Karavadara *et al. EURASIP Journal on Embedded Systems* (2016) 2016:14
DOI 10.1186/s13639-016-0035-9

EURASIP Journal on Embedded Systems

RESEARCH Open Access

Dynamic power management for reactive stream processing on the SCC tiled architecture

Nilesh Karavadara[1*], Michael Zolda[1], Vu Thien Nga Nguyen[1], Jens Knoop[2] and Raimund Kirner[1]

Abstract

Dynamic voltage and frequency scaling (DVFS) is a means to adjust the computing capacity and power consumption of computing systems to the application demands. DVFS is generally useful to provide a compromise between computing demands and power consumption, especially in the areas of resource-constrained computing systems. Many modern processors support some form of DVFS.

In this article, we focus on the development of an execution framework that provides lightweight DVFS support for reactive stream processing systems (RSPS). RSPs are a common form of embedded control systems, operating in direct response to inputs from their environment. At the execution framework, we focus on support for many-core scheduling for parallel execution of concurrent programs. We provide a DVFS strategy for RSPs that is simple and lightweight, to be used for dynamic adaptation of the power consumption at runtime. The simplicity of the DVFS strategy became possible by the sole focus on the application domain of RSPs. The presented DVFS strategy does not require specific assumptions about the message arrival rate or the underlying scheduling method.

While DVFS is a very active field, in contrast to most existing research, our approach works also for platforms like many-core processors, where the power settings typically cannot be controlled individually for each computational unit. We also support dynamic scheduling with variable workload. While many research results are provided with simulators, in our approach, we present a parallel execution framework with experiments conducted on real hardware, using the single-chip cloud computer many-core processor. The results of our experimental evaluation confirm that our simple DVFS strategy provides potential for significant energy saving on RSPs.

Keywords: Tiled architectures, Stream-processing, Reactive systems, Many-core processor, Power management, DVFS

1 Introduction

With the ever-increasing demand of applications for computational performance, tiled architectures have become a promising new technology for parallel processing that combine the computational power of GPUs/DSPs with the flexibility of synchronous multiprocessing. In particular, tiled architectures were designed to address the problems of long line lengths and performance bottlenecks caused by shared resources that have been occurring in traditional symmetric multi-core designs with the increasing number of cores. At the same time, tiled architectures try to retain the flexibility of traditional desktop microprocessor architectures.

The decentralised nature of tiled architectures requires programming and execution models that differ from those used for symmetric multi-core architectures. Stream processing is a promising parallel execution model that is well-suited for modern tiled many-core architectures, as it embraces distributed processes that interact solely through explicit, unidirectional data streams. High-level coordination languages based on principles from dataflow programming allow software engineers to rapidly build maintainable parallel applications from sequential building blocks. A lightweight load-balancing middleware layer can then be used to effectively schedule the execution of these building blocks on tiled architectures.

Reactive stream processing systems (RSPs) designate the case where stream processing is used to implement reactive systems, i.e. systems that operate in direct

*Correspondence: n.karavadara@herts.ac.uk
[1] School of Computer Science, University of Hertfordshire, AL109AB Hatfield, UK
Full list of author information is available at the end of the article

response to inputs from their environment. An example of RSPs is real-time video encoding, where the encoder needs to process incoming video frames as they arrive.

Optimisation of power consumption is important, especially if RSPs are used in embedded resource-constrained environments. For example, if the system load imposed by the environment varies over time, the use of dynamic power management techniques like *dynamic voltage and frequency scaling* (DVFS) can be used to effectively reduce the power consumption of such systems. Expanding on the example of real-time video encoding, the computational resources needed vary greatly depending on the captured scenery: detailed sceneries with fast, complex movements are much more computationally intensive than still scenes with few details.

It is important to note that this situation is very different from non-reactive stream processing, where the system is not subject to such stringent timing constraints. The challenge in dynamic power management for reactive stream processing is to save power at the right time, without violating the system's throughput and latency constraints. This is a hard problem, because the future resource requirements can be unpredictable for many important applications.

In this article, we present an execution framework that provides lightweight DVFS support for RSPs. As described in Section 2, most DVFS research, in contrast to our approach, have been targeted towards generic systems with observation of the system state at OS or message passing interface (MPI) level and/or are limited to offline approaches with static workload. Our specialisation to RSPs allows to deploy a rather simple DVFS strategy, which is fast enough to optimise power consumption dynamically during runtime. And while many research results are provided with simulators, in our approach, we present a parallel execution framework with experiments contacted on real hardware, using the single-chip cloud computer (SCC) many-core processor as a concrete example. The SCC provides 48 cores and a flexible infrastructure for DVFS. Compared to other commercially available tiled architectures, it supports more independent on-die voltage and frequency domains. Moreover, since the SCC is based on IA-32 cores, legacy benchmark code can easily be ported. A brief overview of the SCC architecture is given in Section 3. The programming and execution model we used to test our framework is described in Section 4.

Our lightweight strategy for DVFS on RSPs, which is described in Section 5, reduces the frequency and voltage in case that the system has more than enough resources available to cope with the current input rate. It increase frequency and voltage in case when the load changes and the system becomes overloaded. To do so, we observe the input and output rates of the RSPs, as well as the number of workers waiting for new work to predict overload and underload situations. We then use this information for DVFS so that the system can cope with the input rate and save energy at the same time. Our evaluation in Section 6 indicates that our DVFS strategy for RSPs can reduce energy consumption significantly, even down to half of the non-regulated one. Section 7 concludes this research.

2 Related work

DVFS has attracted several research projects for its applications in efficient power management. The majority of the work in this area aims to minimise energy consumption by dynamically adjusting the voltage and frequency of CPUs depending on the state of the system at different levels.

By observing memory operations like *memget*, *memput*, and *memcpy*, Gamell et al. in [1] identify slack periods of partitioned global address space (PGAS) applications during these operations. By exploiting these slack periods, the authors propose a power management middleware to allow adjusting the power configuration at both PGAS application and runtime layers. The runtime layer makes decisions for power configurations based on the combination of predefined thresholds and adaptive thresholds based on the history. At the application layer, the middleware allows the programmer to use language extensions to define different power/performance policies including maximum performance, balance power/performance, minimum power, and maximum power reduction. The middleware is facilitated with a power controller to dynamically adjust the voltage and frequency depending on the power configuration. To achieve the desired configuration, the power controller can operate in different modes including asynchronous adjusting both the voltage and frequency (DVFS) or adjusting only the frequency (DFS).

Cai et al. proposed a mechanism to save energy consumption based on the identification of critical threads in parallel regions [2]. These threads are detected by the meeting point thread characterisation mechanism and are executed on cores with maximum frequency. The voltage and frequency are scaled down for cores executing other non-critical threads.

Several works focus on dynamic power adaptation for MPI applications by observing the behaviours of MPI calls. For example, Lim et al. proposed a method which aims to dynamically reduce the power state (p-state) of CPUs during communication phases where computation is not extensive [3]. This approach uses different training algorithms to identify communication regions in the MPI program and to derive the appropriate p-state of each region. A shifting component is then used to determine when the MPI program enters and leaves a

communication region. This component is also responsible for changing the *p*-state to the desired level. The *p*-state is changed by writing the appropriate values of the frequency identifier (FID) and voltage identifier (VID) to the model-specific register (MSR). Similarly, Iannou et al. proposed a method that aims to detect recurring communication and execution phases in MPI applications [4]. The phase detector is designed by instrumenting MPI calls to measure the execution time of each call and the execution time of the program in between calls. The instrumentation information is then used in the so-called *supermaximal repeat string algorithm* [5] to detect different phases in the MPI program. The work also proposes a hierarchical power controller to automatically adjust the voltage and frequency during each detected phase. The adjustment is determined based not only on the information of the local power domains but also on the previously observed execution times of the phase on multiple frequency and voltage domains. As another example, Kappiah et al. proposed a method which exploits inter-node slack to detect non-bottleneck nodes [6]. The frequency of these nodes is scaled down so that their computations are potentially completed at the same time with bottleneck nodes.

As a static power adjustment approach, Rountree et al. use linear programming to derive the offline schedule for CPU frequency so that energy saving is optimal [7]. The linear programming solver relies on the application communication trace to identify the critical path of execution. Using the power characteristics of the cluster, the solver derives a schedule that ensures that the node executing the critical path is never slowed down. Although this approach provides a nearly optimal schedule, it requires a complete trace of the program at each different frequency level. In addition, the linear programming solver is too slow to be efficiently utilised at runtime for dynamic adaptation. To overcome these problems, the authors combine this static approach and dynamic slack prediction into the *Adagio* runtime system [8]. Adagio first generates a static schedule based on the predicted execution time. The schedule is dynamically adapted during the runtime based on the prediction of critical paths.

Wang and Lu proposed a threshold-based approach to reduce energy consumption for heterogeneous clusters [9]. First, an offline analysis is performed to generate thresholds. These thresholds are then used to divide the workload into several ranges with different power configurations. At runtime, the workload for cluster is measured and based on this measurement, future workload is predicted. The predicted workload is then used to decide the range and appropriate power configuration to be used. Similarly, Chen at al. proposed a DVFS scheme for heterogeneous clusters that satisfy quality of service (QoS) requirements [10]. They use approximation algorithms to analyse the trade-off of time/space complexity and power consumption minimisation for three different QoS models. A latest survey by Bambagini and Marinoni [11] presents an in-depth analysis of state-of-the-art energy-aware scheduling algorithms for real-time systems including DVFS, DPM, and hybrid approaches.

Chen et al. have presented a DVFS (and also static power consumption) approach that targets special multicore platforms with time-triggered communication, i.e. time-division multiple access (TDMA) [12]. Our approach is not limited to such TDMA-based architectures. Also, the SCC used in our experiments does not have a time-triggered network on chip (NoC). In addition, TDMA schemes are normally applied for highly dependable architectures, where the focus is more on dependability rather than efficient resource utilisation.

