Space-shared and frequency scaling based task scheduler for many-core OS

András Vajda
Group Function Technology, Ericsson
Hirsalantie 11, Jorvas, Finland
andras.vajda@ericsson.com

ABSTRACT
It is predicted that within the next decade chips with several 100s or 1000s of processor cores will be possible to build and will become available [1]. However, efficient usage of many-core chips will have to overcome the limits imposed by Amdahl's law ([3], [4]). In this paper we introduce a novel task scheduling method for chip multi-processors with at least several tens, but scaling up to thousands of cores. It relies on a few principles: *space-shared approach* for scheduling processes on a many-core chip; adapting performance of individual cores and the chip as a whole by *controlled* and *scheduled variation of the frequency* at which different cores execute, in contrast with today's thread migration strategies; *reliance on application-generated requests* for computing resources instead of thread assignment policies in the operating system.

Keywords
Multi-core systems on chip, parallel programming models, operating systems, Amdahl's law

INTRODUCTION
The computing industry's development over the last three decades has been largely defined by a dramatic increase of the available computing power, driven primarily by the self-fulfillment of Moore's law. Recently this development has been heavily impacted by the inability to scale performance further through increased frequency and complexity of processor devices, due primarily to power and cooling issues; as an answer, the industry turned to chip-multiprocessors (CMP) or multi-core devices. It is predicted that the advancement of manufacturing technology – in fact, the continued development according to Moore's law – will allow for chips with 100s or even 1000s of processor cores within the next decade.

Chip multiprocessors pose however new requirements on operating systems and applications. It has been shown by several studies (e.g.[2]) that current symmetric multi-processing, time-shared operating systems will be difficult to scale to several tens, hundreds or thousands of processor cores. On the application side, Amdahl's law ([3], [4]) will put a limit to the amount of performance increase that can be achieved even for highly parallelized applications, assuming static symmetric or asymmetric CMPs.

Better scalability could theoretically be obtained with chips where computing resources can be utilized either as several cores executing in parallel (for the execution of the parallel sections of the applications) or as a single, fast, powerful core for running the single-threaded, sequential portions of the applications ([4]). However, so far only small-scale proposals were put forward relying mostly on ILP.

In this paper, we suggest a different approach, which however is possible to realize with existing HW technology. As we'll detail it in further sections, it relies on dynamically adjusting the frequency at which the cores execute, temporarily boosting the frequency of selected – very few – cores in order to cope with the demands put by single-threaded portions of the applications. As long as sequential code amounts for a small fraction of the applications, this shall be possible without breaking the limits set by the overall power envelope and without overheating the chip. Running cores at various frequencies and adjusting these 'on-the-fly' shall be feasible, see [1].

On top of this mechanism we build a scheduling algorithm relying on *geometry-aware space-sharing of processing cores* and securing that overheating, breaking of the power envelope and big hotspots are avoided. The focus of the paper is OS scheduling under the assumption of available hardware, hence the realization alternatives in hardware are beyond the scope of this paper.

ANALYSIS OF THE FREQUENCY SCALING APPROACH
In our model we start from the assumptions that (a) each application has a certain amount of cores exclusively allocated to it and (b) when the sequential portion of the application is executing, only one core will actively execute application code.

Based on these assumptions, we conclude that when the sequential portion of the application is executing, all but one of the cores allocated to the application can be switched completely off and the available power used for temporarily boosting the performance of the single core that will execute the sequential code.

In [4] a model for applying Amdahl's law to dynamic multi-core chips was introduced. Based on that model, if we assume that for the duration of the execution of the sequential code *all other cores* would be switched off and once core's performance would be enhanced, we would actually get a much better result, close to linear speed-up, even for applications with a sizable sequential portion.

Hardware considerations

In [1], a HW design with three levels of execution frequency for each core – 0, 0,5x, 1x – was proposed. That approach allowed for the usage of only two voltages (nominal and 0,7x). However, in order to achieve a better scaling of applications with sequential code, more fine-grained granularity is needed. Even for 1024 and 2048 core systems however we believe five frequency levels (0, 1x, 2x, 4x, 8x) will suffice, with an acceptable impact on performance. *All* cores shall support at least levels 0 and 1x.

