CHALMERS

Distributed sleep mode handling and task processing in massive multi-core processors

Master of Science Thesis in the Programme Computer Systems and Networks

KARL STAAF

MARTIN ÅBERG

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, April 2013

Distributed sleep mode handling and task processing in massive multi-core processors

K. Staaf,
M. Åberg

Examiner: J. Jonsson

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden April 2013

Distributed sleep mode handling and task processing in massive multi-core processors

KARL STAAF
MARTIN ÅBERG

Distributed sleep mode handling and task processing in massive multi-core processors

KARL STAAF

MARTIN ÅBERG

Department of Computer Science and Engineering

Chalmers University of Technology

SUMMARY

This thesis addresses the problem of energy efficient real-time task processing in massive multi-core environments. A sleep mode handling method for processor architectures with support for per-core power gating is developed and the solution is inspired by cellular automata. In a cellular automaton each subcomponent has behavior based on local information which together makes up the behavior of the system. In order to evaluate the method, a simulator is tool is developed and used. Presented and evaluated are a set of qualitative measures for benchmarking the sleep mode handling policies.

A key result is that it is possible to construct a distributed core sleep mode handling policy which only depends on local information e.g. the neighboring cores power states, and still performs comparable to policies that are not distributed. Simulation results show that it is possible to save a considerable amount of energy without extending the lateness compared to a system without power handling.

The developed simulator tool is useful for understanding how the system behaves as the number of cores increases beyond 1000 and simulation results show that sudden changes in task load can affect lateness momentarily.

Table of contents

Chapter 1, Introduction

Chips with hundreds of computer cores are already reality, and chips with thousands of cores could easily be manufactured [24]. As the number of cores continue to increase, the question of how to orchestrate these machines, and how to use them to perform maximum amount of meaningful work becomes more and more important. So does the issue of power consumption. The problem of how to schedule work packages or tasks onto such massive multi-core chips is now intimately connected to the question of which parts of the machine to switch on or off, and thereby to the question of power efficiency.

As the Information and Communication Technology (ICT) sector grows, telecom vendor companies such as Ericsson face increased customer pressure regarding energy efficiency. With corresponding annual energy consumption in the mobile network infrastructure per mobile subscriber estimated to around 50 kWh [25], there is clearly a strong economic incentive to work actively on power efficiency optimization in this sector. At the same time, requirements on increased data bandwidth forces telecom vendors into extensive use of multi-core processors. The number of computer cores in a single Ericsson radio base station is even today typically several hundreds.

This work considers sleep mode handling and load balancing algorithms which are distributed and scalable, and which as much as possible rely on local knowledge only, and therefore only affect the system locally. The word *local*, in this sense, would mean the core itself, and its nearest neighbors on all sides. The motivation is to eliminate bottlenecks in the sense of centralized shared data structures. To understand what we are thinking about, one could compare to a two dimensional

cellular automaton, e.g. the one used for *the forest-fire model* [36] or *the Game of Life* [11]. In the case of sleep-mode algorithms, these local decisions should eventually lead to a predictive global system behavior in terms of power consumption and real-time responsiveness.

The problem that we study is how to reduce energy consumption in massive multi-core systems while still providing service. It will also be investigated if there are solutions that scale on systems with thousands of cores.

There has been extensive research done on the field of dynamic voltage and frequency scaling (DVFS) as a mean of reducing power consumption for traditional multi-core processors [5, 13, 27, 29]. With this technique both voltage and processor frequency is changed dynamically in order to save power. DVFS can effectively reduce power in situations where full performance is not required. As DVFS works by only reducing the voltage, leakage current is still present. With power gating it is possible to reduce leakage current while still maintaining fine grained power management if combined with a massive multi-core processor.

The energy efficient scheduling problem is today usually solved by a centralized unit managing both tasks and power. As an example, consider a system with 100 cores and a mean task execution time of 50 µs. The centralized unit must handle up to 2 million status updates from the cores each second. This system is not easily scalable to several hundreds of cores.

For multi-core chips with huge number of cores, it may be advantageous to go for decentralized control of task scheduling and sleep modes, as this would remove possible bottle necks of the system. This work is intended to investigate scheduling and power efficiency questions for such homogeneous massive multi-cores used for implementation of a task processing system with soft real-time requirements. A similar approach has been used in [19, 22] to study thermal properties of task processing multi-core systems, but the used scheduling and power control mechanism was still completely centralized, real-time behavior was not considered, and the number of nodes was never greater than 64. In our case the number of nodes can be virtually unlimited. This will give the system an often very complex behavior even when sleep mode handling and task processing are controlled by a "simple program" for each core individually. Further, no implementation of the studied system may yet be available. Our method is therefore to develop and study a simple simulator. The authors agree that it is important to judge the qualities of any studied system by use of quantitative measures. It can however be hard to foresee the system level consequences of local decisions. Analysis of such behavior is much helped by visualization. The simulator is therefore designed to offer such possibilities.

With our simulator we show that good power saving can be achieved. We have managed to save 70% power of what is theoretically possible using our sleep mode handling algorithms. A fully distributed sleep mode handling algorithm is proposed and evaluated.

Chapter 2, Theory

2.1 Energy aware computing

This section presents background theory on energy aware computing and concepts in real-time systems that are later used in the report.

2.1.1 Static and dynamic power dissipation

The power consumption of a transistor can be divided in two parts: static and dynamic power dissipation. Total power consumption is their sum. Dynamic power consumption is the power dissipated by a transistor when it is switching on and off. This power dissipation is reduced when there is no load on the system. Static power consumption on the other hand is always present even if the transistor is not performing any kind of activity.

According to Nam Sung Kim et al. in the paper "Leakage Current: Moore's Law Meets Static Power" [16], static power dissipation has historically had very little impact on the total power consumption of a transistor. But as technology evolves and transistor manufacturing processes improve, resulting in smaller and smaller transistors, the static power consumption will make up an increasingly larger portion of transistors total power dissipation.

An equation explaining the relation is:

(1) $P = ACV^2 f + VI_{leak}$

Here A is the fraction of gates actively switching, C is the equivalent capacitance load of all gates, V is the supply voltage, f is the frequency and l_{leak} is the leakage current. The first term correspond to dynamic power consumption whereas the latter term corresponds to static power consumption. This equation is presented in [16]. From (1) it can be seen that a change in supply voltage will have a quadratic effect on the dynamic power consumption.

I_{leak} is in itself a sum of $I_{sub} + I_{ox}$ where I_{sub} is sub-threshold leakage and I_{ox} is gate-oxide power leakage. The equation of the first term is:

(2) $I_{sub} = K_1 W e^{-Vth/nV0}(1-e^{-V/V0})$

In (2) K_1 and n are empirically derived, W is the gate width and V_0 is the thermal voltage which increase as temperature increase. To reduce the leakage current of I_{sub} it is possible to either set the supply voltage (V) to zero or to increase threshold

voltage (V_{th}). Increasing V_{th} would however lead to a decrease in performance as presented in [16] and we will in this report only focus on *power gating* which corresponds to setting the supply voltage to zero. The gate-oxide leakage (I_{ox}) is harder to reduce as presented in [16] and we will in fact not focus on this at all.

2.1.2 Power gating

As processor cores consume static power even though they are not utilized, an intuitive way to save more power is to shut them down, or make them sleep, when able to. This technique is called "power gating". It is shown that in a system with processors (cores) that can be turned off individually there is the potential to save additional power in comparison with a system utilizing DVFS alone [8].

Power gating is no free lunch however. There are several issues that need to be solved in order for the technique to be effective [14]. One drawback is that performance will decrease with the introduction of power gating as certain types of transistors used will introduce a voltage drop. This performance loss can be compensated for by increasing the supply voltage although increasing the supply voltage might completely nullify the initial intention of enhancing power consumption efficiency [14].

2.1.3 Core-individual power management

It is possible to perform power gating on an individual core level. This technique is referred to as per-core power gating (PCPG). Utilizing PCPG on a general computing processor it is possible to save 30% more power compared to DVFS and up to 60% in total when used in conjunction with DVFS [18]. However, the use of individual power settings for every core will require as many on-chip-voltage regulators as there are cores, which in turn increase chip complexity as well as chip area and manufacturing cost [14]. A way to work around some these drawbacks is to divide the cores into clusters and have each cluster to be power gated. The drawback of this approach is that a cluster cannot be shut down if it has any cores that are currently active and working, and thus the potential power savings will be less.

2.1.4 The ski rental problem

The ski rental problem asks the question if an occasional skier shall buy or continue to rent his skis under the assumption that he will break his legs on an unknown date. In the case of an early accident it would be more beneficial to rent the equipment, whereas an investment at any point in time may pay itself back in the long run. In computing and resource allocation the same online problem comes up in areas of memory caching [15] and TCP acknowledgements [10]. It is also

applicable in power-mode handling on devices with discrete steps of power such as power gating on multi-processor cores. There is the question of whether an idle core should go to sleep or not when there is no work to perform. In short it depends on when it is expected that new work will arrive.

To formalize the problem, assumed that a transition from the idle state to sleeping is for free but going from sleeping to idle costs a fixed amount, B, of energy. The idle state draws r energy units per time unit. Let T denote the unknown time that the system idles before work has to be done.

An optimal algorithm knowing T at the beginning of the idle period makes the following decision. If T*r is greater than B, then enter the sleeping state, otherwise remain idle. There exists a deterministic online algorithm which is 2-competitive that solves the problem [3]. Being 2-competitive means that for every possible idle period, the energy consumed by the online algorithm is at most 2 times that of the optimal algorithm. The deterministic online algorithm remains in the idle state for B/r time units and then enter the sleeping state [3].

A randomized algorithm exists with expected energy consumption less than 1.58 times that of the optimal algorithm [15]. Even less expected energy consumption can be achieved if the particular application (the length of the idle periods) is known in advance or learned online.

2.1.5 Symmetric multiprocessor systems

A symmetric multiprocessor system consists of two or more homogeneous processors which share a centralized memory. The processors communicate by message passing or by shared memory.

2.1.6 Massive multi-core processors

The homogeneous massive multi-core processor is a class of processors that have multiple, typically hundreds, processing units which are all interconnected by a communication subsystem, Network-on-Chip (NoC) [4]. The cores can have both shared and local memory and use common memory hierarchy employment techniques. A tile-based architecture is presented in Figure 2-1.

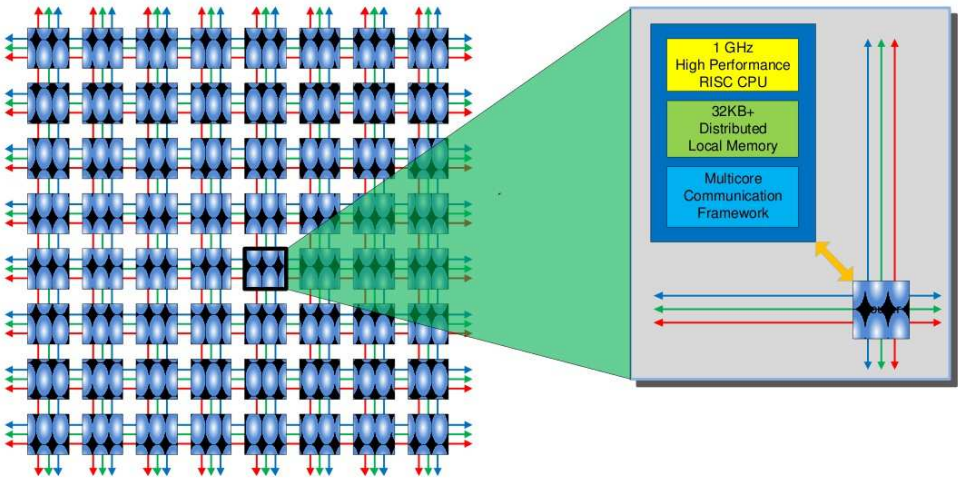

Figure 2-1, The Adapteva Epiphany architecture

The cores in a traditional multi-core architecture communicate by broadcasting on a shared bus. This has the consequence of not being scalable as the number of cores increase and closely located cores interchange data. In the massive multi-core processor architecture, a packet-switching and routing approach to communication is usually taken. This gives good scaling in the number of processors because the communication between neighboring cores does not have to adapt to remotely located ones.

The consequences of a NoC-solution is that core-to-core communication latencies may be unpredictable and that a new problem is introduced into processor architecture design. That is the complexity of network routing. In [4] it is concluded that two-dimensional grid and torus are the most common network topologies in NoC solutions.

Feature-wise the processing units in a homogeneous massive multi-core may contain anything that is suitable for the actual application. Therefore the implemented multi-cores can target fields such as video, network, graphics and cloud computing applications [4]. An overview of actual implementations of massive multi-cores can be found in [4]. In [20], a concept where each tile on a massive multi-core has its own frequency and voltage is presented.

2.1.7 The Adapteva Epiphany architecture

An example of a homogeneous massive multi-core processor is the Adapteva Epiphany architecture [2] which emphases simplicity, scalability and power efficiency. It consists of RISC cores tailored for floating-point operations placed in

a two-dimensional array interconnected with a mesh network. Up to 4096 cores can reside on one chip.

Each core has a local memory and has through the same address space also access to the memory of the other cores. A portion of the memory address is reserved for addressing the local memory of a specific core. There is no memory protection or memory hierarchy. Shared memory is transparently accessed through the NOC, but is affected by relaxed synchronization: the order of memory operations among cores may not always be deterministic. This means that the programmer must take considerations when working on distributed data.

The instruction set architecture features the IDLE instruction which sets the issuing core in a waiting low-power state by means of clock gating. Execution is continued when the core is activated by an interrupt event. A per-core register contains information on the cores chip grid coordinates. This information can be utilized by algorithms that make decisions depending on location.

2.2 Real-time computing

In real-time computing the correctness of a program not only depends on the computational result, but also on the time at which the result becomes available. A common way to model tasks in a real-time context is to assign to each real-time task a set of parameters [17] as will be described here.

A tasks release time is the point in time when the task is ready to begin execution. The deadline is when the task must have completed its execution. A task is periodic if it is released once every P time units where P is the task period.

An upper bound on the tasks execution time is called worst-case execution-time, WCET. If system decisions are based on WCET it is import that this time estimation is realistic and tight. The complexity of calculating WCET increases with the complexity of both the program code and the target hardware. See [35] for an overview of tools and methods for calculating WCET.

Tasks may have precedence constraints which dictate the order in which tasks are allowed to execute. A set of tasks with precedence constraints can be represented in a (directed) precedence graph [17].

Real-time tasks are divided into hard real-time tasks and soft real-time tasks depending on how strict their timing constraints are. When missing a tasks deadline imposes a system failure, the deadline is hard. The deadline is soft when missing a deadline still gives usable results but with lower quality.

An aid in making quality measures of a systems real-time behavior is the *lateness* metric. Lateness is defined as the difference between task finishing time and the deadline [17].

2.2.1 Multiprocessor real-time scheduling

Traditional multiprocessor real-time scheduling can be divided into two major categories; partitioned scheduling and global scheduling. These two scheduling paradigms differ in the way that tasks are distributed onto target processors and cores for execution [26].

Partitioned scheduling consists of two parts. First each task is assigned to exactly one core (partition) and then the tasks on each core are scheduled in isolation using some single-core scheduling algorithm, e.g. *rate monotonic* or *earliest deadline first* [26]. As tasks arrive they are placed in the local queue of the assigned core. No task migration is allowed so a task assigned to a certain core will always execute on that particular core.

In *global scheduling* there exists a single system-wide priority queue where every task resides once it is ready to execute. A central system, scheduler, is responsible for distributing the highest priority tasks on cores.

Crossings between the classes partitioned scheduling and global scheduling exists under the name *hybrid scheduling*.

Regardless of this classification, the scheduler may be *preemptive* or *non-preemptive*. In preemptive scheduling, an executing task may be temporarily be interrupted for a higher priority task to be executed. In the case of global scheduling, there is no guarantee that a preempted task will continue to execute on the same core as it previously did [26].

The scheduling decisions can be made either *online* or *offline*. In offline scheduling, task parameters are known beforehand and every decision is generated before the system is started as opposed to online scheduling where decisions are made as tasks become ready to be executed in the system.

2.2.2 Power-delay product

One performance and quality metric used in NoC and CMOS gate design analysis is the Power-Delay Product, PDP [23], [30]. The definition varies with the domain but common is that some average power is multiplied with a delay to yield a measure of energy. In the case of NoC analysis, the definition is PDP = (average network latency) x (Energy per packet).

In [23], the PDP concept is extended to include the notion of reliability. This is done by defining “Performance, Energy and Fault-tolerance” metric PEF=PDP/(Packet completion probability).

Chapter 3, Related work

This section will give an overview of other publications relevant for this work.

3.1 Thermal-aware scheduling

Thermal-aware scheduling techniques for multiprocessor chips are investigated in [33]. The key concept is to monitor the thermal state of each core and assign processes to cores based on this information. No means of powering down individual cores is mentioned and a system of up to 64 cores is considered.

