Scheduling for Better Energy Efficiency on Many-core Chips

Chanseok Kang, Seungyul Lee, Yong-Jun Lee,
Jaejin Lee, and Bernhard Egger

Department of Computer Science and Engineering,
Seoul National University, Korea

Abstract. *Many-core chips are especially attractive for data center operators providing cloud computing service models. With the advance of many-core chips in such environments energy-conscious scheduling of independent processes or operating systems (OSes) is gaining importance. An important research question is how the scheduler of such a system should assign the cores to the OSes in order to achieve a better energy utilization. In this paper, we demonstrate that many-core chips offer new opportunities for extremely light-weight migration of independent processes (or OSes) running bare-metal on the many-core chip. We then show how this intra-chip migration can be utilized to achieve a better performance per watt ratio by implementing a hierarchical power-management scheme on top of dynamic voltage and frequency scaling (DVFS). We have implemented and tested the proposed techniques on the Intel Single Chip Cloud Computer (SCC). Combining migration with DVFS we achieve, on average, a 25-35% better performance per watt over a DVFS-only solution.*

Keywords: Scheduling, Process Migration, DVFS, Performance per Watt

1 Introduction

The recent trend to integrate more and more cores onto a single chip, so called chip multiprocessors or CMPs [2, 19], has led to chip-level power and thermal constraints becoming one of the primary design constraints and performance limiters [1]. A higher power consumption not only leads to increased energy cost but the higher die temperatures adversely affect chip reliability and lifetime.

Dynamic voltage and frequency scaling (DVFS) allows to lower the operating voltage and frequency of a core to meet its required performance. For current multi-core systems, each core can be controlled individually, however, for CMPs the required logic for individually controlling the voltage and frequency for each core is becoming too costly [12]. Cores are logically clustered into voltage and frequency domains that share a common setting [8, 21]. Researchers have proposed numerous techniques for individually-controllable and clustered cores [4, 5, 10, 20].

With the ongoing server consolidation, an ever increasing number of cores per chip, and the overhead associated with maintaining a coherent global shared

memory, it is more and more common to run several completely independent (sequential or parallel) applications alongside each other on the same physical many-core chip without a common underlying OS [3]. Instead, the independent OSes or applications have full control over the assigned hardware resources and are responsible for scheduling the work on the assigned cores and managing the allocated physical memory. Hosting providers, for example, providing access to bare-metal hosts can execute independent OSes on the different physical cores of a CMP. Existing power management solutions for CMPs are built for a single operating system kernel managing all running (groups of) tasks. To the best of our knowledge, no solutions for CMPs running independent OSes have been proposed.

In this paper, we propose an extremely light-weight OS migration method for independent OSes running on CMPs and show that the scheduler can exploit this OS migration to significantly increase the effectiveness of DVFS policies for CMPs. We have implemented an energy-aware scheduler exploiting light-weight OS migration in the Linux operating system running on the Intel Single-chip Cloud Computer (SCC) [9]. Compared to a state-of-the-art hierarchical power management with DVFS but no OS migration [10], the proposed approach achieves 25-35% better performance per watt ratio over a wide range of workloads.

The remainder of this paper is organized as follows: Section 2 presents related work. Section 3 introduces the many-core architecture. Section 4 describes the implementation of the light-weight OS migration in detail. In Section 5, the integration of OS migration into an energy-aware scheduler with hierarchical power management for many-core chips is discussed. Section 6 presents the experimental setup, and Section 7 discusses the experimental results. Section 8, finally, concludes the paper.

2 Related Work

There is a significant amount of work focusing on the design and implementation of power management techniques for CMPs. Our focus lies on independent OSes executing directly on the hardware in a space-shared manner on the CMP and on exploiting the hardware capabilities of existing and future many-core systems with regard to coarse-grained voltage and/or frequency domains.

One line of related work considers heterogeneous CMP designs in order to consume less power with no or minimal performance loss. Kumar et al. [13] propose heterogeneous CMPs composed of cores supporting the identical ISA but consuming more or less energy depending on the core architecture. Ghiasi [7] proposes CMPs with cores executing at different frequencies. Both works show that such systems offer improved power consumption and thermal management. Our work differs in that our approach modifies the voltage/frequency of cores dynamically, without being bound to certain hardware heterogeneity.

Another line of research has focused on exploiting idle periods. Meisner et al. propose PowerNap [16] and DreamWeaver [17]. Both assume hardware support

for quick transitions between on- and off-states; the latter work batches wake-up events to increase the sleep periods. Our work is orthogonal to such approaches, with one limitation: applying DVFS may lead to a longer execution time which in turn reduces the potential to sleep.

A number of researchers have proposed heterogeneous power management techniques for CMPs [5, 10, 11, 14, 15, 18, 20]. Li et al. [14] provide an analytical model and experiments to show to what extent parallel applications can be parallelized given a power-budget. Isci et al. [11] apply different DVFS policies under a given power budget and show that their best policy performs almost as good as an oracle policy having limited knowledge of the future. Meng et al. [18] proposes an adaptive power saving strategy that adheres to a global chip-wide power budget through run-time adaptation of configurable processor cores. Rangan et al. [20] propose ThreadMotion, a technique that moves threads around in order to improve power consumption. Their approach requires hardware support to quickly move threads from one core to another. Our approach is similar, but can be implemented on available CMPs without extra hardware support. Cai et al. [5] propose to identify critical threads by measuring the slack of threads at fork-join barriers; non-critical threads can then be executed at reduced speed. In our work, we focus on independent OSes as opposed to threads within parallel applications. Ma et al. [15] propose a scalable solution aiming at a mixed group of single-threaded and multi-threaded applications. Unlike our approach with is best-effort, they aim at minimal performance reduction while maintaining a global power budget.

The work most closely related to ours is a hierarchical power manager for the Intel SCC presented by Ioannou et al. [10]. We show that by adding OS migration a significantly improved performance/watt ratio can be achieved.

3 Many-core Architectures

Many-core architectures exhibit a number of typical characteristics [22] in order to effectively manage and utilize the large number of cores. In particular, many-core architectures feature an interconnection network to enable on-chip communication between the cores. This network is also employed when accessing shared resources such as memory. Atomic operations are provided to enable efficient synchronization of multiple cores.

The technique described in this paper does not require any special hardware support and is thus in principle applicable to any many-core architecture. We provide a working implementation on Intel's Single-Chip-Cloud Computer (SCC) [9] as a proof of concept. We leverage the SCC's two-level address translation (see next section) to implement zero-copy OS migration, but the same effect can be achieved – although with some additional overhead – by directly modifying a core's memory translation tables. The remainder of this section describes the architecture of the Intel SCC in more detail.