Poellabauer et al. have developed a feedback-based DVFS approach that aims to reduce the processor speed during input-output (IO) intensive phases where the maximum processor speed cannot be fully exploited [13]. The detection is done by observing the ratio of data cache misses per instruction, which they call *memory access rate* (MAR). A high MAR denotes an IO-intensive period with lots of accesses to data memory, while a low MAR denotes a computation-intensive period. The separation of execution phases into IO and computation works effectively for a single core. With multiple cores, these phases tend to be out of sync among the cores, which reduces the chances that all cores are doing IO at the same time. In contrast, with our DVFS framework, we do not focus on speed reduction at IO phases but rather on speed reduction at temporal underload situations.

Bini et al. have presented a DVFS framework for real-time systems [14]. This approach is based on the calculation of the so-called optimal processor speed which would allow all deadlines to be met. They then use two discrete speeds, one slightly below and the other one slightly above the optimal speed, which are used to approximate the optimal speed, such that it is guaranteed that no deadline is missed. While this work focuses on static calculation of processor speed, our approach targets systems with temporal changes of the system load, which requires a dynamic approach.

The related work discussed in this section addresses different specific platform and software properties. Therefore, none of them can be easily adapted to be integrated into our execution framework in order to compare it with our DVFS strategy. The reasons for this are as follows:

Uniform vs. core-based control

The majority of DVFS research assumes the power controlability of individual computing elements, for example, Cai et al. [2], Gamell et al. [1], Lim et al. [3], Iannou et al. [4, 5], and Poellabauer et al. [13]. This

is also the case for Chen et al. [10] with the focus on server farms.

For many-core processors like the SCC, such an individual control of computing elements is not applicable. While the SCC allows frequency switching at the tile level, the more important voltage setting can only be done at the individual voltage islands of four tiles each. Thus, we support an approach that does not rely on controlability of individual nodes but rather does a processor-wide adaptation of power settings.

Dynamic vs. static scheduling

Some DVFS approaches focus on static control of power settings, for example, Kappiah et al. [6], Rountree et al. [7],[8], and Chen et al. [12].

In contrast, our approach focuses on the support of dynamic scheduling problems, not limiting it only to static ones.

Soft vs. hard real time

Some DVFS approaches target real-time systems, deploying scheduling methods that rely on the knowledge of the computational demand of individual tasks in order to allow response-time analysis, for example, Wang and Lu [9], Bambagini and Marinoni [11], or Bini et al. [14].

Our approach does not rely on the knowledge of computational demands of individual tasks. However, on the downside, our approach is not applicable for frameworks with hard real-time requirements.

In addition, some of the related work described above the results are acquired by means of simulation. In contrast, we use actual hardware platform to obtain results. Nevertheless, our approach is also hardware independent, i.e. it does not require any specific performance registers, and as such can be adapted to a new hardware platform without the need for any additional measures/modifications.

3 The SCC architecture

The SCC [15] is an experimental multi-core processor created by INTEL LABS as a research vehicle for many-core software. As seen in Fig. 1, the processor consists of 24 tiles in a 4×6 grid, connected by a high bandwidth, low latency, on-die 2D mesh network. Such an arrangement of tiles and network resembles a cluster on a single chip.

Each tile contains two modified P54C processor cores that support x86 architecture compilers and operating systems. In standard mode, each core usually runs its own OS, usually Linux and communicate with other cores over a packet-based network. Each core has a private 32 kibibytes (KiB) L1 and 256 KiB L2 caches. Furthermore, each tile features a 16-KiB block of static random access memory (SRAM) called message passing buffer (MPB) that is physically distributed, but logically shared. Each tile connects to a 2D mesh network via router with fixed XY-routing. The SCC chip has four on-die memory controllers (MC) as shown in Fig. 1, which supports 16- to 64-gibibytes (GiB) off-die dynamic random access memory (DRAM) in total and a voltage regulator controller (VRC) to let programs dynamically manage the power and the frequency of cores. In addition, one atomic test-and-set register and two atomic counter registers are available per core via the system interface.

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). The BMC handles commands to initialise

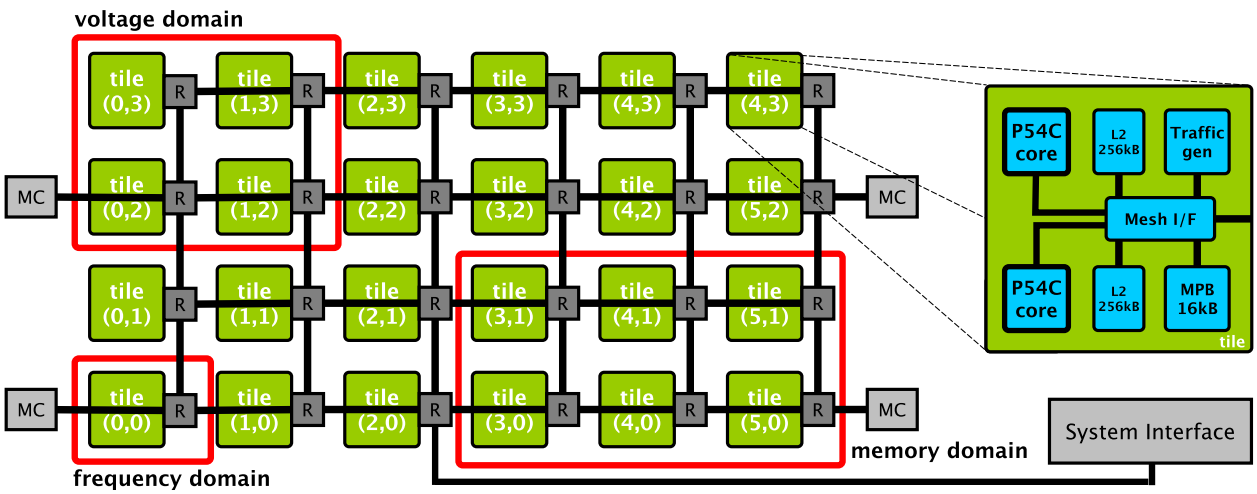


Fig. 1 SCC top-level tile architecture

and shut down the SCC and enables data collection for power consumption. The BMC in-turn connects to the management console PC (MCPC).

Normally, the software (called sccKit) provided by INTEL LABS is used to communicate to BMC.

3.1 Memory architecture

The SCC offers three different address spaces:

- A private off-chip address space in DRAM for each core. This memory is cache coherent with an individual core's L1 and L2 caches.
- A shared off-chip address space in DRAM. This memory may be configured as uncached or cached. If configured as cached, it is the programmer's responsibility to manage coherence.
- The MPB, that is 24×16 KiB of physically distributed, logically shared SRAM.

Tiles on the SCC are organised into four regions, each containing six tiles. Each region maps to a particular MC. The access to private memory goes through the assigned MC while the access to shared memory can go through any of the four MCs.

INTEL has provided a tag for shared data in the MPB called message passing buffer type (MPBT) that identifies cache lines for shared data in L1. All data tagged with MPBT bypasses the L2 cache and directly goes into the L1 cache in case of read accesses. Write operation to MPBT type memory are stored in the write combine buffer (WCB), until an entire cache line is filled or a write access to a different cache line happens. The INTEL instruction set architecture (ISA) is also extended with an CL1INVMB instruction that invalidates all cache lines in L1 tagged as MPBT. Access to this invalidated L1 cache lines forces an update of the L1 cache lines with the data in the shared memory.

3.2 Voltage and frequency scaling

Power management on the SCC consists of three components that work with separate clocks and power sources: tiles, mesh network, and memory controllers. In this study, we will focus only on power management at tile level because voltage/frequency for mesh and memory controller cannot be adjusted at runtime.

As can be seen from Fig. 1, on the SCC, tiles are arranged in voltage and frequency islands. There are seven voltage and 28 frequency islands on SCC. Each tile on SCC is one frequency island, while MCs, VRC, and the mesh make up the rest of the islands. In contrast to frequency island whereby each tile is an island, six voltage island contains 2×2 array of tiles (four tiles per island) each, while entire mesh network is regarded as seventh island. The voltage of six islands made up of tiles can be controlled by using the VRC. As described in the SCC architecture specification document [16], the VRC allows a set of voltages between 0 and 1.3 V in increments of 6.25 mV at island granularity. However, practically, the usable voltage range is 0.7 to 1.3 V (lower voltage setting will require hard reset of the SCC to resume normal operation again). The frequency management system provides a configuration register for each tile. The maximum frequency that can be set is dependent on the current voltage level and can be varied between 100 to 800 MHz. To change the voltage, a value that contains island identifier and actual voltage value is written to VRC. For frequency change, a divider value between 2 and 16 is used to select operating frequency of tile. Divider value is used to divide global frequency of 1600 MHz, providing 800 to 100 MHz and all the values in between. This divider value with tile identifier make up the value that is written to the configuration register. The frequency of the mesh network and MCs can be set at either 800 or 1600 MHz, and it cannot be changed dynamically.

In order to prevent any damage to the SCC chip, scaling the frequency up requires a corresponding change in the voltage. The official SCC documentation [17] and source code of RCCE communication library [18] provides a table with the maximum frequency allowed for each voltage level. However, there were stability issues with certain voltage level and maximum allowed frequency [19]. As reported in [1, 20], we also found that some of the voltage frequency pairs resulted in cores becoming unstable or crashing all together (which requires hard reset of the platform). Through experimental evaluation, we have calibrated the voltage and frequency pairs that works for our SCC unit. Figure 2 shows our calibrated voltage-frequency levels, and the one provided in RCCE source code. For frequency range marked as improved, we were able to lower the voltage required, while for unstable range, we have to increase the volts required to make the core stable and prevent crashes. While RCCE source code only provides two usable voltage levels, our calibrated profile provides four voltage levels.