It is stated that the ideal scenario given a constant workload is when all processors have the same constant temperature, implying zero spatial and temporal diversity. This is said to have positive impact on chip reliability. Given this, a number of scenarios where schedulers that are not aware of thermal information give unfavorable results are presented. One such scenario is when the chip has isolated areas with relatively high thermal stress giving a high temperature gradient. Also temporal fluctuation giving an oscillating temperature profile in a single core is another infeasible scenario.

To minimize the thermal diversities the authors propose thermal-aware scheduler algorithms. The algorithms are suited for implementing in an operating system with centralized process-to-core-assignment running on a multiprocessor system. Scheduling decisions such as the order of processes in the ready queue is not considered, which has the implication that for example real-time behavior is not taken care of by the proposed mechanisms.

The assignment algorithms are stateless, reactive and use input from sensors that monitor the temperature of each core. Being stateless means that no backlog of measurements is kept and that the thermal behavior of the workload is unknown a priory. One complication of demanding temperature sensing for each core is that the number of sensors increases with the number of cores. Also comes the I/O and driver overhead of doing the actual measuring. Though, the authors mention that there exist multiprocessors, for example IBM POWER5, with embedded temperature sensors featuring low sensing overhead.

In the “Coolest” scheme, the scheduler assigns the process to be dispatched to the coolest idle core. The “Neighborhood” scheme is a generalization of the former where the environment of each core is taken into account. This is achieved by calculating for each core, a weighted sum of the temperatures of the core itself, the temperature of the surrounding cores, the number of idle neighboring cores and

also the number of free edges. (A core has a free edge if it is located at the physical border of the chip.) The "Neighborhood" scheme can be reduced into the "Coolest" scheme by zeroing all weights except for the one of the target core's temperature.

The "Neighborhood" algorithm takes into account that a seemingly cold idle core in fact can being warmed up by its neighbors. On the other hand, a hot core located at the chip's corner is guaranteed not to be heated by its missing neighbor and may also benefit from a neighboring heat dissipater.

An extension to the above algorithms is to make decisions based on threshold temperature value. In this scheme the scheduler will postpone a process if it can't find a core with lower temperature cost than a given threshold parameter. Among other things, this will affect real-time properties. Interestingly, according to the authors, using temperature thresholds can actually increase processing performance in cases where the processor uses dynamic thermal management techniques.

The authors discuss the "future of chip multiprocessor" architectures and they also include the term in the title of an article published year 2006. Here, thermal-aware schedulers are said to be necessary to be able to meet the decreasing chip sizes and the increasing number of cores per chip.

3.2 Thermal-friendly load-balancing with power gating

The concept of power gating individual cores in a multi-core processor to achieve power-aware scheduling decisions is investigated in [19]. What makes the work interesting is the study of how the powering down of idle cores affects response time behavior and power consumption given certain workloads.

When waking up a sleeping core, the core is modeled as not being available for execution until after a certain time period, the wakeup latency, has elapsed. Also the power consumption is different in sleep mode, idle mode and executing mode.

It is also assumed that the multiprocessor houses a "load-balancing control unit". This unit is implemented in hardware and has the role of assigning a suitable core to a task. Not much is revealed about the unit by the authors but it is apparently here that the proposed load-balancing algorithm is located. It is certainly a centralized solution and it records all the cores' states and acts according to this information.

An essential tune-in parameter for each of the presented load-balancing techniques is the "waiting idle delay". This is the time interval that an idle core waits before it powers off. The benefit of the waiting idle delay is that the wakeup latency penalty can be avoided if the core is soon being reused. Power consumption becomes higher though.

If there is more than one sleeping core that can receive a process, the problem is to choose which one to wake up. Three regimes for waking up cores are presented by the authors.

In the "Round-Robin" load-balancing technique, each core is powered on in sequence and thus the power consumption is evenly distributed among the cores. The "Lower-Index-First" scheme powers on the sleeping core with the lowest identifier. Cores with low identifiers will make up a thermal hot spot in "Lower-Index-First".

The authors also propose the "Waiting-Idle-First" load-balancing technique. It records the identifier of the core that most recently finished execution, and assigns to that core the next job that arrives. If the most recently finished core is not available, the "Round-Robin" scheme is used. A generalization of "Waiting-Idle-First" (not being mentioned in the article) is to always add the most recently finished core to the back of a FIFO queue and assign the core at the front of the queue to the next job. This scheme needs to have some means of removing cores from the back of the queue as they expire (power off).

Three metrics, latency, power and temperature, are measured for different system parameters. Only a small set of task sets is evaluated by simulation and the results are pure empirical. It is clear that results depend strongly on the properties of the task sets. As workload (CPU utilization) increases, the choice of load-balancing technique and "waiting idle delay" becomes less significant. On the other hand, the choice of scheduling algorithm has great impact burst-like workloads. Waiting-Idle-First is said to give best performance when all three metrics are taken into considerations.

3.3 Clustering of cores

It is possible to reduce the chip area overhead of power gating by sharing the overhead among several cores. Grouping of cores into clusters, where each cluster shares a common power gating facility for all its cores, can be used [22]. The consequence of clustering is that power optimality is compromised for chip area. No specific task processing system is considered in [22] although the proposed load-balancing technique is hardware-based and centralized. Furthermore the load-balancing is stateless which calls for simplicity. Both the power- and thermal properties were investigated.

3.4 Load-balancing packet processors

Scheduling of network flows on massive multi-core processors is the topic of [21]. The author defines a flow as "a sequence of packets from one particular source to a

single destination". Coupled with a flow is a set of jobs that perform operations on the flows packets. One job is started for each packet that arrives.

The characteristics of the processing of packet flows is that the jobs associated with a given flow are highly dependent on each other, but at the same time independent of other flows jobs. Another way to see this is that jobs in different flows can run in parallel whereas the jobs in the same flow have serial constraints among them. Also there is a high communication rate between the jobs in the same flow.

In the context of massive multi-cores where communication cost depends on the cores physical localization, it is desirable to locate jobs of the same flow on cores that are located close together. This is also the load balancing goal as stated in [21]. Spread the processing across the entire chip to prevent thermal hotspots and to increase reliability is a second objective.

A centralized load-balancing technique, named Dynamic Virtual Clustering (DVC), which takes the above in consideration, is proposed. The multi-core is divided into a number (2^{VCL}) of virtual clusters. VCL stands for virtual cluster level and is an algorithm variable that is updated depending on system load. The DVC policy assigns the job associated with an incoming packet to a specific cluster and core.

Each flow is associated with a flow number that will typically be a hash of the source and destination addresses of the packets in the flow. A cluster number is derived from the flow number and the packets job is assigned to that cluster. Note that the number of clusters change dynamically. When the cluster has been selected by the load-balancing policy, a core inside that cluster is selected by using some scheme like round-robin or lowest-index-first. If all the cores in the cluster are busy another cluster and core is selected. By using a hash function on the flow properties, the load-balancer will distribute flows evenly over the clusters and thus the workload is spread evenly across the chip area. At the same time the related jobs are closely located.

When the number of active flows that have packets being processed increase, so does the number of virtual clusters. Very roughly VCL equals the number of currently active flows and the number of virtual clusters is 2^{VCL}. (The exact relation is $VCL=\min(2^{\lceil \log_2(CRF) \rceil}, NC)$ where CRF is the number of currently running flows that have packets being processed by the cores and NC is the total number of cores.)

The presented load-balancing technique requires that the associations between jobs are known before they are being executed. Nothing is said about how the system behaves after a step in the virtual cluster level (VCL).

3.5 Simulating the thermal behavior of a multi-core

In [34] the authors describe a software tool for simulating a model of the thermal properties of a single chip multi-core processor. The tool also simulates task arrivals and it is possible to choose between different core assignment (scheduling) policies. It is possible to tune the model parameters for the simulated multi-core and the task arrival pattern.

The simulator has its main focus on processor chip temperature. For this purpose a comprehensive thermal model for a chip multi-core is presented.

Workloads are generated by the simulator. It is possible to tune the workloads by specifying the average process execution time and system workload. There is no support for creating dynamic task arrival behavior such as bursts arrivals.

A graphical interface presents the multi-core as a square mesh of cores, each showing its local temperature and its running state. It is possible to perform single stepping on the simulators cycle level.

The tool has been used to draw conclusions about thermal-aware scheduling policies [33, 34].

Centralized policies seem to be the only supported scheduling approaches. Also no means of measuring real-time behavior is supported by the simulator (which seems to be a reasonable limitation for this application).

3.5.1 Reliability-Aware Power Management (RAPM)

There is an implication in using DVFS to save power in real-time applications. When the voltage (and frequency) is reduced, it is shown that hardware become much more sensitive to cosmic radiation which in turn leads to more transient faults [9]. In addition, as the voltage and frequency is reduced, the task will take longer time to finish, resulting in less free CPU-time (slack) [12]. This affects the real-time reliability of the system negatively. Real-time reliability is often achieved in two ways; either the same task is executed in parallel over multiple processors with a voting decision made over the results (spatial redundancy), or a task suffering from a transient fault is executed again on the same processor (temporal redundancy) [17]. Temporal redundancy is dependent on that enough slack exist so that a task can execute more than once. This puts power savings with DVFS in direct conflict with the reliability of a system utilizing temporal redundancy as its main method to achieve reliability.

Because of these issues with DVFS, a lot of recent power aware real-time oriented research has been focused on what is referred to as Reliability-Aware Power Management or simply RAPM [6, 9, 12, 32]. The basic way to think about RAPM

is that not all slack is utilized in order to save power. Some slack is left for redundancy executions which may run at full speed in order to utilize as little slack as possible.

Chapter 4, System model

This section describes the system model and assumptions. First the constituting parts of the model will be introduced and then a set of energy mode handling and scheduling algorithms are presented together with an investigation of some of their properties which can be determined offline.

The aim for the core model is to be general enough to map to different present and future massive multi-core chips, and at the same time emphasize the properties of this class of processors. The model is inspired by the NoC processors available from Adapteva and Tilera.

4.1 Task model

In the context of this model, a computation task is synonymous with a job or computer program. Each task is associated with release time, execution time and deadline. The release time is the point in time when the task becomes ready to run. The notion of release time is independent of whether the task can be processed by a processing unit or not at that time instant. Task execution time is assumed to be known before the task starts to execute.

A task is always in exactly one state and the transitions between states are well defined by transition operations. The task states are named ready, running and finished. Figure 4-1 shows all possible transitions between task states.

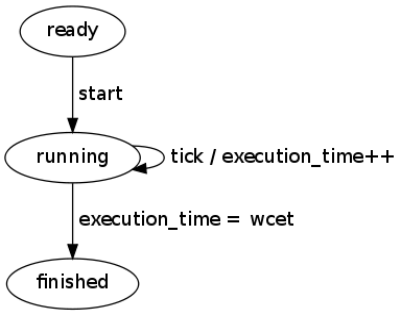

Figure 4-1, The task model state machine

The complete system workload will be modeled as a queue of tasks. When a task is placed in the task queue it is in the ready state. When a task is fetched by a core, the task transitions into the running state where it stays for the duration of execution. Finally the task enters the finished state when its execution is complete.

Any needed activity such as program loading, data input/output etc is considered as a part of the execution. Execution times and deadlines are known as tasks arrive while the actual arrival times are arbitrary.

4.2 Core model

In the model the cores are placed in a two-dimensional grid, each with a local memory and communication machinery. Each core is identified with a unique number. It also has topological coordinates, and thus a well-defined location relative to the other cores. A core has knowledge of, and a direct communication link to, its nearest neighbors in all directions.

4.2.1 The core task manager

Each core has a task manager that can be described as a local operating system or a dispatcher. The task manager is responsible for attaching tasks, one at a time, to the core as well as performing operations on local and shared data structures. When a core has started executing a task, the task will always run to completion. That is, no task preemptions occur. One part of the task manager is the scheduler routine.

4.2.2 Core states and core operations

At every point in time, each core is in exactly one logical state, and the transitions between states are dictated by the core operations issued by the core itself or by a remote core. The core states are *sleeping*, *powering up*, *idle* and *executing* and the core operations that trigger transitions between the states are *enter sleep mode*, *wake up* and *process a task*. Figure 4-2 illustrates the possible transitions between states by means of core operations and task progress. A time delay is associated with the powering up state to model the fact that a sleeping core takes time to become available due to power gating and local hardware and software system initialization. Furthermore, each of the core states are associated with a fixed power consumption. Transition to the powering up state is triggered externally, while the other transitions are issued locally or implicitly.

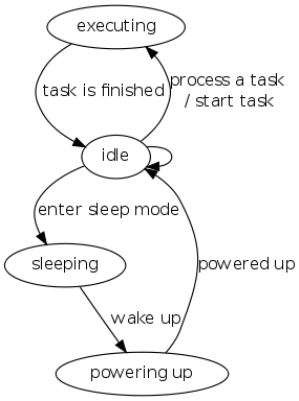

Figure 4-2, The core model state machine

4.3 Distributed power mode handling and scheduling algorithms

The responsibilities of a distributed power mode handling and scheduling algorithm for the presented core and task model are the following.

1. Make sleep mode decisions
2. Pop tasks from task queue
3. Start execution of tasks

Another way to say the same thing is that the scheduling algorithm manages the execution resources and power consumers.

A distributed power mode handling and scheduling algorithm is a routine that runs locally in the core task manager. It can utilize the core operations as well as manipulate data structures that are either local or shared among all core task managers. Every core runs its local algorithm whenever it is in the idle state and all scheduling algorithm are run in parallel.

There exists a centralized task queue from which every core can pop a task. The system model does not cover how tasks are pushed onto the queue, but it is assumed that new tasks arrive in an arbitrary way and out of control of the scheduler. Thus the scheduling problem is *global* in the sense that any core can run any task. The actual scheduling decisions are taken at run-time, so an *online* scheduling algorithm is needed. The situation when all cores are in sleeping state is considered as a system failure no core can ever be activated.

4.3.1 Reasoning about sleep mode handling algorithms

When proposing an algorithm it is important to prove that the algorithm does not enter an infeasible state or stops making progress. Two properties for evaluating the quality of a scheduling algorithm are presented below. The notion I represents the number of cores in, or transition to, the idle state. S and E represent the number of cores in sleeping state and executing state respectively, and |Task Q| is the number of ready tasks in the task queue waiting to be fetched and executed on a core.

Progress property

If not all cores are in executing state (1) and there exist at least one task in the task queue (2) for a time longer than some threshold (3) then after this time has elapsed at least one core will have changed its state to executing (4).

Pre-conditions:

1. I + S > 0, A := I + S
2. |Task Q| > 0
3. Conditions 1. and 2. hold for a duration > threshold

Post-condition:

4. I + S = (A - 1) after threshold

Responsiveness property

If not all cores are in executing state (1), then at least k cores are in idle state (2). The property ensures that the system can serve a certain amount of workload in the task queue.

N := total number of cores in the system

1. IF E < N
2. THEN there are at least min(k, N-E) cores in idle state

With these preparations at hand, a number of algorithms that controls the task assignment and sleep mode handling can be investigated and characterized.

4.4 Algorithm: Fast worker

One main property of this algorithm is that any core has partial knowledge of the systems global state.

4.4.1 Algorithm description

One list and a counter are shared among all cores. The shared list is called the *sleeping list* and contains indexes to the cores that are currently in the sleeping state. The shared counter is called *idle_cores*. Its value is the number of cores that are currently in the idle state.

