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

Marc Gamell, Ivan Rodero, Manish Parashar
NSF Cloud and Autonomic Computing Center
Rutgers Discovery Informatics Institute
Rutgers University, Piscataway, NJ, USA
{mgamell, irodero, parashar}@cac.rutgers.edu

Rajeev Muralidhar
Intel India, Ltd
rajeev.d.muralidhar@intel.com

ABSTRACT

High-performance parallel computing architectures are increasingly based on multi-core processors. While current commercially available processors are at 8 and 16 cores, technological and power constraints are limiting the performance growth of the cores and are resulting in architectures with much higher core counts, such as the experimental many-core Intel Single-chip Cloud Computer (SCC) platform. These trends are presenting new sets of challenges to HPC applications including programming complexity and the need for extreme energy efficiency.

In this paper, we first investigate the power behavior of scientific Partitioned Global Address Space (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 insights that can be used to support similar power management for PGAS applications on other many-core platforms.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*; C.2.4 [**Computer - Communication Networks**]: Distributed Systems; C.4 [**Computer Systems Organization**]: Performance of Systems

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
HPDC'12, June 18–22, 2012, Delft, The Netherlands.
Copyright 2012 ACM 978-1-4503-0805-2/12/06 ...$10.00.

Keywords

Power management, PGAS, SCC, Cross-layer, Application-aware

1. INTRODUCTION

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 [27], hybrid models to improve performance limitations [10] and the implementation of X10 for the Intel SCC [6] and UPC for Tilera's many core [29].

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 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) [26], these approaches do not directly translate to PGAS applications on many-core processors where, for example, such communication and coordination operations are implicit.

This paper explores application-aware cross-layer power management for PGAS applications on many-core platforms, and presents 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, in this paper, 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.

Results obtained from experiments conducted on the SCC platform hosted by Intel 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 and 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 [20]. 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 domains of 8 cores. Finally, based on our observations, we provide a set of insights that can be used to support similar power management for PGAS applications on other many-core platforms. While the current work presented here is on the Intel SCC platform (which is only an experimental one), the focus of this research is to look beyond - on whether programming language level (and programming model aware) power management is meaningful for scientific applications, what are the right abstractions and mechanisms, and what are the constraints and tradeoffs therein - these aspects translate across the platforms.

The rest of the paper is organized as follows. Section 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. Section 3 discusses relevant related work. Section 4 contains a study of power behaviors of PGAS applications based on application profiling, with the goal of identifying opportunities for power management. Section 5 presents the proposed programming extensions and power management system for UPC PGAS applications on SCC, while section 6 presents their evaluation. Finally, section 7 concludes the paper and outlines directions for future work.

2. BACKGROUND

In order to help accelerate many-core research and development, Intel Labs' Tera-scale Computing Research Program has developed the experimental Single-chip Cloud Computer (SCC) processor [21]. Although this processor is not expected to be deployed in future HPC systems, it is representative of current trends, i.e., large numbers of relatively simpler cores, and we believe that our results will translate beyond SCC to such systems. For example, future architectures such as the Intel Knights Corner [28] is touted to have per core power gating, and per-core voltage and frequency switching support, which will only enhance the capabilities that we can exploit from a power/performance point of view.

The hardware consists of 48 x86 Pentium P54C cores each with 32 KB of L1 cache – increased from the standard 16 KB – where 16 KB are for data and 16 KB for instructions, and 256 KB of L2 cache. The cache memory is non-coherent, however customized libraries offer software-based cache coherency. As shown in Figure 1, the 48 cores are configured into tiles with 2 cores per tile. The SCC architecture features a fast (256 GB/s bisection bandwidth) 24-router on-die mesh network which enables communication between tiles and provides hardware support for message-passing. The message-passing support is implemented in the form of a special per-tile 16 KB fast read-write buffer called the Message Passing Buffer (MPB).

The on-chip network also allows each tile to access four dual-channel DDR3 Memory Controllers (MC) which manage, typically, 32 GB (maximum of 64 GB) of main memory for the entire chip. Each core has the ability to run as an individual compute node with its own OS (usually Linux) and software stack and communicate with other compute nodes over a packet-based network. The SCC offers fine-grained power management by allowing dynamic frequency scaling for each tile and dynamic voltage scaling for groups of 4 tiles (as shown in Figure 1). As a result of these power management techniques, the power dissipation can range from 125W to as low as 25W. The frequency and voltage scaling can be controlled by utilizing specific operations exposed by

RCCE. RCCE is an optimized library for the SCC architecture which serves as a high level abstraction by providing functions for message passing, power management and shared memory allocation.

The SCC chip is not directly bootable and requires additional hardware to manage and control it. This is done by an FPGA called the Board Management Controller (BMC) which handles commands to initialize and shut down the SCC and enable power data collection. The BMC is, subsequently, connected to a commodity PC which acts as the Management Console PC (MCPC). The SCC allows power management of three components that work with separate clocks and power sources: mesh network, memory controllers and tiles. In this study, we will focus only on tile power management because mesh and memory controller power management cannot be performed during runtime.

The SCC tiles are arranged in a 6 by 4 grid and further decomposed into distinct voltage and frequency domains (shown in Figure 1). As a result, the dynamic voltage and frequency scaling will affect all tiles within the voltage and frequency domains, respectively. In total there are 6 voltage domains with 4 tiles each and 24 frequency domains with 1 tile each. The voltage can be controlled by using the Voltage Regulator Controller (VRC) which works in a command-based manner: cores send messages to the VRC in order to change the voltage. As described in [2], the VRC allows a set of voltages between 0 and 1.3V in increments of 6.25mV. In contrast to the voltage management system which provides a single VRC for all voltage domains, the frequency management system provides a configuration register for each tile (i.e. for each frequency domain). This configuration register is used to change the frequency divider and, as a result, reduce the clock frequency from the global frequency of 1.6 GHz. Each tile can set the configuration register to integer values between 2 and 16 which correspond to frequency values between 800 MHz to 100 MHz. Figure 1 also shows the default configuration of memory domains, i.e., each tile within a memory domain accesses its corresponding memory controller. However, this configuration can be modified via SCC Address Lookup Table (LUT).

In order to scale voltage, the frequency must be scaled accordingly for the associated tiles. Similarly, scaling frequency up requires a corresponding change in the voltage of the associated voltage domain(s). Through experimental evaluation we found that the lower subrange of voltages cannot be used in practice, as the cores either crash or become unstable. We found empirically that the lowest stable voltage level (using the corresponding frequency) is 0.65625 V. We also found that, on average, the overhead of performing voltage scaling is 40.2 ms.

The official SCC documentation (and the RCCE source code) provides a table with the maximum frequency allowed for each voltage level (in [9] see "Table 9: Voltage and Frequency values"). However, we experimentally found that some of the voltage levels were not stable, as also described in [13]. The voltage-frequency levels that worked robustly during our experiments, are shown in Table 1.