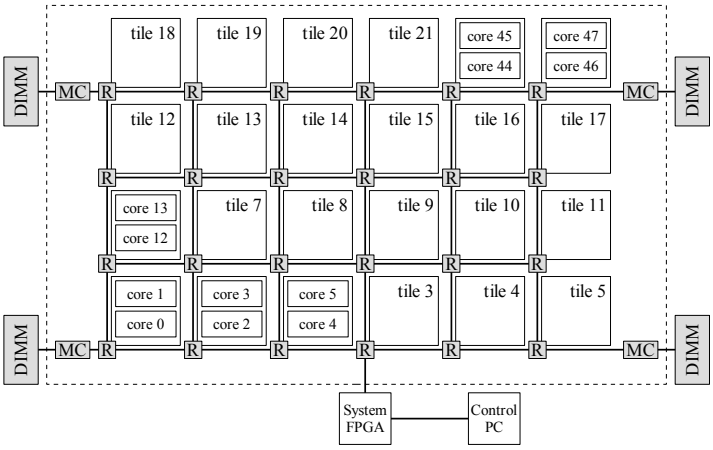

Fig. 1. Intel SCC block diagram

3.1 The Intel Single-chip Cloud Computer

Architecture Overview The Intel SCC is a concept vehicle created by Intel Labs as a platform for many-core research. It consists of 48 independent cores interconnected by a routed network-on-chip (NoC). The cores are Intel P54C Pentium® cores with bigger L1 caches (16KB) and additional support for managing the on-chip scratchpad memory, the so-called *message passing buffer* (MPB). The Intel SCC provides no cache coherence for the core-local L1 and L2 caches. Always two cores are grouped together to form a *tile*; the 24 tiles are organized on a 6 by 4 grid. Four memory controllers in the four corners of the chip provide access to up to 64 GBs of memory. A system FPGA provides the interface between the CMP and the management-console PC (MCPC). Figure 1 shows the SCC block diagram. For better readability, not all cores are shown.

Memory Addressing Each core provides the standard virtual-physical memory translation; all addresses leaving the core are 32-bit physical addresses. 32-bit addresses are not wide enough to address the entire 64-GB address range; to allow access to a total of 4 GB of memory located somewhere in the SCC's 64 GB address space, an additional address translation takes place.

The address translation from core (physical) addresses to system addresses is provided by a core-local lookup table (LUT). Each LUT has 256 entries and is indexed by the top eight bits of the 32-bit core address. Without going into much detail, a LUT entry contains an 8-bit destination ID `destID` designating one of the four memory controllers (MC), and 10 address bits that are prepended to the remaining 24 bits of the core address to form a 34-bit address. One LUT entry thus maps 16 GB of memory. Together with the memory controller designation, this translation allows to access the entire 64 GB memory space of the SCC.

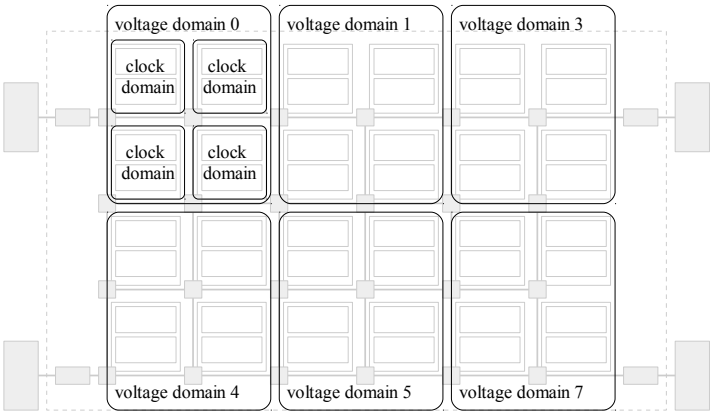

Fig. 2. Voltage and clock domains on the Intel SCC

DVFS Capabilities The SCC provides voltage and frequency control over the cores and the NoC. The frequency can be controlled *per tile*, that is, the two cores located on the same time always run at the same frequency and constitute a clock domain (CD). The voltage can be regulated for a group of four tiles, i.e., a voltage domain (VD) comprises a total of eight cores. Figure 2 illustrates the clock and voltage domains on the SCC. Voltage domains 2 and 6 are not shown in the figure; they are logically the same and regulate the NoC and the system interface.

Voltage and frequency are controlled and queried through registers. Each tile has specific registers to set/read the tile's frequency; voltage changes are similarly controlled through a register interface. Frequency changes happen almost immediately, however, measurements on the SCC revealed that voltage changes may take up to 100ms to complete (this value was obtained by comparing the progress of two cores, one control core running on a unchanged voltage domain, and one running on the domain whose voltage was changed). In addition, voltage change requests can only affect a single domain; requests for several domains must be serialized.

The SCC supports seven different supply voltage levels, however, only four are of practical interest: $1.1V$ to run at a frequency of 800MHz, $0.9V$ to run at 533MHz, $0.8V$ for 400MHz, and $0.7V$ for frequencies below 400MHz. The frequency is set by writing a divisor between 2 and 16 for the 1600MHz clock resulting in core frequencies from 800 to 100MHz.

Power Measurement The SCC provides a number of voltage and ampere meters on-board. The total power consumed by the SCC chip is obtained by multiplying the (constant) supply voltage with the supply current for the entire SCC chip. The power consumption of individual voltage domains cannot be

computed because only the per-domain supply voltage is available but not the current consumed by the domain. We thus always report the total chip power in our experiments in Section 7.

The sensors and meters can be read by the management console via telnet from the system FPGA or by directly querying the system FPGA from a core in the SCC. We chose the latter approach because of its comparatively low overhead.

4 Zero-copy OS Migration

In order to optimize the execution of workloads on a CMP towards a specific goal, the scheduler is often required to move workloads from one core to another. Such goals include but are not limited to even heat dissipation and adherence to a given power budget. In the first case, busy and idle workloads are distributed evenly over the cores of the CMP to even out the sources of the heat generation. The second case is motivated by the need to cluster workloads with similar performance requirements in voltage and/or frequency domains to achieve optimal results when applying DVFS.

A workload in this context can refer to anything from a thread of a parallel application to an entire operating system running exclusively on a core. Clustering threads or processes in the presence of an operating system with shared memory amounts to re-scheduling them on a different core is straight-forward. For applications exhibiting periodic behavior and fork-join parallel programs, special techniques allow accurate estimation of the expected performance requirements and thus more aggressive DVFS policies [10, 15, 20].

For independent programs (such as a Linux kernel) running bare-metal on the assigned cores, migration is not trivial. In this section, we describe the technical details on how such kernels can be migrated from one core to another, the scheduler's *migration policies* are discussed in the section on power management (Section 5).