4 Programming and execution model

4.1 Stream programming

A stream program consists of a set of computational nodes that communicate by exchanging messages via directed streams. The program consumes messages from a dedicated *entry* stream and produces messages to a dedicated *exit* stream. Nodes can be connected in different patterns, such as pipelines and parallel networks. Some examples of stream programming can be found in [21–23].

A stream program can be viewed as a graph whose vertices are nodes and whose edges are streams. For example, Fig. 3 shows an image filter application which includes

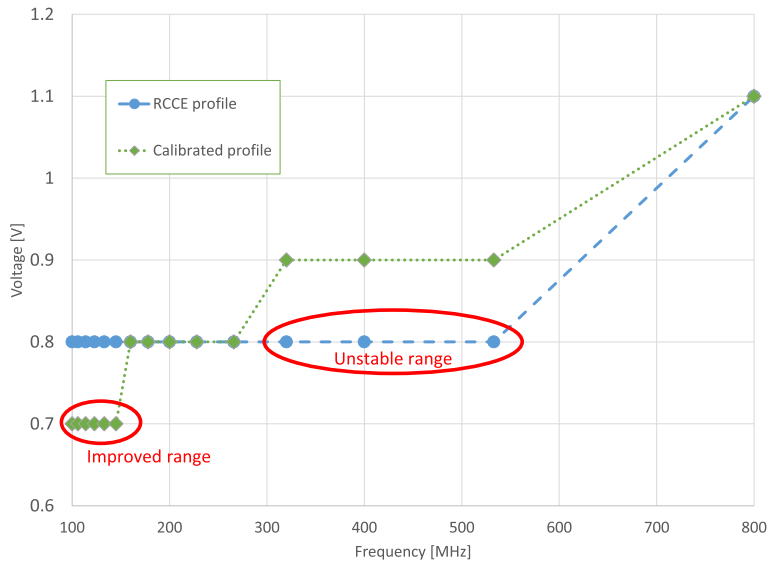


Fig. 2 Frequency-voltage profile of the SCC

a *Splitter* node that reads messages containing images and splits them into messages containing sub-images that are distributed into separate branches, where *Filter* nodes perform some transformation. Messages containing the filtered sub-images are then sent to the *Merger* node, which combines them into complete images.

As can be seen in this example, a node may produce multiple output messages from a single input message, and it may produce a single output message from multiple input messages. In general, a node that always produces $m > 0$ output messages from $n > 0$ input messages is said to have a *multiplicity* of

$$k = \frac{m}{n}.$$

The notion of multiplicity can be naturally extended to networks.

4.2 Task model

The programming model of stream programs described in 4.1 results in a task dependence graph. This graph $G = \langle T_I, T_O, T_C, E \rangle$ has a set of tasks $T = T_I \cup T_O \cup T_C$ and a set of directed dependency edges E: $\forall \langle \tau_1, \tau_2 \rangle \in E.\ \tau_1, \tau_2 \in T$. An edge $\langle \tau_1, \tau_2 \rangle \in E$ denotes that there is a communication stream from task τ_1 to task τ_2. There is no restriction assumed on the possible structure of that task dependence graph. T_I is the set of tasks that read external input, T_O is the set of tasks that write external output, and T_C is the set of task that have only internal communication within the task dependence graph. T_I and T_O can theoretically also overlap: $T_I \cap T_O \supseteq \emptyset$, while by definition, there cannot be an overlap between T_C and T_I or T_O: $T_C \cap (T_I \cup T_O) = \emptyset$.

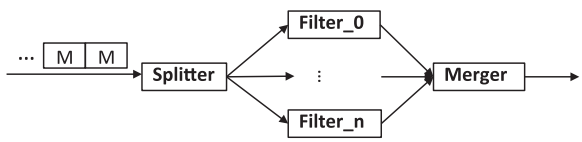


Fig. 3 Program structure of image filter application

As a stream processing system, the computations are triggered by the arrival of an external input message. We require no particular assumption about the arrival rate of the input messages.

There is also no particular assumption about the deployed scheduling method, as our DVFS method does not depend on some specific properties of the task schedule. This makes our DVFS approach outstanding compared to those approaches that depend on a particular schedule. The underlying scheduling method may also have real-time requirements. However, our approach would be limited to soft real-time scheduling only. This is due to the nature of our DVFS method where the system might be sometimes for a short time in overload situations, before the overload gets detected and the speed is increased again.

4.3 Execution model

Conceptually, the stream execution model includes two layers: a runtime system (RTS) and an execution layer. At the RTS layer, each stream is represented as a FIFO buffer for storing messages and each node of the stream program is transformed into one task. A task is an iterating process that reads messages from its input streams, performs the associated node's computations, and writes output messages to its output streams. The role of the RTS

is to enforce the semantic of stream programs, i.e. make sure that each task reads from and writes to appropriate streams.

The execution layer, which lies under the RTS, provides essential facilities for task and stream managements. The execution layer also provides a scheduler to distribute tasks to physical resources.

4.4 LPEL—a stream execution layer with efficient scheduling

The lightweight parallel execution layer (LPEL) [24] is an execution layer that supports stream programs on shared memory platforms. LPEL adopts a user-level threading scheme providing the necessary threading and communication mechanisms in the user space. In LPEL, there is a stream component to support stream creation, stream reading, stream writing, and stream replacing. Additionally, a task component is provided to create a wrapper around each node before sending them to the scheduler.

Figure 4 shows an abstract design of the LPEL execution layer. In LPEL, each CPU core is modelled as a worker. LPEL is facilitated with a centralised scheduler to obtain automatic load balancing [25]. In this scheduler, one worker is dedicated as the conductor to manage the central task queue (CTQ) which stores all ready tasks. Each worker once free sends a request to the conductor. The conductor selects one task in the CTQ to send to the worker. To obtain good performance in terms of throughput and latency, the scheduler uses the notion of data demands on streams to derive the task priority. All conductor-worker communications are exercised via mailboxes.

LPEL is also facilitated with a monitoring framework which can provide the information of internal behaviours at different levels of details ranging from scheduling activities to performance measurement and resource utilisations [26].

5 DVFS strategy

In reactive stream processing, the system operates in direct response to inputs from its environment. If the load imposed by the environment varies, dynamic power management techniques can be used to effectively reduce the power consumption of such systems.

Our strategy is to reduce the frequency and voltage when there are more than enough resources and the system can easily cope with the input rate and to increase them when the system becomes overloaded.

We have extended LPEL to support this automatic DVFS strategy. Each SCC core runs an LPEL worker. We use the built-in monitoring framework of LPEL to observe the input and output rates of the program, as well as the number of workers waiting for new work. We then use

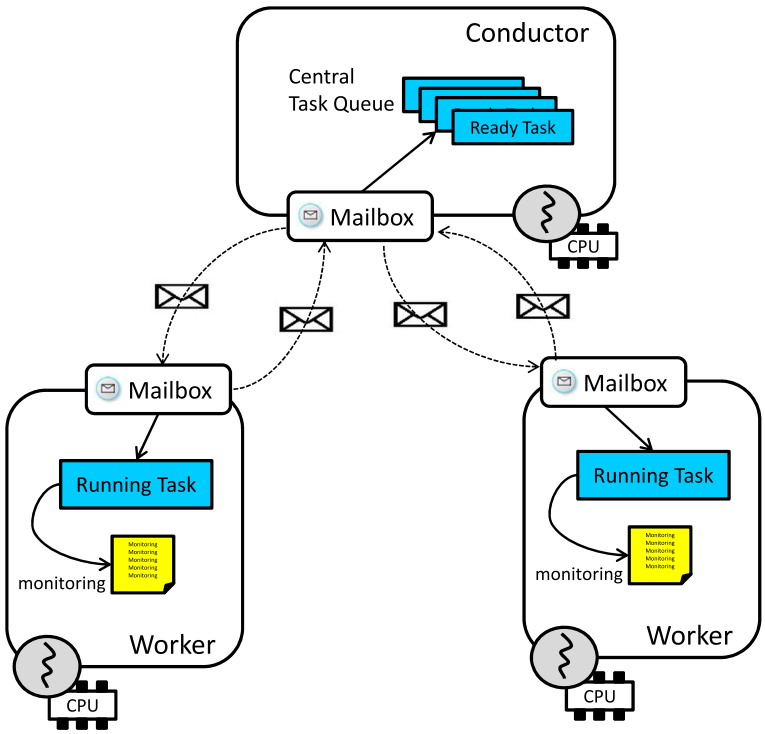


Fig. 4 LPEL execution layer

this information to dynamically adjust the voltage and frequency so that the system can cope with the input rate and save energy at the same time.

As mentioned in Section 4.1, stream programs consume messages from a dedicated entry stream and produce output messages to a dedicated exit stream. At runtime, we would ideally want a situation where

$$ir \approx k \cdot or,$$

where ir is the rate at which messages enter the network, or is the rate at which messages leave the network, and k is the multiplicity of the network (cf. Section 4.1). More precisely, we want to avoid the case of $ir \gg k \cdot or$, which means that the system is internally accumulating messages, a behaviour that will eventually lead to memory exhaustion, whereas the case of $ir \ll k \cdot or$ is just impossible.