The algorithm starts to execute when a core enters the idle state and then executes one loop per system time unit. Transition from the idle state occurs when the core decides to go to sleep or when a task is fetched and executed. A textual description is given together with a program flow graph in Figure 4-3.

```
1. when core enters idle state:
2.    idle_cores := idle_cores + 1
3.    idle_timeout := idle_delay

4. loop:
5.    if task queue is empty then
6.       idle_timeout := idle_timeout - 1
7.       if idle_timeout <= 0 AND idle_cores > min_idle_cores then
8.          idle_cores := idle_cores - 1
9.          add self to sleeping list
10.         enter sleeping state
11.      endif
12.   else
13.      pop a task from the task queue
14.      if  sleeping_list.length > 0
15.         wake up n sleeping cores
16.      endif
17.      idle_cores := idle_cores - 1
18.      enter executing state (process the task)
19.   endif
20.endloop
```

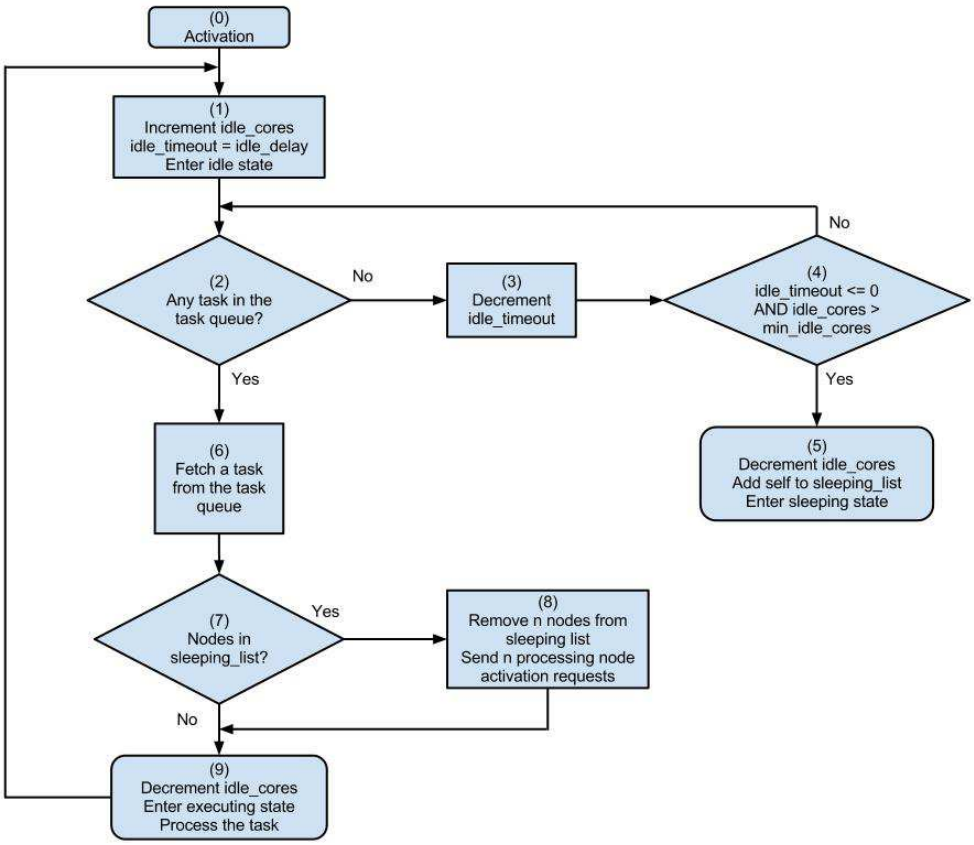

Figure 4-3, Fast worker algorithm

The algorithm has a parameter *min_idle_cores* which is a goal for the minimum number of idling cores in the system. The number of cores to wake up before processing a task is given by the parameter n. Which cores to wake up is chosen random. Every core has a timer which controls how long a core is idling. The algorithm integer parameter *idle_delay* decides the timeout value.

4.4.2 Characteristics, features and properties

The fact that two data structures are shared among all the cores makes this algorithm less distributed. Given a fixed per core average load, then as the number of cores in the system increases, so also does the number of operations on the two shared lists.

The goal is that at least a treshold of *min_idle_cores* cores are ready to accept new work. When a burst of tasks arrive, active cores will immediately wake other cores before they start to process one of the tasks. Eventually the second generation of cores will in turn follow this behavior. If there exist additional work when the first

cores have completed processing their tasks they will pick new tasks, but before that they will again wake yet more cores from their sleeping state. In this way the wake up rate of cores is exponential.

Because of its fast wake up rate, the Fast worker algorithm is suitable in cases when the workload increases in big steps.

4.4.3 Feasibility

It will here be investigated which of the feasibility properties defined above hold for the Fast worker algorithm.

Progress property

We identify three possible scenarios and prove weather the progress property holds.

Scenario 1: There is at least one idle core.

Proof: According to the algorithm *Fast worker*, a core that is idle will run the idle loop as specified by lines 4-19. Should there be anything in the task queue at the time that the core checks the task queue, that task will be assigned to the core according to line 16. The core will then enter executing state according to line 17.

Scenario 2: There are no idle cores, at least one core is executing and the rest are sleeping

Proof: Assuming a situation where no idle core exists but there is at least one core executing. If a new task arrives at this point in time there will be no core available to process that task. However, as task execution time is bounded, any of the cores that are currently in executing state will eventually finish and transition to idle state. In the idle state, a core will wake up other cores if the task queue is non-empty, and then start execution. Now when the new cores are waken up, they will in turn enter executing state as needed, like in scenario 1.

Scenario 3: There are no idle or executing cores, only sleeping cores.

Proof by contradiction: For every core to enter sleep mode it would require that the last core awake should make the decision to also go to sleep. This is however impossible as guaranteed by (1) in the algorithm, assuming that the idle_cores shared variable is correctly updated.

Responsiveness property

There exist two scenarios that may break this property.

Scenario 1: Every core enters sleep mode

Proof: This is impossible as presented in scenario 1 in the **progress property** above.

Scenario 2: Some cores are executing and some cores are sleeping while no cores are in idle state (or in transition to idle state).

Proof by contradiction: Assuming that there are cores in sleeping state and cores in executing state but no core is in idle state (or in transition to idle state). It has already been proved in scenario 1 that it is impossible for a single core that is among the last ones, as specified by the min_idle_cores variable, to enter sleep mode. Then the only way for scenario 2 to occur is if there are no cores woken up when a single idle core transitions into executing state. In the *Fast worker* algorithm there is however at least one core activated **before** the previously idle core is starting to execute its task (as long as there are sleeping cores to activate), as seen on (8) and (9). So scenario 2 must be an impossible scenario.

4.5 Algorithm: Friend worker

In the quest for distributed solutions to the task scheduling and core activation problem, the shared sleeping list is now dropped. The incentive is that the need to lock a data structure and thus delaying transactions on the centralized data will decrease.

4.5.1 Algorithm description

In this algorithm, the global counter *idle_cores* is still used and it has the same meaning as in the Fast worker algorithm. The use of a central sleeping list is replaced by forcing a core to instead of consulting the shared sleeping list, try to wake up some of its nearby neighbors. A core may be able to wake up zero or more of its neighbors depending on their current state. This requires that each core has knowledge of the neighbor's actual states, which is assumed to be provided by the runtime system.

The Friend worker algorithm is presented in the code below and Figure 4-4.

```
1.  when core enters idle state:
2.     idle_cores := idle_cores + 1
3.     idle_timeout := idle_delay

4.  loop
5.     if task queue is empty then
6.        idle_timeout := idle_timeout - 1
7.        if idle_timeout <= 0 AND idle_cores > min_idle_cores then
8.           idle_cores := idle_cores - 1
9.           enter sleeping state
10.       endif
11.    else
12.       pop a task from the task queue
13.       wake up n sleeping neighbors
14.       idle_cores := idle_cores - 1
15.       enter executing state (process the task)
16.    endif
17. endloop
```

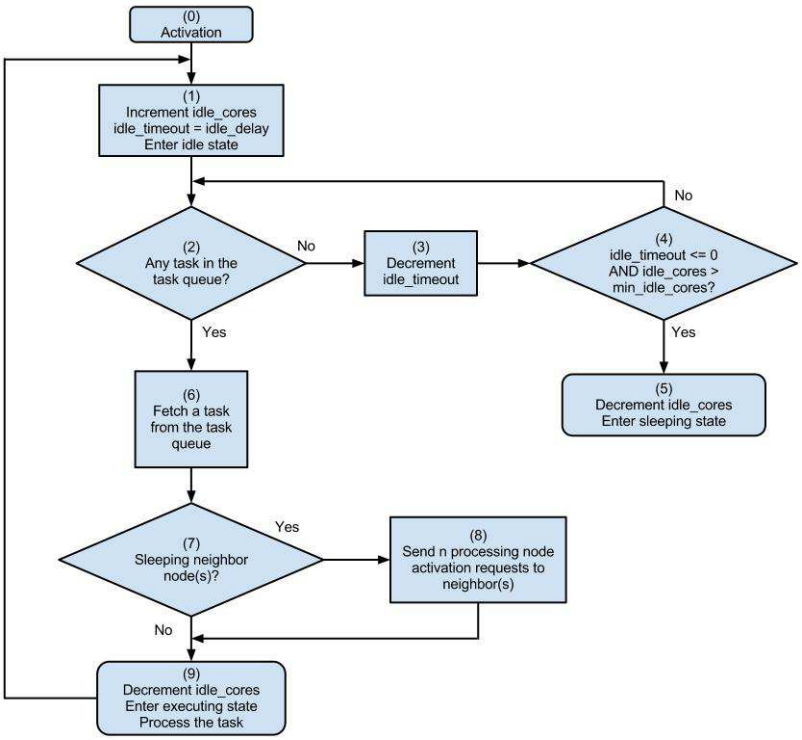

Figure 4-4, Friend worker

4.5.2 Characteristics, features and properties

The requirement that each core has local knowledge of its neighbors' state has no implications on scalability. Each core has the same number of neighbors independent of how many cores that makes up the full system. There are scenarios

where a core fails to wake up any of its neighbors as when it for example is surrounded by executing cores as illustrated in Figure 4-5. This may affect the responsiveness of the system. On the other hand the system makes progress over time as all execution times are finite, and when the executing, *blocking*, cores eventually enter idle state, they will process the workload if still available.

This makes the algorithm suitable for workloads with moderate increase in workload. For a constant workload, the Friend worker is more energy efficient than the Fast worker. It is because Fast worker will always wake up new cores for every arriving task, while Friend worker does not as described above. The powering down behavior of cores is the same as in Fast worker.

4.5.3 Feasibility

It will here be proved whether the feasibility properties hold for the Friend worker algorithm. If they do not, it is explained why.

Progress property

Scenario 1: There is at least one idle core.

Proof: According to the algorithm *Friend worker*, a core that is idle will run the idle loop as specified by lines 4-17. Should there be anything in the task queue at the time that the core checks the task queue, that task will be assigned to the core according to line 14. The core will then enter executing state according to line 15.

Scenario 2: There are no idle cores, at least one core is executing and the rest are sleeping

Proof: Assuming a situation where no idle core exists but there is at least one core executing. If a new task arrives at this point in time there will be no core available to process that task. However, as task execution time is finite, any of the cores that are currently in executing state will eventually finish and transition to idle state. When such a core enter idle state, it will wake up new cores and start a new task. As the cores are waken up, scenario 1 takes over.

Scenario 3: There are no idle or executing cores, only sleeping cores.

Proof by contradiction: For every core to enter sleep mode it would require that the last core awake should make the decision to also go to sleep. This is however impossible as guaranteed by line 7 in the algorithm, assuming that the idle_cores variable is correctly updated.

Responsiveness property

Scenario 1: Every core enters sleep mode

Proof: This is impossible as presented in scenario 3 in the *progress property* above.

Scenario 2: Some cores are executing and some cores are sleeping while no cores are in idle state (or in transition to idle state).

Negative proof: In the *Friend worker* algorithm, as it is only possible for a core to communicate with a certain set of its neighbors, there might arise a scenario where all the neighbors of a core are busy in executing state. In that case, the core with busy neighbors will not be able to activate any other core before it will transition into executing state and the result will be that no core will be in idle state even though that there may be a lot of cores sleeping, see Figure 4-5.

If the neighbor set of a core contains at least one sleeping core then this scenario is the same as the one for the scenario 2 for the *responsiveness property* of the *Fast worker* algorithm with the same positive proof.

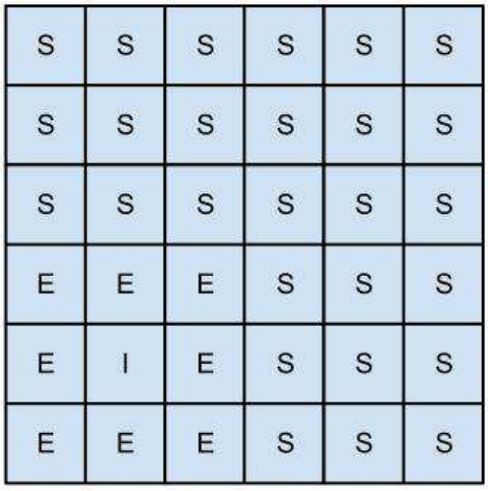

Figure 4-5, A single idle core (I) with only executing neighbors (E). No sleeping core (S) can be activated by the idle core if the neighborhood is set to the eight closest cores.

4.6 Algorithm: Path home

Now let's see if also the *idle_cores* counter can be dropped, and what the consequences of that would be. As it must be ensured that not all cores are in the sleeping state, the algorithm needs some guard for keeping the system awake. One possible solution would be to pick a single core, call it the sink core, and forbid that

particular core to ever enter the sleeping state. Then every other core can always fall asleep knowing that it will not cause the system to die. But that would also imply that the sink core can never execute a task. Because if it did, then it could happen that the sink is the only core that executes when all other cores are in the sleeping state. This would starve the system until the sink finishes execution. It is clearly an infeasible scenario. In the following, an algorithm that allows the sink to execute while not being responsible for starving the system is presented.

One core is selected as the sink core for the lifetime of the system. The algorithm assumes that each core is aware of the state of its four closest neighbors. Also each core has a private constant set, *path_to_sink*, which contains references to the neighbors which are closer to the sink than the core itself. The set contains one or two references, except for the sink core whose *path_to_sink* set is empty.

Before a core enters sleep mode, it checks to see if the cores in the *path_to_sink* set are sleeping. If they all are, then one of them is waken up. In this way the responsibility of locating the sink is forwarded and moved closer to the sink. This policy sets up paths to the sink where active cores move along.

4.6.1 Algorithm description

The algorithm assumes that each core is aware of the state of its neighbors. Also each core has a private constant set, *path_to_sink*, which contains references to the neighbors which are closer to the sink than the core itself. In the case of a two dimensional topology, the set contains one or two references, except for the sink core whose *path_to_sink* set is empty. Below is the algorithm description together with a program control flow graph in Figure 4-6.

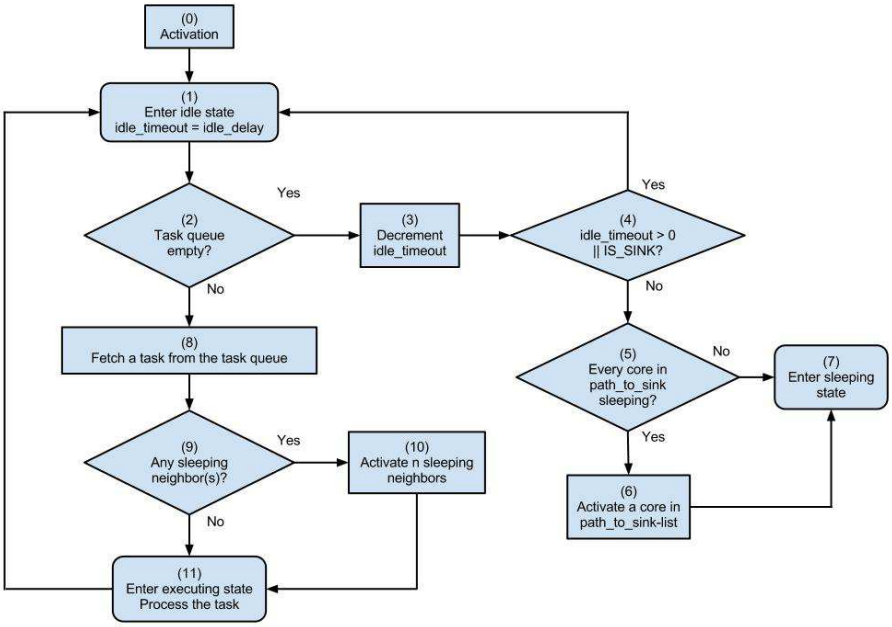

Figure 4-6, Path home algorithm

4.6.2 Characteristics, features and properties

In situations of constant workload, the algorithm behaves as the Friend worker. When going from a high workload to a lower workload, the response time is held high for some time because of the presence of idling cores on their way to the sink. This makes the algorithm suitable for periodic workloads or when load peaks are following each other.

The most characteristic feature of the Path home algorithm is the lack of data structures shared between the cores for handling power modes. There could be a number of sink cores in the system. This number then corresponds to min_idle_cores parameter of the previous two algorithms. Non-sink nodes path_to_sink list will then contain the neighbors on the path to the closest sink.

4.6.3 Feasibility

It will now be investigated whether the feasibility properties hold for the *Path home* algorithm.

Progress property

Scenario 1: There is at least one idle core.

Proof: According to the algorithm *Path home*, a core that is idle will run the idle loop as specified by lines 3-16. Should there be anything in the task queue at the

time that the core checks the task queue, that task will be assigned to the core according to line 13. The core will then enter executing state according to line 14.

Scenario 2: There are no idle cores, at least one core is executing and the rest are sleeping

Proof: Assuming a situation where no idle core exists but there is at least one core executing. If a new task arrives at this point in time there will be no core available to process that task. However, as task execution time is finite, any of the cores that are currently in executing state will eventually finish and transition to idle state. Now some of these finished cores will wake up at least one neighbor and scenario 0 takes over.

Scenario 3: There are no idle or executing cores, only sleeping cores.

Proof by contradiction: For every core to enter sleep mode it would require that the last core awake should make the decision to also go to sleep. This is however impossible as the *Path home* algorithm use a sink core that may never go to sleep.

Responsiveness property

There exist two scenarios that may break this property.

Scenario 1: Every core enters sleep mode

Proof: This is impossible as presented above in the *progress property,* scenario 3.

Scenario 2: Some cores are executing and some cores are sleeping while no cores are in idle state (or in transition to idle state).

Proof: See the proof for this scenario and property for the *Friend worker* algorithm.

4.7 Algorithm: All active

In the *All active* power mode handling and task processing algorithm, an idle core never falls asleep once it has been activated. It can be seen as a special case of the *Fast worker* algorithms, with the minimum idle cores parameter set to the number of cores in the system. The All active algorithm is useful in comparing online algorithm qualities, and in particular it gives an upper bound on the system power consumption.