In practice, this means that at least *some* cores shall be able to execute – for a limited amount of time – at frequencies up to 8 times higher than the usual frequency. We will call these cores *scalable cores* while for the other cores we'll use the term *non-scalable cores*. Having just 4 levels of frequency, issues related to input voltages, power delivery and synchronization shall be possible to manage with a reasonable amount of extra logic and without increasing latency or impacting meta-stability. However, this requires further architecture studies, beyond the scope of this paper. It's important to note, that the nominal frequency shall not be the maximum frequency that the core is capable of running at for a sustained period of time; indeed, assuming workloads that are primarily parallel and thus can take advantage of high number of cores, it's more economical to have chips with cores running at low frequencies (e.g. few hundreds of MHz) as their nominal 1x frequency and instead have support for radical scale-up when necessary. For practical reasons the scalable cores are assumed to be fixed during chip design.

These scaling values are possible to obtain for just *one* core at a time for a given chip with the given number of cores (e.g. on a 64 core chip, only one core can run at 4x speed while *all* other cores are shut off). Thus, it's important to analyze *how many* cores can run on a given chip at 2x, 4x, 8x speeds simultaneously, without breaking the power envelope, based on the assumption that power is function of square of frequency, an approximation of real world values between power of 2 and 3, 2 being the worse case assumption. The results are shown below; each value indicates the maximum number of cores that can run at a specific frequency level, on a chip with a given number of cores.

Number of cores	**8**	**16**	**32**	**64**	**256**	**1024**	**2048**
Speed-up factor							
2	1	2	4	8	32	128	256
4	0	0	1	1	4	16	32
8	0	0	0	0	1	2	4

SPACE-SHARED SCHEDULING

Concepts

The concept of space-shared OS has been suggested on several occasions, most notably in [2], [5], [12]. The basic idea is that the OS executes as services on some of the cores, while all the other cores execute just a microkernel without threading support. Cores are allocated to applications upon request.

We build on this model when introducing our scheduling algorithm proposal. We start from the current approach for single-core chips where the computational resource managed by the OS is the *time of the single processor core*, but we replace this with another quantity, the *total power the many-core chip may use to execute applications*. This global resource is allocated to applications through two resources: *cores* and *frequencies* at which these cores execute. Each application will be allocated a number of processor cores, but the total computing power of these - the frequency at which these will execute – will be adjusted based on application needs – expressed as resource requests – and competing resource requests, within the limits set by the amount of scalable cores allocated to an application, the maximum frequency scaling factor that can be used for those cores and the total power budget of the chip.

Each application may request two types of computing resources from the OS, much in the same way as allocating memory. The first type of resource is *processor core resource*, which may be of type *non-scalable (fixed frequency)* or *scalable (scalable frequency)*. The allocation of this type of resource can occur at any time (at application start-up or during execution), based on application requests. The second type of resource is *execution speed*. There are two scenarios for an application to request such resources: (1) temporary *frequency boost/restore to normal* for the scalable cores allocated to it and (2) s*hut off/restore to normal* for a core allocated to it.

From application perspective, the algorithm assumes a behavior similar to handling memory today: (a) the application shall specify its minimal initial computing power requirements at startup; (b) the application shall request shutting off cores when cores are not needed any further; (c) the application shall request frequency

boosting as a resource allocation request; (d) the application is responsible for requesting restoring to normal operating frequency of any core that was previously shut down; (e) the application is responsible for requesting allocation/release of processor cores of various types.

These requirements are in-line with the approach that processing resources are managed similarly to other OS-provided resources such as memory. It does require a shift from today's approach where the application just provides a set of threads that the OS will deploy on available resources; in our approach, the application itself is responsible to decide on which processor core a certain computation is executed.