4.1 Cooperative vs. Transparent Guest OS Migration

Moving an OS from one physical core to another can be implemented with or without cooperation of the migrated OS. In a co-operative setting, the OS enters a safe state in which it is moved to the newly assigned core and then resumed. The OS itself takes care of changed memory mappings and the like. Transparent OS migration, on the other hand, happens without any interaction or knowledge of the migrated OS.

The main caveat is how to deal with the volatile state, i.e., the assigned memory of the workload and values currently held in registers inside the CPU core. If the CMP implements a global shared address space, the assigned memory does not have to be moved physically; the same physical addresses are still valid on the new core. Since such designs cannot provide total isolation of independently running workloads, CMPs often implement an additional step in the

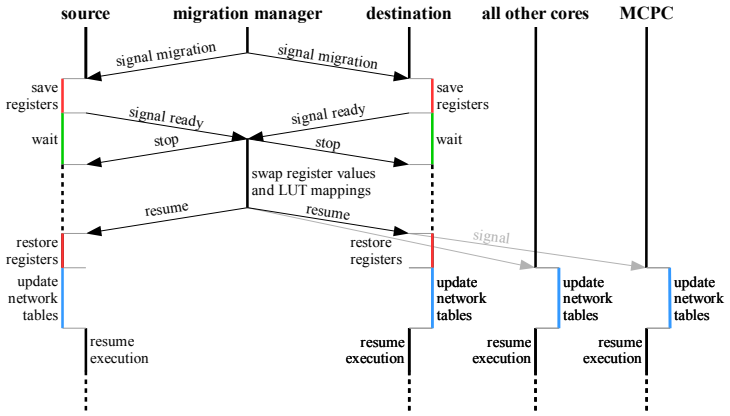

Fig. 3. OS Migration Steps

memory translation process from physical to system addresses. The physical-to-system address translation operates in almost the same way virtual-to-physical translation works: instead of per-process page tables, the CMP provides per-core translation tables indexed by the higher part of the core address (see Section 3.1). We exploit this additional translation step to realize an extremely light-weight migration of OSes.

The volatile state of the OS also includes the data values kept in the registers of the CPU core. Similar to a preemptive task switch, these register values need to be saved on the source core and restored on the destination core. If the CMP provides the means to read and write the register file of physical cores, migration can be implemented without any cooperation from the migrated OS. To this day, however, no many-core chip we are aware of provides such a feature. As a consequence, a minimal amount of cooperation from the OS is necessary.

The following sections describe the necessary steps and the implementation on the SCC in more detail.

4.2 Migration Steps

In the proposed implementation, zero-copy OS migration is orchestrated by a migration manager that is part of the global scheduler. The steps are illustrated in Figure 3. It reveals that migration is, in fact, rather a *circular swap* of two (or more) OSes rather than a unidirectional migration from one core to another. Since we require a minimal amount of cooperation from all involved cores, we assume that a cooperative OS runs on all (including the currently unused) cores. The migration signal is sent by the migration manager in form of an interrupt to the affected cores. This interrupt is handled by the cooperative OS' interrupt handler which saves the necessary registers into a per-core designated memory area. After all registers have been saved, the affected cores signal completion to

the migration manager and completely flush their caches. The migration manager then stops all cores involved in the migration by gating their clock, and swaps the cores' register values and the memory mappings. Next, the migration manager signals completion of the migration by resuming the clock on the migrated cores. The cores proceed by restoring the (new) register values from memory, exit the interrupt handler and resume operation. In addition, all cores, including the MCPC need to update internal network routing tables to reflect the new locations of the cores (see Section 4.4 below).

4.3 Migrating Volatile State

The migrated cores save/restore register values to a designated memory area in a custom interrupt handler. In principle, exactly the same registers need to be saved/restored as when performing a context switch in a preemptive multitasking system. After saving the registers, the migrated cores flush all caches and enter a busy loop. It is impossible to flush a core's cache externally; as a consequence the code of the busy loop will reside in a migrated core's instruction cache. In addition, it is impossible to set the program counter immediately after resuming the clock since we do not know what instruction of the busy loop the core was executing when the clock was gated. However, we can ignore this technical difficulty by assuring that the busy loop, including the code to save and restore the registers, is located at identical virtual addresses on all migrated cores. Since we currently only use a modified version of the sccLinux OS, this condition is always met. If several different OSes are involved in migration, it may be necessary to turn off virtual-to-physical memory translation temporarily and re-enable it once the new page table base register has been set.

4.4 Networking

Cores on a CMP communicate with other cores through the NoC. IP addresses are translated to the destination core's x/y coordinates on the grid in the data link layer of the network stack. Migrated OSes keep their IP addresses, hence additional steps are necessary to update the IP to link-layer translation tables in all cores on the chip.

On the SCC, two separate networks exist: one network for on-chip networking, and a subnet for communication with the MCPC. Data packets sent on-chip from one core to another are first stored in the MPB (see Section 3.1) on the sender side. The sender then signals the receiver with an interrupt, and the receiver fetches the message data directly from the sender's buffer. For migrated cores the location of their MPB remains unchanged; storing/retrieving network packets from the buffer is thus unaffected by migration. However, the target core of a network interrupt is identified by its physical core ID which corresponds to the x/y-coordinates of the core on the grid. In the original sccLinux the interrupt target ID is computed from the core ID. In order to support migration, a table containing the IP-to-coreID mappings is added and kept up-to-date by

each core. After each migration, the migration manager thus notifies *all* cores about the changes to the IP-to-coreID mapping table.

A very similar data structure is maintained by the MCPC to route data packets from external sources to each of the individual cores. The migration manager notifies the MCPC through the system interface about migrations taking place such that the MCPC can keep an up-to-date list of IP-to-coreIDs.

These two simple modifications are enough to keep networking, including open connections, alive across migrations. DMA is not supported, and no other devices exist on the SCC; input/output, including access to permanent storage, are routed through the network.

5 Hierarchical Power Management

In order to remain scalable, power management techniques for many-core CMPs are either organized in a hierarchical manner [10, 11, 15] or operate with independent local agents [6]. Our goal is to apply OS migration in order to improve the effectiveness of DVFS which necessitates a hierarchical design. This section describes the organization of the hierarchical power manager and the DVFS and migration policies in detail.

5.1 Organization

The structure of the hierarchical power manager reflects the structure of the SCC with its different voltage and frequency domains. At the lowest level in the hierarchy is a single core because individual cores exhibit different performance values. The next level represents a tile which comprises two cores and represents a clock domain. Decisions about which clock frequency to run at are made at this level. One level up is the voltage domain. A voltage domain consists of four tiles and represents the unit where voltage changes can be initiated. The highest level models the entire chip.