Chapter 5, Experiment setup

The distributed power mode handling and task processing algorithms are evaluated by running a multi-core simulator with a variety of configurations. How the actual system workload is designed may affect which policies that will be efficient, and is thus a central part of our work. Three experiment series are carried out to investigate properties of the energy- and load balancing algorithm.

No attention is given to the absolute time and power values or their physical units because it is the relative times and powers that matter. Hence generalized power units and time units are used. For simplicity, we consider the power consumption of the cores to be 50% leakage and 50% dynamic. This corresponds to a power consumption of 1 power units for the core states *powering up* and *idle*, and 2 power units for the *executing* state. No power is assumed to be consumed when a core is in the *sleeping* state. Our model does only consider one central task queue for the complete system.

5.1 The simulator

The system model is simulated in a computer program that captures and visualizes the behavior of the system. This simulator is constructed by the authors and is implemented using the computer programming language *Python*, the graphics toolkits *Tk* and the plotting library *matplotlib*. A screenshot of the simulator is provided in Figure 5-1. All simulator parameters can be controlled by a configuration file read by the simulator. This allows for running a set of simulations in batch mode.

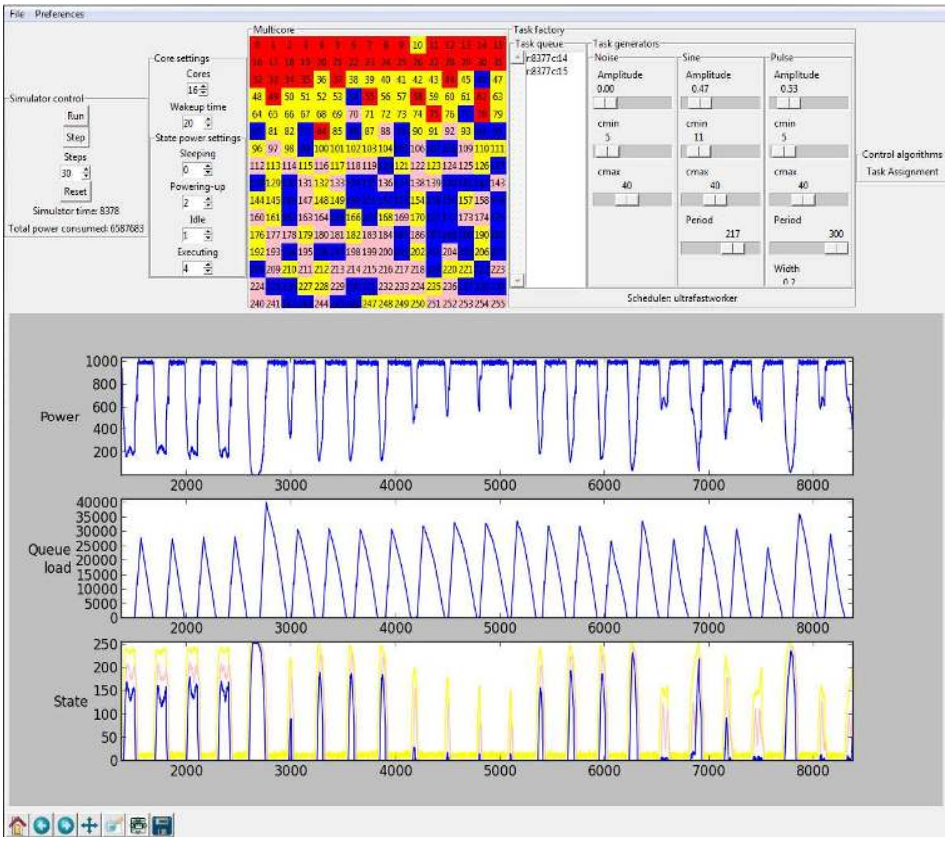

Figure 5-1, The simulator used for evaluation of core sleep mode handling and task scheduling.

The first column in Figure 5-1 contains buttons for starting and stopping the simulator. Next to it are the system configuration parameters. The grid in the top middle shows a representation for a chip grid. In this example the chip is made up of 16 columns and 16 rows of cores. The fields representing cores are color coded according to core state: blue is *sleeping*, pink is *powering up*, *yellow* is idle and red is *executing*. Colours are updated at run-time according to state transitions.

Next to the core area is the task queue. Each task in the queue is represented by a textual description showing when the task arrived at the queue together with its execution time. A task disappears from the task queue as it is acquired by a core. The *task generators* at top right in the figure allows for injecting tasks in the system interactively. Task load with the shape of noise, sine or pulse can be generated with different amplitude and period. In the simulation runs presented later in this report, the task generators are turned off and tasks are instead injected by pre-made task load data files.

A visualization area at the lower part of Figure 5-1 displays run-time statistics in real-time. The *power* graph displays the momentary, per time unit, power consumption. *Queue load* is the sum of all execution times currently in the task queue. At the bottom is the *state* graph which illustrates the distribution of core states.

5.2 Task load

We have tried to design a workload that mimics a simplified but typical LTE radio base station receiver baseband similar to what is described in [31] and [1]. The tasks are processed in a run-to-completion manner. In a real base station, time deadlines are set equal for groups of tasks that together perform a complete work package corresponding to e.g. all signal processing needed for reception of a data transport block of a mobile user. The deadline would then be limited by the standard requirements on delay time for transport block CRC ACK/NACK signaling. In our current experiments, deadlines are set individually for each task as our model yet cannot handle ordered task graphs.

For the purpose of evaluating the previously presented algorithms, the simulator is fed with fixed task sets. The task sets consist of release times paired with execution times and deadlines, e.g. one task is released at time t_i with execution time c_i and deadline d_i.

For every task the execution time is uniformly distributed and ranges from 10 to 80 simulator time units. A task sequence is Poisson distributed and spans 500 time units, with a peak at the first few time units. Each such task sequence has a well defined average load for the period.

The task sets devised are made up of concatenated task sequences, all with their own average load. In this way the workload of arriving tasks vary over time and thus it is possible to capture potential execution scenarios. The three task load sets, *ramp*, *peak* and *overload*, are presented in Figure 5-2, Figure 5-3 and Figure 5-4. Percentages in the figures represent the fraction of the systems total computational capacity. These task sets are used throughout the experiments and the workload is scaled as the number of cores varies.

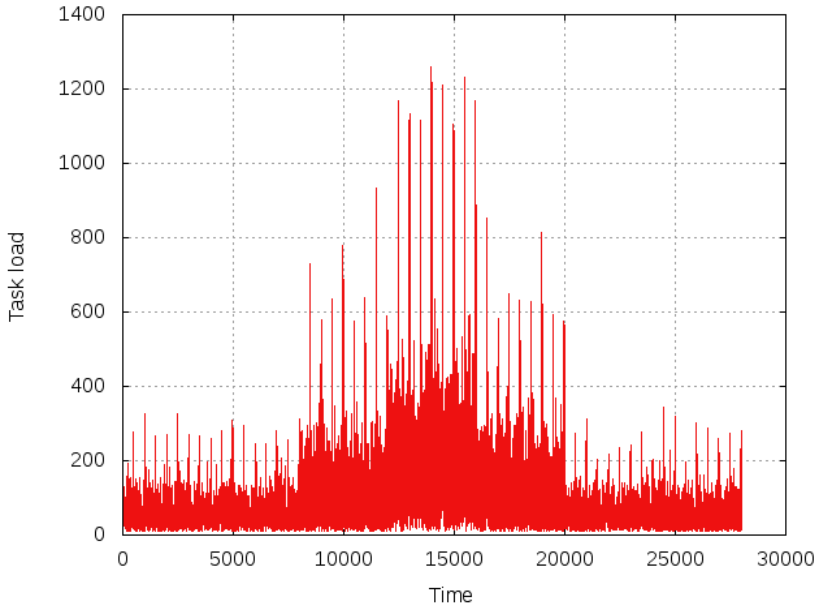

Figure 5-2, Ramp task set for 256 cores. The load sequence is 10%, 40%, 80, 40%, 10% of the systems computational capacity.

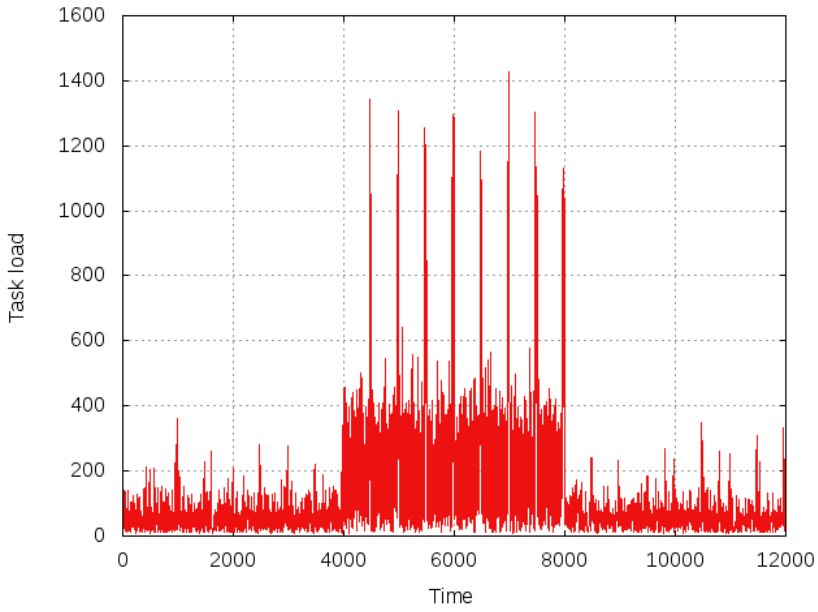

Figure 5-3, Peak task set for 256 cores. The load sequence is 10%, 90%, 10%.

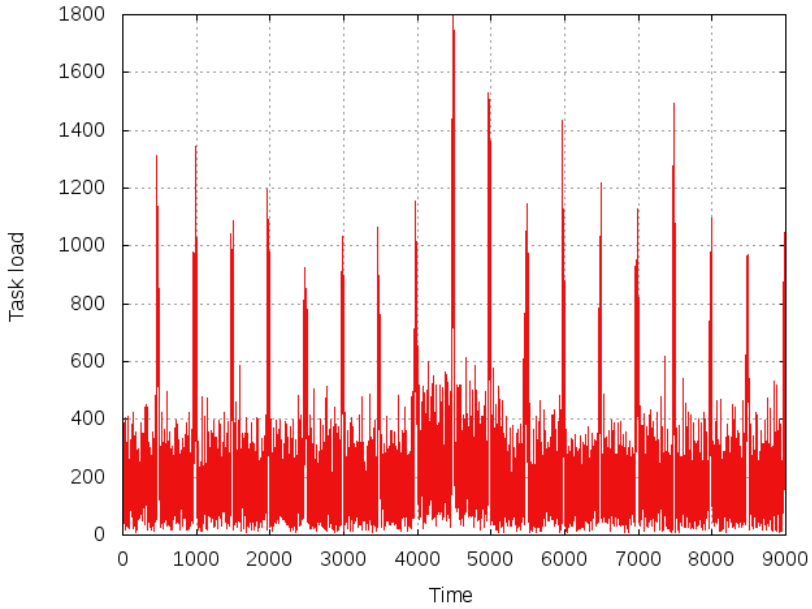

Figure 5-4, Overload task set for 256 cores. The load sequence corresponds to 120% of the systems computational capacity inside the interval 4000 to 5000 time units, and 80% otherwise.

5.3 Power consumption lower bound

Given a task set, it is possible to estimate a lower bound on the energy consumption for the set, for any online algorithm. The optimistic bound that will be used is calculated as follows. Multiply the task sets sum of execution times with the executing state power consumption. Then divide this product with the time span of the task set and the number of cores. The time span is the time from that the first task arrives until the last task finish executing.

It is important to stress that this construction really gives a lower bound because no power consumed in the powering-up state nor in the idle state is considered. On the other hand, exactly the power needed to process the tasks in the execution state is accounted for.

5.4 Simulation parameters

The simulated system and the algorithms have a set of parameters that can be set and tuned according to system constraints, system requirements and task sets.

5.4.1 System parameters

Systems with 64, 256 and 4096 cores are investigated. Transitions from *sleeping* state to *idle* state are chosen to take 20 time units, which is approximately the

length of a short task. Power consumption for the core states is 0 power units for sleeping, 1 unit for powering-up, 1 unit for idle and 2 units for executing.

5.4.2 Algorithm parameters

Idle delay, which is the minimum time a core is in the idle state before going to sleep, is selected as one of 5, 10, 20, 40, 160 or 640 time units. The number of minimum idle cores applies to the Fast worker and Friend worker algorithms are selected depending on the number of cores. The number of cores to preferably be woken up by the algorithms is set to two. This corresponds to setting the parameter n = 2 in the scheduling algorithms.

5.5 Metrics

We are using different metrics to evaluate the qualities of the algorithms and configurations.

- The *average power* metric is defined as the total energy consumed by the chip cores divided by the number of cores and simulation time. This metric measures how energy efficient the execution is.
- The *average queue time* metric is defined as the average time a task is in the task queue before being executed. This metric measures the responsiveness of the system when loaded with the given task set.
- The *power-delay product* (PDP) is constructed as the product of metrics *average power* and *average queue* time. This gives the possibility to compare these metrics between different configurations with one single metric.
- *Average lateness* represents the total lateness divided by the number of tasks. This metric is used for measuring real-time properties. In our lateness calculations we define task lateness to zero if the deadline is met.

Chapter 6, Queue time and Power-delay-product analysis

The purpose of this experiment is to investigate how the sleep mode handling and task processing algorithms impact the average queue time and also how PDP can be used as a measure. As average queue time measure is not connected to deadlines, the results in this chapter are independent of deadlines.

6.1 Results

Presented below is a set of graphs showing PDP for scheduling algorithms Fast worker, Friend worker, Path home and All active. The algorithms are evaluated with the task loads presented in section 5.2 The vertical axis represents the PDP value, and the scheduling parameter *Idle delay* is on the horizontal axis. *Idle delay* represents the idle_delay parameter of the algorithms and denotes the amount of time a core idles before entering sleep mode. Scheduling algorithms Fast worker and Friend worker are represented in two versions where the difference is *min*, the algorithm threshold parameter min_idle_cores, which is the goal for the minimum number of idling cores. For 64 cores the parameter is either 1 or 25 and for 256 cores it is 1 or 81.

6.1.1 Task set: Peak

Figure 6-1 shows the PDP results of running the simulator with 64 cores.

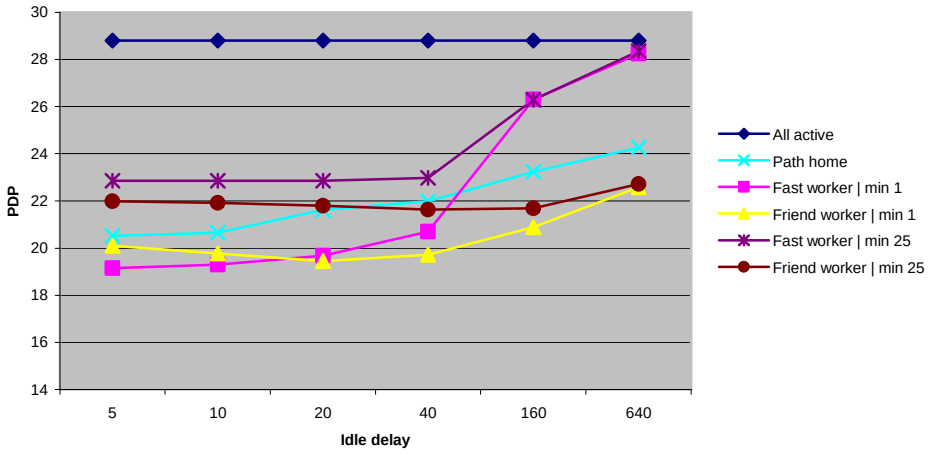

Figure 6-1, Power delay product for task set Peak, evaluated on 64 cores.

We notice that running with all cores powered on all the time (All active) results in a PDP of 28.8. The algorithm with the lowest PDP is “Fast worker | min 1” with a PDP of 19.1 which is 34% less than All active. Each proposed algorithm has at least 20% lower PDP than algorithm All active when the Idle delay is less than or equal to 40. Fast worker with 1 minimum active core performs best with low idle delay but worst (together with its 25 minimum active cores counterpart) when the Idle delay approaches 640.

