to adjust the voltage, it does it, after adjusting the frequency level accordingly to the table 2.2 shown previously. To maintain the voltage domain safe and stable, the controller must decrease the frequency before decreasing the voltage, and must increase the frequency after increasing the voltage, depending on whether the adjust is aimed to rise or lower the power.

The algorithm determines on run-time which core is the controller and which the clients, in a fixed way.

The client's L2CFG bit is only modified by itself, and it indicates, all the time, it's state: waiting for low power (1), or waiting for high power (0). When a client modifies it's own bit, it sets the controller's bit too. This action lets the controller know that something happened in the network. On the other hand note that, the notification of the desire of the own controller is trivial and can be implemented by using a global variable or an awakening call. When the controller detects that its own L2CFG bit is set (1), it checks the client's bits in order to know if all are set (in this case, the controller may drop the voltage) or some are unset (in this case, the controller may rise the voltage). Note that this construct minimize the network traffic, as polling packets are only sent when a core's state changes.

Note that the synchronization algorithm only allows to submit the desire to change only at two voltage levels (high or low). However, we can also tune it in order to implement a third level: medium voltage. This third level is indicated by the clients by setting (1) their bits, exactly the same way as if they wanted low frequency/voltage. In case the controller detects all bits are set (1), identifies which is the globally desired level (low or medium) by looking at its own state, which, as it is local, it can be more expressive and encode one of the three states. The controller then adjust the level of what its own 'state' indicates low or medium frequency and voltage.

At this point, when the controller determines that the voltage (and, therefore, frequency) must be adjusted, it will send a command to the Voltage Regulator Controller, as explained in the SCC's Power management capabilities section (see 2.8).

For testing purposes, the power manager can be configured to change only the frequency, but synchronizing a whole power domain.

6.8 Configuration module

The configuration module loads a configuration file which contains parameters dedicated to tune the power manager components. It can be used to accommodate the power manager to the particular characteristics of each application. Specifically, the user can choose:

- Which method to filter the requests. For the moment, only three options are available:
  - Fixed time threshold.
  - Variable time threshold.
  - Mixed solution.
- Which method the system should apply to reduce the energy?
  - Direct DFS.
  - DFS with intratile synchronization.
  - DVFS (or DFS, for testing) with voltage domain synchronization.
  - Linux cpu-freq interface.

6.9 Statistics

In order to study the behavior of the system, the different modules of this experimental power manager collects statistical data. This data can be treated or plotted later, when the execution is done.

Chapter 7

Experimental results

In this section, we will try to show how the policies described above can be parametrized correctly in order to achieve energy savings.

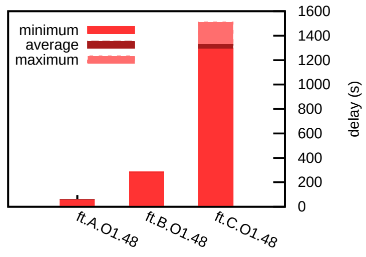
As mentioned, the experimental environment have been done on a SCC prototype given by Intel Labs, configured with the Berkeley UPC runtime and RCKMPI.

7.1 Test suite

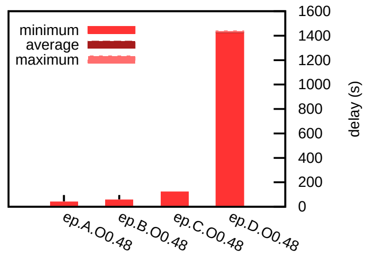
On the one hand, we used the previously studied NAS benchmarks in order to study the performance of our approach. In figure 7.1 we can see the execution time and the consumed energy of several classes of the studied NAS: FT, EP and MG. In our tests we are using the highest working class of each of them: FT class C, EP class D and MG class C.

On the other hand, we used the matmul synthetic application in order to show the effect in the energy savings of the application's imbalance.

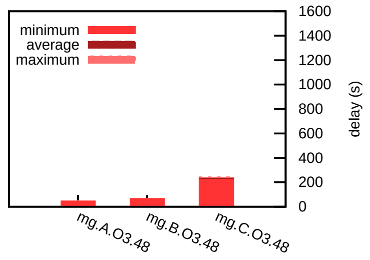
Finally, we analyzed how application-layer hint's can help improve the energy savings policies.



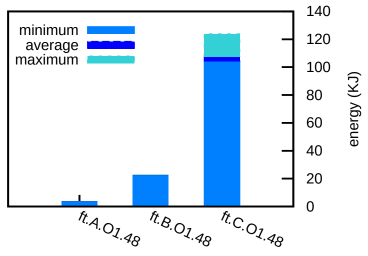
(a) FT delay



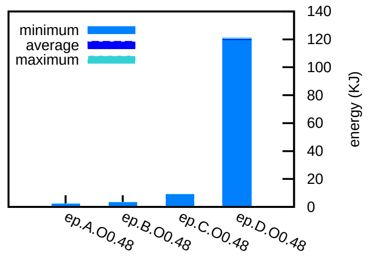
(b) EP delay



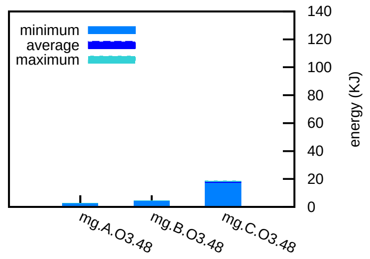
(c) MG delay



(d) FT energy



(e) EP energy



(f) MG energy

Figure 7.1: Measured execution time and energy of several benchmarks in the NAS test suite.

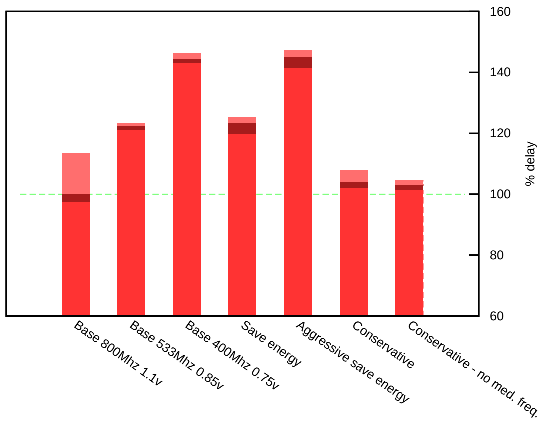
7.2 FT

Figure 7.2 shows several tests aimed to compare the application layer-hints with the practical maximum energy savings. Note that each column shows, with different color tonality, the minimum, the average and the maximum of several samples with the same parameters. In the first three columns of figure 7.2 we can see the results of a several base tests, performed, manually, at different frequency and voltage levels.

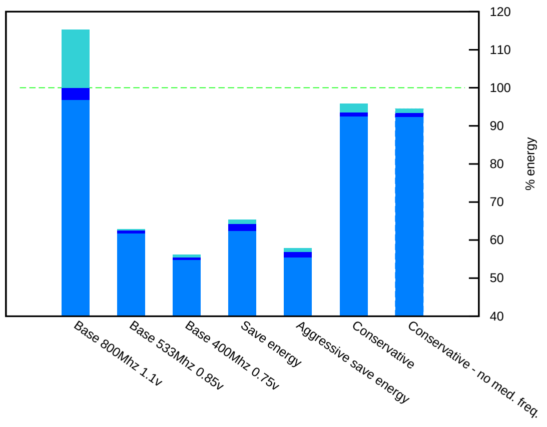
The first test represents the base test, which was executed at maximum SCC performance (800Mhx-1.1V), and, therefore, needing maximum power resources (this test corresponds to a use of the PM_PERFORMANCE application-layer hint). The next two shows the time and energy results that corresponds to the same execution, but reducing, previously, the performance of the platform (SCC) to 533MHz-0.85V and 400MHz-0.75V, respectively.

The following two tests were performed by adding only the PM_SAVE_ENERGY in the beginning of the FT's main function (or PM_AGGRESSIVE_SAVE_ENERGY, respectively). As we can see, there is a correlation between tests base-533MHz and Save energy, and between test base-400MHz and Aggressive save energy.

The last two ones follows a similar idea, but with PM_CONSERVATIVE hint instead of save energy (this is the default behavior, if no hint is specified, while the power manager is enabled). The conservative test is a good example of a little energy savings with almost-no time penalty. These tests are done using DVFS technique enabling only wait and memget requests and filtering these requests with thresholds policy (20-20 for wait and 300-1000 for memget). The main difference in both tests are that in the second one the medium voltage level is disabled. Although this difference is negligible, it have got an explanation: the main goal of medium frequency was to avoid unclusterized call lengths (i.e. applications that have got calls with almost all length) and,



(a) Delay penalty.



(b) Energy savings.

Figure 7.2: Delay penalty and energy savings in several FT class C executions with hints.

as we observed in the profiling phase, FT's calls can be perfectly clusterized and, therefore, the medium frequency is not convenient in this case.

In table 7.1 we can see the same results previously analyzed in figure 7.2. In this table, however, the results are numerically detailed, showing more clearly only the detailed average results (in absolute values and in percentage). Note that the application reduced the energy to a 55 % (with a time degradation of 145 %) when the user asks for an aggressive approach. A moderated case can be configured with the non-aggressive save energy hint, which decreases the energy to 65 % but with a much smaller time penalty (120 %).

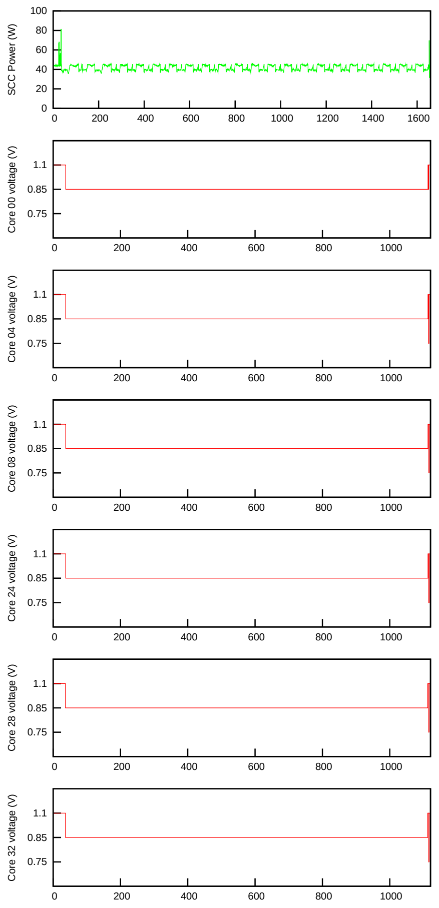
Test description	**Delay s (%)**	**Energy J (%)**
Base 800Mhz 1.1v	1331.53 (**100.0**)	107426.70 (**100.0**)
Base 533Mhz 0.85v	1629.54 (**122.3**)	67166.95 (**62.5**)
Base 400Mhz 0.75v	1925.59 (**144.6**)	59579.00 (**55.4**)
Save energy	1640.67 (**123.2**)	69116.97 (**64.3**)
Aggressive save energy	1932.16 (**145.1**)	61111.82 (**56.8**)
Conservative	1385.39 (**104.0**)	100567.05 (**93.6**)
Conservative - no med. freq.	1372.91 (**103.1**)	100339.04 (**93.4**)

Table 7.1: Delay penalty and energy savings in several FT class C executions with hints. Average of 50 samples.