For simplicity, we will (w.l.o.g.) assume a multiplicity of $k = 1$, i.e. the network produces exactly one output message for each input message. In that case, we can drop the factor k and obtain

$$ir \approx or. \tag{1}$$

To achieve the near balance of Eq. 1, we have to consider a dynamic input rate from the environment. The system must be equipped with sufficient computational resources to process messages fast enough under the maximal input rate ir_{max}; otherwise, the system may become *overloaded*. In this case, the output rate will stay close to some maximal achievable output rate or_{max}, i.e.

$$or \approx or_{max} \ll ir, \tag{2}$$

and messages will either start to accumulate inside the system or they will be dropped. On the other hand, if the input rate falls below or_{max}, the system becomes *underloaded*, i.e.

$$ir \approx or \ll or_{max}, \tag{3}$$

and unused system resources may cause unnecessary power drain.

One way to deal with this situation is to use a platform where we can dynamically adjust the performance in response to demand. If we use voltage and frequency scaling to adjust the performance, or_{max} becomes dependent on the current voltage V and frequency f.

According to Condition 2, we can detect overload by checking if the average output rate falls below the average input rate. The overload at time t is given by

$$ol(t) = \frac{ir(t) - or(t)}{ir(t)}. \tag{4}$$

5.1 Detecting overload

In many streaming applications, the input rate depends on a stateful stochastic process. Thus, it makes sense to consider the overload history to predict the potential future overload. We wanted to keep the overhead of our strategy low and therefore decided to use a simple exponential moving average (EMA) predictor, which is extremely fast to calculate and has minimal memory overhead. At a given time t, we calculate the predicted future overload $ol_{pred}(t+1)$ as

$$ol_{pred}(t+1) = \begin{cases} 0 & \text{if } t = 0 \\ \alpha_{ol} \cdot ol(t) + (1 - \alpha_{\mathrm{ol}}) \cdot ol_{pred}(t) & \text{if } t \geq 1. \end{cases} \tag{5}$$

The smoothing factor $0 < \alpha_{\mathrm{ol}} < 1$ is application specific. A high value makes the prediction depend mostly on the recent history, whereas a low value makes the prediction more dependent on the long-term history. A suitable smoothing factor can be found by minimising an error measure, like the sum of square errors (SSE), on representative sample data. This can either be achieved by graphically checking the fit for different parameters or by applying a least squares approach [27].

5.2 Detecting underload

We detect underload situations by observing used resources. More precisely, we examine the slack, e.g. the number of workers that are waiting for work to be assigned. Like in the overload case, we use an EMA to make a prediction about the future underload. At a given time t, we calculate the predicted future number of unused workers $ww_{pred}(t+1)$ as

$$ww_{pred}(t+1) = \begin{cases} 0 & \text{if } t = 0 \\ \alpha_{ww} \cdot ww(t) + (1 - \alpha_{ww}) \cdot ww_{pred}(t) & \text{if } t \geq 1. \end{cases} \tag{6}$$

Again, the smoothing factor $0 < \alpha_{ww} < 1$ is application dependent. A high value makes the prediction depend mostly on the recent history, whereas a low value makes the prediction more dependent on the long-term history. A suitable smoothing factor can, again, be found by minimising an error measure on representative sample data.

5.3 Policy

Our policy for adjusting the frequency at runtime can be summarised by the following set of rules:

1. Increase the frequency of all the islands by one step, when the number of waiting workers is predicted to fall short of a given lower threshold ($ww_{pred}(t+1) \leq ww_{th}$) and the overload is predicted to exceed a given upper threshold ($ol_{pred}(t+1) > ol_{th}$).
2. Decrease the frequency of all the islands by one step, when the number of waiting workers is predicted to

exceed the given lower threshold ($ww_{pred}(t+1) > ww_{th}$).
3. Otherwise, do not change the frequency.

Table 1 summarises all possible situations and the action implied by the above rules. Of course, the frequency is only varied within the limits given in Fig. 2.

If some messages are half-way processed in the system, the effect of the change is not seen immediately in its full extent. We therefore limit the rate of frequency adjustments. The maximal allowable rate of frequency adjustments depends on the application-specific maximal processing latency l_{max} and on the smoothing factor α_{ol}.

In Section 6.4, we develop a few general guidelines for choosing suitable thresholds ww_{th} and ol_{th}. Notwithstanding, the concrete values are application specific. In practice, these values could be determined using metaheuristics to perform guided profiling of the application under consideration.

5.4 Frequency adjustment

As pointed out in [28], the operating speed of a processing unit on a multicore platform is approximately proportional to its operating clock frequency,

$$speed \propto f, \tag{7}$$

whereas its dynamic power consumption is approximately proportional to the product of its operating frequency by the square of its operating voltage,

$$Power \propto f \cdot V^2. \tag{8}$$

Since a lower operating voltage increases the circuit delay, the operating voltage always imposes an upper bound on the operating frequency, and it follows that the dynamic power consumption of a processing unit is approximately proportional to the cube of its operating voltage,

$$Power \propto V^3, \tag{9}$$

or, equivalently, to the cube of its operating speed or frequency,

$$Power \propto speed^3 \propto f^3. \tag{10}$$

Table 1 Policy for changing the frequency

Underload condition	Overload condition	Action
$ww_{pred}(t+1) \leq ww_{th}$	$ol_{pred}(t+1) \leq ol_{th}$	None
$ww_{pred}(t+1) \leq ww_{th}$	$ol_{pred}(t+1) > ol_{th}$	Increase frequency
$ww_{pred}(t+1) > ww_{th}$	$ol_{pred}(t+1) \leq ol_{th}$	Decrease frequency
$ww_{pred}(t+1) > ww_{th}$	$ol_{pred}(t+1) > ol_{th}$	Decrease frequency

We use the full range of available frequency settings, that is, from 100 to 800 MHz. We change the frequency by adjusting the operating frequency divider $f_{div_{cur}}$

$$\begin{aligned} f_{div_{inc}} &= f_{div_{cur}} - 1 \\ f_{div_{dec}} &= f_{div_{cur}} + 1. \end{aligned} \tag{11}$$

As pointed out above, there is a minimal required voltage level for each frequency, which we also have to adjust in order to avoid chip failures (cf. Fig. 2).

5.5 Influence of load balancing

The SCC allows to individually set the frequency of each tile (of two cores) and the voltage of each island (of eight cores).

Although the SCC offers a theoretical feature to put individual islands to sleep by setting their voltage to 0 V, we found that this feature did not work properly in practice, as pointed out in Section 3.2.

Considering that putting processing units to sleep in order to reduce their static power consumption is thus not an option, maximal dynamic power savings are achieved on a perfectly load-balanced systems, by reducing the speed of all islands by the same amount and avoiding different speeds on different cores. This can easily be seen for a two-core system, where the total power

$$Power = Power_{core_1} + Power_{core_2} \propto speed^3_{core_1} + speed^3_{core_2}. \tag{12}$$

Since we are considering reactive systems with a data input rate that is determined by the environment, we want to run the system at a momentary speed of

$$s = speed_{core_1} + speed_{core_2}, \tag{13}$$

that is determined by the current workload, hence,

$$\begin{aligned} Power &\propto speed^3_{core_1} + (s - speed_{core_1})^3 \\ &= s^3 - 3s^2 speed_{core_1} + 3s\, speed^2_{core_1}, \end{aligned} \tag{14}$$

which is minimal for $speed_{core_1} = s/2$. In other words, $speed_{core_1} = speed_{core_2} (= s/2)$ yields the minimal power consumption for any momentary system speed s. It is not hard to generalised this result to n cores.

Since LPEL already performs automatic load balancing (cf. Section 4.4), we may restrict ourselves to chip-level DVFS, i.e. we merely have to adjust the total system speed, such that the system can cope with the data input rate of the environment. Any required balancing is then automatically performed by LPEL. This greatly simplifies the act of choosing the right voltages/frequencies in the presence of core-level load fluctuations.

6 Evaluation

To evaluate our power-aware scheduler, we deployed a resource aware LPEL (RALPEL) on the SCC. We

used different policies to measure the impact of power saving.

6.1 Experimental setup

We used the default sccKit 1.4.2 configuration, with memory and mesh running at 800 MHz.

As our benchmark, we chose a fast fourier transform (FFT), face detection (FD), and data encryption standard (DES) applications implemented using the S-Net [22] streaming language.

Each benchmark contains the main structure which performs the computation. Normally, this main structure is comprised of multiple sub-tasks. To increase the level of concurrency, S-Net provides parallel replication to create multiple instances of the main structure. The number of instances depends on the number of workers. We reserved some cores to simulate the environment, consisting of a source/producer and a sink/consumer.

The FFT benchmark applies FFT to messages containing 2^{13} discrete complex values.

The FD benchmark applies a sequence of classifiers to detect different features to be found in an image, like the eyes, mouth, nose, etc. The way the algorithm behaves is that it calls in a pipeline all the different classifiers. Each classifier searches on a list of feature predictors until the first matching one has been found. For simplicity of the implementation, we used a simulation of the actual feature detectors and decided randomly whether the feature predictors were matched.

The main structure of DES is made of three sub-tasks. First, sub-task performs initial permutation and splits block of bits into two blocks of equal size. In the next sub-task, actual ciphering is applied to the two blocks. The last sub-task joins up the two blocks into one cipher text block and applies the final permutations. Each sub-task has variable execution time that depends on the size of input messages.

Table 2 shows the minimal and maximal execution time with difference in percentage for each task in the benchmarks. The value reported are arithmetic mean of five runs of each benchmark. For these experiments, we did not employ voltage or frequency scaling. We can see that all benchmarks show a considerable variation in execution times of tasks.