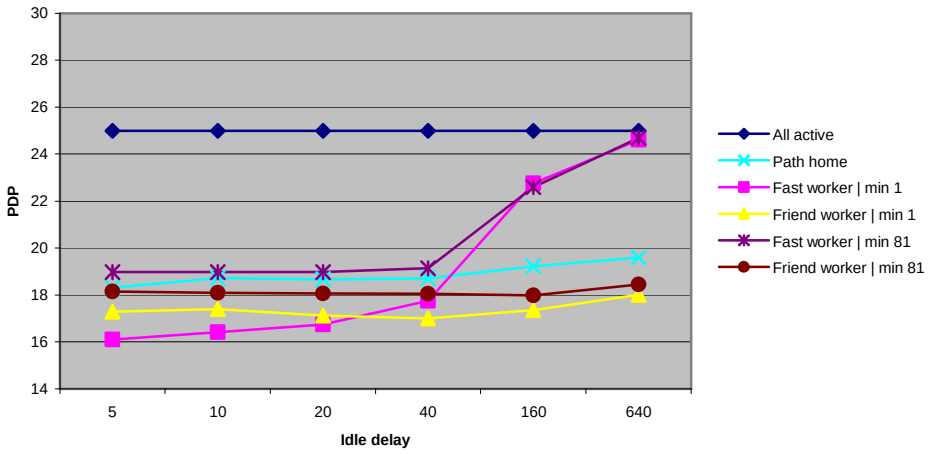

Figure 6-2, Power delay product for task set Peak, evaluated on 256 cores.

On 256 cores we see lower PDP values according to Figure 6-2, Power delay product for task set Peak, evaluated on 256 cores.. Running with no sleep mode (All active) at all result in a PDP of 25. Again, the algorithm with the lowest PDP is “Fast worker | min 1” with a PDP of 16.1. This is 34% less than the “All active” scheduler. Every algorithm has at least 24% lower PDP than scheduling algorithm “All active”, when the Idle delay is less than or equal to 40. “Fast worker | min 1” performs best with low idle delay but worst (together with its 81 minimum active cores counterpart) when idle delay approach 640.

6.1.2 Task set: Ramp

Figure 6-3 shows that the PDP results of the Ramp task set simulated on 64 cores are similar to those of the Peak task set. With the Ramp task set the maximum PDP of the “All active” scheduler is 9.1. The algorithm yielding the lowest PDP is “Fast worker | min 1” with a PDP of 6.2 which is 31.9% less than “All active”. The same algorithm together with it’s “| min 25”-core counterpart has the highest PDP when the idle delay approaches 640. With an idle delay of 40 or lower all scheduling algorithms are at least 20% better than the “All active” scheduler.

On 256 cores, we notice a few differences. “Friend worker | min 1” actually improves as the idle delay increases but its PDP with a low idle delay is much larger (7.9) than for 64 cores (6.9).

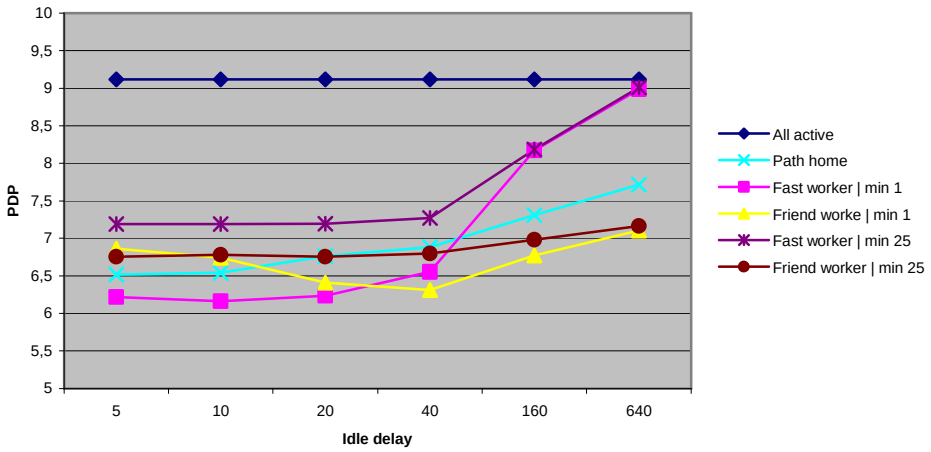

Figure 6-3, Power delay product for task set ramp, evaluated on 64 cores.

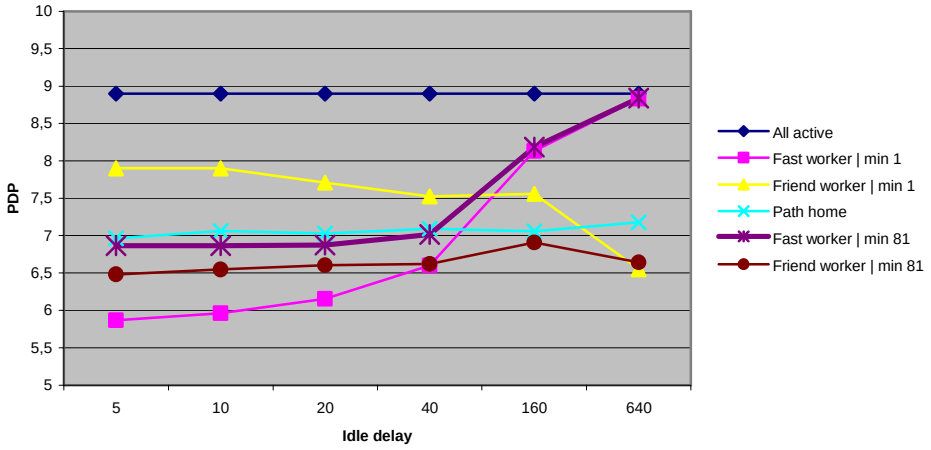

Figure 6-4, Power delay product for task set Ramp, evaluated on 256 cores.

It is interesting to note in Figure 6-4 that the algorithms Fast worker and Friend worker indicate disparate behavior. First, the Fast worker algorithm shows growing PDP for higher idle delay values, all other parameters being the same. At the same time, Friend worker gets lower PDP with increasing idle delay. Also the minimum idle cores threshold parameter impacts the PDP metric differently. Especially the Friend worker algorithm shows lower PDP results when increasing the threshold as

seen in Figure 6-4, whereas Fast worker does not. What is also evident is that this behavior depends on the task set and the number of cores.

Average power consumption for the ramp task set on 256 cores is studied in isolation by means of the average power graph in Figure 6-5. The calculated theoretical lower bound sets a lower limit on the power consumption, and at the same time the "All active" scheduling algorithm is an upper limit. Other sleep-mode handling algorithms fall in between.

In Figure 6-6, the average queue time for the same simulation run is presented. No theoretical lower bound on the queue time is calculated, although an over-optimistic one would be zero.

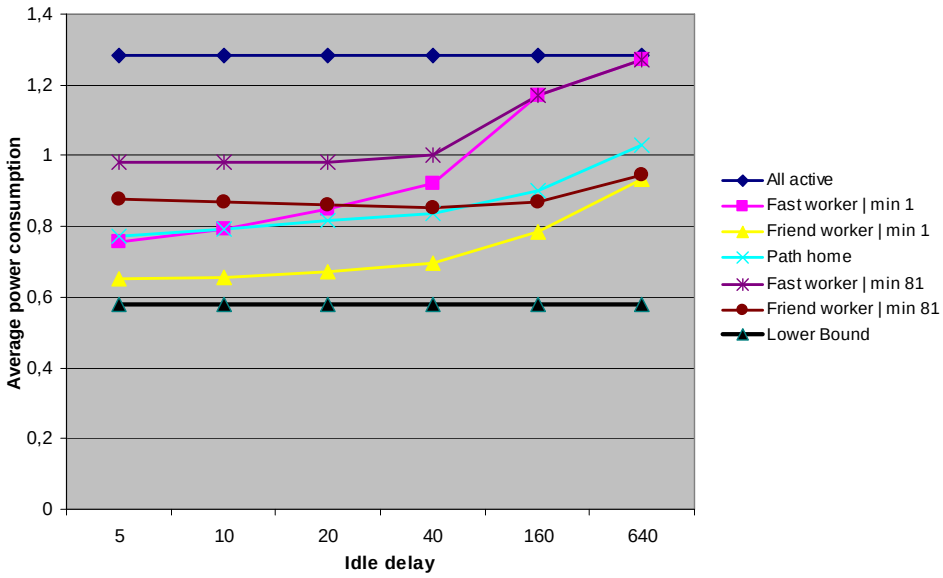

Figure 6-5, Average power consumption for task set Ramp, evaluated on 256 cores.

It is observed that the Friend worker algorithm and Path home algorithm have the common property of being stable in PDP when the idle delay setting is varied but the other parameters remaining the same. Furthermore these two algorithms show the same power consumption behavior as the idle delay is increased, compared to that of the Fast worker algorithm whose power consumption reaches that of the All active algorithm at an idle delay of 640. This can be seen in Figure 6-5.

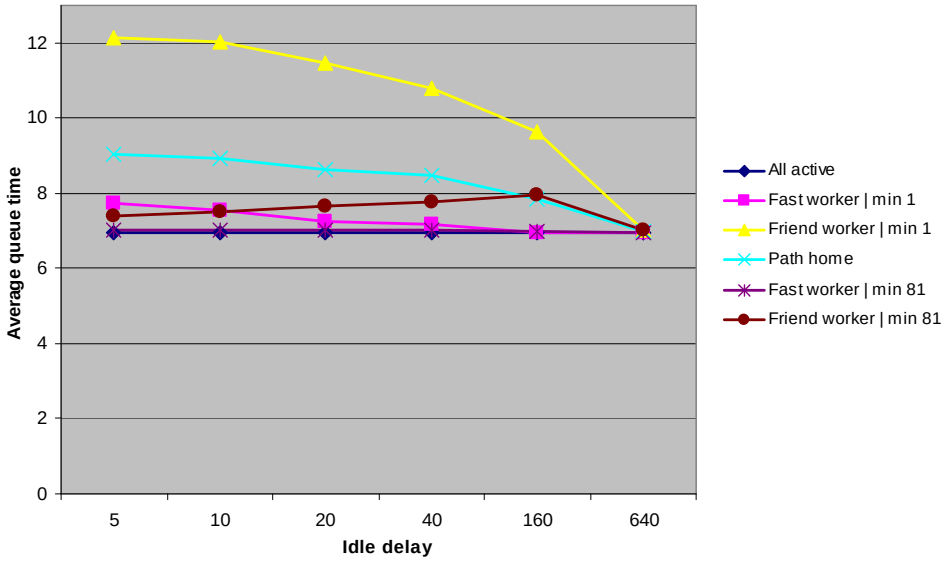

Figure 6-6, Average queue time for task set Ramp, evaluated on 256 cores.

6.1.3 Task set: Overload

The situation is changed when investigating the PDP-results for the overload task set in Figure 6-7. Here we see that the algorithm with the highest PDP result is the one with no sleep mode handling. However "Fast worker | 25" reaches the same PDP result, 80.5, as "All active". "Friend worker | 25" is also close to this value. With an idle delay of 160 there is little difference between any of the algorithms.

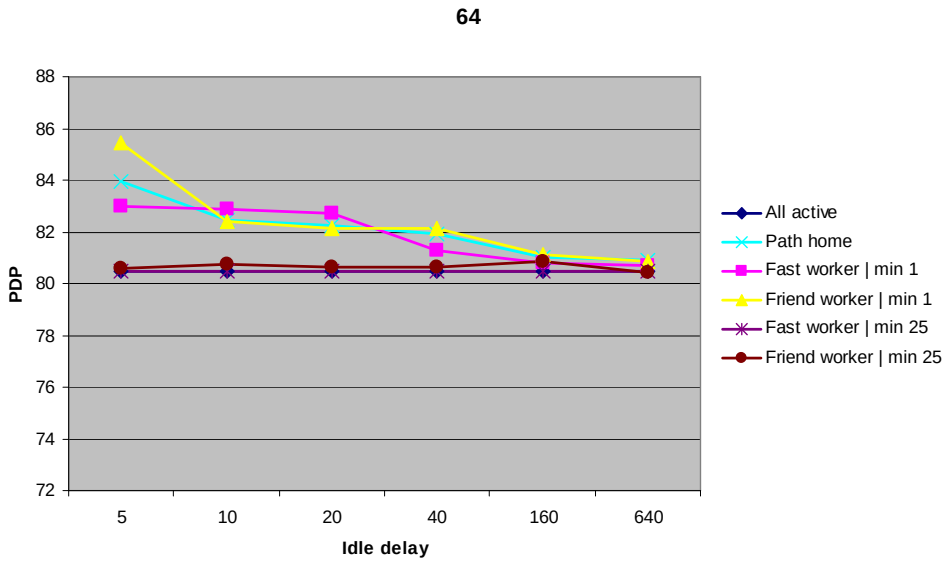

Figure 6-7, Power delay product for task set Overload, evaluated on 64 cores.

In Figure 6-8 we see that "Path home" and "Friend worker | 1" has a higher PDP than the others. None of these algorithms ever manage to reach the levels of the "All active" scheduler. Of the algorithms with distributed sleep mode handling, "Fast worker | 81" is the one with the lowest PDP.

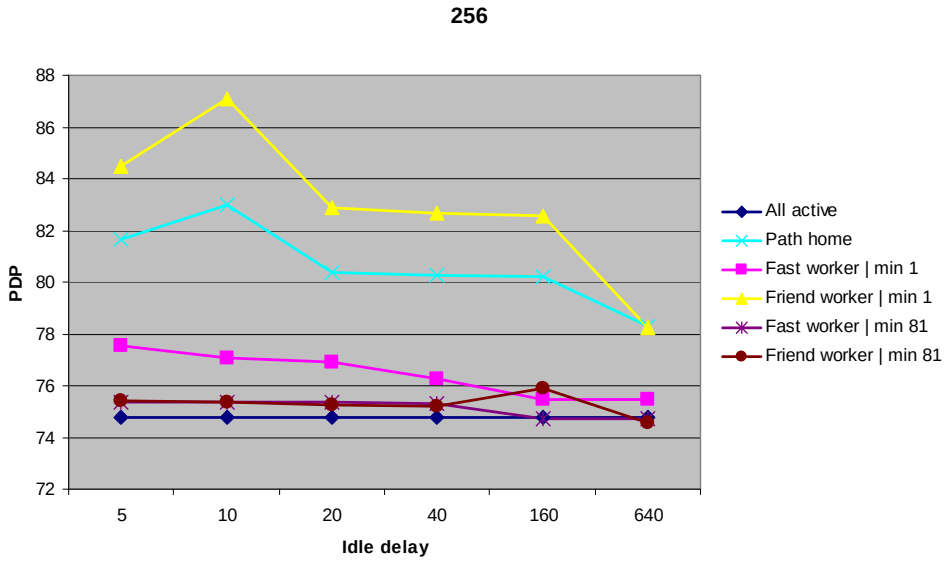

Figure 6-8, Power delay product for task set Overload, evaluated on 256 cores.

6.2 Discussion

6.2.1 Fast worker adapts fast to increased workload

One general observation is that the Fast worker algorithms give good performance in terms of PDP when the idle cores threshold is low and at the same time the idle delay is short. Also the average queue time is close to optimum. This holds for the ramp and peak workloads. The property comes from the fast adaption to a sudden increase in workload inherit in Fast worker. Namely, the algorithm can always activate the number of cores it requests, as long as the system is not overloaded. The algorithms shared counter makes this possible.

6.2.2 Friend worker and Path home have damped waking-up behavior

The similarity between Friend worker and Path home as described in section 6.1.2 can be explained by the damping mechanism inherent in their waking-up schemes: it is not always possible for a core to wake up its wanted number of cores. The more filled a particular chip area is with idling or executing cores, the fewer cores in that particular area can be woken up. If all the cores in this hot spot area are active then the area can only grow outwards, so the number of cores that are waken up in a peak load situation is dependent on the perimeter of that hot spot area. The Fast worker algorithm does not have this property because it can always wake up a core if there is a sleeping one at any location on the chip. What is particularly interesting is that this damping mechanism of the friend worker and path home algorithms, and its system impact, is maintained by local decisions.

6.2.3 Cores can survive low load periods

One property of the application at hand is that a temporal peak load relative to the average load arrive every 500 time units. The proposed algorithms do not take care of this particular behavior. One consequence is that if the algorithm's idle delay parameter is set greater than 500, then a once activated core will stay awake for the full time of such 500 time units period. This means that the temporal response (queue time) of the next peak will be at least as fast as the last one. Another consequence is that the system is staying prepared to process a higher load than the actual average, which is favorable in the case of suddenly increasing loads but unfavorable in the sense of power efficiency for constant load.

6.2.4 Overload

For the overload task set, it is apparent that all the proposed power mode handling and task processing algorithms perform worse in the PDP sense than the All active regime.

First, one could argue that this task set represents an extreme scenario so it is in some sense reasonable that it gives extreme results. What the result tells us is that the three proposed algorithms are not useful in an overload-like scenario in terms of the PDP quality as stated in this work; it is better to not bother putting cores to sleep in this particular scenario.

Second, the result sheds some light on the chosen PDP metric and the fact that both the average power and queue times are weighted equally.

6.2.5 Knowledge of the application at hand could be implemented in the algorithm.

Because of the damped wake-up in Friend worker and Path home, the average queue (response) time is strongly dependent on the minimum idle cores threshold and the number of sinks. It seems reasonable that this parameter is tuned for the anticipated task set or changed online based on profiling behavior over time. The parameter can also be set based on calendar time if the application can benefit from it.