The SCC chip does not provide monitoring tools; however, measured voltage and power dissipation can be obtained by sending a query command to the BMC from the MCPC. Using this feature we were able to automatically collect the measured power dissipation and utilized a sampling frequency of about 6.5 measures per second. We then estimate the consumed energy by integrating the power measures over time.

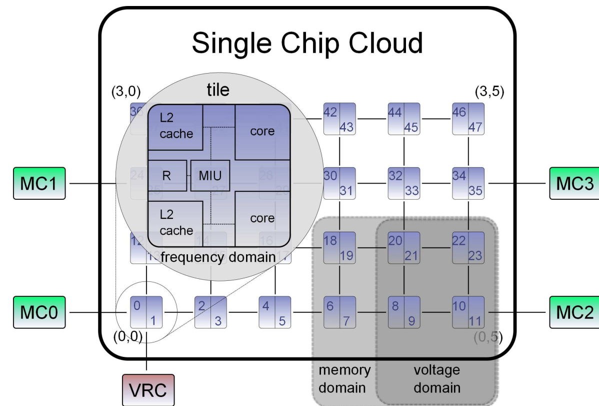


Figure 1: SCC architecture overview

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

Table 1: Stable SCC voltage and frequency values

3. RELATED WORK

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. Since processors dominate the system power consumption in HPC systems [18], 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 [14]. OS-level CPU power management involves controlling the sleep states or the C-states and the P-states of the processor when the processor is idle [22]. 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 [31]. 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 [16], based on application profiles [24], scheduling mechanisms [7] or exploiting low power modes when a task is not in the critical path [26]. Existing work has also addressed power efficiency and power management at the application and compiler levels. For example, Eon [32] is a coordination language for power-aware computing that enables developers to adapt their algorithms to different energy contexts. Wu et al. [36] introduced a dynamic-compiler-driven control for energy efficiency. However, any of the existing programming extensions have been developed in a cross-layer approach.

Existing research has also addressed many-core systems. For example, Majzoub et al. [19] introduced a chip-design approach to voltage-island formation, for the energy opti-

mization of many core architectures. Other approaches have been conducted using the SCC platform. Rotta [25] discussed how to efficiently design and implement the different strategies for message passing on SCC. Pankratius [23] introduced an application-level automatic performance tuning approach on the SCC. Urena et al. [34] implemented an MPI runtime optimized for the SCC message passing capabilities. Clauss et al. [8] improved message passing performance on SCC by adding a non-blocking communication extension to RCCE library. Van Tol et al. [35] introduced an efficient memcpy implementation. Alonso et al. [3] proposed extending power-aware Dense Linear Algebra algorithms to SCC. Although these approaches exploit the SCC along multiple dimensions (i.e., library, runtime, application, etc.), none of them address power and performance tradeoffs from a cross-layer perspective.

Existing PGAS research has focussed mainly on its performance and scalability. Salama et al. [27] proposed a potential improvement of collective operations in UPC. Dinan et al. [10] proposed an hybrid programming paradigm based upon MPI and UPC models that try to improve performance limitations. Chen et al. [15] compare the performance of benchmarks compiled with their own optimizations in BUPC with that of HP UPC compiler. Tarek et al. [12] benchmark UPC and propose compiler optimizations. Kuchera et al. [17] study the UPC memory model and memory consistency issues. Existing PGAS research on many-core systems is not very large. Chapman et al. [6] implemented the X10 programming language on the Intel SCC and performed a comparative study versus MPI using different benchmark applications. Serres et al. [29] 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.

4. PROFILING OF PGAS APPLICATIONS

4.1 Scope for power management

Exploiting load imbalance in MPI applications has been the traditional method of saving CPU energy without significantly penalizing execution time [26]. Here we define an application as load imbalanced when some nodes are assigned more computation than others. In such cases, the nodes with less computation can be run at lower frequency because, otherwise, they waste energy while waiting for other nodes during synchronization calls. However, existing approaches do not translate to PGAS due to the one-sided communication for remote memory accesses, as opposed to the two-sided synchronization (send/receive) in MPI. Additionally, no explicit language-level APIs/library calls are required to access remote memory with PGAS, whereas MPI utilizes specific MPI calls. As such, there are no easily obvious places to interject power management calls. However, like MPI, PGAS implementations (e.g., UPC) also support barriers for synchronization.

In this section we study the behavior of PGAS applications in order to understand the use of different operations and identify opportunities to perform power management. Several intermediate-level operations, such as `wait` (barrier), `lock` (critical section), `memget`, `memput` or `memcpy`, might be candidate operations for power management, however, application profiling reveals that only `wait` and `memget` operations have potential for power management.

Existing profiling tools for UPC such as PPW [33] or GASP [30] are designed to be executed in traditional systems with large amounts of memory. Using such tools in the SCC environment (i.e., a large number of cores running an OS but sharing the memory), would interfere with the profiling due to the overhead of the instrumentation. Additionally, existing profiling tools for PGAS must be adapted to the SCC framework. For this reason we decided to implement a lightweight instrumentation system (power management instrumentation - `pmi`). `pmi` is currently available on UPC, but extensible to any other PGAS runtime. We achieve low overhead by writing lightweight intermediate files containing data in raw binary format. Then, the stored data is parsed in order to generate begin/end timestamps for each call and automatically create log files. Plots are generated postmortem. Through evaluation we found that the measured overhead of `pmi` is less than 1%.