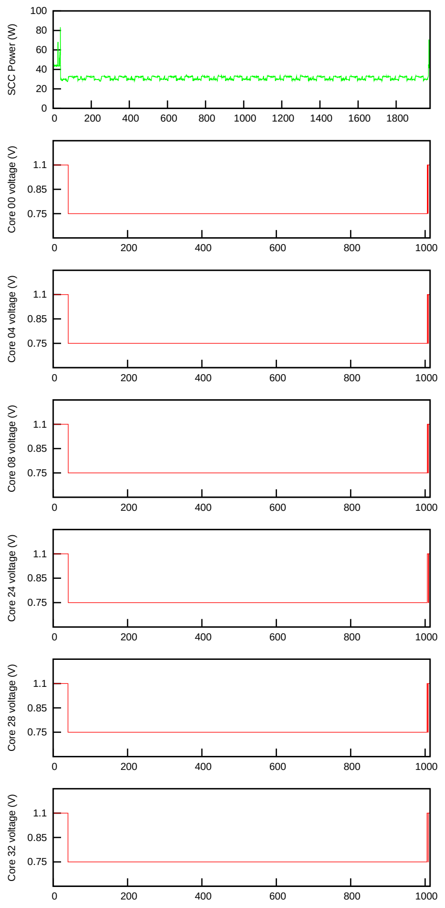
As stated before, the last two executions have been done with the most important hint: conservative. It is the most important in the automation sense, as in this state, the power manager is the responsible of decision making, instead of the programmer. The power manager will try to decrease the energy with no- or little- performance penalty by adjusting the frequency and/or the voltage to the better level for each situation. With FT, this technique allows a reduction of almost 7 % in energy, with as little as 3 % time penalty.

Note that although this reduction seems useless, the aim of this technique is to automatically reduce the energy with low time penalty. Furthermore, we should take in mind that this benchmark is very well-balanced and, therefore, it's energy saving opportunities decrease, as stated in the profiling phase.

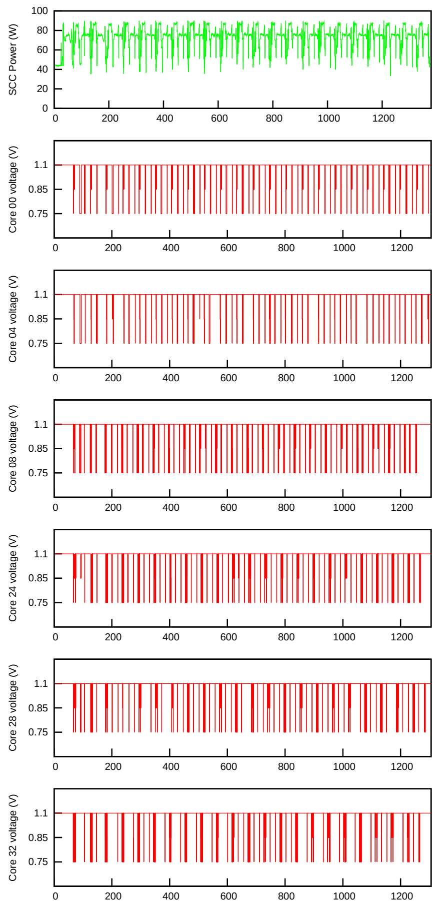
Note that the hints in the aforementioned tests have been included in the beginning of the application and non changed during the execution in order to allow a logical comparison needed in this experimentation part. Remember, however, that the hint's main goal is to be flexible, robust and highly powerful for the user, allowing the user to indicate which region of a program is more suitable to save energy. A better use case will be described later, when talking about Sobel.



(a) SAVE_ENERGY hint.



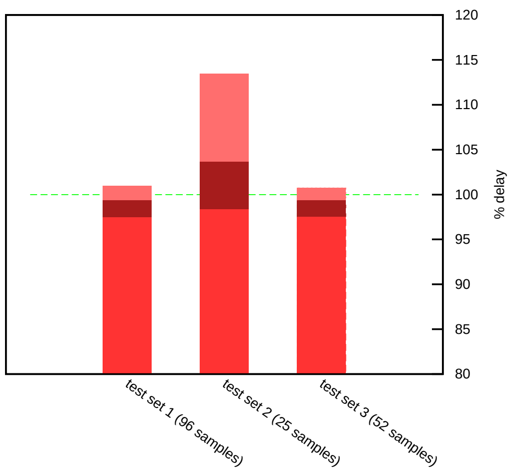
(b) AGGRESSIVE_SAVE_ENERGY hint.



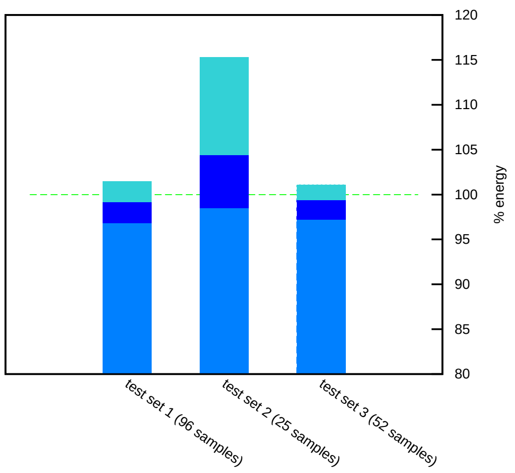
(c) CONSERVATIVE hint.

Figure 7.3: Measured power and PM decisions in NAS FT class C benchmark. Controller core decisions are shown in voltage levels. Note that this levels have a direct correspondence with a frequency: 1.1V-800MHz, 0.85V-533MHz and 0.75V-400MHz.

In order to show a fine-grained experimental result, we have collected some runtime data of the application-layer FT hint's tests. Specifically, the measured real power and the power management controller's decisions through a complete execution have been collected, and are shown in figure 7.3. Note that the save energy hint produces a decrease of the voltage and frequency levels to 0.85 V and 533 MHz and the aggressive version produces a larger decrease, achieving 0.75 V and 400 MHz. Conservative version, as we know, changes the frequency or voltage level only in the non-filtered waits and memgets. In the corresponding measured power plot we can see the real effect of this technique.



(a) Delay penalty.



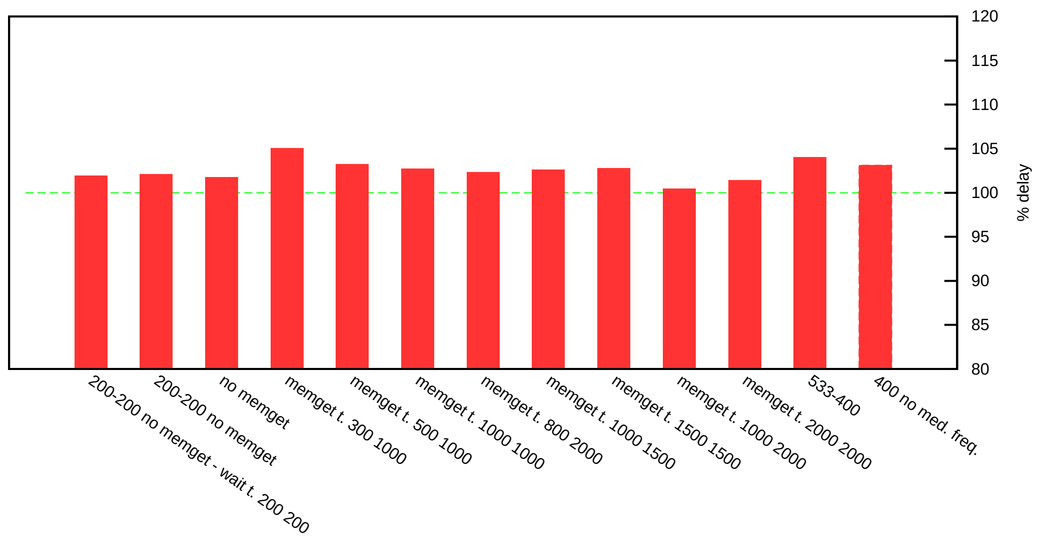
(b) Energy savings.

Figure 7.4: Delay penalty and energy savings in several base FT executions.

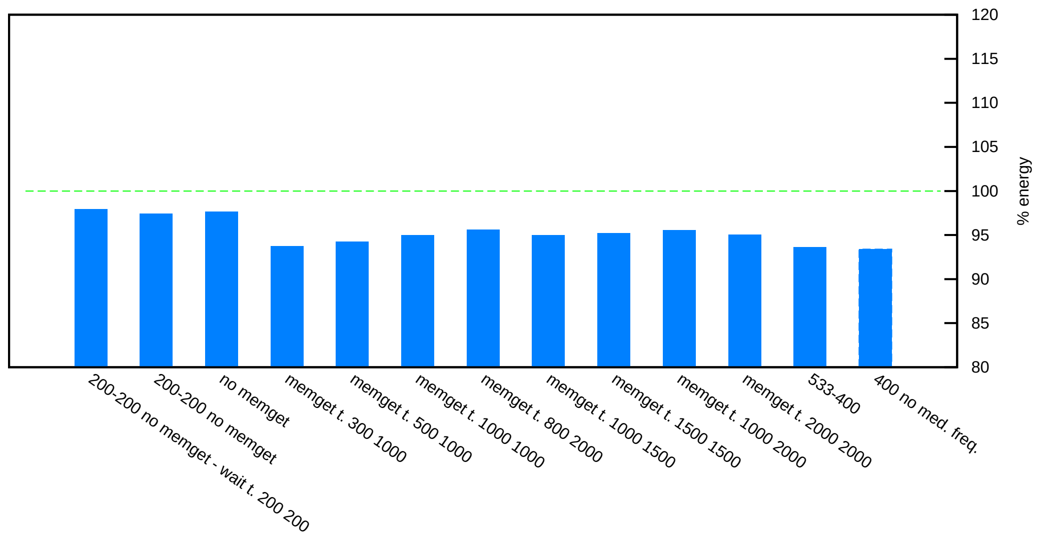
Figure 7.4 shows three executions of the same base test done in the same conditions, which achieved different variability. This shows that SCC platform exhibits a little instability in the execution time of several tests. Due to this instability, all tests have been performed several times, collecting lots of samples (for example, 173 in the case of base test) and showing the average of them, expecting it to be the most adjusted to the reality.

Until this point, we have described only the most representative tests regarding the cross-layer power management part of our research. We performed, however, a large test suite with great variety of configuration parameters and input values, which we are about to introduce.

Figure 7.5 shows several FT class C execution sets using DVFS technique and thresholds filtering policy. On the one hand, we can remark that the great majority of executions achieves a energy reduction with little time penalty. On the other hand, note that disabling memget operation allows better stability: lower energy savings, but minor time penalty. Enabling both memget and wait operation filtering, and tuning the threshold to 1000-2000, we can obtain the best time penalty tests: only 0.4 % of delay penalty. However, thresholds of 300-1000 offers the best energy savings in respect to the base execution: 93.75 % with a 400 MHz low and medium frequencies, 93.61 % with 533MHz and 400MHz frequencies and 93.40 % with medium frequency disabled and lower frequency set to 400 MHz.



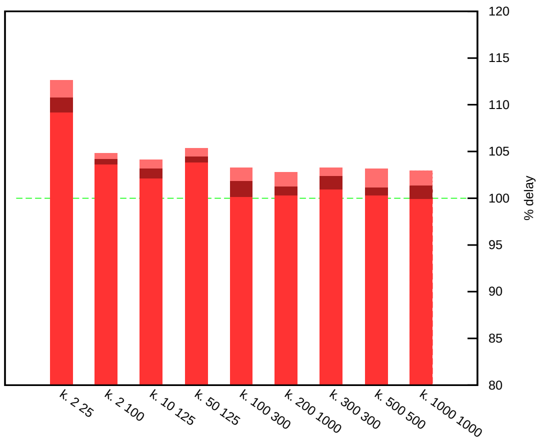
(a) Delay penalty.



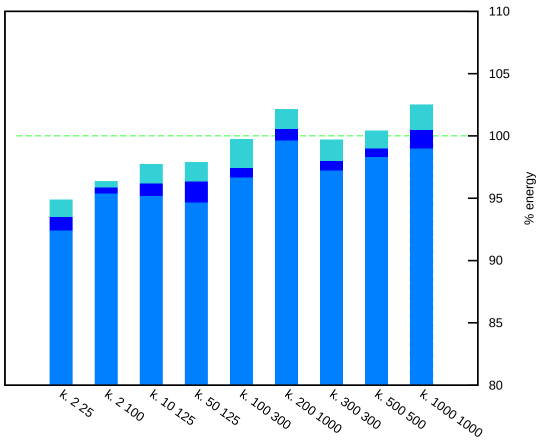
(b) Energy savings.