On the short scale the sleep-mode handling and task scheduling algorithms could be tailored for the specific application. In the case of this work and the studied application it would be suitable to prepare cores for the 500 time unit peaks. The more an algorithm is tailored for one application, the more it will lose in generality and predictability on arbitrary workloads.

6.3 Summary

The PDP is stable for Friend worker and Path home when idle delay varies. It cannot be said whether this behavior is true even for slightly altered task sets, like for example if all loads were scaled up by some fraction.

Chapter 7, Lateness analysis

One can wonder how much the average queue time of a simulation says about the system qualities. As it is an arithmetic average measure the distribution of queue time is hidden. For the analysis of a real-time system this is unacceptable. In fact, the notion of task deadline was not taken into account at all. With the addition of the notion of deadlines to tasks, it is possible to use the lateness measure as described in Chapter 5,.

The purpose of this experiment is to investigate how the sleep mode handling and task processing algorithms impact the average lateness on the given task sets with the addition of task deadlines.

A similar experiment setup as before is used but with the addition of per-task deadlines. Deadlines are chosen so that they consist of the tasks execution time plus a random number from the interval 40 to 240. This means that task lengths are c = random(10,80) with deadlines d = c + random(40,240). The system parameters and sleep mode algorithms are preserved.

Examined multi-core configurations consist of two different settings with 256 and 4096 cores. Fast worker and Friend worker algorithms are evaluated using three different values for the min idle parameter.

7.1 Results

For each algorithm and hardware configuration, two graphs are presented. The first one shows average power consumption and the second shows average lateness.

7.1.1 Ramp and peak task sets

In Figure 7-1, power consumptions range from 0.6 to 1.3 on the scale from 0.0 to 2.0, with All active at 1.3. This means that the average power consumption spread covers roughly 40% of the energy scale. The same holds for Figure 7-3, Figure 7-5 and Figure 7-7.

Ramp

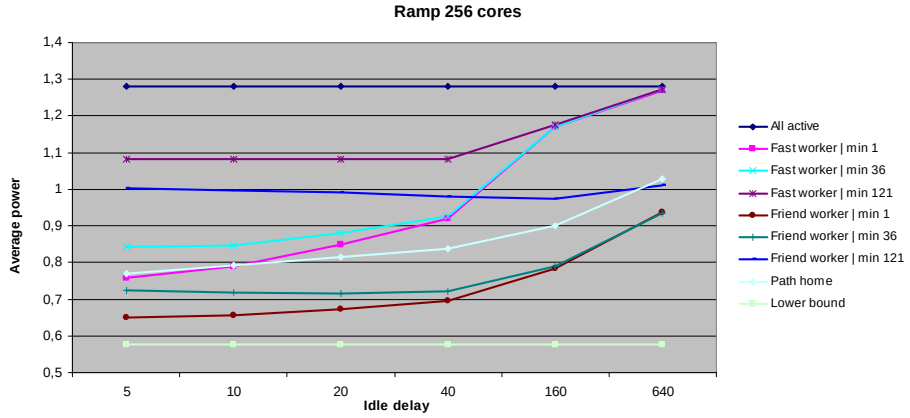

Figure 7-1 Average power consumption for task set Ramp, evaluated on 256 cores.

Figure 7-1, shows average power consumption for each algorithm and configuration tested. "Friend worker | min 1" stands out as having the lowest power consumption.

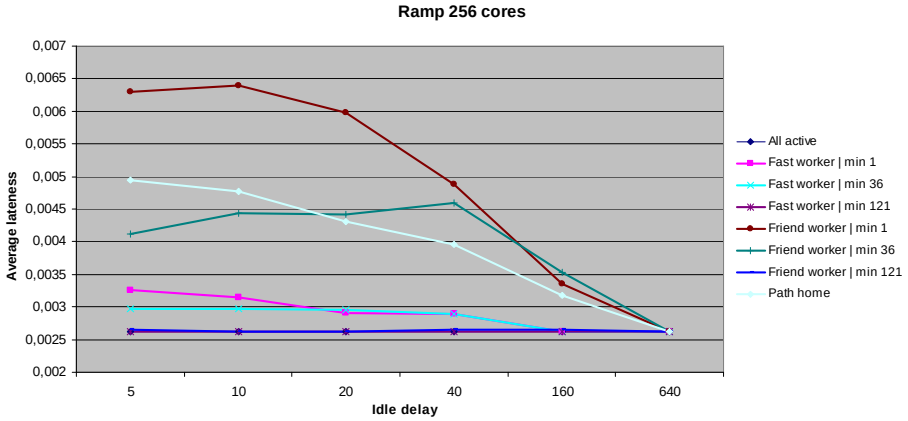

Figure 7-2, Average lateness for task set Ramp, evaluated on 256 cores.

The price for low power consumption is apparent in Figure 7-2 where it can be seen that "Friend worker | min 1" has an average lateness which is 110% higher than that of "Fast worker | min 36", when the idle delay is set to 5.

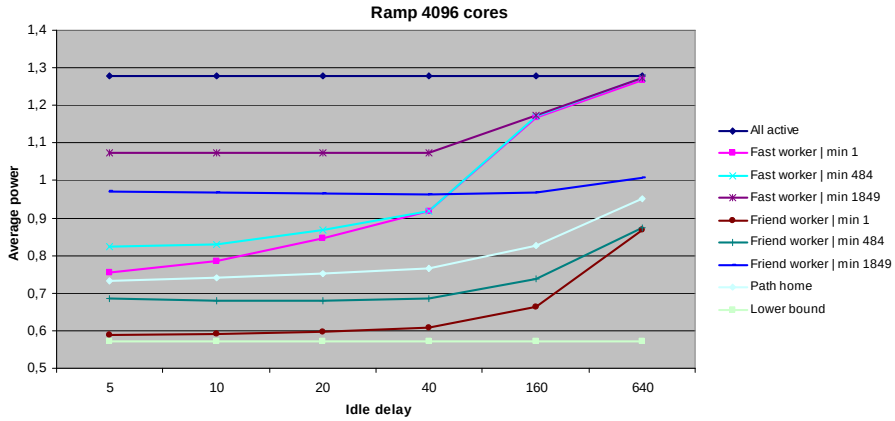

Figure 7-3, Average power consumption for task set Ramp, evaluated on 4096 cores.

Figure 7-3 and Figure 7-4 present power and lateness correlations to the idle delay in a system with 4096 cores. The power performance results are similar to those for the system with 256 cores as shown in Figure 7-1.

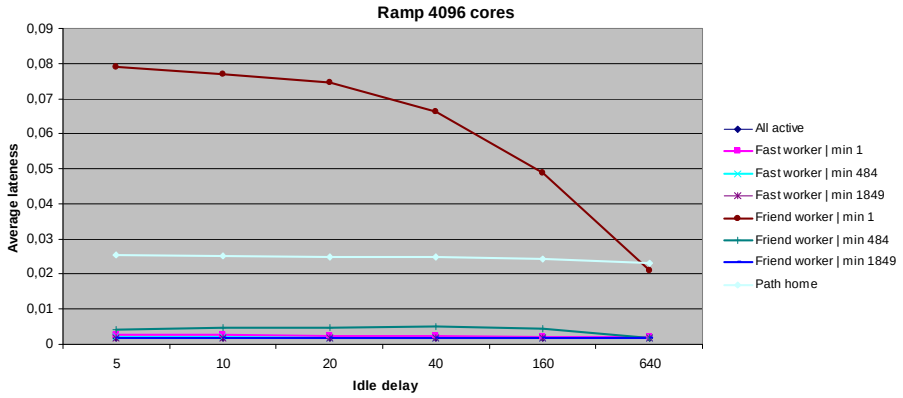

Figure 7-4, Average lateness for task set Ramp, evaluated on 4096 cores.

In Figure 7-4 we see how "Friend worker | min 1" stands out from the rest. With an idle delay between 5 and 20 it has an average lateness being at least 300% higher than the closest competitor, Path home.

We see that the lateness of "Friend worker | min 484" does not differ considerably from that of "All active", although the power saved by the former algorithm is about 80% of what is theoretically possible.

Peak

Figure 7-5 to Figure 7-8 show the results for the peak task set.

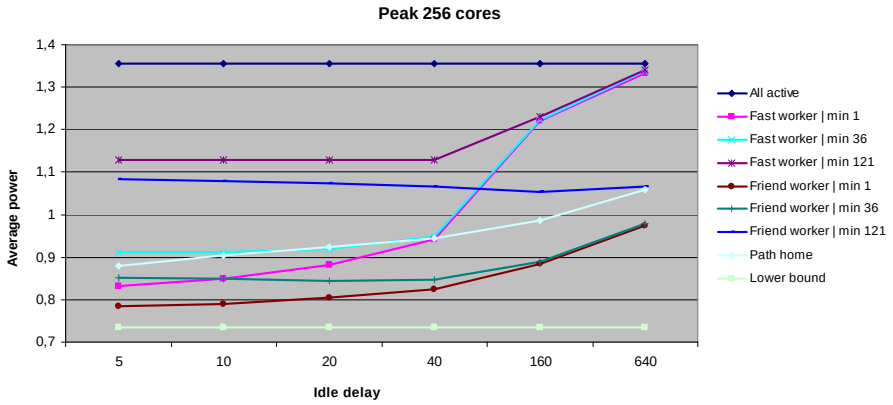

Figure 7-5, Average power consumption for task set Peak, evaluated on 256 cores.

"Friend worker | min 1" with the idle delay set to 5 has again the lowest average power consumption of all algorithms. It is also the one with the highest average lateness as seen in Figure 7-6. Note the difference between the lateness of the algorithms for the Peak task load in Figure 7-6 compared to that of the Ramp task load in Figure 7-2.

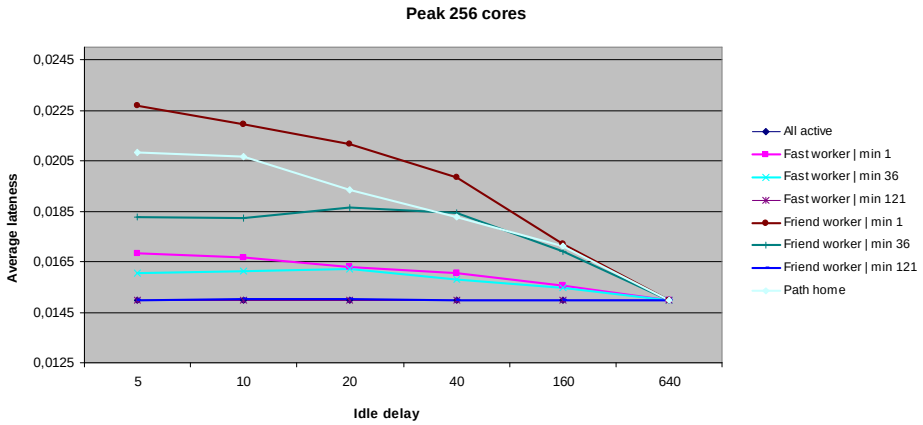

Figure 7-6, Average lateness for task set Peak, evaluated on 256 cores.

With 4096 cores the "Friend worker | min 1" algorithm comes close to the theoretically lower bound at 0.73. The algorithms perform similar in relation to each other when compared to the results for the 256-core configuration.

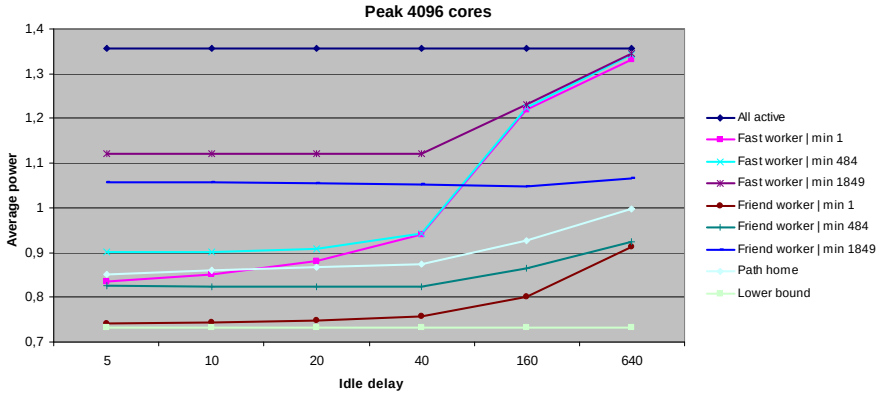

Figure 7-7, Average power consumption for task set Peak, evaluated on 4096 cores.

Looking at the lateness results in Figure 7-8 however, the "Friend worker | min 1" stands out. It has an average lateness being 1700% higher than the closest algorithm, Path home.

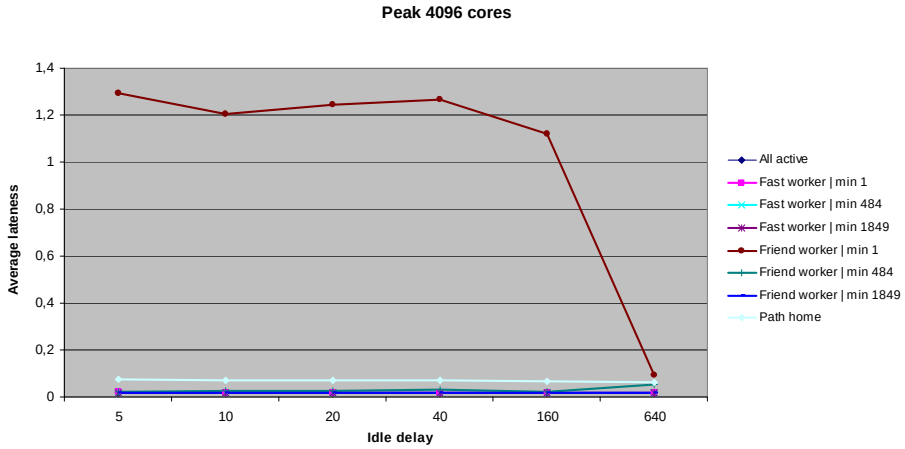

Figure 7-8, Average lateness for task set Peak, evaluated on 4096 cores.

7.1.2 Overload task set

One general observation for the system responses to the overload task set is that the spread of power consumption is less than 10% of the scale from 0.0 to 2.0. Also two groups of lines can be recognized in the average power plots: Path home and Friend worker with low values on the minimum idle cores parameter consume significantly less power than the rest. For example the Fast worker algorithms always consume more power than Friend worker when given the same parameters.

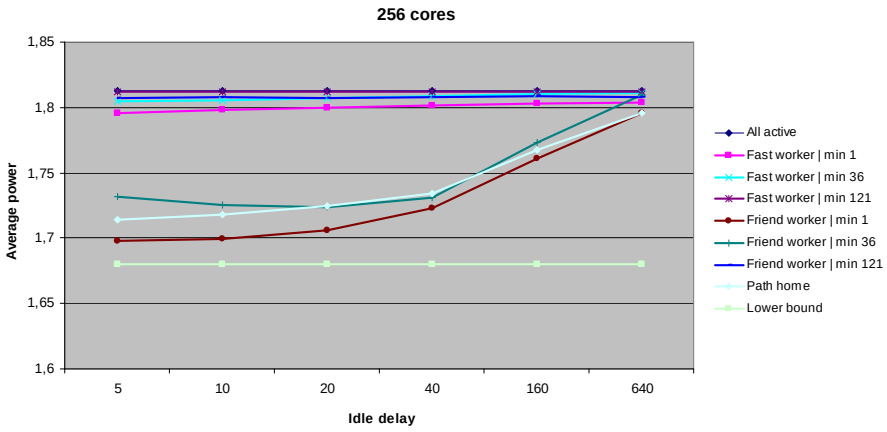

Figure 7-9, Average power consumption for task set Overload, evaluated on 256 cores.

For the 256-core system the algorithms make two groups as seen in Figure 7-9. One group where it seems like the idle delay parameter makes no impact on the power consumption and the other group where it does. The algorithms affected by the idle delay setting are "Friend worker | min 1", "Friend worker | min 36" and "Path home". Note that the bound of possible power consumption figures are not that wide as the average power consumption only can range between 1.68 to 1.81 due to the high load of the system.

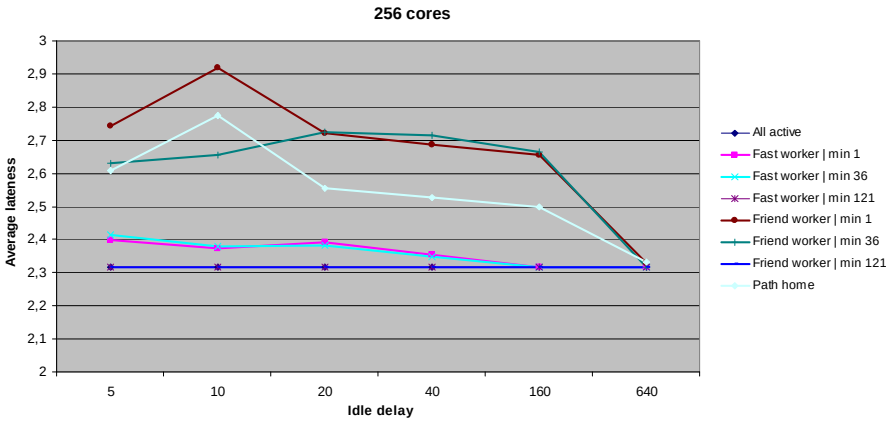

Figure 7-10 Average lateness for task set Overload, evaluated on 256 cores.