4.2 Benchmarks

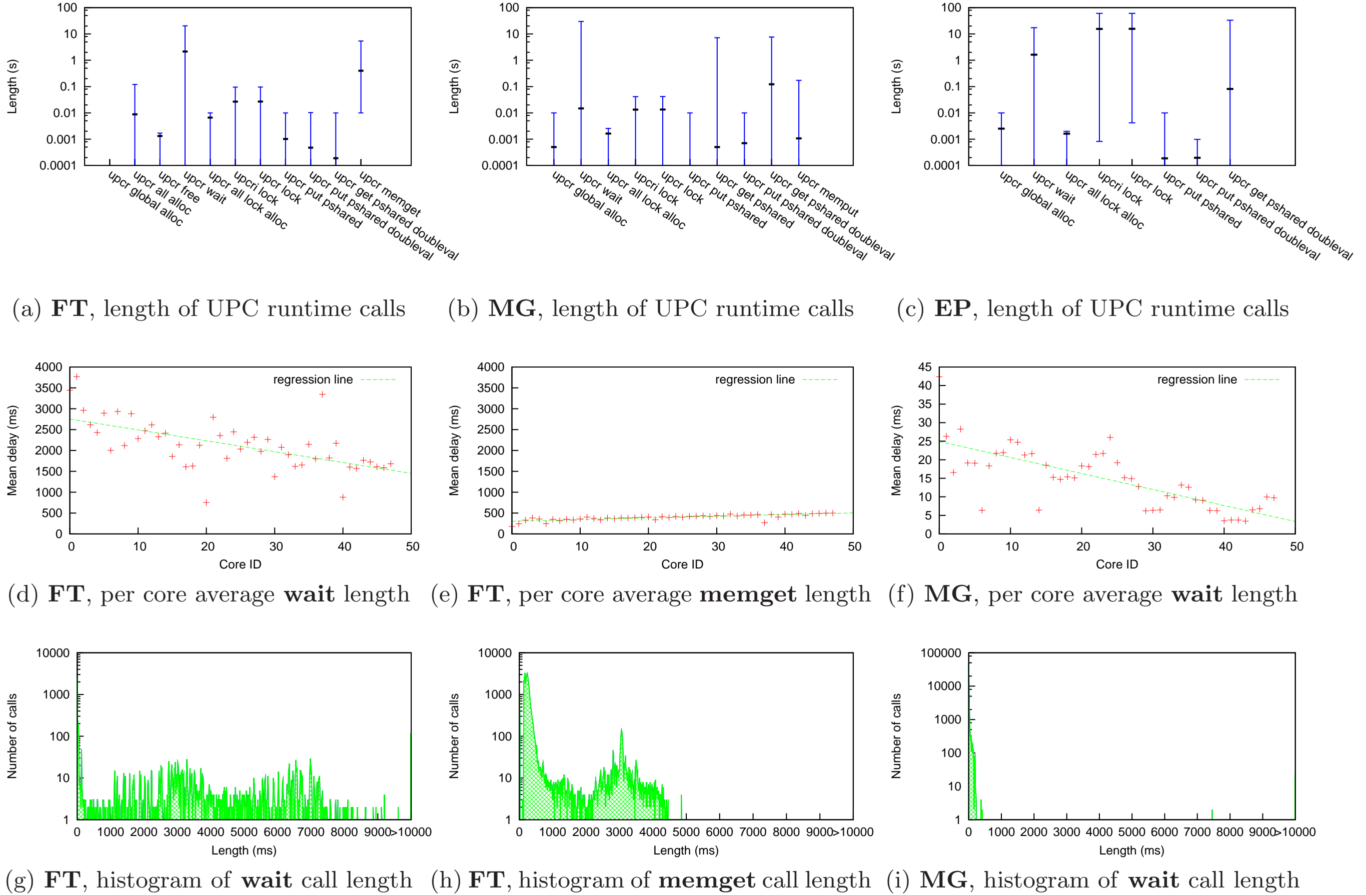
Three different types of kernels were used: (1) the George Washington University UPC implementation of the NAS Parallel Benchmarks (NPB) [11], (2) the Sobel edge detector kernel [20], and (3) a customizable synthetic application. Specifically, we used the FT (Fast Fourier Transform), EP (Embarrassingly Parallel) and MG (Multi Grid) kernels from NPB.

In order to identify the potential for power management in a single many-core system, we aimed to stress the SCC platform by executing each kernel on all 48 cores. We used the largest problem size class of each benchmark that fits in the SCC memory (i.e., class C for FT and MG and class D for EP). Sobel is an edge detection application widely used in computer vision. The parallelized version of this algorithm partitions the image among the cores, performs calculations locally and accesses elements from another execution thread when data shifting is required.

Since the SCC does not support per-core frequency and voltage scaling, we have defined a synthetic application (`matmul`) to determine how to exploit application load imbalance to save energy. It specifically performs a set of matrix multiplications as shown in the algorithm below.

```
for (i=0 ; i<N ; i++) {
  perform A matrix multiplications
  (distributed in 48 cores);
  if(MYTHREAD is in the last Voltage Domain)
     perform B matrix multiplications;
  upc_barrier;
}
```

`N` is the number of iterations, `A` is the number of matrix multiplications performed by all the 6 voltage domains, and `B` is the number of matrix multiplications performed by only one domain. We can use these parameters to configure the load imbalance, which in this case is the percentage of time that threads are blocked in a barrier while waiting for other threads. For example, we used `N`=36, `A`=20 and `B`=5,000 to obtain 33% of load imbalance. These parameters, led to a 33 % of load imbalance due to: (1) the first phase of the `matmul` benchmark is executed in parallel while the second phase is performed sequentially, and (2) since the matrices used are not very large, a parallel multiplication requires lot of syn-



(a) **FT**, length of UPC runtime calls (b) **MG**, length of UPC runtime calls (c) **EP**, length of UPC runtime calls

(d) **FT**, per core average **wait** length (e) **FT**, per core average **memget** length (f) **MG**, per core average **wait** length

(g) **FT**, histogram of **wait** call length (h) **FT**, histogram of **memget** call length (i) **MG**, histogram of **wait** call length

Figure 2: NAS Parallel benchmarks UPC calls behaviour

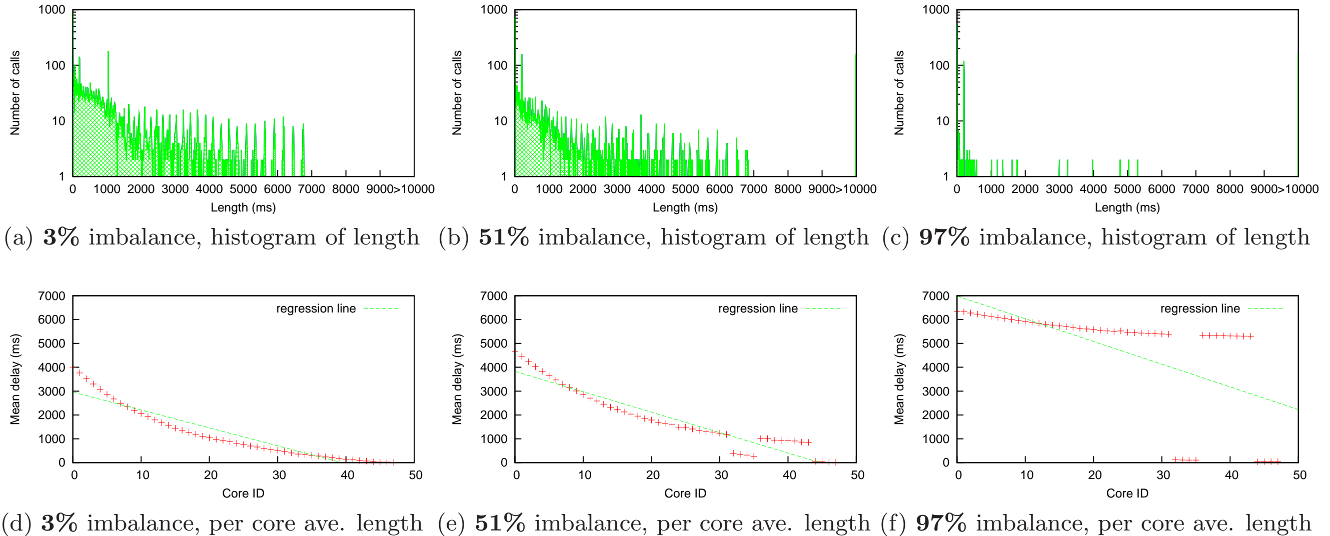
chronization and, consequently, a sequential multiplication is far faster than a distributed one.