Table 2 Minimal and maximal task execution time for different benchmarks, mean values of five runs with maximal voltage and frequency energy policy

Benchmark	Task	Min (s)	Max (s)	Difference (%)
FFT	initP	6.5868	8.2640	25.4623
	stepP	70.5236	113.0189	60.2568
FD	classifier1	4.0704	14.6089	258.9072
	classifier2	4.7971	14.1402	194.7675
	classifier3	4.4682	13.1498	194.2975
DES	initP	3.6370	6.8986	89.6753
	subRound	4.7194	14.3530	204.1283
	finalP	2.1357	4.0790	90.9877

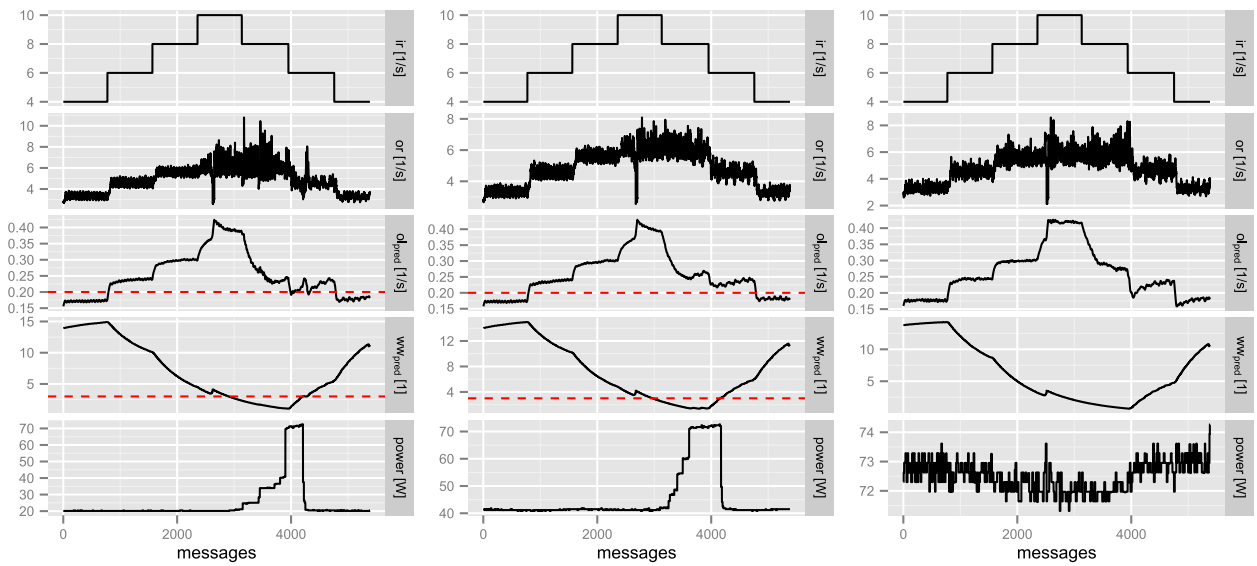
6.2 Data collection and post-processing

As we mentioned in Section 3, the SCC is connected to the FPGA called the BMC. While SCC is equipped with many sensor ADCs that can measure the supply voltages for individual islands, it does not, however, provide current at this level. In order to calculate power, we need both supply voltage and the current. Both of these values are available at chip level on the 3.3 V rails. Which means power can only be measured of all cores and the mesh together as a whole.

The power measurement controller (PMC) situated in the FPGA/BMC periodically collects the data from the measurement ADCs and stores them in the power measurement registers. These registers can be memory mapped and read by the cores or the MCPC. Furthermore, FPGA also provides global timestamp counter that can be used across cores to have reliable/consistent time source. Since global timestamp counter is located on the FPGA, it does not get affected by frequency change on the SCC cores.

Table 3 Total wall clock time, average power level, and total energy consumption of each benchmark under three different energy policies, as mean over five runs

Benchmark	Policy	Wall clock time (s)	Power (W)	Energy (kJ)
FFT	DVFS	1646.045	30.600	50.378
	DFS	1662.049	47.934	79.670
	MVF	1655.352	72.579	120.143
	σ^2	43.069	296.665	818.136
	σ/μ	0.40 %	34.19 %	34.30 %
FD	DVFS	1797.958	42.044	75.757
	DFS	1774.602	53.726	95.513
	MVF	1776.495	74.030	131.508
	σ^2	112.186	174.648	532.678
	σ/μ	0.59 %	23.35 %	22.87 %
DES	DVFS	1447.779	46.625	67.507
	DFS	1448.767	56.862	82.399
	MVF	1451.745	71.838	104.291
	σ^2	2.842	107.199	228.231
	σ/μ	0.12 %	17.72 %	17.83 %



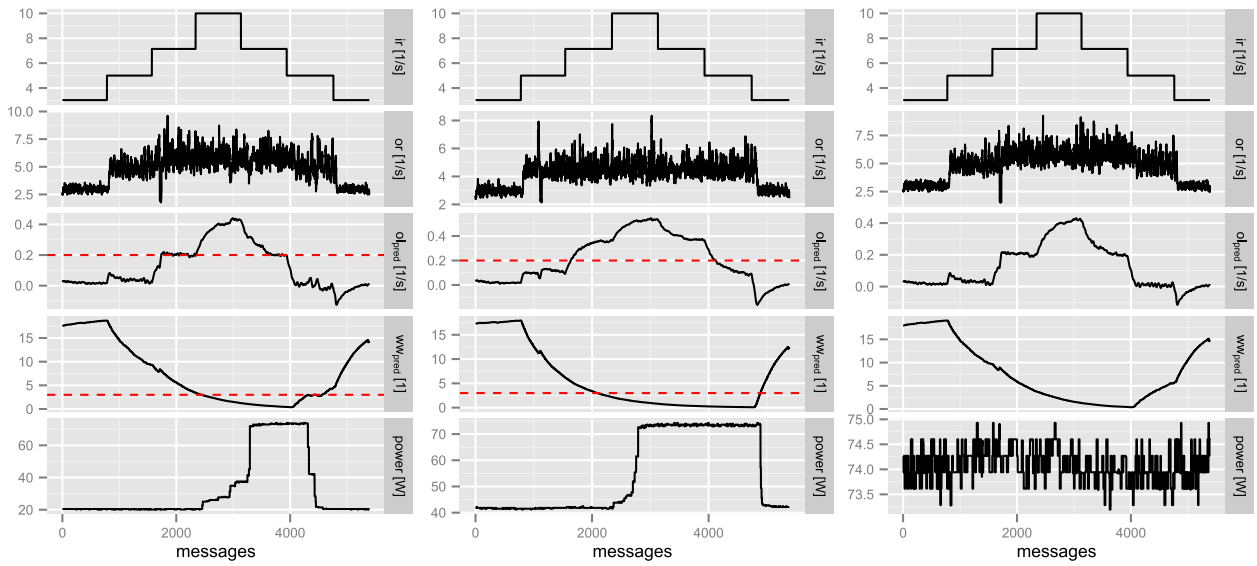
(a) DVFS (b) DFS (c) MVF

Fig. 5 Fast fourier transform with thresholds $ol_{th} = 0.20$ and $ww_{th} = 3.0$. **a** DVFS. **b** DFS. **c** MVF

As we mentioned before, we only need to observer input rate, output rate, and waiting workers. In addition, we also keep track of voltage and current of the SCC chip. We achieve this by memory mapping registers mentioned above. All of this information is collected and used by power manager (a part of conductor) at runtime. Power manager also writes all the information in to a log file for post-processing. Post-processing is only required to generate graphs and analyse the effects of different energy policies. Power consumption is calculated by multiplying the voltage and current consumption at any given moment (i.e. when each measurement is taken). Average power consumption of a benchmark execution is arithmetic mean of all the power readings.

6.3 Effectiveness of power saving

We tested our DVFS strategy using different thresholds for overload (ol_{th}) and waiting workers (ww_{th}), using a fixed input pattern consisting of 7000 messages for FFT and FD and 2300 messages for DES. We expected different



(a) DVFS (b) DFS (c) MVF

Fig. 6 Face detection with thresholds $ol_{th} = 0.20$ and $ww_{th} = 3.0$. **a** DVFS. **b** DFS. **c** MVF

thresholds to yield different behaviours with respect to energy efficiency, which would allow us to pick thresholds with a balanced behaviour, i.e. which would save a significant amount of energy without sacrificing too much computational performance.

Table 3 summarises the main result of our experiments, indicating the total wall clock time, the average power level, and the total energy consumption of each benchmark under three different energy policies: dynamic voltage and frequency scaling (DVFS) as described in Section 5, dynamic frequency scaling at maximal voltage (DFS), and maximal voltage and frequency settings (MVF).

The total wall clock time that each benchmark takes to run is roughly the same ($\sigma/\mu < 0.6\,\%$) under DVFS, DFS, and MVF; the reason being that we are considering a reactive scenario, where the pace of the system is not determined by the core frequency but by the data input rate imposed by the environment. In other words, the system must complete a given workload within a specified wall clock time. It cannot complete the workload significantly faster, because the input data only becomes available in real time, and it must not complete its workload significantly slower (the latter would indicate an abnormal overload situation).

Tiny differences in the wall clock times are mainly due to variations in the dynamic scheduling of tasks: for example, even a slight difference in the execution time of a sole task (e.g. due to a cache hit vs. miss) can influence numerous subsequent scheduling decisions, like the assignment of individual tasks to particular cores, which can in turn cause changes in the memory access times of these tasks, due to the use of differing NoC routes.