In the lateness graph in Figure 7-10 we see the same two groups of configurations as in Figure 7-9. Here the group which had the lowest power consumption in Figure 7-9 has the highest lateness results. Interestingly enough all of the algorithms in the high lateness group gets an increase in lateness as the min idle

core parameter is increased. Intuitively a higher amount of idle cores would give the system an increased response time and lower lateness values.

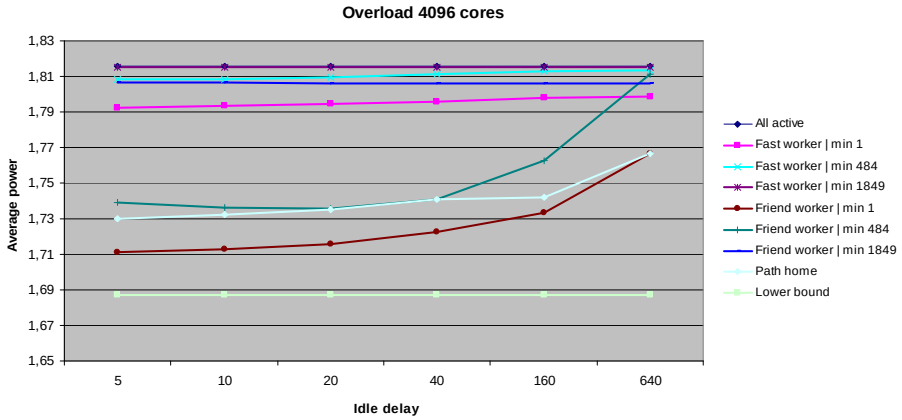

Figure 7-11, Average power for task set Overload, evaluated on 4096 cores.

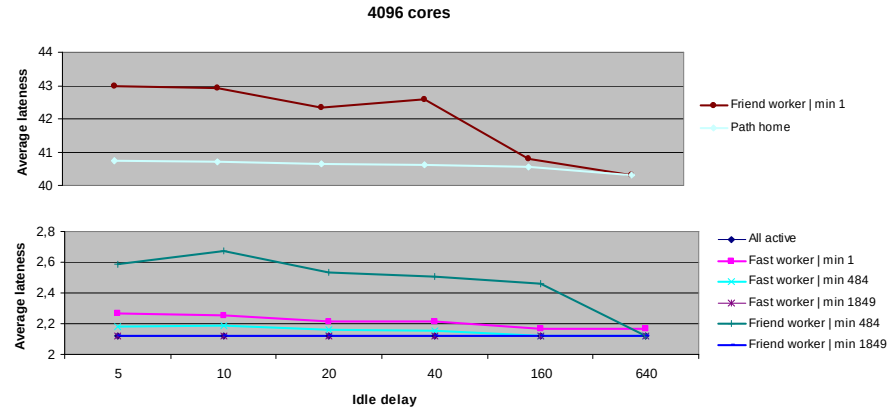

Figure 7-12, Average lateness for task set Overload, evaluated on 4096 cores.

7.2 Discussion

The general trend for all simulated scenarios is that the energy consumption increases and the lateness decreases when the idle delay parameter is increased. But this behavior is in fact no necessity as is also evident from the simulations: see for example “Fast worker | min 36” in Figure 7-5 and Figure 7-6. Modifying the idle delay parameter will impact which cores are available for execution and thus where on the chip new tasks will end up. Different execution locations for a certain task may result in different core activation patterns.

The lateness tends to converge for all load-balancing algorithms as idle delay reaches 640. The value converged to be not necessarily the lowest possible lateness.

7.2.1 Ramp and peak task sets

Anomalous behavior is seen in Figure 7-8 regarding lateness for the Friend worker algorithm with a minimum idle cores parameter set to one on systems with 4096 cores. The anomaly is the extremely high average lateness for this configuration. To investigate this case in detail, Figure 7-13 and Figure 7-14 present task loads overlaid with information on where lateness actually occurs for two configurations. On the *Time* axis is the actual simulation time. *Task load* is the cumulative execution time of arriving tasks at a certain simulation time and *Lateness* is the total lateness for tasks arriving at this same time.

The extra lateness at the beginning of the simulation is explained by the fact that the initial workload cannot be handled with the responsiveness delivered by a sole core. More cores are waked up by the power-mode handler and after some wake-up time the task load can be processed. In the meantime, task deadlines may have timed out. At time 500, the system has stabilized at system load of 10% and deadlines are no longer missed.

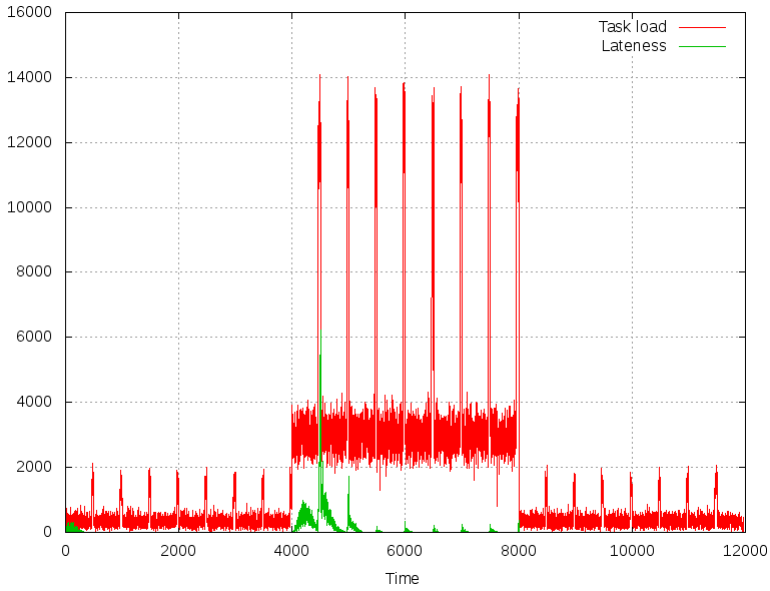

Figure 7-13, Workload and lateness for Friend worker | min 1 with idle delay 5 on 4096 cores. The task set is Peak.

A similar reasoning goes for the load change at time 4000: the system with the goal of having one core idle is not prepared for the increased task load whereas the

system with 484 idle cores apparently can handle this without missing too many deadlines. It is also interesting to note that the system in Figure 7-13 stabilizes with regard to lateness and has a lateness pattern similar to Figure 7-14 after 6000 time units. The later configuration doesn't have the transient behavior.

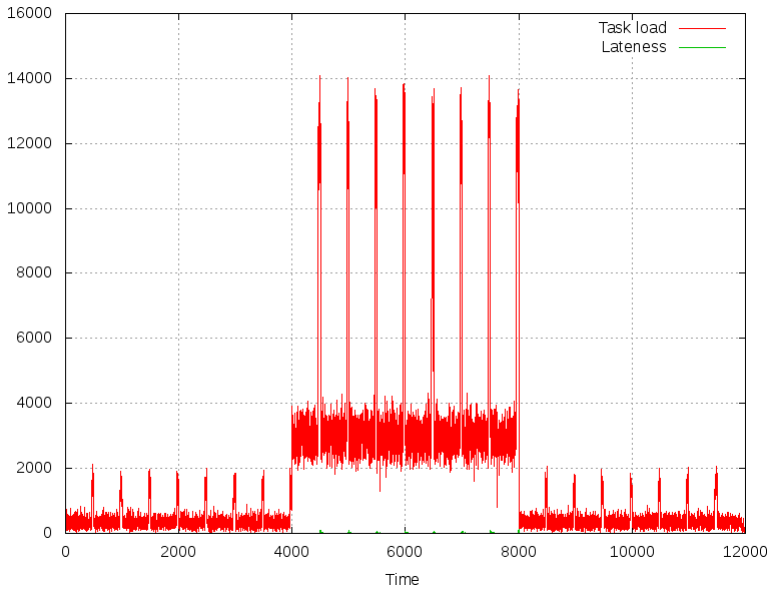

Figure 7-14, Workload and lateness for Friend worker | min 484 with idle delay 5 on 4096 cores. The task set is Peak.

Another perspective that can be taken on the same scenarios is to investigate the power consumption with regard to core state for the full simulation run. This is shown in Figure 7-15 and Figure 7-16. In Figure 7-15 the minimum number of idle cores is 1 out of 4096 (< 1%) and in Figure 7-16 it is 484 (12%). All other parameters are held the same. The difference in lateness occurs at the beginning and after the task load steps up from 10% system load to 90% system load at time 4000.

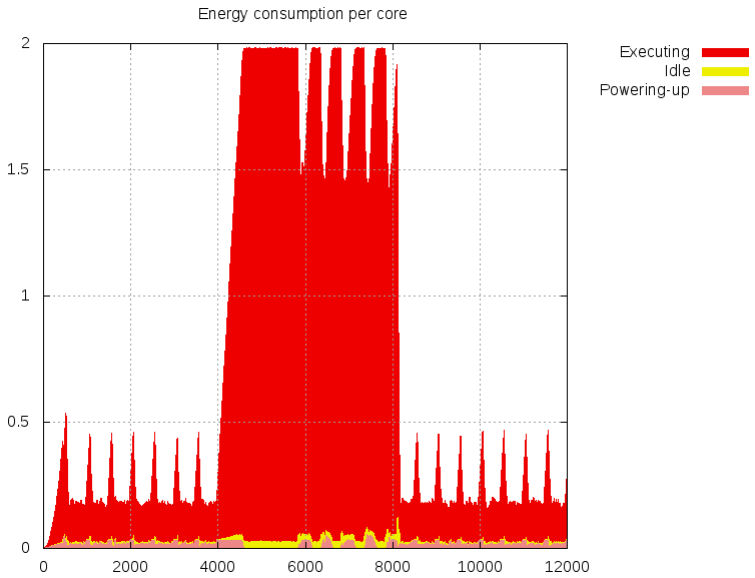

Figure 7-15, Energy consumption per core for different core states. The configuration is Friend worker | min 1 with idle delay 5 on 4096 cores. The horizontal axis is simulation time.

In Figure 7-15 we clearly see how the slower adaption to the increased workload affects the power consumption profile. Especially the slope of the area (red) corresponding to energy consumed by executing cores at time 0 and time 4000 differs. Another key point to note is how the yellow areas, energy consumed by idling cores, differ in Figure 7-16. The height of these stripes roughly corresponds to the minimum number of idling cores algorithm parameter, and so corresponds to the momentary responsiveness to new tasks.

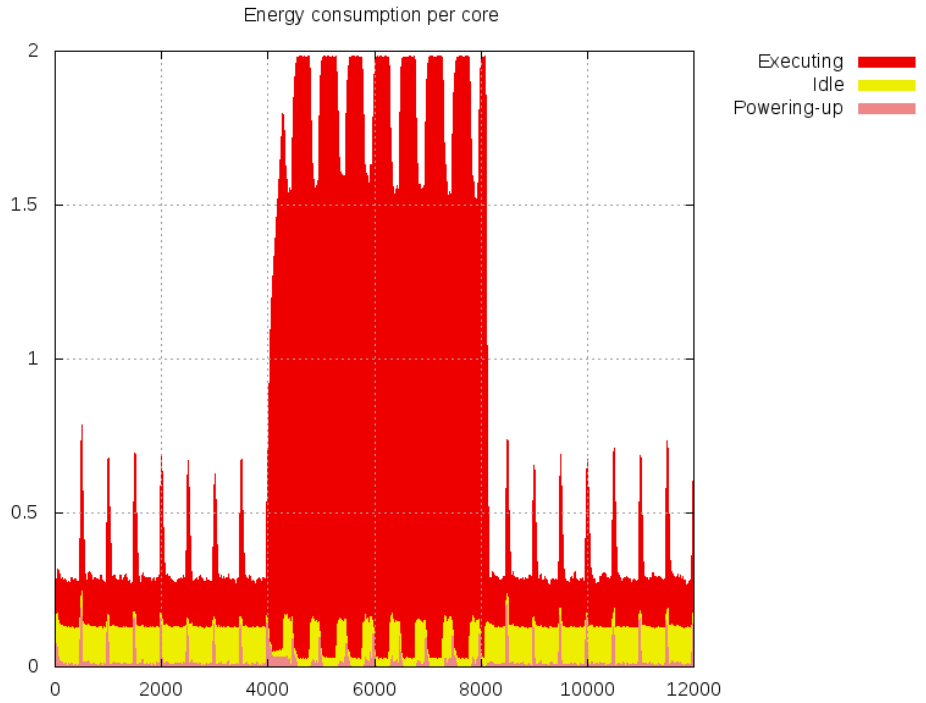

Figure 7-16, Energy consumption per core for different core states. The configuration is Friend worker | min 484 with idle delay 5 on 4096 cores.

The observed anomaly can be connected to the actual algorithm as described in Chapter 4,. Friend worker is characterized by waking up cores on the edge of an active spot, and because of this the wake up rate is bounded by the perimeter of that spot area. Figure 7-17 shows this behavior on a chip with 1024 cores. Notice the broad area on the middle of the chip where no cores can be activated as no neighboring cores are active. The Fast worker algorithm has different behavior: as long as there are sleeping cores and there is movement in the task queue, new cores will be woken up at random locations on the chip when asked for, even if the net workload does not in fact increase.

One solution to this problem is to enlarge the neighborhood of each core in the Friend worker algorithm. Then the parameter *n* on line 13 in the description of the Friend worker algorithm in Figure 4-4 could be modified to increase the number of cores that may be activated.

Another extension for solving the problem is to implement sporadic core activation to create new spots of active cores and thus increase the number of potential neighbors to wake up resulting in faster workload adaptation.

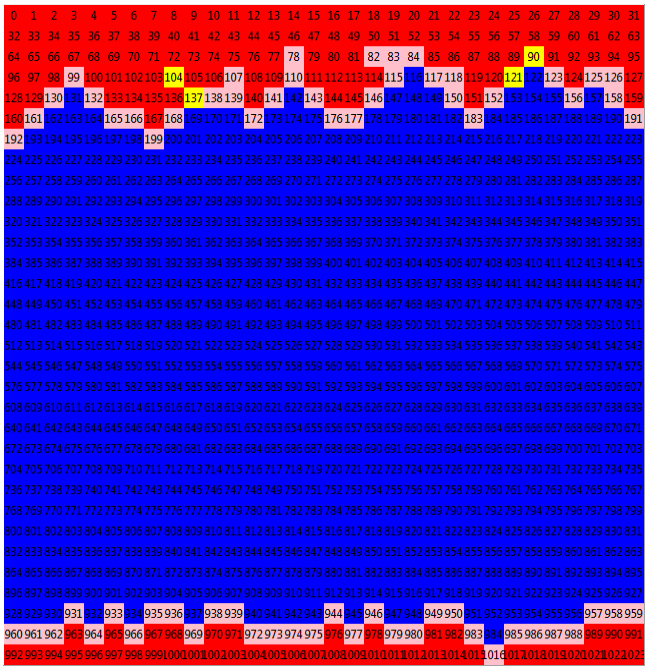

Figure 7-17, Core state geography of Friend worker | min 1, with idle delay 5, visualized on 1024 cores.

7.2.2 Overload task set

In the case of 4096 cores, two algorithm configurations definitely shoot out in lateness. That is the Friend worker with a minimum of one idling core and Path home. The situation doesn't seem to be addressed by increasing the minimum idle delay as can be seen in Figure 7-12. It is therefore reasonable to suspect that the high level of lateness is contributed at the stages where the task load increases step-wise. For an example, see Figure 7-18 and Figure 7-19 which show the lateness related to task load for two simulations with the Friend worker having different minimum idle cores.

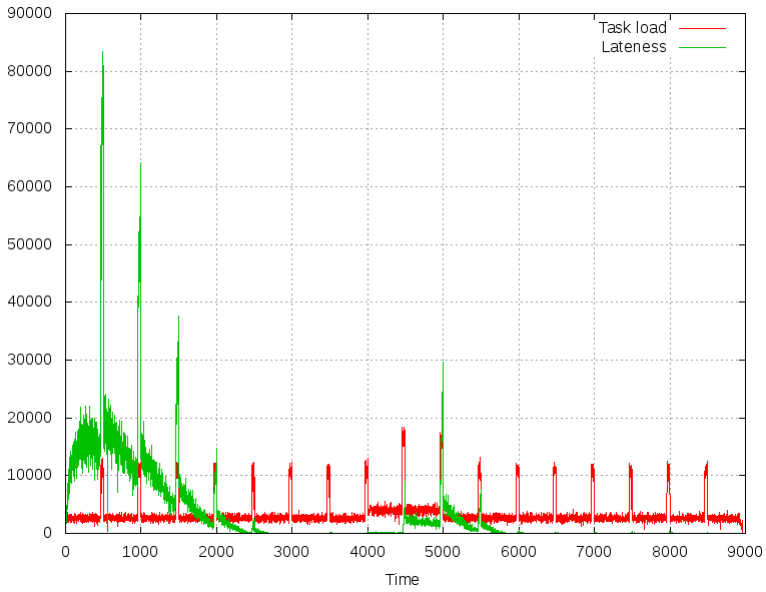

Figure 7-18, Workload and lateness for Friend worker | min 1 with idle delay 5 on 4096 cores. The task set is Overload.