5.2 Local Performance Monitoring and Prediction

On each active core, a local agent monitors the current performance of the core. Depending on the load factor, it requests a higher, the same, or a lower frequency from the next-higher level in the hierarchy. The local agent uses the core's performance monitoring unit (PMU) to gather statistics about the number of executed and memory-bound instructions. At regular intervals the local agent predicts the load of a core based on a weighted average of sampled PMU data. When the core is not fully utilized, the optimal operating frequency can be analytically computed. For purely CPU-bound benchmarks, for example, and a utilization of 50% at 800MHz we expect 100% load at 400MHz. Frequency values are discreet, the computed frequency is thus always rounded up to the next higher available frequency.

If the core is fully utilized, however, it is not clear by how much the frequency should be increased. We have experimented with three policies: step-up one, step-up two, and half-way step-up. The first two policies increase the operating frequency by one and two steps, respectively, while the half-way step-up computes the requested frequency by adding half the difference of the current frequency to the maximal frequency. Experiments have shown that in our framework the performance and power consumption are indifferent in regard to the three policies.

Performance is measured periodically; experiments have shown that values between 0.5 and one second are short enough to quickly react to changing performance requirements, but long enough to avoid too much noise in the signal.

5.3 Domain Managers

Each domain, clock, voltage, and global, maintains its own domain manager. Each level only communicates directly with the level above or below, i.e., the clock domain manager interacts with the voltage domain manager, the voltage domain manager interacts downstream with the clock domain, and upstream with the global domain manager. The functionality of the different domain managers is elaborated in more detail in the following sections.

Clock Domain Manager For each clock domain, its clock domain manager computes and sets the appropriate frequency. The frequency of a clock domain is constrained by the current voltage level of the corresponding voltage domain and computed based on the performance counters reported by the local agents and the currently active DVFS policy (see Section 5.4 below). Each clock domain manager maintains sorted lists of the current and requested frequencies for all of its cores. The clock domain managers communicate with their voltage domain manager by periodically sending the list of requested frequencies. The voltage domain manager signals changes in the voltage level.

Voltage Domain Manager The voltage domain manager computes and sets the operating voltage of a voltage domain. Due to the nature of DVFS, voltage changes must happen in close collaboration with frequency changes: before lowering the voltage, all frequencies must be lowered to a values supported by the lower voltage. Similarly, for higher voltages, the voltage must be increased before the frequencies can be raised. Similar to the clock domain managers, voltage domain managers also maintain sorted lists of the current and requested voltages per clock domain. The voltage domain managers communicate with the global manager by periodically sending the list of requested frequencies and voltages upstream.

Global Domain Manager The global manager gathers the sorted voltage/ frequency requests from the domain managers and determines which cores to

migrate based on the migration policy. After migration has completed, the global domain managers informs the voltage domain managers of the migration such that the voltage may be changed immediately. This is not absolutely necessary since the information will eventually be sent from the local agents to the voltage domain managers, however, giving the voltage domain managers a chance to react immediately to migration leads to better results.

5.4 Scheduler Power Management Policies

The goal of the scheduler's power management policy is to optimize the *performance per watt* ratio of the overall chip. Other policies, such as, for example, even heat dissipation or adhering to a given power budget, are part of future work.

The power management policy is implemented in the global domain manager. The core migration and DVFS algorithms are invoked at regular intervals by the scheduler. The DVFS and migration policy, though the former depends on the latter, are completely separated in order to be able to freely combine different migration and DVFS policies. The following sections describe our DVFS and migration policies in detail.

OS Migration Policy Without migration OSes are pinned to their cores. For voltage domains containing both very busy cores and idle cores there is no optimal voltage setting: if the voltage is too low, idle cores run at the optimal frequency but the performance of busy cores is severely affected because the low voltage prevents the frequency domain manager from selecting the required frequency. On the other hand, if the voltage is set high enough to satisfy the performance needs of busy cores, idle cores waste energy because they operate at a higher than necessary voltage.

OS migration enables consolidation of cores with similar performance requirements into one voltage/frequency domain. This allows setting the voltage/frequency of the domain to a value that is close to the optimal value for most involved cores.

A naïve algorithm is to sort the OSes by their performance requirements and then assign them in order to the voltage and frequency domains. While the resulting allocation of cores to domains is optimal for one time quantum, this algorithm fails to consider the overhead of OS migration. The actual live migration of an OS is very quick ($\leq 3ms$), each time an OS is migrated it will experience a lot of cold misses in the local instruction and data caches which will lead to both a performance reduction as well as increased memory traffic. The migration algorithm must thus also consider the current positions of the OSes and minimize the number of migrations.

We currently employ a buyer-seller heuristic where domain managers for a given target frequency put up cores for sale that are expected to require a lower than the given target frequency. A market manager then matches the sellers to buyers. At the moment, the market manager has knowledge of all voltage domains; however, a hierarchical model is possible if the number of voltage domains

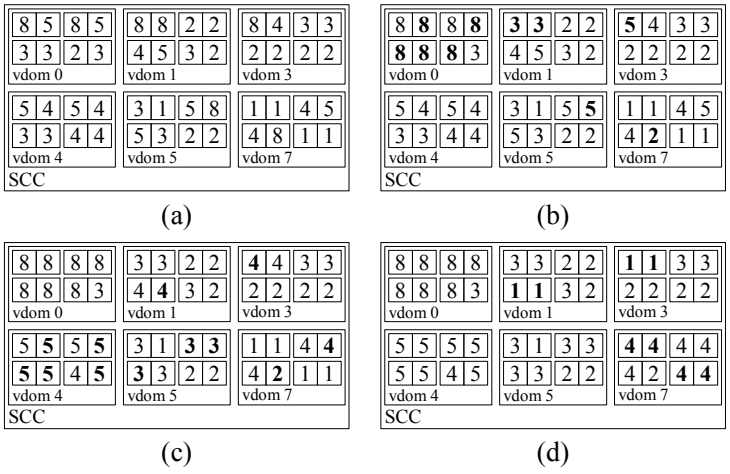

Fig. 4. Buyer-seller algorithm: (a) initial configuration, (b) configuration after running the first round for $v = 8$, (c) configuration after running the second round for $v = 5$, (d) final configuration after running the last round for $v = 4$. Bold values represent tiles/cores migrated in that iteration.

prohibits a global analysis. A limitation of the current heuristic is that it does not consider the location of a core's data. Developing a memory-location and contention aware OS migration policy is part of future work.

Figure 4 shows the effect of our heuristic. Figure 4 (a) displays the estimated frequencies for each core before the buyer-seller algorithm starts. Figures 4 (b)-(d) show the layout after each repetition for $v_i = 8, 5$ and 4, respectively; (d) represents the final configuration.

The heuristic returns the instructions to perform the actual migration in form of several circular lists of cores that are to be migrated. This list is then processed by the migration manager as discussed in Section 4.2.