Space-shared, frequency scaling scheduling algorithm

The scheduling solution relies on the following principles:

- Processor cores are *space-shared* rather than *time-shared* - each core is allocated to only one application
- Allocation of processor cores is *geometry aware,* in the sense that the system will group resources allocated to the same application close to each other
- The OS will only shut off or free up allocated processor cores at the *explicit request of the application* or when the application exits
- The *OS may adjust the speed of individual scalable cores* if needed to accommodate new requests, but always based on requests from applications, similarly to memory allocation requests
- All *non-allocated cores are kept in shut-off state*
- Processing resource requests will have a *wait time* attached; a request with a zero wait time will return immediately – either with allocated resources or a negative response; a request with a positive wait-time will either return immediately with successfully allocated resources or will be deferred until it can be fulfilled or the timer expires.

The algorithm works as follows:

1. At start cores not executing OS code are shut off

2. When a new application is started, it gets allocated, if possible, the amount of processor cores it indicates. Initially, all cores are started at normal, 1x speed. Allocation of processor cores is possible if power requirements can be met (see below)

3. When an application requests the allocation of more processor cores, the request will be fulfilled if power requirements can be met (see below)

4. The OS actively re-calculates power budgets every time a application requests shut off or restore of individual cores and will process pending requests if possible; restoring will only succeed if power requirements can be met (see below)

5. When an application requests frequency boosting or restoring for a core, the OS will fulfill the request to the extent that the result does not go over the overall power budget; if the application has a priority that may trigger re-scheduling of power budgets, such re-scheduling will be performed. If the request was to reduce frequency, the OS will process pending resource requests.

6. The OS shall keep track of the *period* for which a scalable core was running at boosted frequency; in order to avoid over-heating of the chip, the frequency shall be scaled back after the pre-set maximum amount of time the core is allowed to run at such high frequency; the scale-back will also trigger re-scheduling and servicing of outstanding requests. A possible enhancement of this technique is to monitor the amount of time a core needs to 'cool down' after running at high frequency; for this period of time, the core shall not support frequency boosting.

Power requirements are said to be met if (a) the resulting power budget does not exceeds the total budget, or (b) the application has a priority level that triggers re-scheduling and the re-scheduling results in having enough power budget freed up.

Re-scheduling of resource allocations is a key component of our proposal. It will occur every time when a request cannot be fulfilled without over-stepping the power budget *and* the requesting application has a higher priority than at least some of the other executing applications. During re-scheduling the OS will basically modify the clocking of frequency-boosted cores. As we stated before, an already allocated core is *never* taken away forcefully by the OS, neither is it shut down. This is a pre-requisite to allow applications to perform correctly. However, modifying the frequency at which single-threaded portions of lower-prioritized applications execute is a sensible way of prioritizing applications, without starving any. Hence, the OS will calculate how much it shall reduce the frequency scaling factor of some of the scalable cores executing lower-prioritized applications in order to fit in the power budget the requirements of higher prioritized applications. The applications will be notified about the change and the application with higher priority will get its resources allocated. There may be several different policies used by the scheduler such as spreading the scale-back as evenly as possible to several applications, reducing the impact on each individual application, but impacting several applications; or focusing on just enough

applications (with the lowest priorities) to free up sufficient power budget to fulfill the request.

Requests with non-zero wait time that cannot be fulfilled will be queued and will be serviced based on the priority of the application making the request. If a request cannot be fulfilled before its timer expires, the request will be rejected and the application notified.

PERFORMANCE ANALYSIS

The scheduling algorithm will handle two types of resources: non-scalable cores and scalable-cores. Both of these resources scale – quantity-wise – linearly with the total number of cores; in fact, the maximum number of scalable cores is *n/8*, where *n* is the total number of processor cores in the chip.