Table 4 Average power level, wall clock time, and total energy consumption for different threshold values of ol_{th} and ww_{th} under the DVFS policy, as mean over three runs

Benchmark	ol_{th}	ww_{th}	Wall clock time (s)	Power (W)	Energy (kJ)
FFT	0.24	2.4	1647.944	26.868	44.278
	0.16	2.4	1638.741	28.729	47.080
	0.20	3.0	1646.045	30.600	50.378
	0.24	3.6	1635.508	32.971	53.930
	0.16	3.6	1655.718	32.010	53.024
FD	0.24	2.4	1766.394	33.599	59.714
	0.16	2.4	1785.962	34.912	62.487
	0.20	3.0	1797.958	42.044	75.757
	0.24	3.6	1785.684	40.825	73.130
	0.16	3.6	1760.555	44.493	78.333
DES	0.24	2.4	1443.617	40.944	59.113
	0.16	2.4	1454.066	44.094	64.168
	0.20	3.0	1447.779	46.625	67.507
	0.24	3.6	1441.094	43.402	62.539
	0.16	3.6	1447.406	46.882	67.849

Our results indicate that the DVFS strategy cut the energy consumption of the FFT benchmark by roughly a half and the energy consumption of the FD benchmark by a third. The reduction for the DES benchmark is about

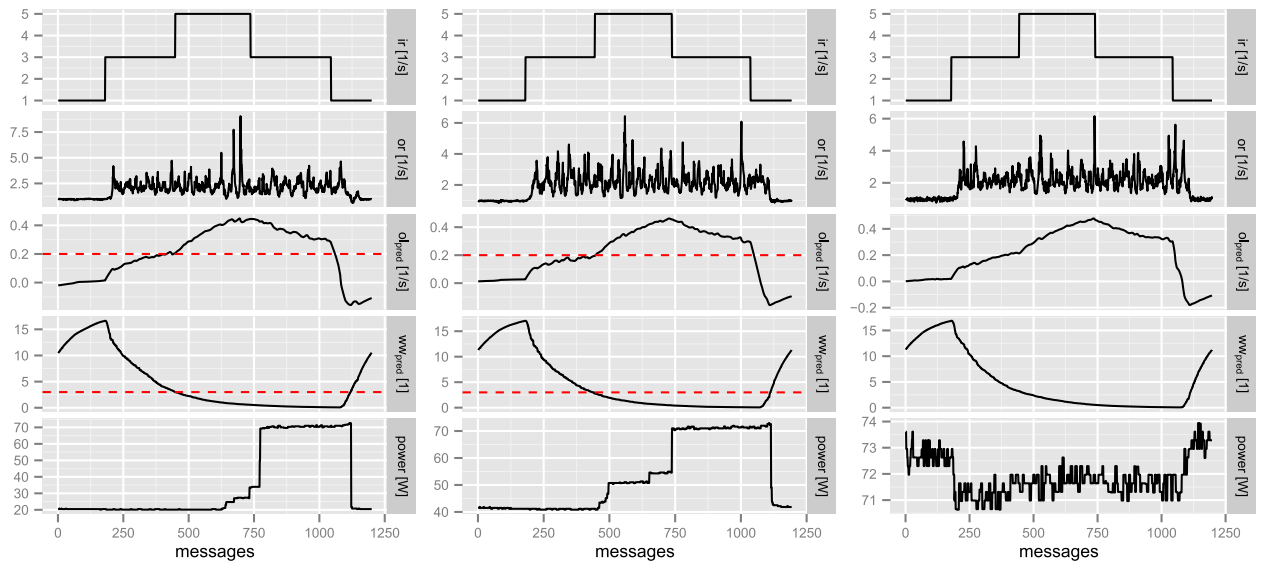


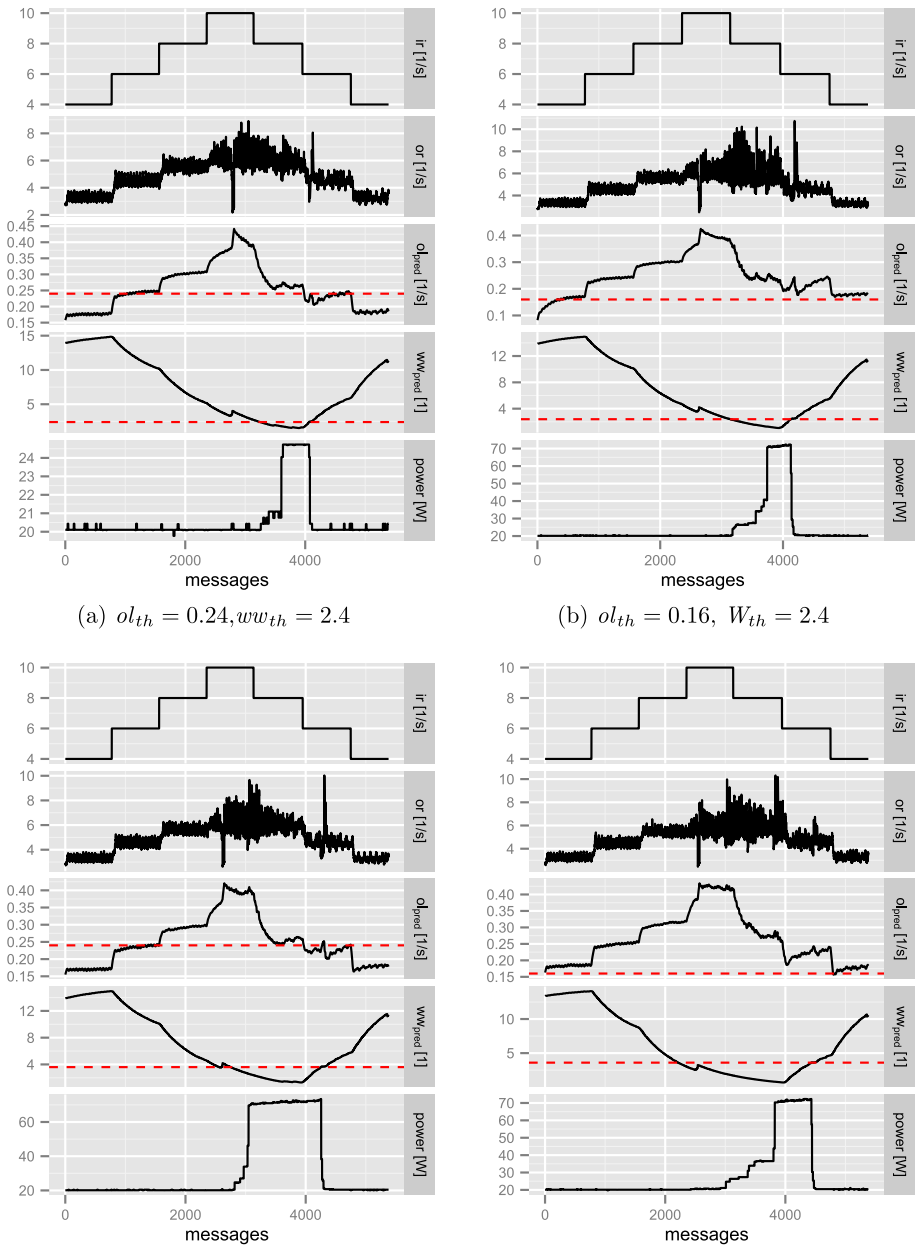
Fig. 7 DES with thresholds $ol_{th} = 0.20$ and $ww_{th} = 3.0$. **a** DVFS. **b** DFS. **c** MVF

30 %. As expected, the exclusive use of frequency scaling without voltage scaling (DFS) saves significantly less energy in all benchmarks.

6.4 Influence of thresholds

Figures 5, 6 and 7 show time series of our experiments under the three energy policies. Each sub-figure shows the progress of the input rate *ir*, the output rate *or*, the predicted overload ol_{pred}, the predicted number of waiting workers ww_{pred}, and power.

The red dashed lines mark our balanced overload threshold $ol_{th} = 0.2$ and our balanced waiting threshold $ww_{th} = 3.0$. In the topmost panel, we can see our input rate pattern, in this case, a synthetic ascending/descending step pattern. The panel below shows the corresponding output rate, which can be seen to carry a significant jitter. The next two panels show the predicted overload ol_{pred} and the predicted number of waiting workers ww_{pred}, respectively. In the last panel, we see that the power level for DVFS follows roughly the



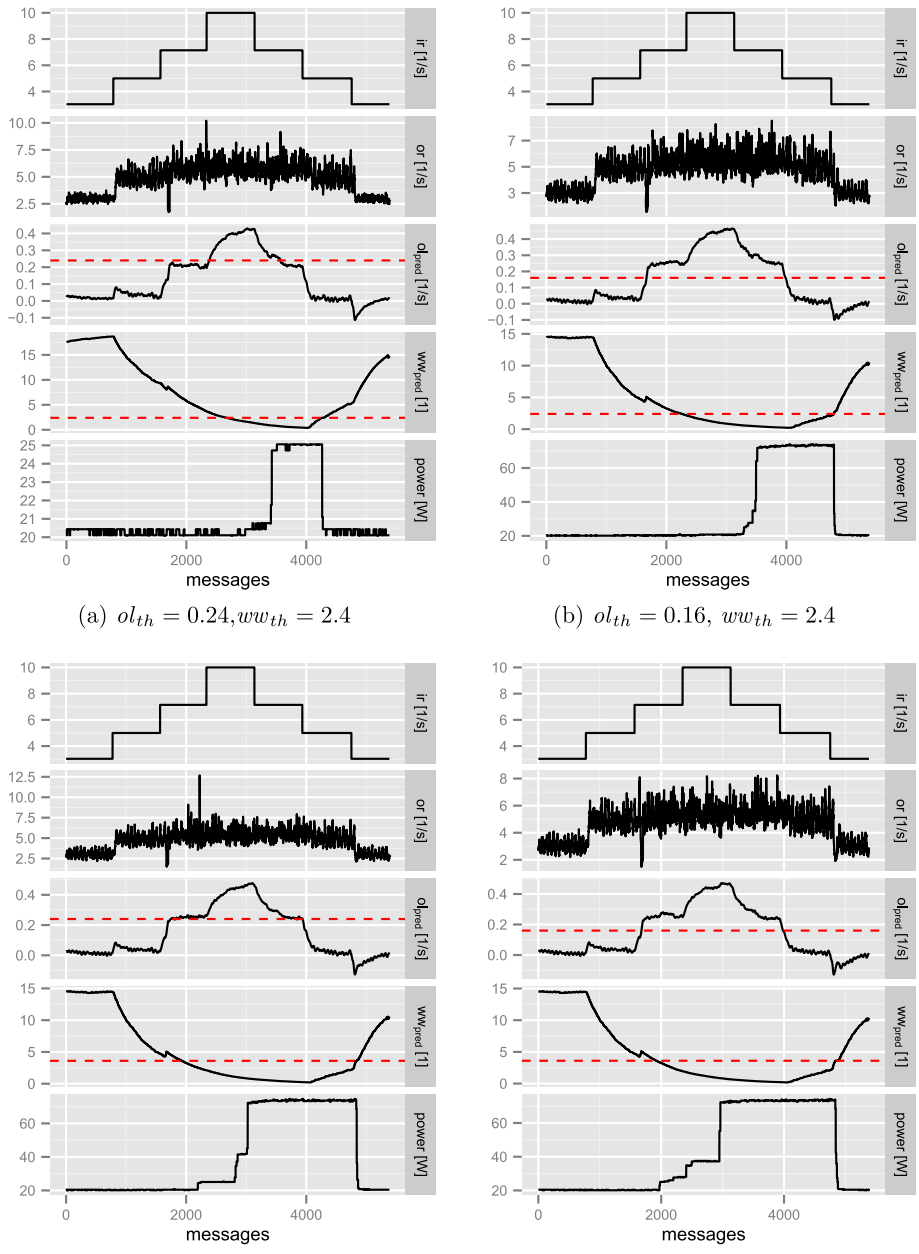
(a) $ol_{th} = 0.24, ww_{th} = 2.4$ (b) $ol_{th} = 0.16,\ W_{th} = 2.4$