DVFS Policies We implement two DVFS policies that are similar to policies used in the hierarchical power manager for CMPs proposed by Ioannou *et al.* [10]. Their work has been implemented on the Intel SCC chip and thus provides a good reference point. The policies proposed in [10] and reproduced here are:

- `Allhigh`: this policy runs all cores within a voltage domain at the highest requested frequency.
- `Tile`: grants the requested frequency to each clock domain and sets the voltage accordingly. Within each clock domain, the higher of the requested frequencies is chosen. Note: in [10] this policy is denoted `Simple`.

We have not implemented the `Alllow` and `Allmean` policies since they sacrifice too much performance in return for power savings.

For both policies the voltages of the voltage domains are computed such that all clock frequencies of the associated clock domains can be satisfied, i.e., $v_{VD} = \max_i v(f_{CDi})$. $v(f)$ for a given frequency f is a simple table lookup.

Phase Ordering and Frequency Considerations In order to achieve maximum power savings, migration should occur before applying DVFS. The frequency of migration, voltage, and frequency changes is determined by the cost of these operations: the time for migration is largely unaffected by the number of cores being migrated because all involved cores can store/restore the volatile state in parallel. Migrated cores are stopped and have their caches flushed while unaffected cores continue to run during migration operations. Voltage changes are quite expensive because the clock of all affected cores is stopped during the rather long voltage adjustment. Frequency changes, on the other hand, are almost instantaneous and can thus be performed often. On the SCC specifically, we have measured the following latency: $\leq 3ms$ for migration and $\leq 10ms$ for voltage changes. On this particular architecture, migration is cheaper than voltage changes. In addition, the SCC only supports one voltage change at a time; i.e., different domains cannot change the voltage in parallel. Nevertheless, for our server/desktop benchmark scenarios with rather slow changes in the CPU load, migration and voltage changes can be performed at every step. Section 7 discusses the benchmarks and results in more detail.

6 Experimental Setup

All experiments were conducted on an Intel Single-chip Cloud Computer. The scheduler runs on a dedicated core in voltage domain 3 on the SCC itself. All cores from the other voltage domains run a modified version of the sccLinux.

In addition to the scheduler we also run a few monitoring and logging processes on dedicated cores in voltage domain 3. In order not to pollute the migration algorithm with these processes, voltage domain 3 is excluded from the hierarchical power management. However, the reported results show the *total chip power* and therefore also include the power consumed by `vdom3`.

A benchmark scenario comprises a number of OSes with distinct workloads and an initial placement of the different OSes onto the SCC's cores.

The workloads running on the cores are either synthetic workloads used to demonstrate the operation of the proposed scheduler or represent profiled workloads that we have gathered by profiling 20 desktop and development computers of graduate students over a period of several months.

The baseline of the experiments is obtained by running the benchmark scenario on the SCC at full speed (800MHz) with no power management enabled. Unlike the work in [10] we do not use a phase-detector based on message passing since we are aiming at independent OSes running on a CMP. Instead, we apply the workload prediction method based on a weighted average. We compare the DVFS policies of [10] without OS migration, `Allhigh` and `Tile`, against the same policies with OS migration.

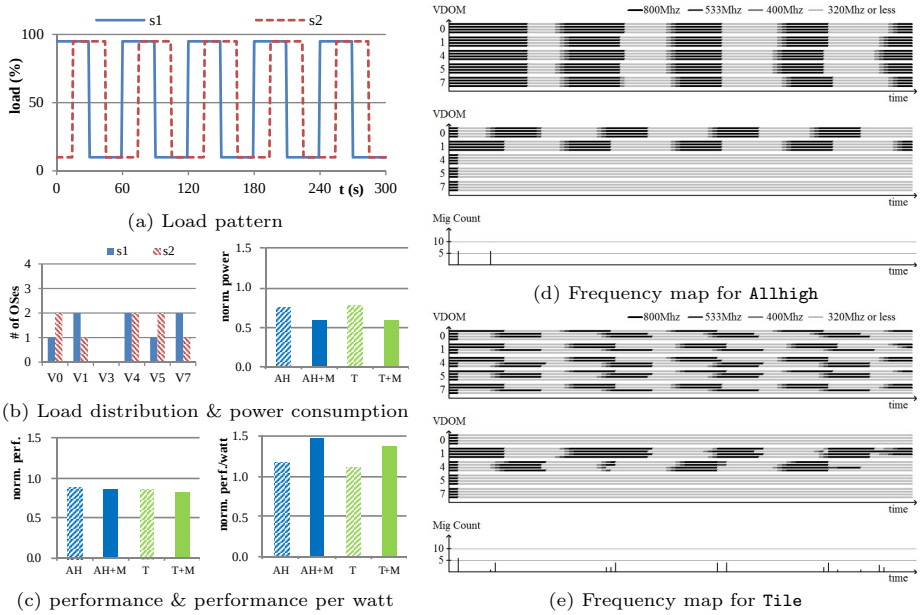

Fig. 5. Results for a simple alternating synthetic load

7 Results

We have conducted a wide range of experiments on the proposed scheduler. To show the effect of OS migration on DVFS, we first present the results of a synthetic periodic workload. The second example details the results of applying the proposed method on a workload pattern obtained from desktop machines of our graduate students. We conclude this section with the overall results over all benchmark scenarios.

Synthetic Periodic Workload The setup and the results of this workload are shown in Figure 5. The load pattern is shown in Figure 5 (a) and consists of two simple periodic synthetic workloads that alternate between 10% and 90% CPU load. The second workload `s2` is slightly time-shifted compared to `s1`. The initial OS distribution onto the different voltage domains of the SCC is shown in the left chart of Figure 5 (b). Each domain initially contains three or four OSes running one of the load patterns.

The results of running this benchmark scenario are shown in Figure 5 (b) and (c). The right-hand of Figure 5 (b) and Figure 5 (c) show the normalized power consumption, performance, and performance per watt, respectively, for the `Allhigh` and the `Tile` policy, denoted `AH` and `T`, without and with (appended `+M` postfix) OS migration.

We observe that both DVFS only and DVFS+migration suffer from a performance loss. In the case of DVFS only, there are two reasons for this loss:

first, a voltage change operation of an island stops execution on all cores, adjusts the voltage, and then resumes the core clock. This process is implemented in hardware and takes about 10ms per voltage change. The second reason for reduced performance is observed when the workload prediction model fails to predict a sudden raise in the load and selects a too low operating frequency for a workload. OS migration incurs additional overhead: migration requires stopping and resuming all involved cores. Additionally, the changes in the routing tables are propagated to all active OSes through external interrupt; processing these interrupts on the individual cores also causes a minimal performance overhead.