Figure 7.5: Delay penalty and energy savings in several FT class C executions (DVFS and thresholds filtering policy). By default, memget and wait operations are enabled, medium and low frequency both at 400MHz and wait threshold 20-20. Changes to this default values are indicated in the X-axis of the figure.

Continuing with DVFS technique, at figure 7.6 we can see some results obtained with the moving average filtering policy. The main conclusion is: the the lower the filtering parameter is, the greater the energy savings is and the more time penalty requires. An experimentally determined balance point is the corresponding of values 100-300 and 300-300, which corresponds to the 97 % of energy and 101 % of time and 98 % of energy and 102 % of time, respectively.



(a) Delay penalty.

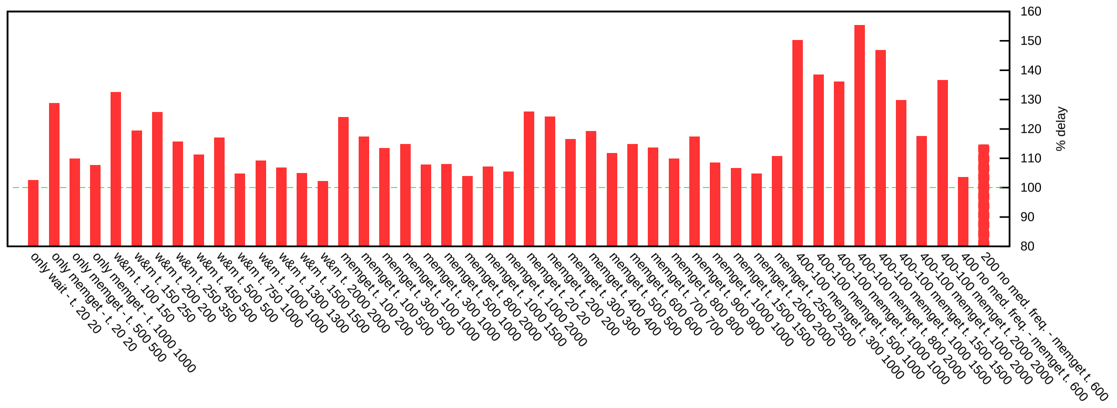


(b) Energy savings.

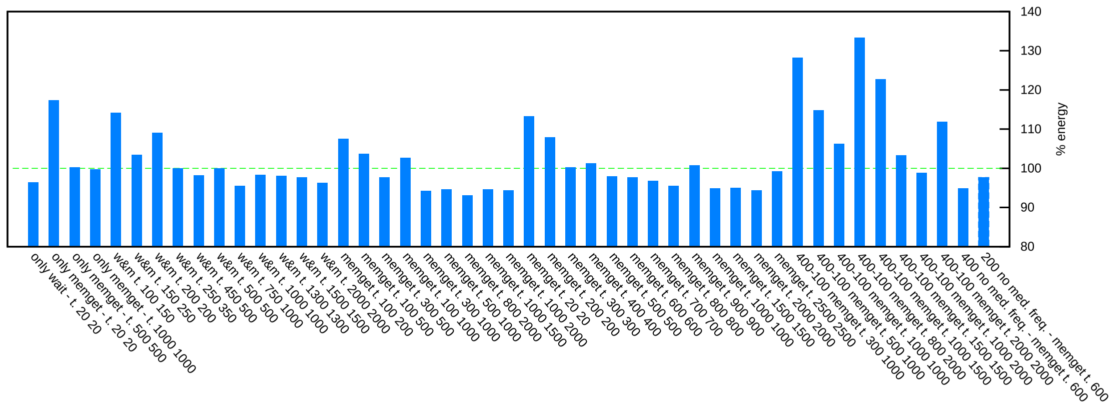
Figure 7.6: Delay penalty and energy savings in several FT executions with DVFS and moving average filtering policy. Other default values are the same as in figure 7.5.

Although DFS technique offers less energy reduction in the same amount of time (as described in the SCC introductory section), we will show it's results using intra-tile communication.

Figure 7.7 shows a great variety of experiments that obtained very different range of results in time penalty and energy savings. Note that there are several tests with big time penalties, achieving 110-115 % and up to 150 %. On the other hand, fine-tunning the input parameters having in mind the results of the profiling phase, we can achieve far better results. For example, disabling memget we achieve 96,4 % of energy with a 2.5 % of time penalty; enabling it and configuring the threshold to 800-2000 we obtain an average of 93.06 % of energy and a 3.9 time penalty. The best test case using this configuration, however, achieved 91% of energy savings with no time penalty.



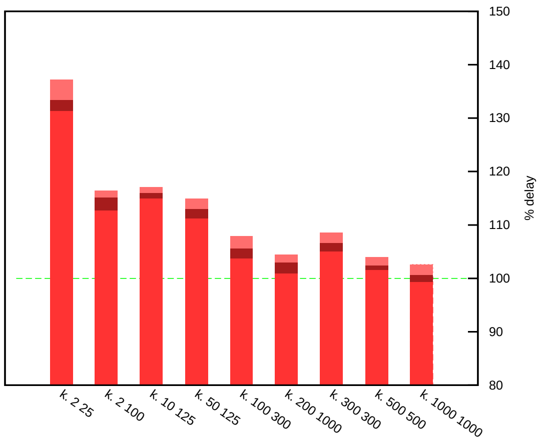
(a) Delay penalty.



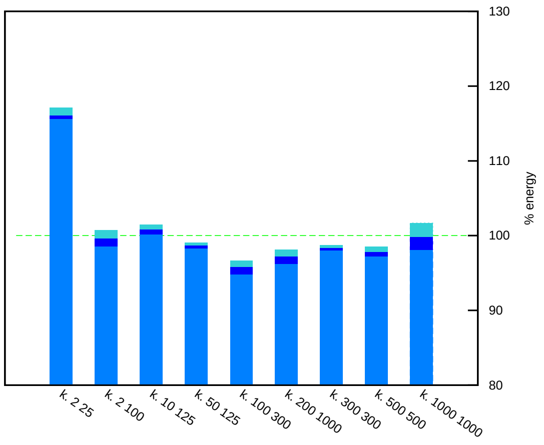
(b) Energy savings.

Figure 7.7: Delay penalty and energy savings in several FT class C executions (DFSi and thresholds filtering policy). By default, memget and wait operations are enabled, medium and low frequency at 400MHz and 200 MHz and wait threshold 20-20. Changes to this default values are indicated in the X-axis of the figure. Note that the results shown are the average of 10 samples.

Continuing with the same technique, DFSi, we have performed tests with the moving average filtering policy, which can be seen in figure 7.8. If we are interested in energy savings, in this case, we can apply a filtering parameter of 100-300 achieving a 95.8 % of energy, while if we are interested in no-time penalty we can filter requests with a 1000-1000 parameter achieving, however, minor energy savings.



(a) Delay penalty.

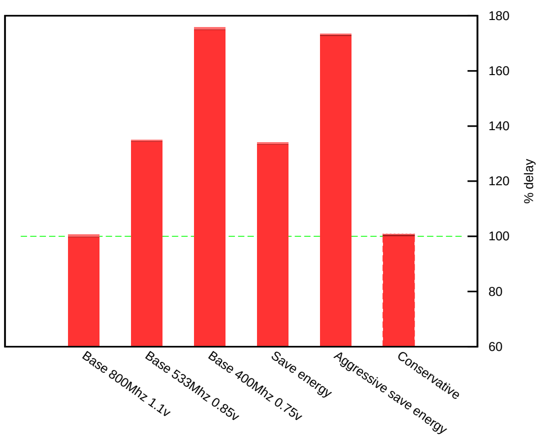


(b) Energy savings.

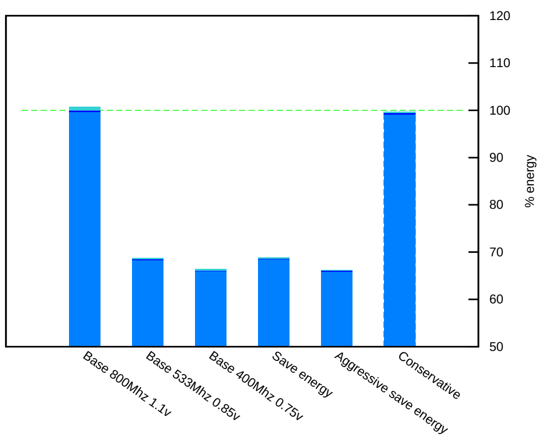
Figure 7.8: Delay penalty and energy savings in several FT executions with DFSi and moving average filtering policy. Other default values are the same as in figure 7.5.

7.3 EP and MG

Figures 7.9 and 7.10 shows the results obtained with NAS EP class D and NAS MG class C. The results show that automatic runtime power management (i.e., `PM_CONSERVATIVE` policy) does not provide significant energy savings. However, application level policies allows the programmer to manage energy/performance tradeoff within a wide range of energy saving and time penalty.

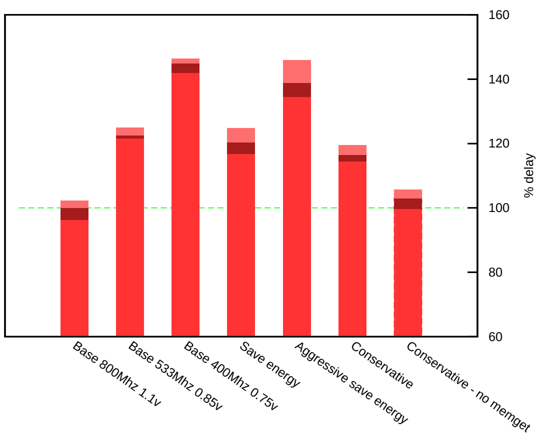


(a) Delay penalty.

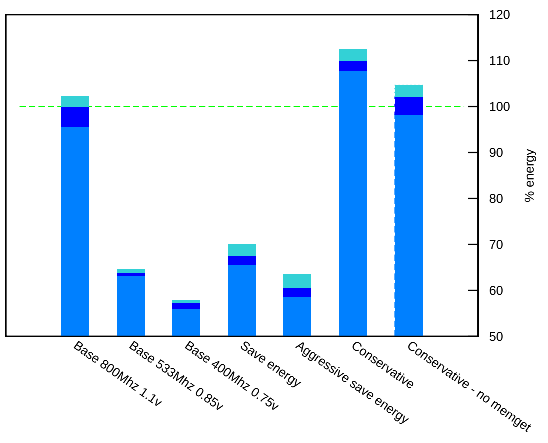


(b) Energy savings.

Figure 7.9: Delay penalty and energy savings in several EP class D executions with hints.



(a) Delay penalty.



(b) Energy savings.

Figure 7.10: Delay penalty and energy savings in several MG class C executions with hints.

7.4 Sobel

Sobel is an edge detection application, useful in computer vision, as we described in the profiling phase. As stated in the corresponding chapter, the most important PGAS call subject to energy savings is `wait` (see figure 5.15), and there are two well distinguished phases: shared matrix initialization and the actual edge detection (see figure 5.16).