From an algorithmic point of view, the scheduling algorithm will perform two tasks: keeping track of allocated cores (which core to which application) and performing re-scheduling calculations. The first task is complexity-wise quite similar to memory management and there are algorithms with a constant complexity, not dependent upon the actual number of resources or users (number of cores and amount of applications). The task of re-scheduling depends on the number of scalable cores. In fact, every time a re-scheduling is needed, the scheduler will perform the following steps:

- *Step 1*: Identify the number of scalable cores that are candidate for scaling back: these are cores allocated to applications with priority lower than the application whose request that triggered the re-scheduling. As this information may be stored for each active scalable core (limited to *n/8*), it can be obtained by traversing a list with *n/8* elements, yielding a complexity of *O(n/8)*.

- *Step 2*: Out of the candidate scalable cores, the algorithm shall select which shall have their frequency reduced and by how much. The gain with each frequency-step can be pre-calculated, hence the total number of frequency changes can be easily calculated independently of the total number of cores, but dependent on the chosen policy; in any case it will be limited to the number of scalable cores, hence *O(n/8)*.

In conclusion, the complexity of the re-scheduling algorithm will be *O(n/8)* in best case, with a worst case complexity of *O(n/4)*, hence scaling linearly with the potential number of scalable cores and involves.

Concerning memory, the scheduler has to store only four pieces of information for each scalable core – to which application it is allocated, current frequency scaling factor, dead-line until it can execute at this frequency and application priority (for performance reasons) – the amount of memory needed for even a 1024 core system will be just a few kilobytes. For each non-scalable cores information such as current state (on/off) and application information (identity) needs to be stored, adding a few extra kilobytes.

An issue that needs to be considered is the access latency and speed to memory and buses. There are a number of potential ways to address these issues, namely, using intelligent placing of data in memory, design of big enough on-chip local memory around scalable cores, pre-fetching of data just before entering the single-threaded portion (for both data and code), sharing of on-chip memories between scalable and non-scalable cores, with direct access from both cores (memory local to one scalable and a few non-scalable cores) etc. These issues need to be addressed; however we consider these beyond the scope of this paper.

Empirical evaluation

Empirical evaluation was done using a cycle-by-cycle simulator. The simulator was executing workloads defined in a table on an abstract HW defined in terms of total number of cores, number of scalable cores and number of non-scalable cores. The workloads were defined in terms of *stages*. Each stage is either *sequential* requiring one scalable core for a certain amount of instructions, or *parallel*, requiring a configured number of non-scalable cores, each executing a configured number of instructions. Each simulation was run up to a pre-configured number of cycles, sufficient to execute all the workloads. For simplicity, memory access issues were not considered and it was assumed that a core running at say, 4x speed can execute 4 instructions for each 'regular' cycle.

For the simulation we used 64 cores and seven scenarios, each with 1-4 workloads. Each workload contained two or three stages, usually one or two short sequential and one parallel stage. The degrees of parallelism chosen were f = 0.63, 0.98, 0.99 and 0.999.

The seven scenarios were: (I) one workload, $f = 0.999$, 62 non-scalable cores; (II) two workloads, both at the same priority level, both with $f = 0.999$, both using 30 non-scalable cores; (III) two workloads with $f = 0.999$, both using 40 non-scalable cores, but running at different priorities; (IV) four different workloads, each using 15 non-scalable cores, at the same priority; (V) same workloads as in III, but each using 30 non-scalable cores at different priority levels ($f=0.999$); (VI) same as scenario V, in reverse priority order; (VII) one workload ($f=0.999$), 15 non-scalable cores.

In the results table the rows represent workloads; the columns contain the measured speedup (2); speedup achievable according to [4] (3); total amount of cycles needed on a single core (4); total amount of cycles needed to execute the scenario (5); power used to execute the scenario (6).

A few notes on the results: the speedup according to Amdahl's law assumes that specific workloads execute alone, while the empirical result is observed while all the workloads are executed at the same time; the consumed power is calculated assuming 1 unit/1 cycle/1 non-scalable core, and (frequency factor)*(frequency factor) /1 cycle/1 scalable core; the simulator did not increase the frequency to the maximum available in one step, but rather in several smaller steps, each time with a factor of 2x. However, this proved to be a good way to factor in HW latency when adjusting frequencies.