Nonetheless, as can be seen from the normalized power consumption in Figure 5 (b) on the right-hand side, the reduction in power by far outweighs the loss in performance. In terms of performance per watt (right-hand of Figure 5 (c)), both DVFS only and DVFS+migration show better results. In particular, the proposed method of combining DVFS with OS migration achieves about a 30% improved performance per watt compared to DVFS-only policies.

Figures 5 (d) and (e), finally, visualize the effect of DVFS only and DVFS+ migration on the individual voltage domains' frequency settings for the two DVFS policies `Allhigh` and `Tile`. The frequency over time is shown for each voltage domain for DVFS only (upper part) and DVFS+migration (middle part of the figure). Higher frequency (and thus voltage) settings are represented by darker levels of gray. The lower part of the chart shows the number of migrations over time.

Comparing DVFS and DVFS+migration with the `Allhigh` DVFS policy (Figure 5 (d)) clearly shows how migration is able to group OSes with similar performance characteristics together and thus select voltages that are closer to the optimal value. In the `Allhigh` policy in particular, if only one core in a particular voltage domain requests a frequency of 800MHz and thus the highest voltage setting, the entire domain will run at $1.1V$. Since in this artificial example the OSes are evenly spread over all voltage domains, without OS migration all domains run at maximum voltage most of the time. In comparison to DVFS+migration, we clearly observe that the migration policy first migrates all OSes into the first two domains, `vdom0` and `vdom1`. About 30 seconds into the benchmark, the OSes running load pattern `s1` drop to 10% load which causes another batch of migrations and results in grouping the OSes running the same load pattern together. After this, the OSes running in the same voltage domain observe similar load patterns and no more migrations are necessary. The DVFS policy can select the appropriate voltage and frequency for the first two domains.

For the `Tile` DVFS policy we observe a similar pattern (Figure 5 (e)). Here, the frequency can be set on a tile-basis. Again, DVFS only cannot consolidate OSes with similar load patterns, resulting in voltage settings that are too high for most cores in a domain. DVFS+migration, on the other hand, groups all OSes into `vdom1` and `vdom4`. We see that migration fails to group the OSes running identical load patterns into distinct domains at first which causes some migration activity after about one third of the benchmark's runtime. From then on, the two load patterns are nicely separated. Even though the load patterns are perfectly

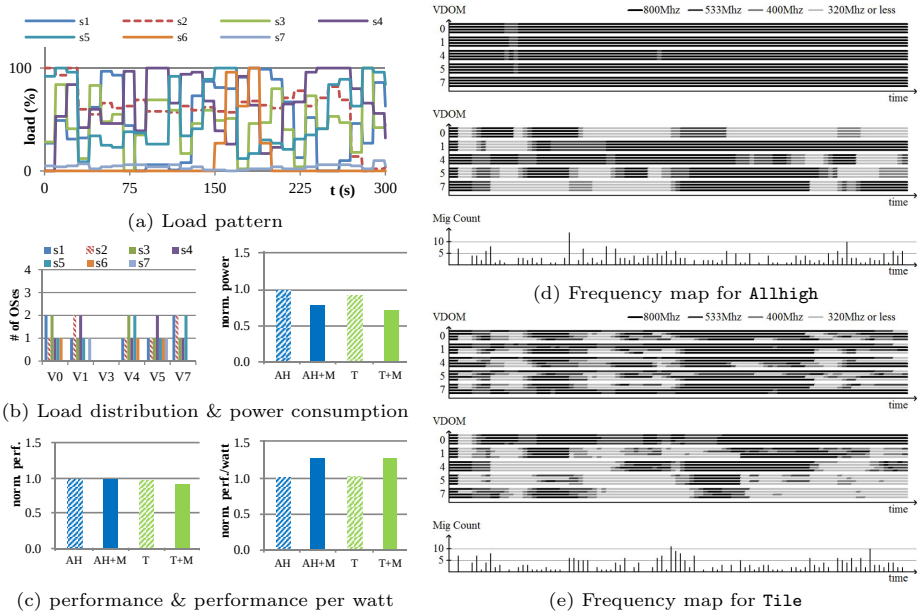

Fig. 6. Results for a profiled load pattern

synchronized at the beginning of the run, the overhead of DVFS and migration causes them to drift apart slowly which then again triggers migrations.

Profiled Workload Figure 6 shows the results of a scenario based on actual, measured workloads patterns. Seven different load patterns obtained from profiling data of graduate students' computers, `s1` to `s7`, are assigned to a total of 40 OSes and initially placed onto the different voltage domains.

Compared to the synthetic workload, the performance loss (left-hand in Figure 6 (c)) is much less severe (1% and 2.5% for `AH` and `AH+M`, and 3% and 8% for `T` and `T+M`, respectively). This is because profiled workloads exhibit smoother workload changes; the local performance prediction is thus much more accurate. The DVFS-only policies cannot group OSes with similar workload characteristics together, and all voltage domains run at maximal voltage during most of the benchmark (upper-hand VDOM charts in Figures 6 (d) and (e)). As a consequence only minimal total energy savings are obtained (1% and 7% for `AH` and `T`).

With OS migration, however, the scheduler is able to group workloads exhibiting similar load patterns into voltage domains as shown by the lower-hand VDOM charts in Figures 6 (d) and (e). The total energy savings are significant (23% and 26% for `AH+M` and `T+M`) and lead to a much better performance per watt increase compared to DVFS only (1% and 4% for DVFS only, 26% and 26% for DVFS+migration).

Table 1. Normalized power, performance, and performance per watt (PPW)

	AH			**AH+M**			**T**			**T+M**		
BM	Power	Perf	PPW	Power	Perf	PPW	Power	Perf	PPW	Power	Perf	PPW
1	99.2	99.9	100.8	77.2	97.7	126.5	73.6	97.4	104.0	74.2	93.5	126.0
2	88.2	98.8	112.0	62.6	96.3	153.8	80.3	96.6	120.2	62.2	94.0	151.2
3	92.1	99.5	108.0	63.7	100	157.2	84.9	97.8	115.3	62.5	96.4	154.1
4	98.1	99.9	101.9	77.1	98.6	127.9	91.9	96.5	105.0	71.7	90.7	126.6
Avg.	**94.4**	**99.5**	**105.7**	**70.2**	**98.2**	**141.4**	**82.7**	**97.1**	**111.1**	**67.7**	**93.7**	**139.5**

Overall Results Table 1, finally, displays the normalized power, performance, and performance per watt over the baseline, respectively, for the `Allghigh` and the `Tile` policy, denoted `AH` and `T`, without and with (appended `+M` postfix) OS migration for three different benchmark scenarios all based on profiled workload patterns (details the benchmark scenarios are given in Appendix A).

