Systems Research Group

School of Electrical and Electronic Engineering
School of Computing Science

Adaptive resource control in multi-core systems

F. Xia, A. Mokhov, A. Yakovlev, A. Iliasov, A. Rafiev, A. Romanovsky

Technical Report Series

NCL-EEE-MICRO-TR-2013-183

Dec 2013

Contact: fei.xia@newcastle.ac.uk, nai11@ncl.ac.uk

NCL-EEE-MICRO-TR-2013-183
Copyright © 2013 Newcastle University

Systems Research Group
School of Electrical and Electronic Engineering
Merz Court
Newcastle University
Newcastle upon Tyne, NE1 7RU, UK

http://async.org.uk/

Adaptive resource control in multi-core systems

F. Xia, A. Mokhov, A. Yakovlev, A. Iliasov, A. Rafiev, A. Romanovsky

Dec 2013

Abstract

Multi-core systems present a set of unique challenges and opportunities. In this paper we discuss the issues of power-proportional computing in a multi-core environment and argue that a cross-layer approach spanning from hardware to user-facing software is necessary to successfully address this problem. $\not\subset$

1 Introduction

Translating integration scaling to performance growth is challenged by such factors as the utilization wall [1]. Processor clock frequencies have not increased since 2005 despite increasing transistor speed [2]. Using multi-core to maintain the predictions of Moore's Law [3] will only delay the inevitable [4], as the near-threshold computing (NTC) advantages are limited by such factors as variability and reliability concerns when voltage is radically scaled down. When other issues, such as reliability, are considered in addition to performance in the context of increasing parallelism (e.g. increasing the number of cores), there is little concrete research result to be found in the literature and the picture is even more uncertain.

Modern computing systems, especially mobile and/or embedded systems, must deal with a high degree of uncertainty from within the systems themselves and without in the environment. Examples include the not-always predictable inherent physical parameters such as temperature, voltage, energy availability, external noise, variability etc. and from unpredictable user demands. In communication-heavy devices the environment a device is communicating to can also be highly unpredictable, a natural consequence of networks based more on the concept of best effort and probabilistic behaviours than guaranteed service and deterministic behaviours. Managing and adapting to on-chip conditions such as power, thermal dissipation, and computation flow, therefore, are vital for pushing computing forward. Powering off inactive cores [5], dynamically changing clock frequencies [6] and other power saving and performance boosting measures [7] already exist, but runtime monitoring, feedback, management and control are needed for systems to operate close to their power and thermal budgets under process and environment variation and workload conditions difficult to predict at design time.

A comprehensive runtime management system must take into account all of the natural layers of computation, from application to core software (including OS) to hardware, able to receive and process information from all layers and maintain cross-layer communications and interactions. For this to be meaningful, a solid theoretical foundation based on concrete mathematical models at all levels of detail is necessary.

In Section 2 we present an initial investigation of the interplay of the essential parameters of multi-core computation, including voltage, throughput, and reliability as well as how they relate to the scale of parallelism, i.e. the number of cores being used. In Section 3 we apply the studied principles to design a runtime adaptive

management system that tries to optimize resource utilisation in a multi-core environment by involving user-space software in the process of planning and scheduling resource allocation.

2 Power, energy and reliability

In digital CMOS systems, a higher supply voltage (V) usually allows a higher operating (clock) frequency and hence a higher throughput, at the cost of higher power dissipation. When power is limited, it is possible to obtain an increase in system throughput by scaling to multiple computation units, i.e., cores, if the computation can be reasonably parallelized. Parallelization scaling in this fashion could also be used to tackle the related problem of reducing power consumption while faced with a certain throughput requirement.

This type of parallelization scaling has been known to provide the best advantage when V is scaled down to just above the threshold voltage of the CMOS node concerned. This is known as near threshold computing (NTC), which maximally takes advantage of the parallelization scaling without entering the sub-threshold region, where a number of other factors limit the throughput and efficiency under further parallelization. However, what happens to reliability in this region is not well known, because these general knowledge points were obtained usually by considering reliability as a separate issue. Here we study the inter-relationship of all these issues together, namely voltage, throughput, power, reliability and parallelization scaling.