This phenomenon is much like what was previously described for the Peak task set. That is, the wake-up rate is limited by the perimeter of the active chip areas.

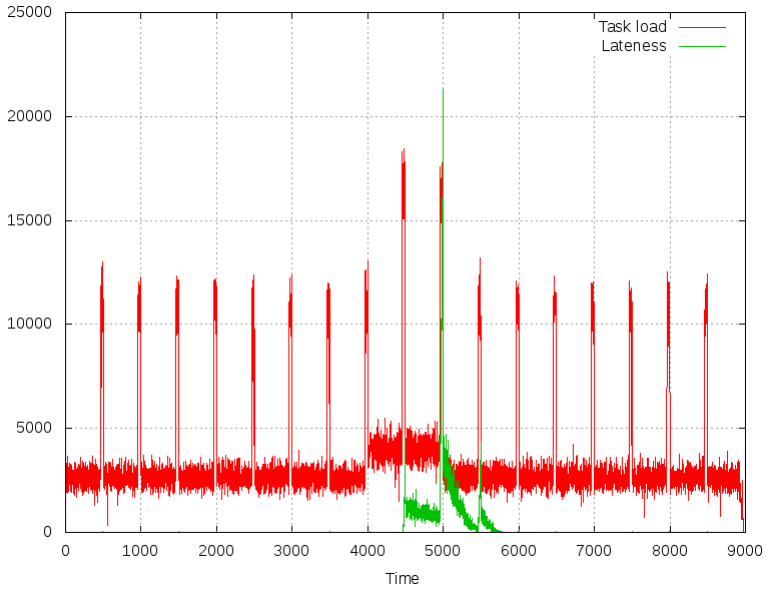

Figure 7-19, Workload and lateness for Friend worker | min 484 with idle delay 5 on 4096 cores. The task set is Overload.

Chapter 8, EDF fetching of tasks

As stated in the model description it is assumed that the task queue is centralized and that tasks are handled in FIFO order. A common policy in both single- and multi-processor systems is to dispatch the most urgent task: the task with the earliest deadline. This is called earliest deadline first (EDF).

The purpose of this experiment is to investigate what the EDF policy can do to the studied job processing system in terms of power consumption and lateness.

8.1 Results

Having the task queue sorted using EDF results in that almost no lateness is present for the ramp and peak task sets and therefore we have chosen not to study them further.

The system setup is identical to the previous experiment but only the Overload task sets for 256 and 4096 cores are presented as seen in Figure 8-1 to Figure 8-4. The power consumption in the EDF case, as seen in Figure 8-1 and Figure 8-3, does not change considerably compared to the non-EDF policy studied in the previous chapter. This indicates that the wake-up behavior of cores is not affected by EDF.

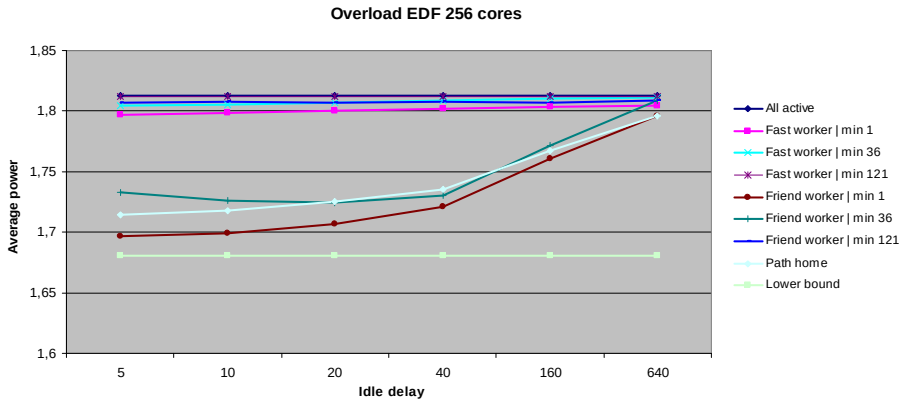

Figure 8-1, Average power consumption for task set Overload, evaluated on 256 cores.

Real-time behavior on the other hand is greatly affected. For the 256 core system in Figure 8-2, lateness has been reduced by factor 50.

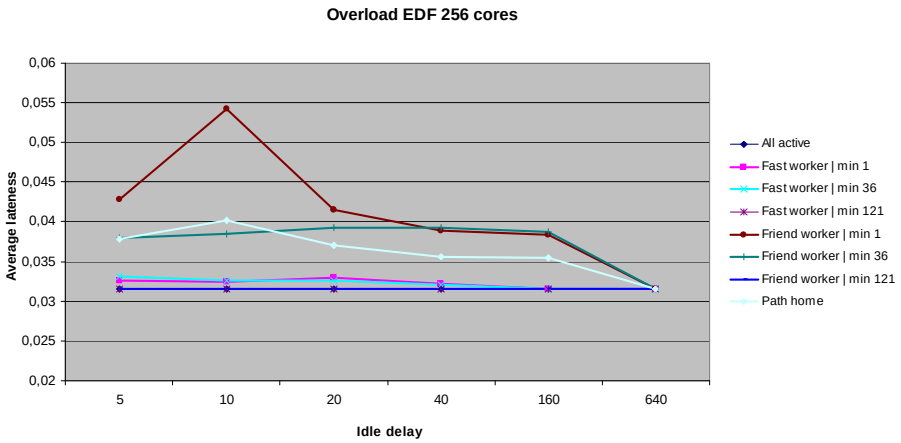

Figure 8-2, Average lateness for task set Overload, evaluated on 256 cores.

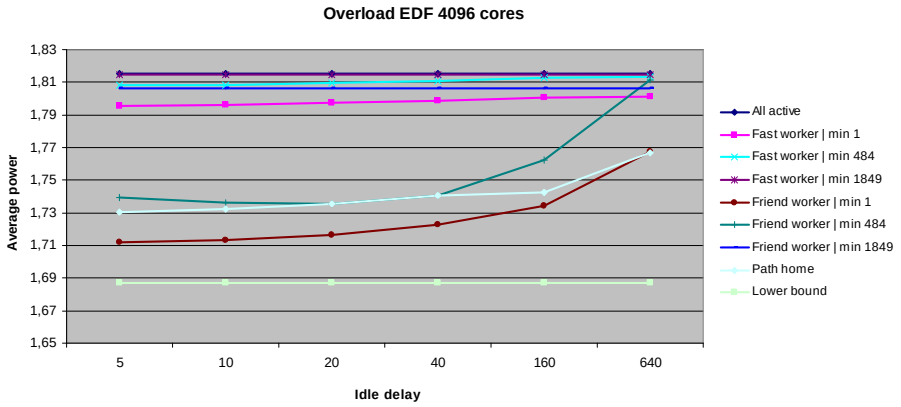

Figure 8-3, Average power consumption for task set Overload, evaluated on 4096 cores.

In Figure 8-4 we see that with EDF, algorithms “Friend worker | min 1” and “Path home” are improved by 15% when compared to the non-EDF case. The other algorithms reach an improvement of factor 10.

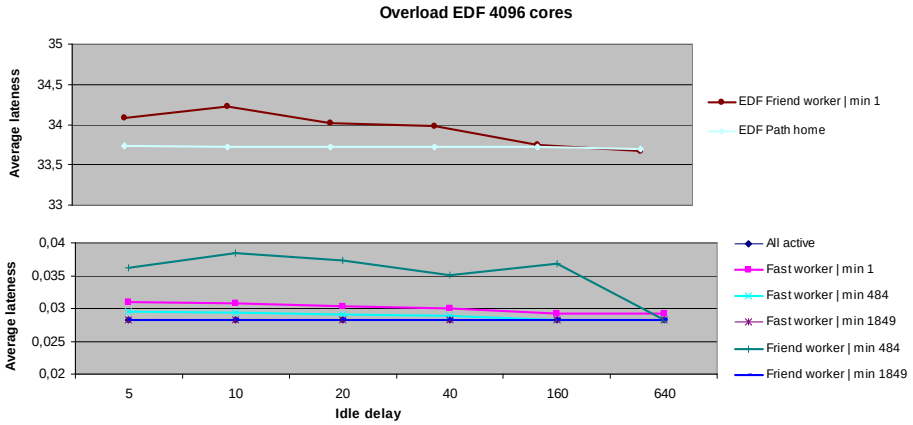

Figure 8-4, Average lateness for task set Overload, evaluated on 4096 cores.

8.2 Discussion

For a system with a centralized task queue, EDF does not incur greater demands than that of inserting a task in order in an already sorted list of tasks. The system studied in this work benefit of this policy, as was shown. However, as the focus in this work is on distributing the job processing decisions, and ultimately also the task queue, it must be noted that the notion of a centralized EDF queue does not always apply. For example, the task queue could be clustered so that a fixed set of cores always fetch tasks from a predefined queue. Or it could be that any core in the system could supply any other core with tasks, and thus removing the concept of task queues altogether.

Chapter 9, Conclusion

This work presents a distributed sleep mode handling and task processing method. In order to evaluate the method, a simulator tool was developed and used. With the simulator, it is shown that the proposed method can be used to handle power and tasks in a massive multi-core computer. The model is thought to be general enough to be adapted to suit an actual hardware platform. The simulator tool is also useful for understanding how the system behaves as the number of cores increases far above 1000.

A fully distributed sleep mode handling policy, *Path home*, which only depends on local information has been constructed which still performs comparable to policies that are not fully distributed. This shows that the strategy of mimicking a cellular automaton is successful in this problem domain.

The simulation results show that sudden changes in task load can incur transients in lateness. Algorithm parameters have to be tuned to account for this. It is also shown that there exist configurations that give only modest increase in lateness but at the same time decrease power consumption up to 80% of what is theoretically possible.

Sleep mode handling and task processing algorithms which performs well on 256 cores might in fact produce very poor power-per-core results when executed on a higher number of cores. With simulator produced data we are able to investigate, explain and give solutions to this anomalous behavior.

Chapter 10, Future work

A sleep mode handling a task processing algorithm could be developed that takes advantage of the qualities of the tasks as they are fetched. For example, a core may fetch more than one task at once to run in sequence, and before execution starts, decide to wake up an appropriate number of neighbors depending on perceived urgency of the fetched tasks. This gives an additional level of feedback.

Below are areas of how the simulator and model could be extended.

- Task could be scheduled as belonging to task graphs with dependency constraints.
- Fully distributed task queues and scheduling can be implemented. For example “work stealing” [7] or the ability for each task to spawn new tasks onto neighboring cores e.g. “Cuckoo scheduling” [28].
- The chip geometry can be generalized to include n-dimensional chip networks.
- The assumption that each core could individually be fully switched off may be wrong for some multi-core systems and thus ways to group the cores into “power islands” form another possible addition to the simulator.

For more realistic simulations there is a need to implement additional hardware parameters such as network on chip latencies and power consumption for core-to-core data transfers. Implementing latency penalties for utilizing shared data structures can also give additional information and help in deciding how beneficial it is to create even more distributed algorithms.

Chapter 11, References

[1] Lte uplink receiver phy benchmark. *http://sourceforge.net/projects/lte-benchmark/*.

[2] Adapteva. *Epiphany Architecture Reference (G3)*. Adapteva Inc., 2012.

[3] Susanne Albers. Energy-efficient algorithms. *Commun. ACM*, 53(5):86–96, May 2010.

[4] Jos L. Ayala. *Communication Architectures for Systems-on-Chip*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2011.

[5] H. Aydin, R. Melhem, D. Mossé, and P. Mejá-Alvarez. Power-aware scheduling for periodic real-time tasks. *Computers, IEEE Transactions on*, 53(5):584–600, 2004.

[6] H. Aydin and Dakai Zhu. Reliability-aware energy management for periodic real-time tasks. *Computers, IEEE Transactions on*, 58(10):1382 – 1397, oct. 2009.

[7] R.D. Blumofe and C.E. Leiserson. Scheduling multithreaded computations by work stealing. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 356–368. IEEE, 1994.

[8] Sangyeun Cho and R.G. Melhem. On the interplay of parallelization, program performance, and energy consumption. *Parallel and Distributed Systems, IEEE Transactions on*, 21(3):342 –353, march 2010.

[9] Hakan Aydin Dakai Zhu. *Handbook of Energy-Aware and Green Computing*, volume 2. Chapman and Hall/CRC, jan. 2012.

[10] Daniel R. Dooly, Sally A. Goldman, and Stephen D. Scott. Tcp dynamic acknowledgment delay (extended abstract): theory and practice. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC '98, pages 389–398, New York, NY, USA, 1998. ACM.

[11] Martin Gardner. Mathematical games - the fantastic combinations of john conway's new solitaire game "life". *Scientific American*, 223:120–123, 1970.

[12] Yifeng Guo, Dakai Zhu, and H. Aydin. Reliability-aware power management for parallel real-time applications with precedence constraints. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1 –8, july 2011.

[13] I. Hong, D. Kirovski, Gang Qu, M. Potkonjak, and M.B. Srivastava. Power optimization of variable-voltage core-based systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 18(12):1702 –1714, dec 1999.

[14] Hailin Jiang, M. Marek-Sadowska, and S.R. Nassif. Benefits and costs of power-gating technique. In *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*, pages 559 – 566, oct. 2005.

[15] Anna R. Karlin, Mark S. Manasse, Lyle A. McGeoch, and Susan Owicki. Competitive randomized algorithms for non-uniform problems. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, SODA '90, pages 301–309, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[16] N.S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J.S. Hu, M.J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68 – 75, dec. 2003.

[17] C. M. Krishna. *Real-Time Systems*. John Wiley & Sons, Inc., 2001.

[18] J. Leverich, M. Monchiero, V. Talwar, P. Ranganathan, and C. Kozyrakis. Power management of datacenter workloads using per-core power gating. *Computer Architecture Letters*, 8(2):48 –51, feb. 2009.

[19] E. Musoll. A thermal-friendly load-balancing technique for multi-core processors. In *Quality Electronic Design, 2008. ISQED 2008. 9th International Symposium on*, pages 549 –552, march 2008.

[20] E. Musoll. A process-variation aware technique for tile-based, massive multicore processors. *Computer Architecture Letters*, 8(2):52 –55, feb. 2009.

[21] E Musoll. A cost-effective load-balancing policy for tile-based, massive multi-core packet processors. *ACM Trans. Embed. Comput. Syst.*, 9(3):24:1–24:25, March 2010.

[22] E. Musoll. Hardware-based load balancing for massive multicore architectures implementing power gating. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(3):493 –497, march 2010.

[23] C.A. Nicopoulos and The Pennsylvania State University. *Network-on-Chip Architectures: A Holistic Design Exploration*. Pennsylvania State University, 2007.

[24] Andreas Olofsson. A 1024-core 70 gflop/w floating point manycore microprocessor. In *HPEC 2011*, 2011.

[25] The Climate Group on behalf of the Global e Sustainability Initiative (GeSI). Smart 2020 : Enabling the low carbon economy in the information age. *GeSI's Activity Report June 2008*, 2008.

[26] Michael Palis. Real-time scheduling algorithms for multiprocessor systems. In *Handbook of Parallel Computing*, chapter 24, pages 1–26. Chapman and Hall/CRC, December 2007.

[27] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. *SIGOPS Oper. Syst. Rev.*, 35(5):89–102, October 2001.

[28] M. Prakash, R. Saranya, K.R. Jothi, and A. Vigneshwaran. An optimal job scheduling in grid using cuckoo algorithm.

[29] Xuan Qi and Da-Kai Zhu. Energy efficient block-partitioned multicore processors for parallel applications. *Journal of Computer Science and Technology*, 26:418–433, 2011.

[30] B. P. Singh. *Electronic Devices And Integrated Circuits*. Pearson Education, 2006.

[31] Magnus Själander, Sally A. McKee, Peter Brauer, David Engdal, and Andras Vajda. An lte uplink receiver phy benchmark and subframe-based power management. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software,*, 2012.

[32] R. Sridharan, N. Gupta, and R. Mahapatra. Feedback-controlled reliability-aware power management for real-time embedded systems. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 185 –190, june 2008.

[33] K. Stavrou and P. Trancoso. Thermal-aware scheduling: A solution for future chip multiprocessors thermal problems. In *Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on*, pages 123 –126, 0-0 2006.

[34] Kyriakos Stavrou and Pedro Trancoso. Tsic: Thermal scheduling simulator for chip multiprocessors. In Panayiotis Bozanis and Elias Houstis, editors, *Advances in Informatics*, volume 3746 of *Lecture Notes in Computer Science*, pages 589–599. Springer Berlin / Heidelberg, 2005. 10.1007/11573036_56.

[35] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problemoverview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[36] Stephen Wolfram. Cellular automata as models of complexity. *Nature*, 311:419–424, 1984.