Independent of the workload at hand, migration OSes before applying a DVFS policy results in a significantly reduced power consumption at the expense of a very moderate performance degradation. Taking the DVFS-only policy as the baseline, `Allhigh+Migration` achieves a 36% better power-per-watt energy efficiency than `Allhigh` at a relative performance loss of only 1.3%. Similarly, `Tile+Migration` outperforms `Tile` by 28.4% at a performance loss of 3.4%. We observe that `Tile` outperforms `Allhigh` without migration whereas with migration they achieve similar performance. The reason is that OS migration is able to group OSes with similar performance requirements into voltage domains such that the superior `Tile` DVFS policy has less effect.

8 Conclusion

In this work, we show that energy-aware space-shared scheduling of independent programs running on a CMPs is feasible and the potential to achieve significantly energy savings. We provide a working implementation on the Intel Single-chip Cloud Computer where the individual OSes run bare-metal on the assigned cores, and the global scheduler communicates with cores through interrupts.

Techniques for reducing power consumption of CMPs rely on well-known DVFS techniques. The special organization of CMPs into frequency and voltage domains makes direct application of previous work difficult. We employ zero-copy OS migration implemented in a energy-aware scheduler to consolidate OSes with similar workloads onto the same frequency and voltage domains, thereby allowing DVFS policies to achieve a larger power reduction at the cost of a minimal performance penalty.

The proposed energy-aware scheduler with its integrated hierarchical power management supporting intra-core live migration is put to a test with a wide range of workloads. Experimental results conducted on a real system show that, on average, the proposed techniques achieve an improvement of the performance per watt by 25-35% over previous DVFS approaches.

Acknowledgments

This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2012R1A1A1042938 and NRF-2008-0062609). ICT at Seoul National University provided research facilities for this study.

References

1. Tilak Agerwala and Siddhartha Chatterjee. Computer architecture: Challenges and opportunities for the next decade. *IEEE Micro*, 25(3):58–69, 2005.
2. Luiz André Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 282–293, New York, NY, USA, 2000. ACM.
3. 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.
4. Thomas D. Burd and Robert W. Brodersen. Energy efficient cmos microprocessor design. In *Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences*, volume 1, pages 288–297, Jan 1995.
5. Qiong Cai, José González, Grigorios Magklis, Pedro Chaparro, and Antonio González. Thread shuffling: Combining dvfs and thread migration to reduce energy consumptions for multi-core systems. In *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design*, ISLPED '11, pages 379–384, Piscataway, NJ, USA, 2011. IEEE Press.
6. Thomas Ebi, Mohammad Faruque, and Jörg Henkel. Tape: Thermal-aware agent-based power econom multi/many-core architectures. In *IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers*, ICCAD 2009, pages 302–309, Nov 2009.
7. Soraya Ghiasi. *Aide De Camp: Asymmetric Multi-core Design for Dynamic Thermal Management*. PhD thesis, Boulder, CO, USA, 2004. AAI3136618.
8. Sebastian Herbert and Diana Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *2007 ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, pages 38–43, Aug 2007.
9. J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van der Wijngaart, and T. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109, Feb 2010.
10. Nikolas Ioannou, Michael Kauschke, Matthias Gries, and Marcelo Cintra. Phase-based application-driven hierarchical power management on the single-chip cloud computer. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 131–142, Washington, DC, USA, 2011. IEEE Computer Society.
11. Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the 39th*

Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 39, pages 347–358, Washington, DC, USA, 2006. IEEE Computer Society.

12. Wonyoung Kim, Meeta S. Gupta, Gu-Yeon Wei, and D. Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *IEEE 14th International Symposium on High Performance Computer Architecture (HPCA 2008)*, pages 123–134, Feb 2008.
13. Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 64–, Washington, DC, USA, 2004. IEEE Computer Society.
14. Jian Li and José F. Martínez. Power-performance implications of thread-level parallelism on chip multiprocessors. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2005)*, pages 124–134, March 2005.
15. Kai Ma, Xue Li, Ming Chen, and Xiaorui Wang. Scalable power control for many-core architectures running multi-threaded applications. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 449–460, New York, NY, USA, 2011. ACM.
16. David Meisner, Brian T. Gold, and Thomas F. Wenisch. Powernap: Eliminating server idle power. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 205–216, New York, NY, USA, 2009. ACM.
17. David Meisner and Thomas F. Wenisch. Dreamweaver: Architectural support for deep sleep. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 313–324, New York, NY, USA, 2012. ACM.
18. Ke Meng, Russ Joseph, Robert P. Dick, and Li Shang. Multi-optimization power management for chip multiprocessors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 177–186, New York, NY, USA, 2008. ACM.
19. Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 2–11, New York, NY, USA, 1996. ACM.
20. Krishna K. Rangan, Gu-Yeon Wei, and David Brooks. Thread motion: Fine-grained power management for multi-core systems. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 302–313, New York, NY, USA, 2009. ACM.
21. Efraim Rotem, Avi Mendelson, Ran Ginosar, and Uri Weiser. Multiple clock and voltage domains for chip multi processors. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 459–468, New York, NY, USA, 2009. ACM.
22. András Vajda. Multi-core and many-core processor architectures. In *Programming Many-Core Chips*, pages 9–43. Springer, 2011.

Appendix A Profiled Workload Benchmark Scenarios

This appendix describes the details of the benchmarks evaluated in this work. Each benchmark scenario consists of two parts:

- Two or more workload pattern that describe how the workload changes over time.
- An initial assignment of the workloads to the 48 cores of the exercised Intel SCC.

Each workload pattern (WL), denoted S{1-7} in the tables below, lists the CPU workload for every epoch (10 or 15 seconds, depending on the benchmark) for the duration of one period (300 seconds). A workload never stops, it keeps repeating the workload pattern period after period. Note that all workloads are pure CPU-based workloads; memory-based workloads are part of future work.

The core assignment tables below show what workload pattern are assigned to which cores when the experiment starts. In our setup, voltage domain 3 runs various logging and monitoring services and is thus not available for user benchmarks. The power measurements include the power consumed by `vdom3` because power is only reported for the entire chip and not for individual voltage domains.

A benchmark ends after a predefined number of seconds (in our example after 300 seconds). The total progress of each workload is measured externally and thus includes all overheads caused by migration, voltage changes or slowdowns cause by too low frequency settings.

A.1 Synthetic Benchmark Scenario based on Periodic Workloads

The synthetic benchmark consists of two identical workload patterns shifted in time. Each voltage domain contains workloads of both patterns. The purpose of this benchmark is to demonstrate the potential of combining DVFS with OS migration. The results of this benchmark are shown in Figure 5.