(c) $ol_{th} = 0.24, ww_{th} = 3.6$ (d) $ol_{th} = 0.16, ww_{th} = 3.6$

Fig. 8 Fast fourier transform with different choices of thresholds ol_{th} and ww_{th} for DVFS (**a**–**d**)

shape of the input rate pattern, although it is delayed. The pattern for DFS is similar, but the system operates at a higher power level, which is visible as a shift on the power axis. Moreover, the system reaches the maximal power level faster and stays there longer. For MVF, the power level is practically constant, and what can be seen in the panel is merely a slight jitter. Bakker et al. [29] have observed a similar fluctuation in power consumption of SCC chip in idle mode. They ascribe this behaviour to the charging and discharging of a stabiliser capacitor in the voltage regulator circuits.

Table 4 summarises the results of our experimentation with different overload and waiting workers thresholds under the DVFS policy. For the same reason as given in Section 6.3 concerning Table 3, the total wall clock time that each benchmark takes to run is roughly the same for all different chosen thresholds.

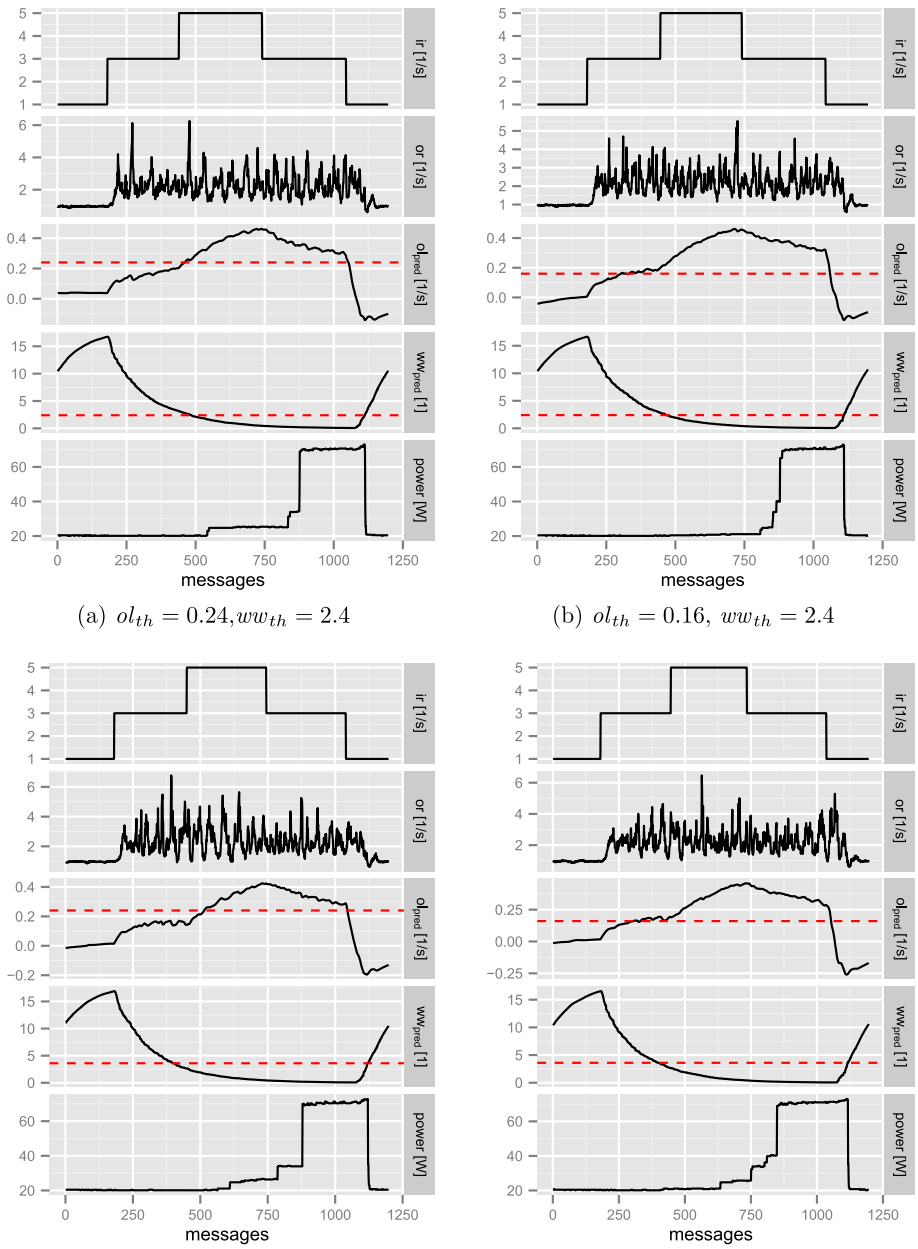


(a) $ol_{th} = 0.24, ww_{th} = 2.4$ (b) $ol_{th} = 0.16,\ ww_{th} = 2.4$

(c) $ol_{th} = 0.24, ww_{th} = 3.6$ (d) $ol_{th} = 0.16, ww_{th} = 3.6$

Fig. 9 Face detection with different choices of thresholds ol_{th} and ww_{th} for DVFS (**a**–**d**)

Let us have a closer look at the effect of the different parameter choices for the FFT benchmark. The combination of a high overload threshold of $ol_{th} = 0.24$ and a low waiting threshold of $ww_{th} = 2.4$, as seen in Fig. 8a, causes the system to increase its performance configuration at a very late point in time, and performance is decreased again shortly afterwards. Although the ol_{pred} is already higher than ol_{th}, the system does not engage DVFS to increase performance, the reason being that ww_{pred} is still a lot lower than ww_{th}. Which means that the overload that the system is experiencing can be/is dealt with by increasing the resource usage, i.e. waiting workers. Once ww_{pred} is below the threshold ww_{th} and if ol_{pred} is still above $\mathrm{ol_{th}}$, then DVFS is engaged. Lowering the overload threshold to $ol_{th} = 0.16$ makes the system reach its maximum performance state earlier and leave it later, as shown in Fig. 8b. If, in addition, the waiting threshold is raised to $ww_{th} = 3.6$, as shown in Fig. 8d, the system spends most of the time in its maximum performance state.



(a) $ol_{th} = 0.24, ww_{th} = 2.4$ (b) $ol_{th} = 0.16,\ ww_{th} = 2.4$

(c) $ol_{th} = 0.24, ww_{th} = 3.6$ (d) $ol_{th} = 0.16, ww_{th} = 3.6$

Fig. 10 DES with different choices of thresholds ol_{th} and ww_{th} for DVFS (**a**–**d**)

Figure 9 shows similar patterns for the face detection benchmark. Note that the power panel in Fig. 8a and Fig. 9a has a different scale, indicating a very good choice of parameters, where the system never needs the maximal performance settings. For the DES benchmark, similar patterns to FFT and FD can be seen in Fig. 10. Figure 10a shows that ol_{pred} and ww_{pred} crosses the thresholds at roughly the same time and we can see that power consumption start to increase around same time (roughly around 500 messages). From Fig. 10c, we can see that even though ww_{pred} is below threshold around 375 messages, we do not see the corresponding change in power consumption, the reason being ol_{pred} has not reached the threshold yet. Once ol_{pred} crosses the threshold $ol_{th} = 0.24$ around 500 messages, we can see that power consumption start to increase reflecting the increase in voltage/frequency.

From all the experiments, we can see that the waiting worker threshold ww_{th} determines the sensitivity towards low resources. If ww_{th} is too high, the speed of the processor (and thus the power consumption) might be increased earlier than necessary to handle the workload even though that are still some workers available that might be able to handle increased workload. If ww_{th} is too low, there is a risk that the system will not be able to handle the workload. The overload threshold ol_{th} determines the sensitivity towards high workload. If ol_{th} is too low, the speed of the cores (and thus the power consumption) might be kept higher than necessary to handle the work load. If ol_{th} is too high, there is a risk that the system will not be able to handle the workload.