In order to retain reliability, a system must operate within certain constraints. For instance, a particular hardware implementation may not behave correctly if the supply voltage V goes below or above certain values. Different hardware components in a system may have different minimum and maximum V values and this means that software components, depending on how they are mapped onto hardware, could also be constrained by different V limits, even within the same system. Another type of constraint is the performance/throughput requirement specification. A system or a part of a system may be required to attain at least a certain level of throughput for the execution to be meaningful. A third type of constraint is the power supply limitation. The amount of available power limits the behaviour of the system.

The minimum latency, which is related to computation throughput, of any specific hardware logic is related to the V supplied to it. This means that if this logic is run on a clock too fast for a certain V the computation may not complete in time before the next clock pulse arrives, which usually leads to unreliable or unusable results. A common technique for determining the appropriate clock frequency for computation logic is to first determine the critical path delay of the logic under the given V condition, and add enough delay margins to account for potential effects of process, voltage and temperature (PVT) variations.

The region of reliable operation for a system within the V/throughput space is bounded by constraints on power, timing reliability, minimum throughput requirements and low and high V boundaries. The minimum and maximum V boundaries and the minimum throughput are defined as

$$\min V \leq V \leq \max V \qquad T \geq \min T \tag{1}$$

The power limit and timing reliability boundaries, however, are more complex. Their general shapes can be derived from theories on semiconductor characteristics. For instance, the timing reliability boundary within a V – throughput/frequency space usually starts off at very low V with the frequency climbing from almost zero upwards exponentially until V increases to around the threshold voltage, from there upwards the increase in frequency is more or less linear, until usually when V is around the nominal V of the technology, where the frequency increase starts to saturate. The region of reliable operation can is illustrated by the diagram in Fig. 1.

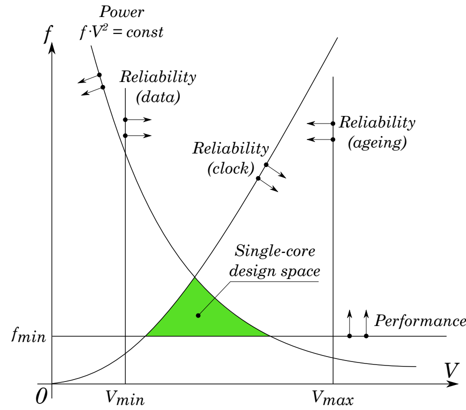

Figure 1: The region of reliable operation

For reliable operation, the system throughput must be restricted below the timing (clock) reliability and power limit lines and must be above the throughput requirement line. The system's V must also be restricted between the high and low V limits. In the super-threshold region, we can hypothesize that dynamic power dominates power consumption and leakage power and its influence can be ignored.

Dynamic power is known to be related to switching activity (and through which to system frequency), switching swing voltage (and through which to system V) and switching element capacitance (and through which to system size/area – which is a constant before hardware scaling). Lumping all the constants together we can say that power is related to frequency and V in the following manner:

$$P = AFV^2 \tag{2}$$

where A is a constant, P is the power and F is the frequency. In this work, we explore the issue of core scaling (increasing or decreasing the degree of parallelization) with an assumption of perfect scaling. Multiplied hardware operating at the same frequency will provide multiplied throughput and require a multiplied amount of power based on the same constant multiplier. Non-zero scaling overheads will be investigated in the future. In perfect scaling with a scaling factor of n, the constant A is scaled in the same way, i.e.

$$A = nA_1 \tag{3}$$

where A_1 is the A for the hardware before scaling (e.g. a single core). In general, scaling with a factor of n will give a new A which is a factor of n of the unscaled A.

For each core in a new scaled set-up, the available power is also changed by a factor of $1/n$. Considering these factors, for each core, equation (2) now becomes

$$P_n = AFV^2/n \tag{4}$$

and the overall system power equation with n cores stays the same as (2).