Workload patterns:

WL	Epoch (1 epoch = 15 sec)																			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
S1	95	95	10	10	95	95	10	10	95	95	10	10	95	95	10	10	95	95	10	10	10
S2	10	95	95	10	10	95	95	10	10	95	95	10	10	95	95	10	10	95	95	10	10

Core assignment:

vdom0		vdom1		vdom3		vdom4		vdom5		vdom7	
-	-	-	-	n/a	n/a	-	-	-	-	-	-
S2	-	S2	-	n/a	n/a	S2	S2	S2	-	S2	-
-	-	-	-	n/a	n/a	-	-	-	-	-	
S1	S2	S1	S1	n/a	n/a	S1	S1	S1	S2	S1	S1

A.2 Benchmark Scenarios based on Profiled Workloads

The following four benchmarks are based on the usage patterns of Linux and Windows desktop computers. Initially, each voltage domain is loaded with different workload patterns. These benchmarks demonstrate the effect of the proposed technique when applied to a multi-user setup (i.e., virtual desktops of employees on a server machine).

The detailed result of the first benchmark are shown in Figure 6, and Table 1 lists the combined results for all four benchmark scenarios shown here.

Benchmark 1 (BM1) Workload patterns:

WL	Epoch (1 epoch = 10 sec)																													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
S1	27	49	31	32	62	77	80	44	0	6	1	1	8	73	87	81	80	91	100	99	89	67	13	52	0	0	10	46	27	86	63
S2	69	57	68	60	55	66	61	63	69	58	56	57	63	59	62	58	57	67	68	64	61	71	78	63	71	82	69	14	0	2	4
S3	28	84	41	12	83	48	55	0	35	69	42	59	17	46	59	49	51	2	46	47	80	40	4	73	41	53	47	18	100	42	45
S4	27	49	31	32	62	77	80	44	0	6	1	1	8	73	87	81	80	91	100	99	89	67	13	52	0	0	10	80	66	56	32
S5	71	53	26	9	34	25	23	38	37	26	96	92	34	41	89	100	100	12	17	30	27	21	31	35	41	84	89	63	100	96	84
S6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	27	96	63	100	27	0	0	0	0	0	0	0	0	0	0	0
S7	5	4	5	7	2	4	5	6	6	4	100	6	2	4	1	1	0	1	2	2	4	2	2	4	6	6	6	5	2	10	5

Core assignment:

vdom0		vdom1		vdom3		vdom4		vdom5		vdom7	
S4	S6	S4	S4	n/a	n/a	S5	S5	S5	S6	S5	S5
S3	S3	S3	S7	n/a	n/a	S3	S3	S4	S4	S2	S2
S2	S5	S2	S2	n/a	n/a	S2	S6	S2	S7	S3	S4
S1	S1	S1	S5	n/a	n/a	S1	S4	S1	S3	S1	S1

Benchmark 2 (BM2) Workload patterns:

WL	Epoch (1 epoch = 10 sec)																													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
S1	27	49	31	32	62	77	80	44	0	6	1	1	8	73	87	81	80	91	100	99	89	67	13	52	0	0	10	46	27	86	63
S2	82	39	55	42	96	42	100	33	53	20	20	10	11	14	13	11	13	13	1	5	1	0	23	45	61	42	83	83	20	15	3
S3	28	84	41	12	83	48	55	0	35	69	42	59	17	46	59	49	51	2	46	47	80	40	4	73	41	53	47	18	100	42	45
S4	27	49	31	32	62	77	80	44	0	6	1	1	8	73	87	81	80	91	100	99	89	67	13	52	0	0	10	10	15	30	27
S5	71	53	26	9	34	25	23	38	37	26	96	92	34	41	89	100	100	12	17	30	27	21	31	35	41	84	89	63	100	96	84
S6	53	21	52	48	33	92	89	100	39	38	29	41	48	4	64	45	36	31	42	41	42	35	15	80	93	62	10	23	48	32	0

Core assignment:

vdom0		vdom1		vdom3		vdom4		vdom5		vdom7	
-	-	-	-	n/a	n/a	-	-	-	-	-	-
S5	S6	S5	S6	n/a	n/a	S3	S6	S4	S5	S4	S5
-	-	-	-	n/a	n/a	-	-	-	-	-	-
S1	S4	S1	S2	n/a	n/a	S1	S2	S2	S3	S1	S3

Benchmark 3 (BM3) Workload patterns:

WL	Epoch (1 epoch = 10 sec)																													
	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
S1	42	77	25	11	34	36	30	14	33	26	22	58	100	52	30	13	15	0	21	39	48	43	40	41	40	42	41	40	39	36	35
S2	45	15	6	27	25	9	64	55	27	28	18	51	46	100	56	20	25	25	12	0	0	0	0	0	0	0	0	0	0	0	0
S3	71	53	26	9	34	25	23	38	37	26	30	23	34	41	39	29	29	12	17	30	27	21	31	35	41	84	89	63	100	96	2
S4	11	22	20	10	27	12	45	100	22	9	4	14	9	43	19	6	17	18	14	21	5	5	5	6	25	16	7	0	0	0	0
S5	42	66	40	67	57	67	66	71	75	72	31	38	59	54	86	80	68	55	95	100	89	85	86	77	64	0	0	0	0	0	0

Core assignment:

vdom0		vdom1		vdom2		vdom4		vdom5		vdom7	
S5	-	-	-	n/a	n/a	S5	-	S5	-	S5	-
-	-	S5	-	n/a	n/a	S4	-	S4	-	S4	-
S2	S4	S2	S4	n/a	n/a	-	S3	S2	S3	S2	-
S1	S3	S1	S3	n/a	n/a	S1	S2	S1	-	S1	S3

Benchmark 4 (BM4) Workload patterns:

WL	Epoch (1 epoch = 10 sec)																													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
S1	27	49	31	32	62	77	80	44	0	6	1	1	8	73	87	81	80	91	100	99	89	67	13	52	0	0	10	46	27	86	63
S2	82	39	55	42	96	42	100	33	53	20	20	10	11	14	13	11	13	13	1	5	1	0	23	45	61	42	83	83	20	15	3
S3	8	20	21	30	80	100	24	50	36	54	83	92	91	73	27	1	0	1	1	1	1	0	1	1	10	1	21	17	33	5	7
S4	27	49	31	32	62	77	80	44	0	6	1	1	8	73	87	81	80	91	100	99	89	67	13	52	0	0	10	10	15	30	27
S5	53	21	52	48	33	92	89	100	39	38	29	41	48	4	64	45	36	31	42	41	42	35	15	80	93	62	10	23	48	32	0

Core assignment:

vdom0		vdom1		vdom2		vdom4		vdom5		vdom7	
-	-	-	-	n/a	n/a	-	-	-	-	-	-
S3	S4	S3	S4	n/a	n/a	S3	S4	S3	S4	S3	S4
-	S5	-	S5	n/a	n/a	-	S5	-	S5	-	S5
S1	S2	S1	S2	n/a	n/a	S1	S2	S1	S2	S1	S2