4.3 Discussion

Figure 2a shows that FT uses 11 UPC runtime operations, but only `memget` and `wait` (barriers) are likely to be long enough to take advantage of power management. Note that, since we observed some variability in the executions, the figures of this subsection show average values along with minimum and maximum. We found that the NAS FT benchmark had a large number `memget` operations and several barriers per core. Figures 2g and 2h show histograms of `wait` and `memget` operation call lengths, respectively. The distribution is not homogenous, especially for `memget` calls. In addition to 2,500 very short `wait` calls, there were many that took 1-4s, 6-7s and longer than 10s. `memget` operation calls were much shorter, only a small percentage of calls were around 3s or longer. Therefore, we can conclude that the UPC `wait` operation is a good candidate for exploring aggressive power management in NAS FT. However, the average length of both `wait` and `memget` operations are not uniform over the cores, as shown in Figures 2d and 2e . The observed trend indicates that we will not be able to apply a fixed power management criteria over all the cores simultaneously.

The dominant operations of MG are `wait`, `get_pshared` and `get_pshared_doubleval` as shown in Figure 2b. Each core performs a large number of `wait` calls. Figures 2f and 2i show that most of the `wait` calls are short (about 5-300ms), however, some of them are quite long (close to 10 seconds). The long `wait` calls correspond to the initialization phase of the kernel but the rest of the execution is well balanced. The behavior of `get_pshared` and `get_pshared_doubleval` calls is similar to the behavior exhibited by `wait` calls. In this case, the challenge is performing power management only on the meaningful calls without penalizing the overall performance. The dominant operations of EP are `wait`, `lock` and `get_pshared_doubleval` as shown in Figure 2c. Like MG, only a few calls are long enough to be considered for power management without incurring too large an overhead. This is because EP's profile is neither communication- nor memory-bound.

We considered `wait` and `global_alloc` operations for the Sobel application but the `upcr_get_pshared` operation was not instrumented because we observed a large number of short calls (about 90 million per core) that impact the execution time considerably. The first core behaved differently than the rest of the cores due to the initialization phase. Specifically, there is a barrier at the beginning of the execution that delays all the cores except one core, which is doing the initialization. Once the initialization is done, it repeats the Sobel algorithm 100 times and synchronizes with



(a) **3%** imbalance, histogram of length (b) **51%** imbalance, histogram of length (c) **97%** imbalance, histogram of length

(d) **3%** imbalance, per core ave. length (e) **51%** imbalance, per core ave. length (f) **97%** imbalance, per core ave. length

Figure 3: Synthetic matmul benchmark UPC wait behaviour for three levels of imbalance: 3%, 51% and 97%

a barrier at the end of each iteration, which results in a slack period of 0.5-2 seconds. Although the potential of runtime power management for Sobel is lower than other kernels such as FT, an application-aware approach can improve energy efficiency. An application-aware approach would allow the runtime system to identify meaningful periods to apply aggresive power management such as the initialization and synchronization periods.

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). We ran `matmul` with load imbalances ranging from 3% to 97%. Figures 3a, 3b and 3c show the number of `wait` calls of each specific length. Figures 3d, 3e and 3f show the average length of calls, per core. Note that in tests with very imbalanced applications (Figures 3b and 3c) many calls are longer than 10 seconds. The results show very different behaviors depending of the percentage of load imbalance. Specifically, longer `wait` calls correspond to larger load imbalance percentages.

5. POWER MANAGEMENT MIDDLEWARE

In this section, we describe the power management middleware that we have developed to perform cross-layer power management for PGAS applications on the SCC. The main goal is exploiting the application's slack periods that we identified in the previous section (i.e., during `wait` and `memget` operations). Although we focus on UPC, the power management middleware implementation is independent of the PGAS instantiation. Additionally, we provide a set of interfaces to give the programmer the ability to easy-tune the runtime power management through programming extensions (in the form of high level hints).

5.1 Architecture

Since an integral approach for overall power management is challenging, we consider a layered model that allows us to reduce complexity by addressing specific problems at different layers and integrating them using a cross-layer approach. Specifically, we consider three different layers: resource, runtime and application (see Figure 4). The left column of Figure 4 represents the standard application stack model of a UPC application on the SCC. Note that, in order to improve the communication, we use RCKMPI [34], which is the MPI implementation for the SCC. RCKMPI uses the SCC's on-die message passing buffers and mesh as a physical resource.

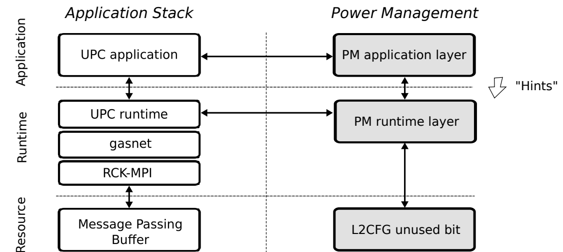


Figure 4: Cross-layer architecture

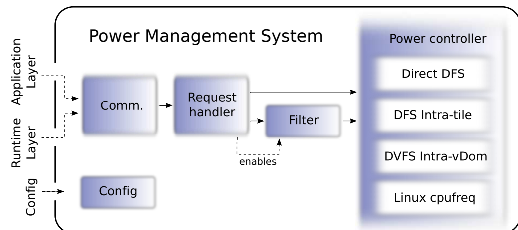


Figure 5: Power management system architecture

The global architecture of the power management system is shown in the right column of the mentioned figure, and its internal architecture is shown in Figure 5. The communication component receives requests from both the runtime (e.g., message-passing call or memory transfer) and the application layers using standard unix sockets. This mechanism adds some overhead to the execution time (< 1%), however, it makes the power management system independent of the PGAS incarnation and runtime. As a result,

it improves the usability, portability and maintainability of the power management system. The request handler identifies the source of each request, applies the established layer priorities (e.g., application extensions might have higher priority than runtime calls; i.e., if it is specified that, during a period of time the program should run using a specific policy, the middleware does not consider the policies at the runtime layer) and manages the coherency of calls (i.e., entry and exit). The filter module implements policies to decide when to use low power modes (e.g., predefined threshold or adaptive threshold based on history), and the power controller is responsible for setting the power modes using different techniques. The configuration module allows us to define threshold values, choose filtering policy, bypass entire modules and other parameters.

5.2 Policies

High level policies can be specified using programming extensions at the application layer. Analogous to CPU governors (i.e., policies of `cpufreq` OS power management) we define a set of performance/power driven user-level policies:

1. `PM_PERFORMANCE` for maximum performance (i.e., working at highest voltage and frequency levels).
2. `PM_CONSERVATIVE` to balance power/performance (i.e., using low power modes during slack).
3. `PM_POWER` for minimum power at the cost of a limited delay penalty (i.e., using a continuous voltage level, regardless of he applications' slack periods).
4. `PM_AGGRESSIVE_POWER` for maximum power reduction (i.e., using lowest voltage and frequency levels).