In order to do an in-depth analysis of the Sobel behavior we should study listing 7.1. This listing shows the pseudocode abstracted from the actual Sobel UPC application, without any modification (this test is considered the *base test* in this dissertation), in which both phases have been marked. Note that the initialization phase is performed only by thread 0 (MYTHREAD==0), which is the responsible of assigning random values to each position of the shared matrix representing the image, while all the other cores wait for it to finish in the first upc_barrier. About the execution phase, which is performed by all threads in parallel, it is composed of 100 iterations of the edge detection algorithm.

```
int main(void)
{
  // Initialization phase!

  if (MYTHREAD==0) {
    for(i=0; i<N; i++) {
      for (j=0; j<N; j++) {
        orig[i].r[j]=rand();
      }
    }
  }

  // Execution phase!

  upc_barrier;
  for (i=0; i<100; i++) {
    Sobel();
    upc_barrier;
  }
  upc_barrier;
}
```

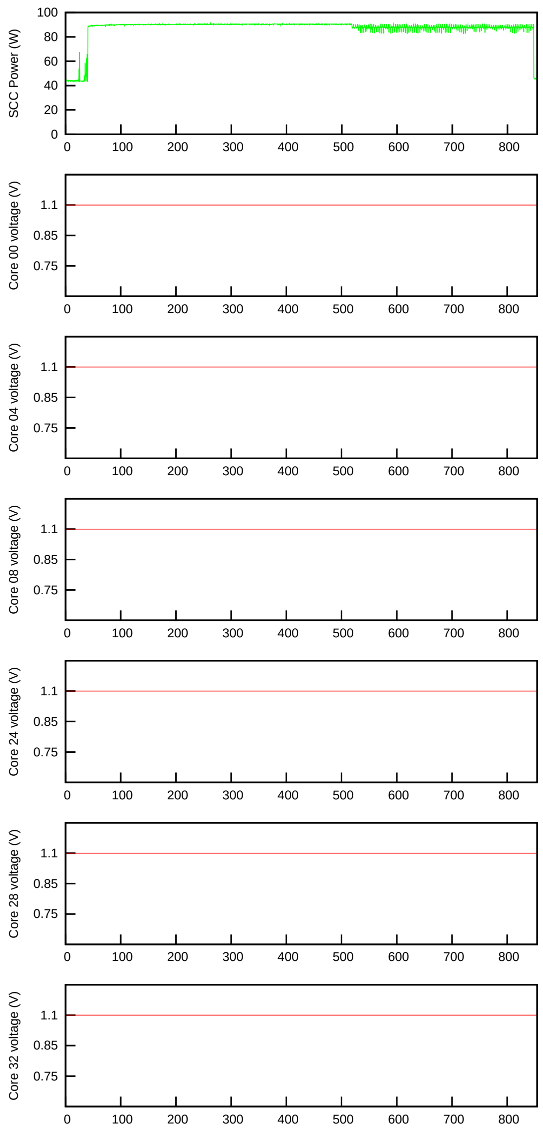
Listing 7.1: Sobel

```c
int main(void)
{
  PM_CONSERVATIVE();
  // PM_SAVE_ENERGY();
  // PM_AGGRESSIVE_SAVE_ENERGY();
  // PM_PERFORMANCE();

  // Initialization phase!

  if (MYTHREAD==0) {
    for(i=0; i<N; i++) {
      for (j=0; j<N; j++) {
        orig[i].r[j]=rand();
      }
    }
  }

  // Execution phase!

  upc_barrier;
  for (i=0; i<100; i++) {
    Sobel();
    upc_barrier;
  }
  upc_barrier;

  PM_CLOSE();
}
```

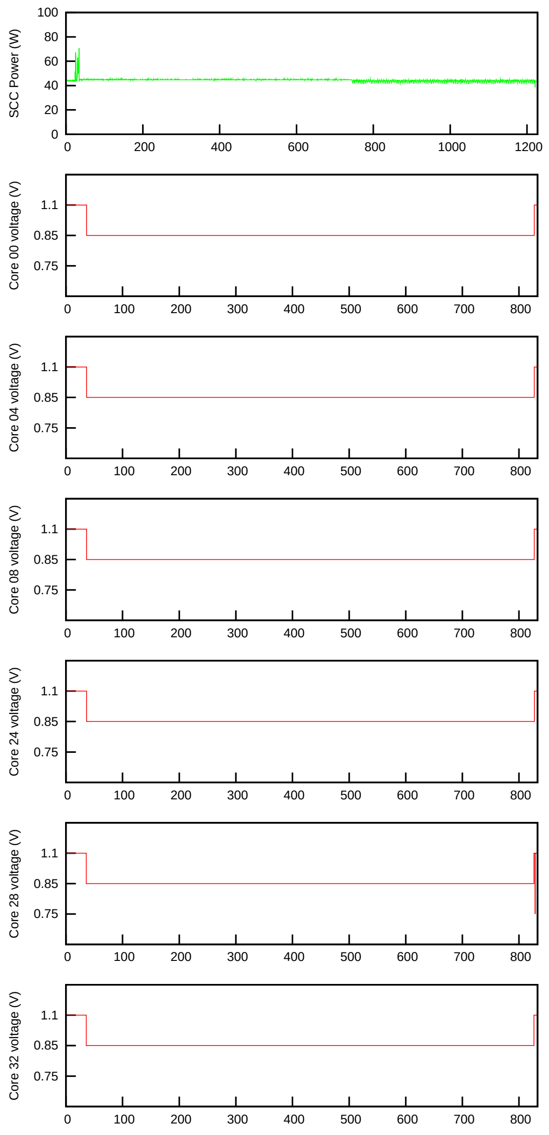
Listing 7.2: Sobel with hints

Let's analyze now the influence of the several existing hints, executed in the beginning of the program (remember that, although this is not useful in real applications, will help us determine the energy savings potential, as done in studied NAS benchmarks). The execution of the code showed in listing 7.2 is reflected in figures 7.11 and 7.12, depending on the hint used:

- Figure 7.11a shows the behavior of the performance hint (which can be compared to base execution, as it maintains 800 MHz and 1.1V during all the execution). In the associated measured power plot we can see that it maintains around 90W along the execution.
- Figure 7.11b is the equivalent, corresponding to the use of save energy hint, which uses 533MHz and 0.85v. With these values, the power plot measures approximately 45W along the execution.
- Figure 7.12a corresponds to the aggressive save energy hint, which associates 400MHz and 0.75V and decreases power consumption to 35W, approximately.
- Finally, the usage of the conservative hint is shown in figure 7.12b. In this case, the power manager adjusts the power levels according to every need.

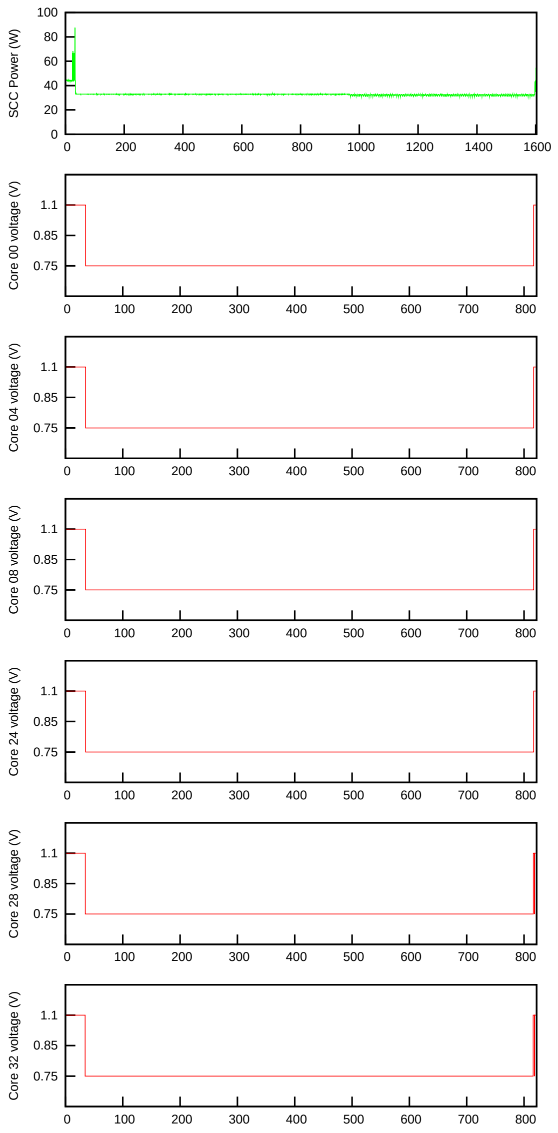


(a) Sobel, with PERFORMANCE hint.

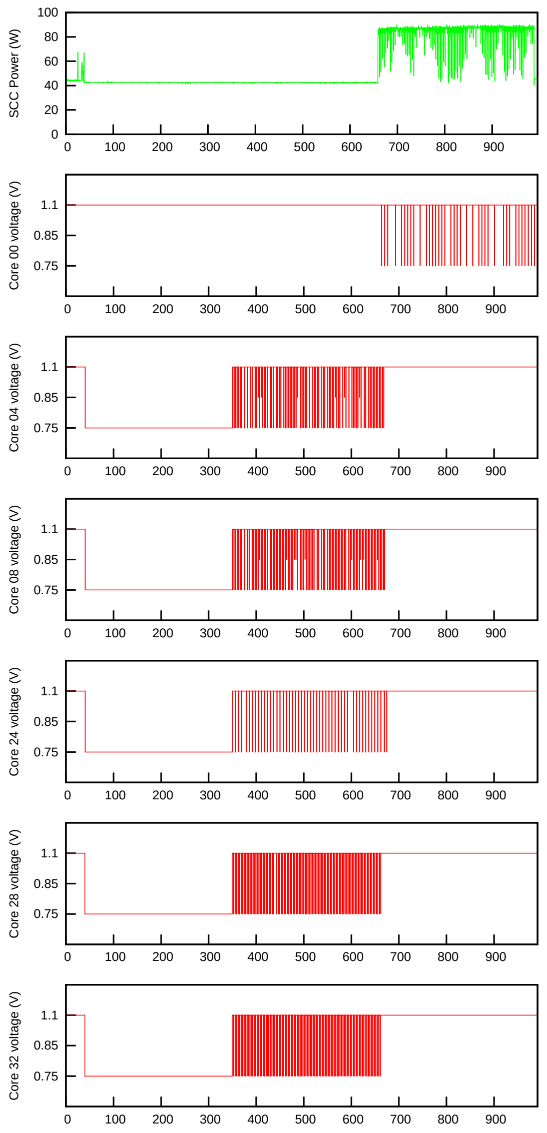


(b) Sobel, with SAVE_ENERGY hint

Figure 7.11: Measured power and PM decisions in Sobel application, part I.

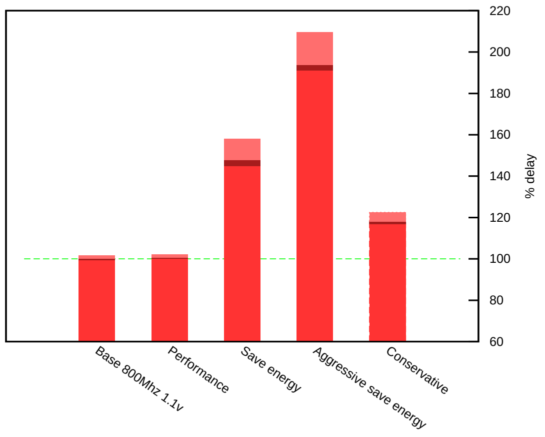


(a) Sobel, with AGGRESSIVE_SAVE_ENERGY hint

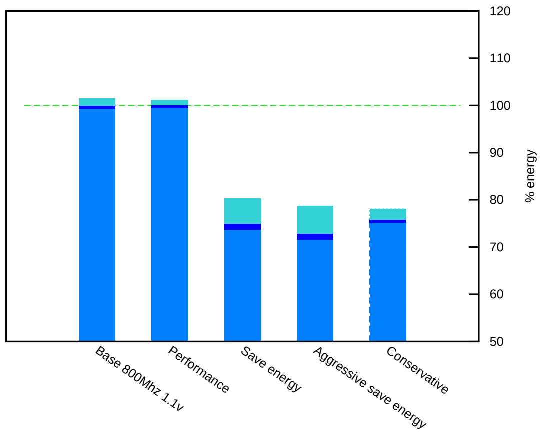


(b) Sobel, with CONSERVATIVE hint

Figure 7.12: Measured power and PM decisions in Sobel application, part II.