The above rules form a set of general guidelines for choosing the suitable thresholds ww_{th} and ol_{th}. Notwithstanding, the concrete values are application specific. In practice, these values could be determined using metaheuristics to perform guided profiling of the application under consideration.

7 Conclusions

Reactive stream processing systems (RSPs) are a common paradigm in embedded computing. The streaming model behind RSPs makes them well suited for computationally intensive applications where parallel execution on multiple cores is required.

In this paper, we present an execution framework for RSPs that provides a *dynamic voltage and frequency scaling* (DVFS) to optimise the power consumption. With our specialisation to RSPs, we were able to deploy a lightweight DVFS strategy that can be used for DVFS during runtime in order to cope with variable system load. Our DVFS strategy reduces the frequency and voltage when there are more than enough resources, and the system can easily cope with the input rate, and increases them when the system becomes overloaded. To do so, we observe the input and output rates of the RSPs, as well as the number of workers waiting for new work to predict overload and underload situations. We then use this information to dynamically adjust the voltage and frequency so that the system can cope with the input rate and save energy at the same time. In contrast to many other approaches, we do not require the ability to control the power setting of individual computational units, making our approach suitable for tiled many-core processors like the SCC.

Our experimental validation of approach does not rely on simulations. We use an execution framework that runs on real parallel hardware, for example, the SCC we used in our experiments. This execution framework has been built into the *lightweight parallel execution layer* (LPEL) middleware, since it already provided a load-balancing dynamic scheduler. Our experimental evaluation indicates that our DVFS strategy for RSPs can significantly reduce energy consumption, even to a half.

Competing interests

The authors declare that they have no competing interests.

Acknowledgements

This work has been supported by the Material Transfer Agreement 2010–2013 for the Intel SCC Research Processor "Lightweight Parallel Execution Layer for Stream Processing Networks on Distributed Memory Architectures", the IST FP7 research project "Asynchronous and Dynamic Virtualization through performance ANalysis to support Concurrency Engineering" (ADVANCE) under contract no. IST-2010-248828, and the FP7 ARTEMIS-JU research project "ConstRaint and Application driven Framework for Tailoring Embedded Real-time Systems" (CRAFTERS) under contract no. 295371.

Author details

[1] School of Computer Science, University of Hertfordshire, AL109AB Hatfield, UK. [2] Institut für Computersprachen, Vienna University of Technology, A-1040 Vienna, Austria.

Received: 30 September 2015 Accepted: 14 May 2016
Published online: 13 June 2016

References

1. M Gamell, I Rodero, M Parashar, R Muralidhar, in *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*. HPDC'12. Exploring cross-layer power management for PGAS applications on the SCC platform (Association for Computing Machinery (ACM), New York, NY, USA, 2012), pp. 235–246. doi:10.1145/2287076.2287113. http://doi.acm.org/10.1145/2287076.2287113
2. Q Cai, J González, R Rakvic, G Magklis, P Chaparro, A González, in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. PACT '08. Meeting points: using thread criticality to adapt multicore hardware to parallel regions (ACM, New York, NY, USA, 2008), pp. 240–249. doi:10.1145/1454115.1454149. http://doi.acm.org/10.1145/1454115.1454149
3. MY Lim, VW Freeh, DK Lowenthal, in *Supercomputing Conf. (SC), Proceedings of the ACM/IEEE*. Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs (IEEE, Washington, DC, USA, 2006), pp. 14–14
4. N Ioannou, M Kauschke, M Gries, M Cintra, in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference On*. Phase-based application-driven hierarchical power management on the single-chip cloud computer (Institute of Electrical & Electronics Engineers (IEEE), Galveston Island, TX, USA, 2011), pp. 131–142. doi:10.1109/pact.2011.19. http://dx.doi.org/10.1109/pact.2011.19

5. D Gusfield, *Algorithms on Strings, Trees, and Sequences—Computer Science and Computational Biology*. (Cambridge University Press, Cambridge, UK, 1997)
6. N Kappiah, VW Freeh, DK Lowenthal, in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. SC '05. Just in time dynamic voltage scaling: exploiting inter-node slack to save energy in MPI programs (IEEE Computer Society, Washington, DC, USA, 2005), p. 33. doi:10.1109/SC.2005.39. http://dx.doi.org/10.1109/SC.2005.39
7. B Rountree, DK Lowenthal, S Funk, VW Freeh, BR de Supinski, M Schulz, in *Supercomputing, 2007. SC'07. Proceedings of the 2007 ACM/IEEE Conference On*. Bounding energy consumption in large-scale MPI programs, (Washington, DC, USA, 2007), pp. 1–9
8. B Rountree, DK Lownenthal, BR de Supinski, M Schulz, VW Freeh, T Bletsch, in *Proceedings of the 23rd International Conference on Supercomputing*. ICS '09. Adagio: making DVS practical for complex HPC applications (ACM, New York, NY, USA, 2009), pp. 460–469. doi:10.1145/1542275.1542340. http://doi.acm.org/10.1145/1542275.1542340
9. L Wang, Y Lu, in *Real-Time Systems Symposium, 2008*. Efficient power management of heterogeneous soft real-time clusters (IEEE, Washington, DC, USA, 2008), pp. 323–332
10. J-J Chen, K Huang, L Thiele, Dynamic frequency scaling schemes for heterogeneous clusters under quality of service requirements. J. Inform. Sci. Eng. **28**(6), 1073–1090 (2012)
11. M Bambagini, M Marinoni, H Aydin, G Buttazzo, Energy-aware scheduling for real-time systems: a survey. ACM Trans. Embedded Comput. Syst. (TECS). **15**(1), 7 (2016)
12. G Chen, K Huang, A Knoll, Energy optimization for real-time multiprocessor system-on-chip with optimal DVFS and DPM combination. ACM Trans. Embed. Comput. Syst. **13**(3s), 111–111121 (2014). doi:10.1145/2567935
13. C Poellabauer, L Singleton, K Schwan, in *Real Time and Embedded Technology and Applications Symposium, 2005. RTAS 2005. 11th IEEE*. Feedback-based dynamic voltage and frequency scaling for memory-bound real-time applications (IEEE, Washington, DC, USA, 2005), pp. 234–243
14. E Bini, G Buttazzo, G Lipari, Minimizing CPU energy in real-time systems with discrete speed management. ACM Trans. Embed. Comput. Syst. **8**(4), 31–13123 (2009). doi:10.1145/1550987.1550994
15. 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, RVD Wijngaart, T Mattson, in *2010 IEEE International Solid-State Circuits Conference - (ISSCC)*. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS, (2010), pp. 108–109. doi:10.1109/isscc.2010.5434077. http://dx.doi.org/10.1109/isscc.2010.5434077
16. Intel Corporation, SCC external architecture specification (EAS). Technical report, Intel Labs (November 2010). Revision 1.1. Available online at https://communities.intel.com/docs/DOC-5852
17. Intel Corporation: The SCC programmer's guide V(1.0) (2010). Intel Corporation. Available online at https://communities.intel.com/docs/DOC-5684
18. R van der Wijngaart, T Mattson, RCCE: a small library for many-core communication. Technical report, Intel Labs (Jan 2011). Available online at https://communities.intel.com/docs/DOC-5628
19. Many-core Applications Research Community, RCCE_iset_power function causes cores to be unstable. https://communities.intel.com/thread/26799 accessed 19-Apr-2014
20. P Gschwandtner, T Fahringer, R Prodan, in *Cluster Computing (CLUSTER), 2011 IEEE International Conference On*. Performance analysis and benchmarking of the Intel SCC (IEEE, Washington, DC, USA, 2011), pp. 139–149
21. I Buck, T Foley, D Horn, J Sugerman, K Fatahalian, M Houston, P Hanrahan, Brook for GPUs: stream computing on graphics hardware. ACM Trans. Graph. **23**(3), 777–786 (2004)
22. C Grelck, S-B Scholz, A Shafarenko, A gentle introduction to S-Net: typed stream processing and declarative coordination of asynchronous components. Parallel Process. Lett. **18**(2), 221–237 (2008)
23. W Thies, M Karczmarek, SP Amarasinghe, in *Proceedings of the 11th International Conference on Compiler Construction*. CC '02. StreamIt: a language for streaming applications (Springer, London, UK, UK, 2002), pp. 179–196. http://dl.acm.org/citation.cfm?id=647478.727935
24. D Prokesch, A light-weight parallel execution layer for shared-memory stream processing. Master's thesis, Technische Universität Wien, Vienna, Austria (Feb. 2010)
25. V Nguyen, R Kirner, in *Algorithms and Architectures for Parallel Processing*. Lecture Notes in Computer Science, ed. by J Kołodziej, B Martino, D Talia, and K Xiong. Demand-based scheduling priorities for performance optimisation of stream programs on parallel platforms, vol. 8285 (Springer, Vietri sul Mare, Italy, 2013), pp. 357–369
26. VTN Nguyen, R Kirner, F Penczek, in *Proc. 12th International Conference on Algorithms and Architectures for Parallel Processing*. LNCS. A multi-level monitoring framework for stream-based coordination programs (Springer, Fukuoka, Japan, 2012)
27. DW Marquardt, An algorithm for least-squares estimation of nonlinear parameters. SIAM J. Appl. Math. **11**(2), 431–441 (1963). doi:10.1137/0111030
28. JH Anderson, SK Baruah, in *ISCA PDCS*, ed. by S-M Yoo, HY Youn. Energy-aware implementation of hard-real-time systems upon multiprocessor platforms (ISCA, Atlantis Hotel, Reno, Nevada, USA, 2003), pp. 430–435
29. R Bakker, MW Van Tol, AD Pimentel, in *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference On*. Emulating asymmetric MPSoCs on the Intel SCC many-core processor (IEEE, 2014), pp. 520–527. doi:10.1109/pdp.2014.104