The filter module is responsible for determining which requests have to be considered to switch the power mode (i.e., frequency and voltage levels) over time. The experimental evaluation of different kernels with different policies stated that policies must be very lightweight to avoid large overhead since they have to run in a performance-constrained environment (i.e., SCC cores). As we observed from the application profiling, only long `wait` and `memget` calls are meaningful for power management. Switching power modes during short calls highly impacts the execution time (and consequently the energy consumption) due to the overheads for changing the power mode. We propose three different policies (all with $O(n)$ cost):

(i) Fixed time *threshold*. When an entry call request is received, the filtering module waits for the amount of time specified by *threshold* value. If this threshold is reached and no exit call has been received, then the request handler redirects the request to the power controller; otherwise, the call is discarded (i.e., is filtered).

(ii) Variable time *threshold*. This approach maintains a historical call-length moving average and calculates the threshold by multiplying a given customizable constant k with the historical average.

(iii) *Mixed* approach. This policy deals with the application's behavior variations according to the program region. It dynamically adapts the threshold value (within given bounds) using recent historical average.

In order to avoid performing power management during those calls whose length is $threshold + \epsilon, \epsilon \rightarrow 0$, we support an intermediate frequency level, in addition to high and low. This allows, in certain circumstances, to distinguish between short, medium and long calls. Shorter calls are discarded by waiting for a short time before accepting the request. When this threshold is reached, the algorithm can step up to a medium power stage.

5.3 Power controller

The power controller is responsible for applying the frequency and voltage levels as the Linux cpu-freq module has not been ported to the SCC yet. Although the SCC standard library (RCCE) offers power management mechanisms, it needs to utilize the Message Passing Buffers whilst they are being used by RCKMPI (i.e., adding more load to the system) (see figure 4). Therefore, the power controller directly accesses the hardware-specific power management controls (bypassing RCCE library). Power adjustments can be done in several ways and each one has been implemented on a separated submodule:

DFS. In order to set a core's frequency we need to set the configuration register, utilized by the frequency divider, to values between 2 and 16. We must take into account, however, that setting the register for a single tile will slow down both cores in the tile. If the frequency is switched without considering this situation (i.e., slowing down both cores), the application's execution time may increase, especially if one core has more load than the other.

DFS with synchronization. DFS Intra-tile submodule synchronizes both cores in a tile and applies frequency reduction only when both cores have agreed (i.e., only when both are in a slack period). To do this synchronization, we use an unused bit inside the hardware-implemented `L2CFG` register (two of these per tile) as a communication mechanism. This method is utilized because the default communication system (MPB) cannot be used, because it is used by RCKMPI. The synchronization algorithm, which has been designed to be distributed, fast, and lightweight, can work in two modes: using 2 frequency levels (high and low) or allowing an additional intermediate frequency level (med).

DVFS with synchronization. In order to adjust the voltage, eight cores must be synchronized to avoid performance loss. This feature has been implemented by the DVFS intra-vDom submodule. To coordinate a voltage domain we have used the same unused bit in the `L2CFG` register. In this case, the algorithm is more complex, and is centralized. A core called the *controller* executes the algorithm in a separate thread (for performance purposes), adjusts the tile frequencies (for all 8 cores) and sends commands to the VRC. The remaining seven cores are considered as the *clients*. When a *client* requests a voltage level, it sets the corresponding bit in the `L2CFG` register. The voltage will be adjusted (with the associated frequency level shown in Table 1) when the *controller* detects that all cores have requested the change. Note that, although the synchronization algorithm only supports two voltage levels (high and low), we can tune it in order to recognize a third level. However, this makes the assumption that all cores request the same voltage. The *client*'s L2CFG bit is only modified by itself, and indicates the state: waiting for low power (1), or waiting for high power (0). When a *client* modifies its own bit, it also sets the *controller*'s bit. The implementation of the *controller* side is quite straightforward and can be done 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* can lower the voltage) or if some are unset (in this case, the *controller* can increase the voltage). Note that this con-

struct minimizes the network traffic, as polling packets are only sent when a core's state changes.

CPU-FREQ. A module that uses the Linux cpu-freq interface for portability purposes.

6. EXPERIMENTAL EVALUATION

Results obtained from experiments conducted on the SCC platform showed that certain PGAS operations (e.g., `wait` and `memget`) need to be considered for power management. From those operations, performing power management during long calls can provide large energy savings and during short calls can penalize both execution time and energy consumption. Furthermore, PGAS operation calls may cluster by length facilitating the differentiation between long and short calls. If they do not cluster using an intermediate power mode works better. Power management during memory accesses surprisingly provides significant energy savings even though the memory is shared among all the cores. We expect larger energy saving in distributed memory systems.

The experiments also show that in 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.

In this section, we present the results obtained from the experimental evaluation of the UPC application kernels described in section 4.2 on the SCC prototype at Intel Labs using the Berkeley UPC runtime and RCKMPI. We first present the results obtained with the NAS kernels, focusing on FT since it showed higher potential for power management than the other NAS kernels (see section 4). Our base tests were executed at high (800MHz – 1.1V), intermediate (533MHz – 0.85V) and low (400MHz – 0.75V) power modes. The figures show average values of 50 runs along with maximum and minimum, and are normalized to the results obtained with the base test. Figures 6 and 7 also show the already well-known energy delay product (EDP) metric (product of runtime and expended energy), which captures the effect of energy management on performance.

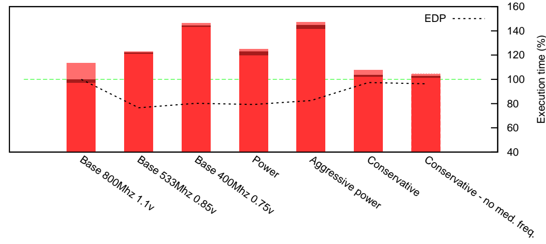
Figures 6a and 6b show the normalized execution time and energy consumption (Joules), respectively, of NAS FT class C using different application level policies. The policies are applied at the beginning of the kernel and are not modified during its whole execution, so here we focus on automatic runtime power management using the different policies. The last two columns of the figures show the results obtained with the `PM_CONSERVATIVE` policy enabling DVFS only during `wait` and `memget` operations, and filtering the calls with *threshold* policy (rejecting calls shorter than 20 ms for `wait` and shorter than 300 ms for `memget`, and considering medium power mode for `memget` calls between 300ms and 1s). However, the last column shows the results obtained using only two power modes (low and high). Overall, the figures show that `PM_CONSERVATIVE` (i.e., automatic runtime power management) can save up to 7% of energy with as little as 3% time penalty, on average. They also show that using the other policies we can obtain higher energy savings (up to 45%) but at the cost of higher time penalty. Thus, policies allow users to define their energy/performance trade off goals. It is also worth noting that correlations between the test base at 533MHz and the `PM_POWER` policy, and between the test base at 400MHz and the `PM_AGGRESSIVE_POWER` policy are shown.