(a) Delay penalty.



(b) Energy savings.

Figure 7.13: Delay penalty and energy savings in several Sobel executions with hints.

On the other hand, this results are summarized in figure 7.13 and detailed in table 7.2, which shows the energy savings and the time penalty compared to base test. Note that performance test have got the same results, due to the same resource allocation. With aggressive and non-aggressive save energy hint we reduce the energy to 72 and 75 %, respectively, and increases the time up to 193 and 147 %. In this case, the non-aggressive version is much better, as it's performance footprint is far lower. The best policy, however, is the automatically determined, i.e. the conservative hint. It saves 25% of the energy, but with a non-desired 18 % of time penalty. The next step shall be the analysis of this problem.

Test description	**Delay s (%)**	**Energy J (%)**
Base 800Mhz 1.1v	839.91 (**100.0**)	73740.44 (**100.0**)
Performance	846.04 (**100.7**)	73809.76 (**100.0**)
Save energy	1240.31 (**147.6**)	55306.68 (**75.0**)
Aggressive save energy	1627.46 (**193.7**)	53696.45 (**72.8**)
Conservative	991.73 (**118.0**)	55882.11 (**75.7**)

Table 7.2: Delay penalty and energy savings in several Sobel executions with hints. Average of 50 samples.

If we look at figure 7.12b, we will verify that the power manager behavior is correct: during init phase (from time t=40 to t=340) core 0 (which is the responsible of the initialization) is running in high frequency and voltage while the other cores are in lowest frequency and voltage. This, however, doesn't consider the fact that core 0 initializes each position by sending a write message to the core that maps that shared memory position. Due to the destination lower frequency, the message will take much more time to be treated, and hence the higher time penalty. To solve this situation, the user may use the application-layer hint tools.

In listing 7.3 we show a possible modification in the sobel's initialization phase that handles this situation and allows the future inclusion of hints, which let to a similar energy reduction. The alteration aim is to be aware of which thread is the destination of each init message, i.e. which is the thread mapping each shared matrix position.

```
int main(void)
{
  // Initialization phase!

  numLines=N/THREADS;
  for(k=0; k<THREADS; k++) {
    for(k2=0; k2<numLines; k2++) {
      i = (k*numLines)+k2;
      if(MYTHREAD==0) {
        for (j=0; j<N; j++) {
          orig[i].r[j]=rand();
        }
      }
    }
    upc_barrier;
  }

  // Execution phase!

  upc_barrier;
  for (i=0; i<100; i++) {
    Sobel();
    upc_barrier;
  }
  upc_barrier;
}
```

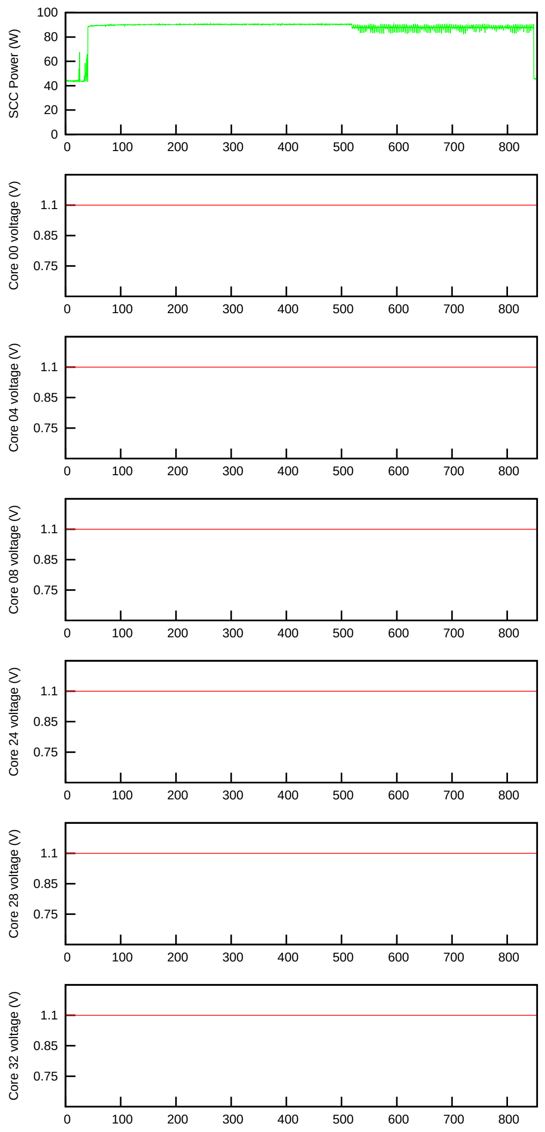
Listing 7.3: Modified sobel

```
int main(void)
{
  PM_CONSERVATIVE();

  // Initialization phase!

  numLines=N/THREADS;
  for(k=0; k<THREADS; k++) {

    if((MYTHREAD==k)||(MYTHREAD==0))
      PM_PERFORMANCE();
    else
      PM_CONSERVATIVE();
      // PM_SAVE_ENERGY();
      // PM_AGGRESSIVE_SAVE_ENERGY();

    for(k2=0; k2<numLines; k2++) {
      i = (k*numLines)+k2;
      if(MYTHREAD==0) {
        for (j=0; j<N; j++) {
          orig[i].r[j]=rand();
        }
      }
    }
    upc_barrier;
  }

  PM_CONSERVATIVE();

  // Execution phase!

  upc_barrier;
  for (i=0; i<100; i++) {
    Sobel();
    upc_barrier;
  }
  upc_barrier;

  PM_CLOSE();
}
```

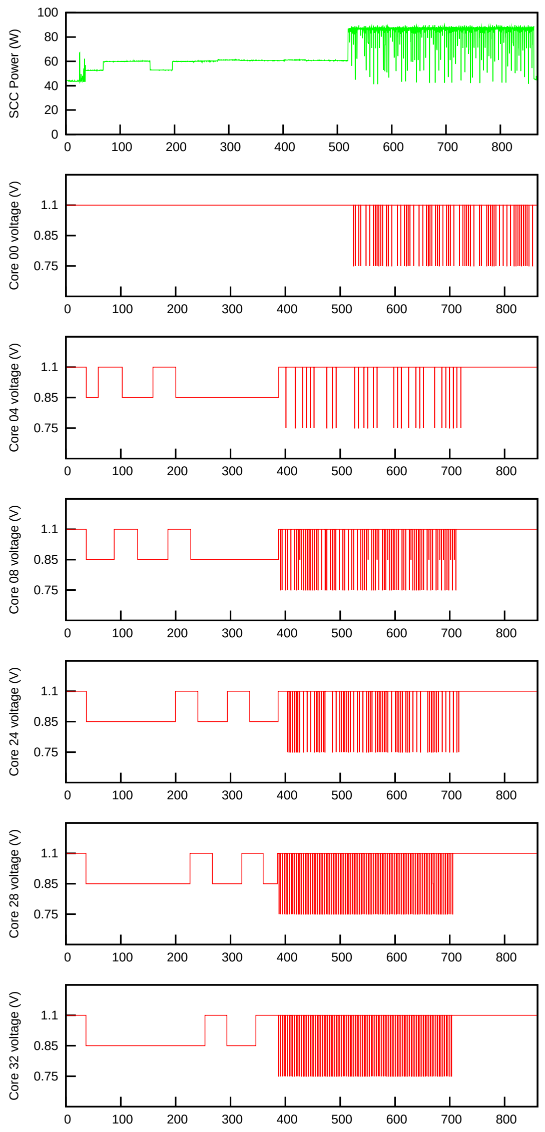
Listing 7.4: Modified sobel with hints

In listing 7.4 we can see how these hints have been included in the modified version, in order to allow a better power management. Note that, apart from an initial PM_CONSERVATIVE hint, inside the initialization loop we have included the convenient power manager hint: PM_PERFORMANCE for the worker thread and PM_CONSERVATIVE, PM_SAVE_ENERGY or PM_AGGRESSIVE_SAVE_ENERGY for the rest. The behavior of the several configurations is shown in figures 7.14 and 7.15:

- Figure 7.14a corresponds to the base execution and is exactly the same as non-modified version (see figure 7.11a). This means that this modification does not produce any time penalty.
- Figure 7.14b shows the behavior of the save energy hint. Unlike figure 7.11b (the corresponding to the unmodified sobel), we can see how during initialization phase core's frequency and voltage is adjusting dynamically, distinguishing when a core's shared memory is accessed. In the measured power plot this difference are noted too.

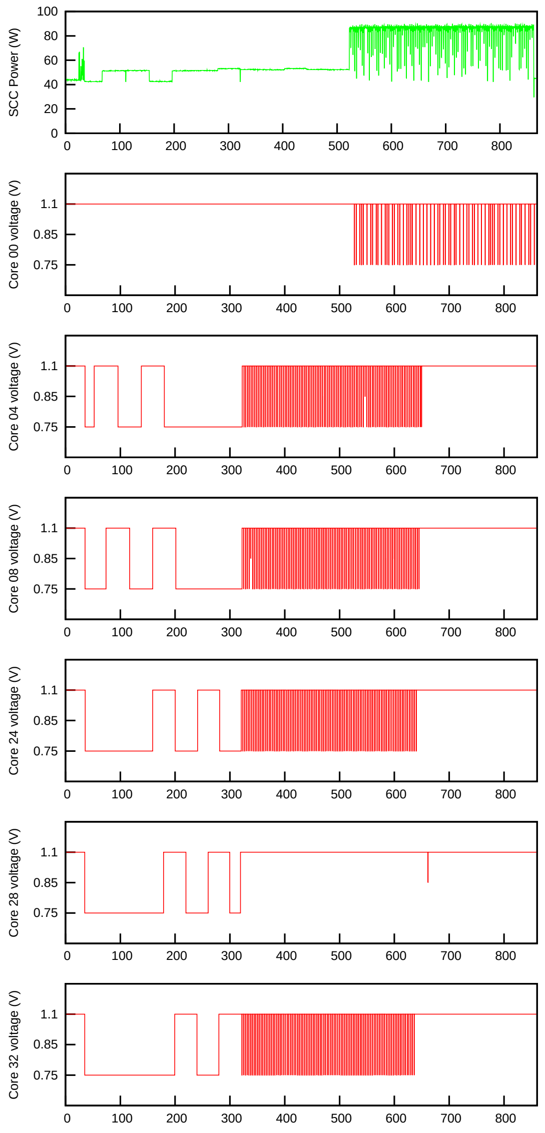


(a) Sobel, with PERFORMANCE hint.

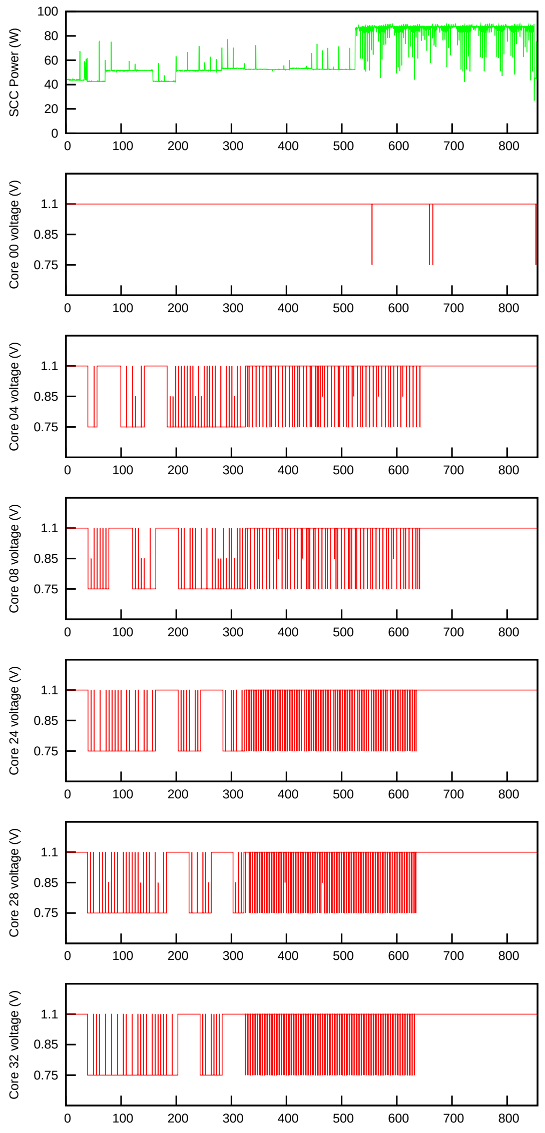


(b) Sobel, with SAVE_ENERGY hint

Figure 7.14: Measured power and PM decisions in modified-Sobel application, part I. This modification places the hint strategically in order to maximize energy savings without penalize the delay.