Scenario	Speedup vs sequential	Speedup according to Amdahl's law	Sequential cycles	Total cycles	Total Power
I	62	61,6	2048	34	2062
II	30	29,87	1982	67	3992
	30	29,87	1982		
III	40	39,79	2002	74	4052
	27,42	39,79	2002		
IV	15	14,96	1982	133	6112
	14,7	14,58	1000		
	14,4	14,19	1010		
	7,5	7,29	950		
V	30	29,87	1982	121	6991
	28,57	28,71	1000		
	15,3	27,56	1010		
	8,6	11,32	950		
VI	8,9	11,32	950	108	6170
	19,8	27,56	1010		
	28,57	28,71	1000		
	18,52	29,87	1982		
VII	11,3	7,29	950	84	3384

Figure 1 *Empirical results*

In non-resource constrained scenarios (I, II, IV, VII) the algorithm matches or outperforms the scalability predicted by Amdahl's law; e.g. in scenario VII the performance is 55% better due to shutting off all cores but one for the sequential part of the application. The processor utilization expressed as consumed power stayed at or above 95% for 75% percent of the time and never drops below the 66% level for all scenarios.

RELATED WORK

An evaluation of technological issues for building 1000 core chips was done in [1], with the conclusion that voltage and frequency scaling has the potential of providing good theoretical performance within a limited power budget. It also proposes the usage of simpler cores, in line with [7], proposing a method for estimating the optimal size of processor cores; the proposal for implementing multiple clock domains with dynamic voltage and frequency scaling in [11] lays the architectural groundwork for the algorithm described in this paper. Various many-core OSes (fOS, Barrelfish, Corey) are described in [2], [12] and [13]. Scheduling algorithms for heterogeneous CMP systems were proposed in [6], [8], [9] and [10]. All rely however on static, heterogeneous architectures and thus propose various thread migration policies. [10] describes an approach based on architectural signatures, with hints about the cores on which the applications shall be scheduled. This is however *statically defined* and hence has a limitation with regards to which application characteristics can be taken into account.

REFERENCES

1. Borkar, S. Thousand Core Chips – A Technology Perspective. *In Proceedings of DAC*, 2007
2. Wentzlaff, D., Agarwal, A. The Case for a Factored Operating System. *MIT CSAIL Report*, 2008
3. Amdahl, G.M. Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities. *In AFIPS Proceedings*, 1967.
4. Hill, M.D., Marty, M.R. Amdahl's Law in the Multicore Era. *IEEE Computer*, July 2008.
5. Smith, B. Many-core Operating Systems, *WIOSCA, keynote speech, in conjunction with ISCA-34*, 2007
6. Kumar, R., Tullsen, D.M., Ranganathan, P., Jouppi, N.P., Farkas, K.I. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proceedings of ISCA-31*
7. Agarwal, A., Levy, M. The KILL Rule for Multicore. At *44th DAC*, June 2007
8. Becchi,M., Crowley,P. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. *Proceed-ings of the Conference on Computing Frontiers*, 2006
9. Fedorova, A., Vengerov, D., Doucette, D. Operating System Scheduling on Heterogeneous Core Systems. In *Proceedings of the Workshop on OS Support for Heterogeneous Multicore Architectures*, 2007
10. Shepelow, D., Fedorova, A. Scheduling on Heterogeneous Multicore Processors Using Architectural Signatures. In *Proceedings of WIOSCA*, at ISCA-35, 2008
11. Semeraro, G., Magklis, G., Balasubramonian, R., Albonesi, D.H., Dwarkadas, S., Scott, M.L. Energy-efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling. In *HPCA*, February 2002
12. Boyd-Wickizer S. et al. Corey: An Operating System for Many Cores. In *Proceedings of the 8th USENIX OSDI Symposium*, , 2008
13. Schüpbach, A., et. al. Embracing diversity in the Barrelfish manycore operating system. *At Workshop on Managed Many-Core Systems*, June 2008