Figures 6c and 6d show the results of NAS FT execution using DVFS and the *threshold* filtering policy with different configurations in terms of thresholds and operations support. The figures show, on the one hand, that the majority of configurations result in significant energy reduction with little time penalty considering the limitations of the platform. On the other hand, they show that not considering the `memget` operation for power management provides lower energy savings and lower time penalties. The lowest time penalty is obtained when considering both `wait` and `memget` operations for filtering and using a threshold of 1000 – 2000 (0.4% time penalty, on average) and the lowest energy consumption is obtained using a threshold of 300–1000 (7%, on average). Figures 6e and 6f show the results of NAS FT execution using the moving average filtering policy. The figures show that larger energy savings resulted in higher time penalties. However, there are some configurations that balance energy and performance such as the k.100-300 test (rejecting calls whose length is less than 100% of historical moving average, and considering a medium power mode for calls between 100% and 300% of historical moving average), results in 3% energy savings with only 1% time penalty.

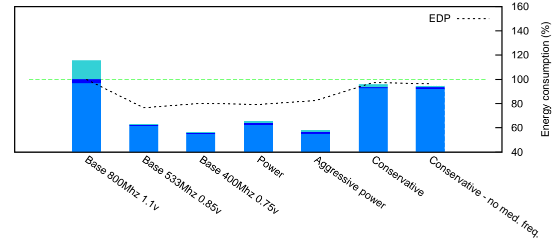
Figure 7 shows the results obtained with NAS EP class D and NAS MG class C. The results show that automatic runtime power management (i.e., `PM_CONSERVATIVE` policy) does not provide significant energy savings. However, application level policies allows the programmer to manage energy/performance tradeoff within a wide range of energy saving and time penalty.

We used Sobel to study the potential of the proposed programming extensions (i.e., application level hints) used throughout the application, in contrast to the previous experiments that uses a given policy during the whole application execution and rely on automatic runtime power management. Specifically, we extended the application's code by including per-thread hints during the initialization phase and a common policy during the iterative phase to facilitate power management. Table 2 shows the normalized average execution time, energy consumption and EDP, obtained with the different policies. In "Original Sobel" we used the same policy during the whole application execution and in "Modified Sobel" we considered the described application extensions. 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. Figures 8a and 8b show the SCC power dissipation over time using the `PM_POWER` policy and using the application level hints described previously. The former shows an almost-continuous power dissipation during the application execution regardless of the application phase. The latter shows different power levels during the different application phases, especially in the second phase (after 500s), which is the actual parallel computation.

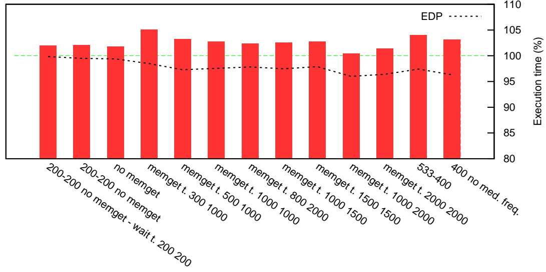
The main goal of the synthetic `matmul` application evaluation is showing the potential energy savings and delay penalty (upper bounds) of both runtime and application-aware power management techniques for different levels of load imbalance. Figure 9 shows the normalized energy savings and time penalty of `matmul`, ranging from 3% to 97% of load imbalance, for different power management strate-



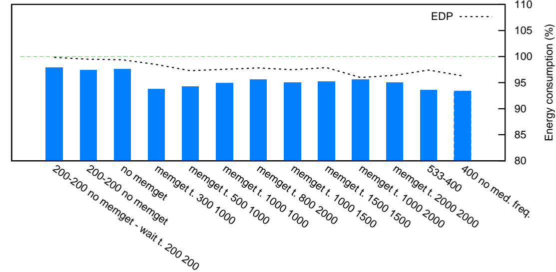
(a) Execution time (hints)



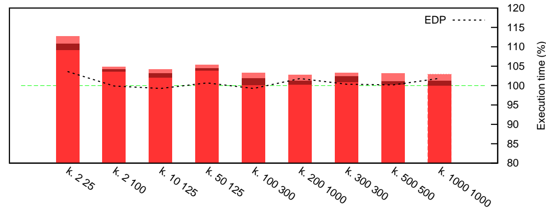
(b) Energy consumption (hints)



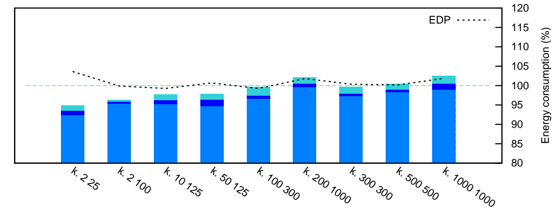
(c) Execution time (DVFS, threshold filtering)



(d) Energy consumption (DVFS, threshold)



(e) Execution time (DVFS, moving ave.filtering)



(f) Energy consumption (DVFS, moving ave.)

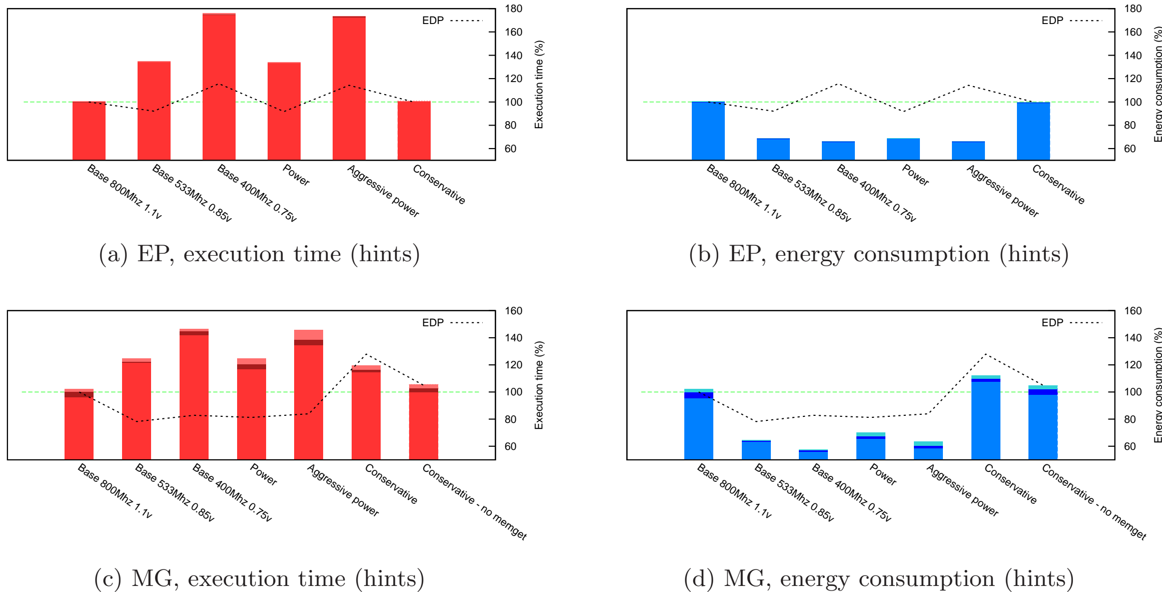
Figure 6: Normalized (%) execution time and energy consumption of FT class C using different policies and strategies. The borderline between two different color tonalities indicates (down to up) the minimum, average, and maximum observed values

gies. 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). Figures 9a and 9b show the results obtained performing power management to all the calls (i.e., no threshold filter) and filtering the calls with a threshold of 500ms, respectively. The figures show that, although the energy savings are similar, the time penalty is much higher when DVFS is applied to all the `wait` calls due to the associated overheads. Figures 9c and 9d 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 homogenous among all the voltage domains. 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 wider set of results can be found at `http://nsfcac.rutgers.edu/GreenHPC/research-scc.php`.