(a) Sobel, with AGGRESSIVE_SAVE_ENERGY hint



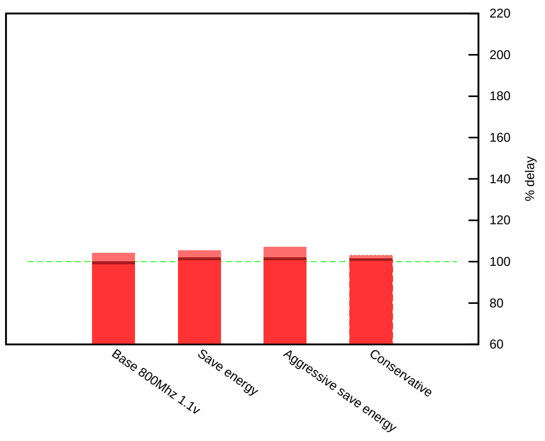
(b) Sobel, with CONSERVATIVE hint

Figure 7.15: Measured power and PM decisions in modified-Sobel application, part II

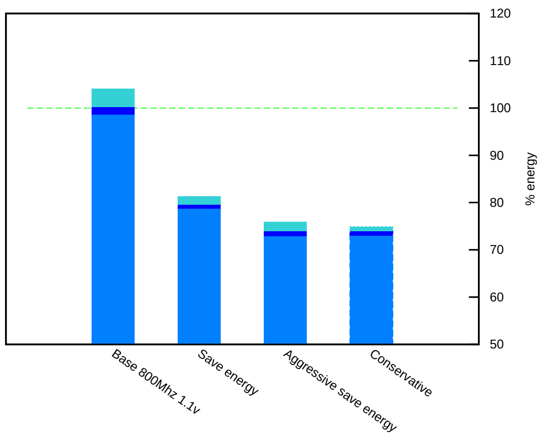
- The same happens with the aggressive hint, as seen in figure 7.15a. The lowest level in this case is, however, 400MHz and 0.75V.
- Finally, figure 7.15b shows the application of conservative hint, which automatically slows down the core frequency when allowed.

As this figures shows the behavior of the controller cores, the high performance adjustments are not only requested by their corresponding power manager, but by each core of the power domain.

As a result of the new hint-policy, in figure 7.16 and table 7.3 we can see that delay penalty have decreased a lot in respect of the unmodified Sobel, achieving at the same time, an even better energy savings.



(a) Delay penalty.



(b) Energy savings.

Figure 7.16: Delay penalty and energy savings in several modified-Sobel executions with hints.

Test description	**Delay s (%)**	**Energy J (%)**
Base 800Mhz 1.1v	842.21 (**100.0**)	73912.96 (**100.0**)
Save energy	857.83 (**101.8**)	58691.08 (**79.4**)
Aggressive save energy	858.87 (**101.9**)	54535.76 (**73.7**)
Conservative	854.99 (**101.5**)	54542.00 (**73.7**)

Table 7.3: Delay penalty and energy savings in several modified-Sobel executions with hints. Average of 50 samples.

Remember that with the default configuration power management techniques impact the execution time very significantly (e.g., in the best case, the energy savings are around 25% with time penalty of 18%). However, with the programming extensions the time delay is reduced drastically. With all the three evaluated policies the energy savings are 21–26% with less than 2% of time penalty.

7.5 Synthetic imbalanced benchmark: matmul

The main goal of the synthetic `matmul` application evaluation is showing the potential energy savings and delay penalty (upper bounds) of both runtime and application-aware power management techniques for different levels of load imbalance.

Figures 7.17 and 7.18 shows the relative energy savings and time penalty of `matmul`, ranging from 3% to 97% of load imbalance, for different power management strategies. Note that the figure shows the results of different runs. The results show that energy savings are proportional to the load imbalance: the more imbalanced the application, the more the energy savings. They also, show that our power management middleware can save up to 50% of energy with little time penalty, which is very significant since power requirements of the SCC are not very large (up to 125W).

In the following subsections we analyze this results step by step.

7.5.1 Power Management, Runtime layer

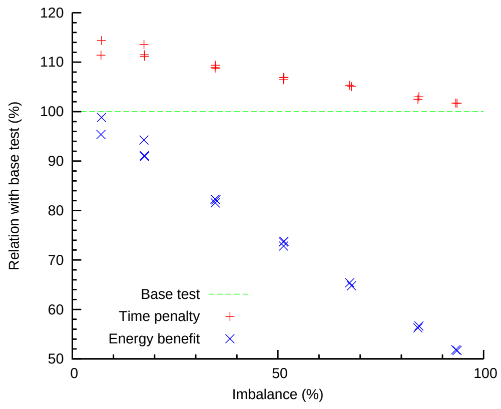
The results using only the runtime layer in the power manager (i.e. using default power manager behavior, which corresponds to PM_CONSERVATIVE hint) are described in the following subsections. Note that in all tests we have been using the same frequencies and voltages (**800 MHz, 1.1 V** for high power profile, and **400 MHz, 0.75 V** for low power profile), in order to help comparison between results.

Without request filtering

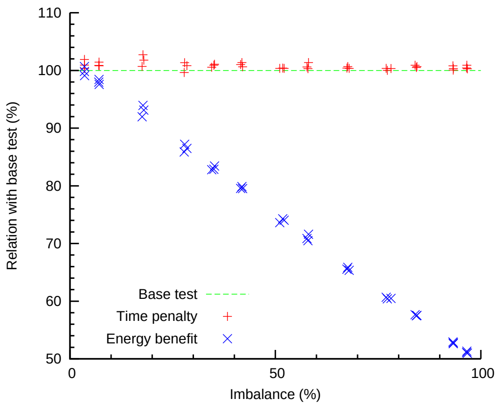
Let's take a look first at the simplest algorithm: no request filtering (i.e. no threshold, threshold=0). If we consider all the power management requests as suitable, and always apply voltage or frequency scaling, we will see a **big time penalty** (7-15 %). This penalty is due to the fact that most of the requests (if we look at logs, about 200-1200, depending on the specified imbalance) are very short, and only 36 (remember, each test iterate 36 times) are really long. The results are shown in figure 7.17a for DVFS, figure 7.17c for DFS with power domain synchronization and figure 7.17e for DFS with only tile synchronization.

With request filtering

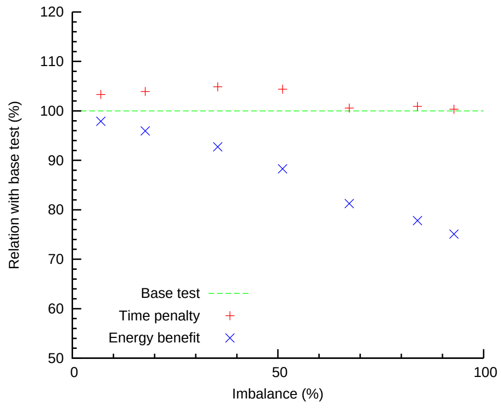
If we try to limit the requests to the 36 significant ones, and discard all others, we will improve the delay penalty and push it almost to the lowest value (100 %). To achieve this result we discarded all the requests that took less than 500 ms to finish (i.e. threshold=0.5s). This is shown in figure 7.17b for DVFS, figure 7.17d for DFS with power domain synchronization and figure 7.17f for DFS with only tile synchronization. Power behavior of executions with the described configuration can be seen in figures 7.19 for the whole execution and in figure 7.20 for a detailed part of only 4 iterations. Both figures shows the whole SCC power dissipation (top subplots) due to power management decisions, for three levels of imbalance: 3%, 51% and 97%.



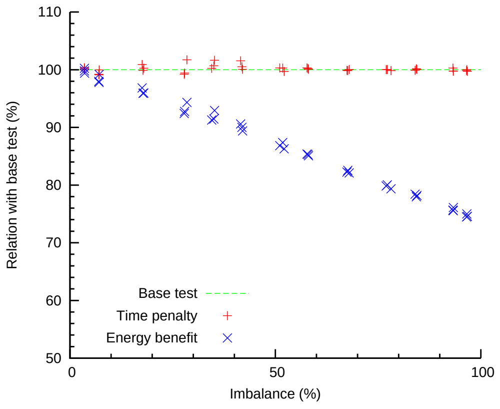
(a) PM no filter, DVFS



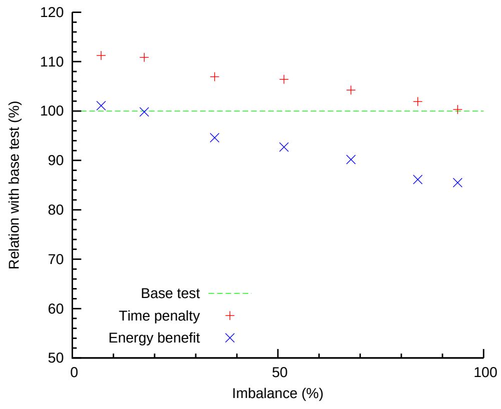
(b) PM filter, DVFS



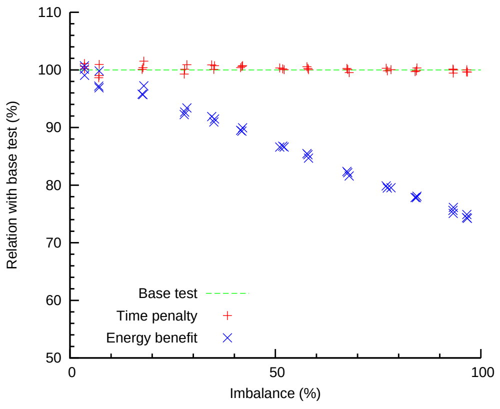
(c) PM no filter, DFS, vDom sync



(d) PM filter, DFS, vDom sync



(e) PM no filter, DFS, tile sync



(f) PM filter, DFS, tile sync

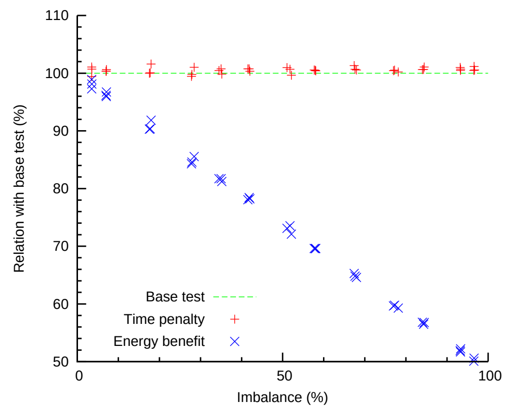
Figure 7.17: Energy savings and time penalty of matmul with different policies and load imbalance levels. Executed with default power manager behavior (corresponding to PM_CONSERVATIVE hint).

7.5.2 Power Management, Application Layer

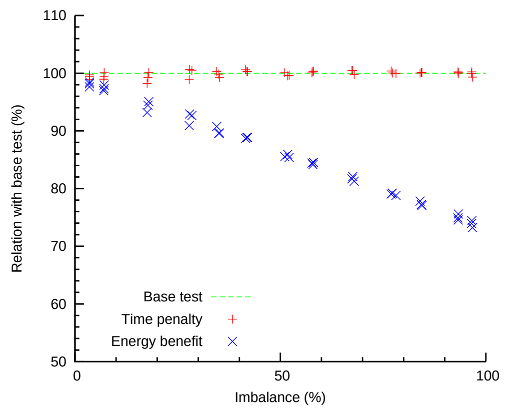
The `matmul` benchmark may be modified in order to allow the usage of application-layer policies. The modified pseudo-code is the following, in which we have strategically placed PM_AGGRESSIVE_SAVE_ENERGY hint in order to exploit imbalance:

```
->  PM_PERFORMANCE();

    for (i=0 ; i<N ; i++) {
      perform A matrix multiplications, distributed in 48 cores;

      if(MYTHREAD is in Voltage Domain X)
        perform B matrix multiplications;
      else
->      PM_AGGRESSIVE_SAVE_ENERGY();

      upc_barrier;
->    PM_PERFORMANCE();
    }
```

Figures 7.18a and 7.18b show the results obtained taking the power management decisions via programming extensions, using DVFS and only DFS, respectively. The time penalty is similar using both techniques; however, the energy savings with DVFS are higher (about twice) than the savings with only DFS. This is possible because the synthetic load imbalance is homogeneous among all the voltage domains.



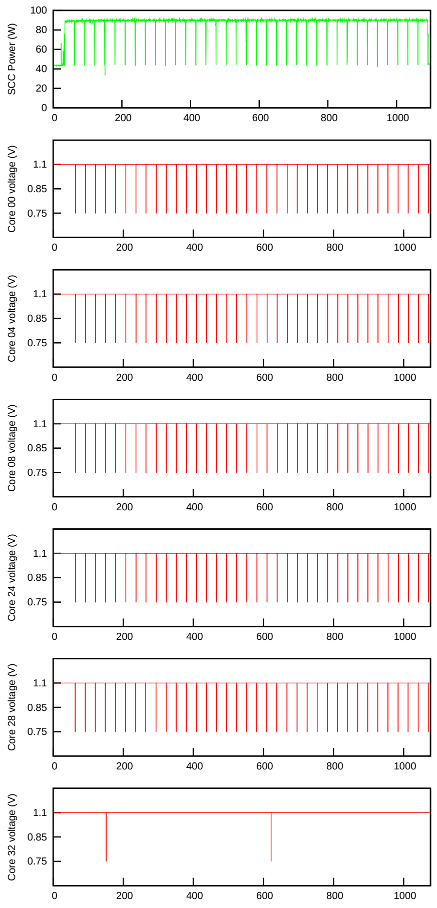
(a) PM hint, DVFS



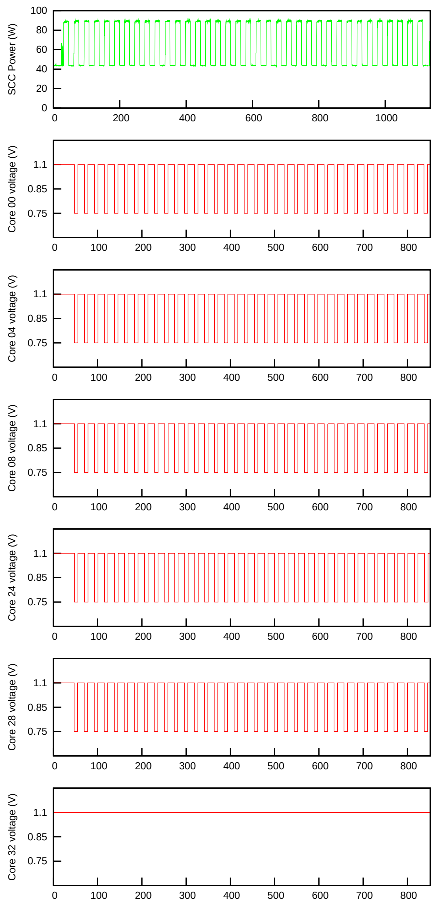
(b) PM hint, DFS, vDom sync

Figure 7.18: Energy savings and time penalty of matmul with different policies and load imbalance levels. Executed with strategically placed PM_AGGRESSIVE_SAVE_ENERGY hint.

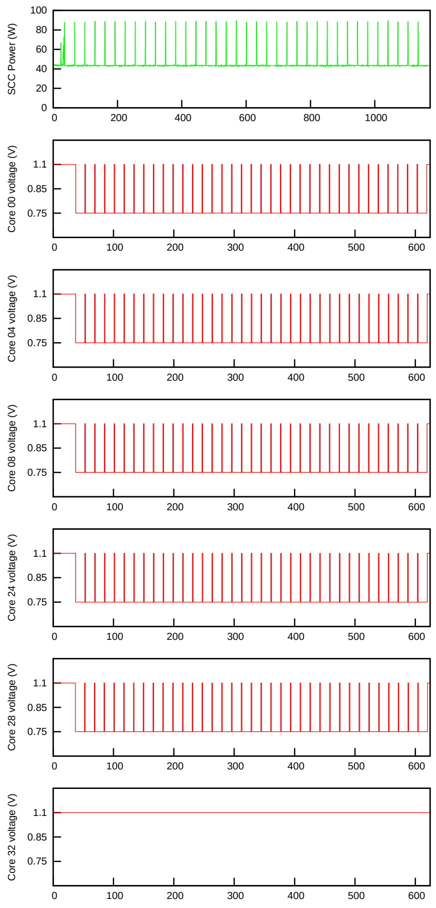
It is worth noting that results obtained using runtime power management and using programming extensions are very similar, which means that runtime power management works efficiently with the proposed filtering mechanisms.



(a) Matmul, with 3 % imbalance.

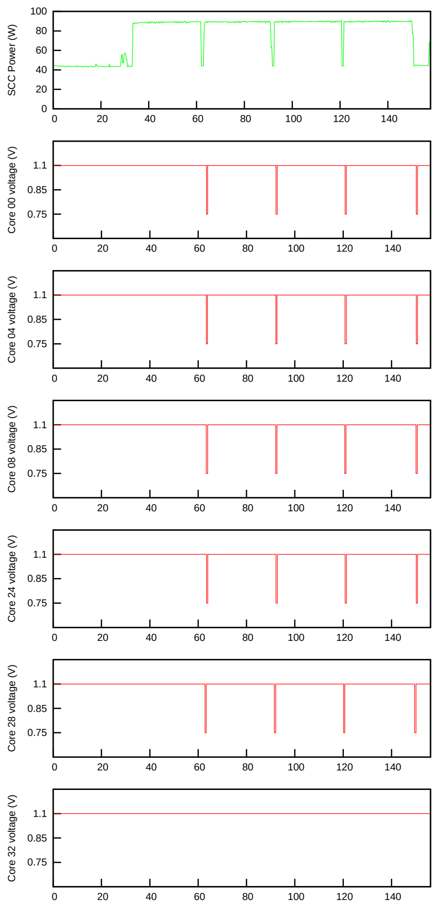


(b) Matmul, with 51 % imbalance.

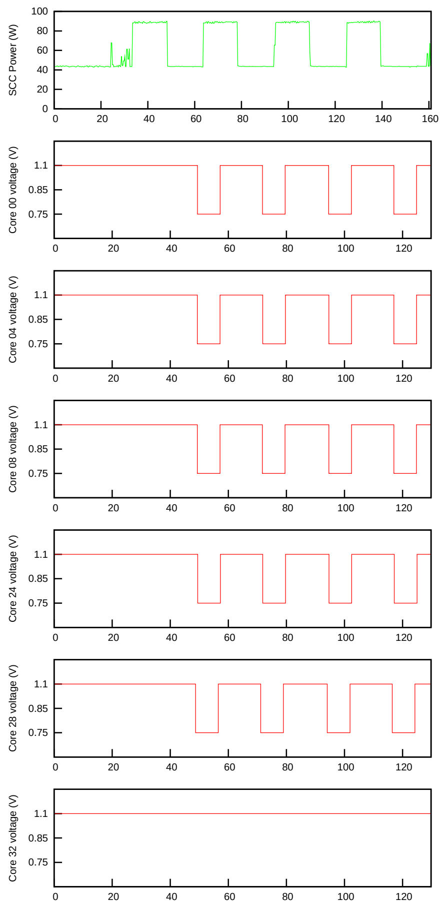


(c) Matmul, with 97 % imbalance.

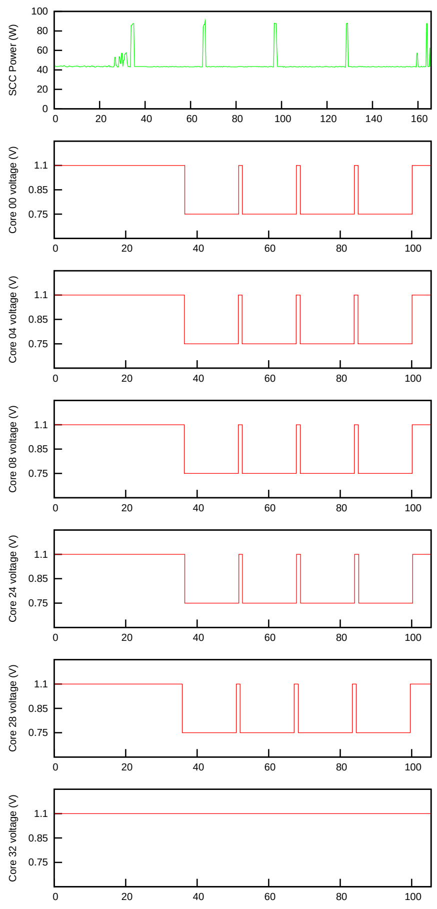
Figure 7.19: Measured power and PM decisions in Matmul application. Whole execution with 36 iterations.



(a) Matmul, with 3 % imbalance.



(b) Matmul, with 51 % imbalance.



(c) Matmul, with 97 % imbalance.

Figure 7.20: Measured power and PM decisions in Matmul application. Detailed reduced execution with only 4 iterations.

7.5.3 Algorithm impact

We can execute all the program logic (allowing initializations, external requests, filtering, core synchronization), but if we disable (thanks to a configuration parameter) the code that actually sends the voltage or frequency scaling command, we would see the impact of the algorithm, compared to the mean of the base executions.

As shown in figure 7.21, the impact is null (points, all surrounding the base test line, are not exactly 100 % due to noise).

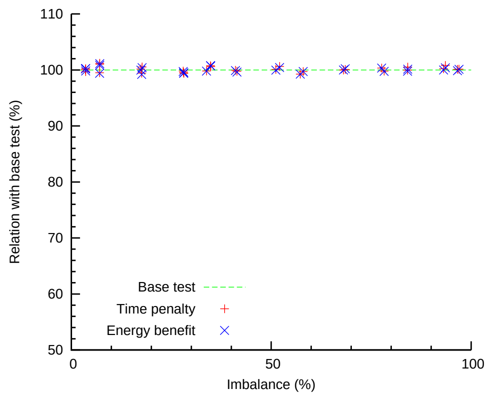


Figure 7.21: Impact of the power manager on delay and energy.

Chapter 8

Conclusion

In this work we have explored application-aware cross-layer power management for PGAS applications on many-core platforms, and presented the design, implementation and experimental evaluation of language-level extensions and a runtime middleware framework for application-aware cross-layer power management of UPC applications on the SCC platform.

Results obtained from our experiments conducted on the SCC platform showed that certain PGAS operations (e.g., `wait` and `memget`) need to be considered for power management. Performing power management during long calls of these operations can provide large energy savings, while during short calls can penalize both execution time and energy consumption. Furthermore, PGAS operation calls may cluster by length, which facilitates the identification of long and short calls. If they do not cluster, using an intermediate power mode results in better energy savings.