7. CONCLUSION AND FUTURE WORK

In this paper 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 provide several insights that can be translated to other PGAS languages and platforms. They 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.



(a) EP, execution time (hints)

(b) EP, energy consumption (hints)

(c) MG, execution time (hints)

(d) MG, energy consumption (hints)

Figure 7: Normalized (%) execution time and energy consumption of EP class D and MG class C, using different policies

Test description	Original Sobel Execution time %	Original Sobel Energy %	Original Sobel EDP %	Modified Sobel Execution time %	Modified Sobel Energy %	Modified Sobel EDP %
Base 800Mhz - 1.1v	100.0	100.0	100.0	100.0	100.0	100.0
Base 533Mhz - 0.85v	146.8	74.4	109.9	101.3	79.0	80.4
PM_PERFORMANCE	100.7	100.0	100.8	N/A	N/A	N/A
PM_POWER	**147.6**	75.0	110.7	**101.8**	79.4	80.8
PM_AGGRESSIVE_POWER	**193.8**	72.8	141.0	**101.9**	73.7	75.2
PM_CONSERVATIVE	**118.0**	75.7	89.4	**101.5**	73.7	74.9

Table 2: Execution time and energy savings for Sobel application (average of 50 samples). Modified Sobel version includes programming extensions and provides dramatic execution time reduction while maintaining similar energy savings

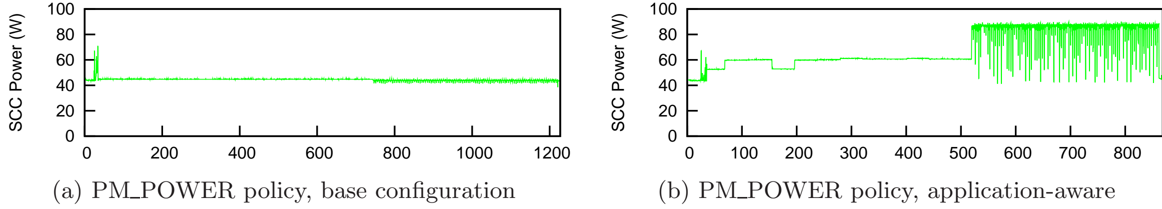
Power management during memory accesses provides surprisingly significant energy savings even though memory is shared among all the cores. We expect larger energy saving in distributed memory systems. We also have observed that large energy savings can be obtained with imbalanced applications; however, blocking times in the barriers are not usually homogenous 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 tradeoffs 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]. Clearly, the tradeoff between power control granularity to enable energy saving and processor performance requires further research.

However, these results need to be considered keeping in mind that the Intel SCC is an experimental platform. Arguably, future architectures such as the Intel Knights Corner, is expected to have much more granular power management capability, with per core clock and power gating, per-core voltage and frequency switching support, which will only enhance the capabilities that we can exploit from a power/performance point of view. Such architectural enhancements will provide aggressive capabilities to software to exploit most idle/high latency periods. For example, typically, OS based DVFS (ondemand governor, for example) will change the CPU frequency if the workload is heavily memory bound - this is done by the governor looking at the ACNT/MCNT ratio every 40 ms (default setting), which indicates how CPU/memory bound the workload is. So the question then arises as to how much more quickly can we determine that the workload is memory bound; this is beyond the range of current OS governors. We believe that if that could be done through workload profiling/analysis or programming model hints, aggressively deploying DVFS and/or per-core clock/power gating can potentially yield more power savings on such future architectures. We intend to continue this line of research on the Intel Knights Corner and other such many-core architectures.

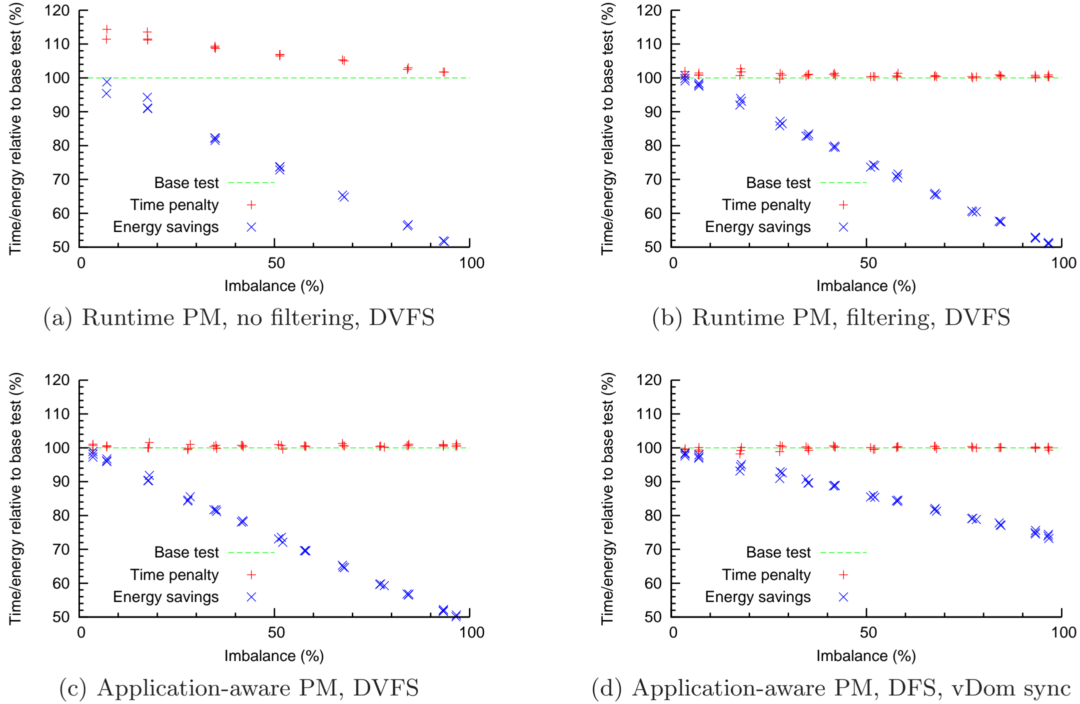
Our ongoing and future work also include: (i) exploring other PGAS models (e.g., Global Arrays and Co-Array For-



(a) PM_POWER policy, base configuration

(b) PM_POWER policy, application-aware

Figure 8: Measured power in Sobel application



(a) Runtime PM, no filtering, DVFS

(b) Runtime PM, filtering, DVFS

(c) Application-aware PM, DVFS

(d) Application-aware PM, DFS, vDom sync

Figure 9: Energy savings and time penalty of matmul with different policies and load imbalance levels

tran), (ii) implementing cross-layer optimizations between application, compiler and runtime levels using the proposed application level hints, (iii) using per-core performance counters in order to detect application profiles and adjust frequency and/or voltage accordingly, (iv) explore other imbalanced applications, such as unstructured graph problems difficult to load balance, and (v) 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.