Power management during memory accesses provides surprisingly significant energy savings even though memory is shared among all the cores. We expect larger energy saving in distributed memory systems.

We also have observed that large energy savings can be obtained with imbalanced applications; however, blocking times in the barriers are not usually homogeneous over all cores.

Our experiments also show that in the case of applications where application-level power management does not provide any significant energy savings, a cross-layer approach can be used to achieve a wide range of energy and performance behaviors, and appropriate trade off can be selected.

Our evaluation also reveals several power management limitations of the SCC platform. For example, frequency scaling is fast (20 clock cycles) and only needs to synchronize 2 cores, but the energy savings are not very large; rather, voltage scaling provides larger energy savings, but the latency is longer (40 ms) and needs to synchronize 8 cores. An ideal power management would scale voltage and frequency per core; however, this level of granularity would require a large amount of the die, increasing at the same time the per-core power requirements [5].

8.1 Future work

Our ongoing and future work include:

- Exploring other PGAS models (e.g., Co-Array Fortran).
- Using the implemented power manager middleware in other hardware platforms like regular power-aware multi-core cluster or a pseudo-cluster with several SCC's via an external network. The framework has been designed to be modular, where the only platform-dependent module is the implementation of the power adjuster. A module that connects to *cpu-freq* (the standard Linux way to change frequency) have been already developed.
- Studying other possibilities that SCC can provide us, for example study which power management possibilities the cache memory can give us.
- Using per-core performance counters in order to detect application profiles and adjust frequency and/or voltage accordingly.
- Exploring distributed memory systems based on multi- and many-cores architectures. The latest might require extending runtime libraries (e.g., GASNET) to bypass memory accesses to external memory (i.e. exploit the hardware-implemented shared memory).
- Improving the core request filtering, by extending the prediction algorithm.
- Completing the cross-layer power management by implementing a *compiler layer* able to determine which are the main loops, more important barriers or simply adding some notifications to the runtime power manager.
- Adding an extra *prediction/post-compiler layer*: after the first compilation of a given user application, automatically execute some tests and collect data that may help power-management decision-making or request filtering, at runtime.

8.2 Personal opinion

As shown in related work, I discovered how competitive is the world of research. Maybe thanks to this competition, the technology progresses so fast.

Apart from improving my technical skills, this project allowed me the opportunity to explore amazing topics and being involved in an actual cutting-edge research project, working with an extremely experimental platform (Intel's SCC), and a novel language paradigm (PGAS) and runtime (Berkeley UPC). It also let me learning important insights of the research methodology during my internship at the Center for Autonomic Computing (CAC) at Rutgers University.

Although this document gives a complete view of the work done, the large amount of taken-decisions and generated-information across the research period make very hard to show the number of blind alleys that we explored before arriving to a dead end and all the engineering decisions. An important lesson from this research is that obtaining novel and significant results is hard but most important, that negative results can be meaningful and very useful.

To sum up, the overall balance is very good, and all the conclusions and solutions drawn compensate by far the invested effort.

I would like to thank my advisor Ivan Rodero and Prof. Manish Parashar for giving me the opportunity to carry out this research, their invaluable help and advising and for encouraging me to do further research in the future. I would like to thank my family and fiancee for their support and patience.

Bibliography

[1] Report to congress on server and data center energy efficiency. Technical report, U.S. Environmental Protection Agency, August 2007.

[2] Scc external architecture specification, revision 1.1, November 2010.

[3] Pedro Alonso, Manuel F. Dolz, Francisco D. Igual, Bryan Marker, Rafael Mayo, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Power-aware dense linear algebra implementations on multi-core and many-core processors. In *MARC Symposium*, pages 103–106, 2011.

[4] Saman Amarasinghe, Mary Hall, Richard Lethin, Keshav Pingali, et al. ASCR Programming Challenges for Exascale Computing. Technical report, U.S. DOE Office of Science (SC), July 2011.

[5] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM.

[6] Kirk W. Cameron, Rong Ge, and Xizhou Feng. High-performance, power-aware distributed computing for scientific applications. *Computer*, 38(11):40–47, 2005.

[7] K Chapman, A Hussein, and AL Hosking. X10 on the single-chip cloud computer. In *Proceedings of the X10 Workshop*, X10, 2011.

[8] Yiyu Chen, Amitayu Das, Wubi Qin, Anand Sivasubramaniam, Qian Wang, and Natarajan Gautam. Managing server energy and operational costs in hosting centers. In *ACM SIGMETRICS Intl. Conf. on Measurement and modeling of computer systems*, pages 303–314, 2005.

[9] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl. Evaluation and Improvements of Programming Models for the Intel SCC Many-core Processor. In *Proceedings of the International Conference on High Performance Computing and Simulation (HPCS2011), Workshop on New Algorithms and Programming Models for the Manycore Era (APMM)*, Istanbul, Turkey, July 2011.

[10] Carsten Clauss, Stefan Lankes, and Thomas Bemmerl. Performance tuning of scc-mpich by means of the proposed mpi-3.0 tool interface. In *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, EuroMPI'11, pages 318–320, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] James Dinan, Pavan Balaji, Ewing Lusk, P. Sadayappan, and Rajeev Thakur. Hybrid parallel programming with mpi and unified parallel c. In *Proceedings of the 7th ACM international conference on Computing frontiers*, CF '10, pages 177–186, New York, NY, USA, 2010. ACM.

[12] Tarek El-Ghazawi and Francois Cantonnet. Upc performance and potential: a npb experimental study. In *2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pages 1–26, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[13] Tarek A. El-Ghazawi and Sébastien Chauvin. Upc benchmarking issues. In *2001 International Conference on Parallel Processing*, ICPP '02, pages 365–372, Washington, DC, USA, 2001. IEEE Computer Society.

[14] Vincent W. Freeh and David K. Lowenthal. Using multiple energy gears in mpi programs on a power-scalable cluster. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 164–173, 2005.

[15] Vincent W. Freeh, Feng Pan, Nandini Kappiah, David K. Lowenthal, and Rob Springer. Exploring the energy-time tradeoff in mpi programs on a power-scalable cluster. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, page 4.1, 2005.

[16] Philipp Gschwandtner, Thomas Fahringer, and Radu Prodan. Performance analysis and benchmarking of the intel scc. In *CLUSTER*, pages 139–149, 2011.

[17] Chung hsing Hsu and Wu chun Feng. A feasibility analysis of power awareness in commodity-based high-performance clusters. In *IEEE International Conference on Cluster Computing*, pages 1–10, 2005.

[18] Chung-Hsing Hsu and Wu chun Feng. A power-aware run-time system for high-performance computing. In *ACM/IEEE Conference on High Performance Networking and Computing (SC'05)*, page 1, 2005.

[19] Parry Husbands, Costin Iancu, and Katherine Yelick. A performance analysis of the berkeley upc compiler. In *17th annual international conference on Supercomputing*, ICS '03, pages 63–73, New York, NY, USA, 2003. ACM.

[20] Nandini Kappiah, Vincent W. Freeh, and David K. Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 33, 2005.

[21] W. Kuchera and C. Wallace. The upc memory model: problems and prospects. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 16, april 2004.

[22] Min Yeol Lim, Vincent W. Freeh, and David K. Lowenthal. Adaptive, transparent frequency and voltage scaling of communication phases in mpi programs. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 107, 2006.

[23] Min Yeol Lim, Vincent W. Freeh, and David K. Lowenthal. Adaptive, transparent frequency and voltage scaling of communication phases in mpi programs. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[24] Chun Liu, Anand Sivasubramaniam, Mahmut Kandemir, and Mary Jane Irwin. Exploiting barriers to optimize power consumption of cmps. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers - Volume 01*, IPDPS '05, pages 5.1–, Washington, DC, USA, 2005. IEEE Computer Society.

[25] Yongpeng Liu and Hong Zhu. A survey of the research on power management techniques for high-performance systems. *Softw. Pract. Exper.*, 40(11):943–964, 2010.

[26] Sohaib S. Majzoub, Resve A. Saleh, Steven J. E. Wilton, and Rabab K. Ward. Energy optimization for many-core platforms: communication and pvt aware voltage-island formation and voltage selection algorithm. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 29:816–829, May 2010.

[27] Damián A. Mallón, Guillermo L. Taboada, Carlos Teijeiro, Juan Touriño, Basilio B. Fraguela, Andrés Gómez, Ramón Doallo, and J. Carlos Mouriño. Performance evaluation of mpi, upc and openmp on multicore architectures. In *16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 174–184, 2009.

[28] Alon Naveh, Efraim Rotem, Avi Mendelson, Simcha Gochman, Rajshree Chabukswar, Karthik Krishnan, and Arun Kumar. Power and Thermal Management in the Intel Core Duo Processor. Technical report, Intel Technology Journal, May 2006.

[29] Venkatesh Pallipadi, Shaohua Li, and Adam Belay. Cpuidle-Do nothing efficiently... In *Ottawa Linux Symposium (OLS'07)*, Ottawa,Ontario, Canada, Jun 2007.

[30] Venkatesh Pallipadi and Suresh B Siddha. Processor Power Management Features and Process Scheduler: Do We Need to Tie Them Together? LinuxConf Europe, 2007.

[31] Victor Pankratius and Sven Bläse. Application level automatic performance tuning on the single-chip cloud computer. In *MARC Symposium*, pages 1–6, 2011.

[32] I. Rodero, S. Chandra, M. Parashar, R. Muralidhar, H. Seshadri, and S. Poole. Investigating the potential of application-centric aggressive power management for hpc workloads. In *2010 International Conference on High Performance Computing (HiPC)*, pages 1 –10, dec. 2010.

[33] Randolf Rotta. On efficient message passing on the intel scc. In *MARC Symposium*, pages 53–58, 2011.

[34] Barry Rountree, David K. Lowenthal, Shelby Funk, Vincent W. Freeh, Bronis R. de Supinski, and Martin Schulz. Bounding energy consumption in large-scale mpi programs. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–9, New York, NY, USA, 2007.

[35] Barry Rountree, David K. Lownenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. Adagio: making dvs practical for complex hpc applications. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 460–469, New York, NY, USA, 2009. ACM.

[36] Rafik A. Salama, Ahmed Sameh, C. Bischof, M. Bcker, P. Gibbon, G. R. Joubert, B. Mohr, F. Peters (eds, Rafik A. Salama, and Ahmed Sameh. Potential performance improvement of collective operations in upc, 2007.

[37] Olivier Serres, Ahmad Anbar, Saumil Merchant, and T El-Ghazawi. *Experiences with UPC on TILE-64 processor*, page 19. IEEE, 2011.

[38] Suresh Siddha, Venkatesh Pallipadi, and Arjan Van De Ven. Getting Maximum Mileage Out of Tickless. In *Ottawa Linux Symposium (OLS'07)*, pages 201–208, Ottawa,Ontario, Canada, Jun 2007.

[39] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. Eon: a language and runtime system for perpetual systems. In *5th international conference on Embedded networked sensor systems*, SenSys '07, pages 161–174, New York, NY, USA, 2007. ACM.

[40] Isaías A. Comprés Ureña, Michael Riepen, and Michael Konow. Rckmpi - lightweight mpi implementation for intel's single-chip cloud computer (scc). In *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, EuroMPI'11, pages 208–217, Berlin, Heidelberg, 2011. Springer-Verlag.

[41] Michiel W. van Tol, Roy Bakker, Merijn Verstraaten, Clemens Grelck, and Chris R. Jesshope. Efficient memory copy operations on the 48-core intel scc processor. In *MARC Symposium*, pages 13–18, 2011.

[42] Qiang Wu, Margaret Martonosi, Douglas W. Clark, Vijay Janapa Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David Brooks. Dynamic-compiler-driven control for microprocessor energy and performance. *IEEE Micro*, 26:119–129, January 2006.