Acknowledgments

The research presented in this work is supported in part by National Science Foundation (NSF) via grants numbers IIP 0758566 and DMS-0835436, by the Department of Energy ExaCT Combustion Co-Design Center via subcontract number 4000110839 from UT Battelle and via the grant numbers DE-SC0007455 and DE-FG02-06ER54857, and by an IBM Faculty Award, and was conducted as part of the NSF Cloud and Autonomic Computing (CAC) Center at Rutgers University. We thank Intel for the access to the SCC and the opportunity to contribute to its MARC (Many-core Applications Research Community) program. We also thank the anonymous reviewers for their constructive comments and recommendations, and Aditya Devarakonda for proofreading this paper.

8. REFERENCES

[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] P. Alonso, M. F. Dolz, F. D. Igual, B. Marker, R. Mayo, E. S. Quintana-Ortí, and R. 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] S. Amarasinghe, M. Hall, R. Lethin, K. Pingali, et al. ASCR Programming Challenges for Exascale Computing. Technical report, U.S. DOE Office of Science (SC), July 2011.

[5] S. Borkar. Thousand core chips: a technology

perspective. In 44th annual Design Automation Conf., pages 746–749, 2007.

[6] K. Chapman, A. Hussein, and A. Hosking. X10 on the single-chip cloud computer. In X10 Workshop, pages 460–469, 2011.

[7] Y. Chen, A. Das, W. Qin, A. Sivasubramaniam, Q. Wang, and N. 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.

[8] C. Clauss, S. Lankes, and T. Bemmerl. Performance tuning of scc-mpich by means of the proposed mpi-3.0 tool interface. In 18th European MPI Users' Group Conf. on Recent advances in the message passing interface, pages 318–320, 2011.

[9] I. Corporation. The scc programmers guide, revision 0.75, 2010.

[10] J. Dinan, P. Balaji, E. Lusk, P. Sadayappan, and R. Thakur. Hybrid parallel programming with mpi and unified parallel c. In 7th Intl. Conf. on Computing frontiers, pages 177–186, 2010.

[11] T. El-Ghazawi and F. Cantonnet. Upc performance and potential: a npb experimental study. In ACM/IEEE Conf. on Supercomputing, pages 1–26, 2002.

[12] T. A. El-Ghazawi and S. Chauvin. Upc benchmarking issues. In 2001 Intl. Conf. on Parallel Processing, pages 365–372, 2001.

[13] P. Gschwandtner, T. Fahringer, and R. Prodan. Performance analysis and benchmarking of the intel scc. In CLUSTER, pages 139–149, 2011.

[14] C.-H. Hsu and W.-C. Feng. A power-aware run-time system for high-performance computing. In ACM/IEEE Conf. on High Performance Networking and Computing, page 1, 2005.

[15] P. Husbands, C. Iancu, and K. Yelick. A performance analysis of the berkeley upc compiler. In 17th Intl. Conf. on Supercomputing, pages 63–73, 2003.

[16] N. Kappiah, V. W. Freeh, and D. K. Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs. In ACM/IEEE Conf. on Supercomputing, page 33, 2005.

[17] W. Kuchera and C. Wallace. The upc memory model: problems and prospects. In 18th Intl. Parallel and Distributed Processing Symposium, page 16, 2004.

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

[19] S. S. Majzoub, R. A. Saleh, S. J. E. Wilton, and R. 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, 2010.

[20] D. A. Mallón, G. L. Taboada, C. Teijeiro, J. Touriño, B. B. Fraguela, A. Gómez, R. Doallo, and J. C. Mouriño. Performance evaluation of mpi, upc and openmp on multicore architectures. In 16th European PVM/MPI Users' Group Meeting, pages 174–184, 2009.

[21] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core scc processor: the programmer's view. In ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis, pages 1–11, 2010.

[22] V. Pallipadi, S. Li, and A. Belay. Cpuidle-Do nothing efficiently... In Ottawa Linux Symposium, Ottawa,Ontario, Canada, 2007.

[23] V. Pankratius and S. Bläse. Application level automatic performance tuning on the single-chip cloud computer. In MARC Symposium, pages 1–6, 2011.

[24] 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 Intl. Conf. on High Performance Computing, pages 1 –10, 2010.

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

[26] B. Rountree, D. K. Lownenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch. Adagio: making dvs practical for complex hpc applications. In Intl. Conf. on Supercomputing, pages 460–469, 2009.

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

[28] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, et al. Larrabee: a many-core x86 architecture for visual computing. ACM Trans. Graph., 27(3):18:1–18:15, 2008.

[29] O. Serres, A. Anbar, S. Merchant, and T. El-Ghazawi. Experiences with UPC on TILE-64 processor, pages 1–9. IEEE, 2011.

[30] H. Shan, F. Blagojević, S.-J. Min, P. Hargrove, H. Jin, K. Fuerlinger, A. Koniges, and N. J. Wright. A programming model performance study using the nas parallel benchmarks. 18:153–167, 2010.

[31] S. Siddha, V. Pallipadi, and A. V. D. Ven. Getting Maximum Mileage Out of Tickless. In Ottawa Linux Symposium, pages 201–208, 2007.

[32] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. Eon: a language and runtime system for perpetual systems. In 5th Intl. Conf. on Embedded networked sensor systems, pages 161–174, 2007.

[33] H.-H. Su, M. Billingsley, and A. D. George. Parallel performance wizard: A performance system for the analysis of partitioned global-address-space applications. Int. J. High Perform. Comput. Appl., 24:485–510, 2010.

[34] I. A. C. Ureña, M. Riepen, and M. Konow. Rckmpi - lightweight mpi implementation for intel's single-chip cloud computer (scc). In 18th European MPI Users' Group Conf., pages 208–217, 2011.

[35] M. W. van Tol, R. Bakker, M. Verstraaten, C. Grelck, and C. R. Jesshope. Efficient memory copy operations on the 48-core intel scc processor. In MARC Symposium, pages 13–18, 2011.

[36] Q. Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks. Dynamic-compiler-driven control for microprocessor energy and performance. IEEE Micro, 26:119–129, 2006.