With a V values near threshold to below threshold, the super-threshold power relation (2) no longer describes the total power as leakage power starts to become an equally important or more important factor compared with dynamic switching power.

Instead of drawing constant power curves from equations (2) and (4) or dealing with the much more complex power equations taking leakage power into account, observed power from experiments can be used to estimate

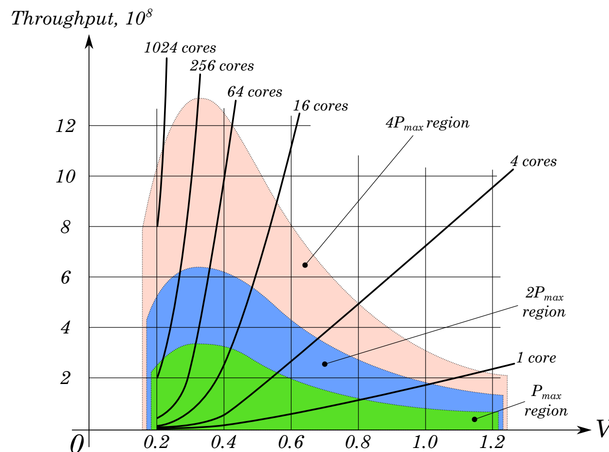

Figure 2: SRAM constant max power curve and scaling lines

the points along the frequency curves pertaining to any known power budget.

Fig. 2 shows the scaling effects on the reliable operation region, based on experimental data collected from an asynchronous SRAM [8], with the total power from experimental data (and not calculated from (2) and (4)) including both dynamic and leakage power. As can be seen after about $n = 256$, further scaling would not improve the system throughput of the system given the constant power limit reached at the max writing frequency and voltage of $1.2V$ without scaling, but the operable region continues to be expanded to the left, although the increase becomes smaller even without considering $\min V$.

To determine the shape of power limit boundary curves, we start by setting a power limit amount, P_{lim}, usually the power consumption when operating an unscaled system at nominal V and the appropriate reliable frequency for this V (P_{max}). Then for each point i where there is experimental power data, we calculate the maximum possible scaling factor n_i for that V based on

$$n_i = P_{\text{lim}}/P_i \tag{5}$$

where Pi is the experimental power observed at data point i. This, similar to (3) and (4), is based on the perfect scaling assumption. Plotting $T = n_i T_i$ gives the power limit curve for the particular P_{lim}.

Fig. 3 is a similar study on the reliable operation region and its relationship with power and parallelization scaling. This is based on experimental data from a low-power ARM M0 core implementation [9]. Both examples demonstrate that the relationship between power limits, parallelization scaling, and the reliable operation region within the V/throughput space is the same with both processors and memory, and the reliable operation region is reasonably straightforward to model given standard experimental data collected during any reasonable hardware design process.

3 Adaptive control

This section describes an approach to adaptive control that aims to increase energy efficiency and reliability of computer systems by introducing an OS level software and hardware management layer. The layer makes use of previously unavailable or under-utilised hardware reconfiguration facilities and makes applications actively

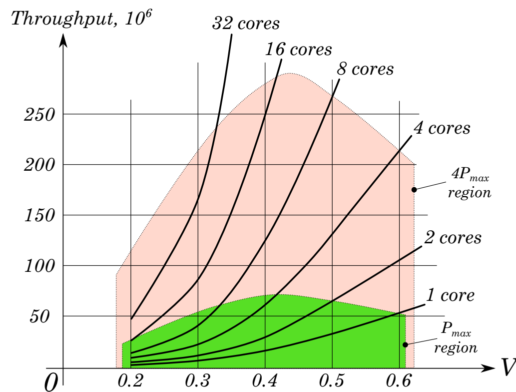

Figure 3: Scaling close to and in the subthreshold region

involved in resource and power management. There are three main parties to the subject of our study: *user* (either a human user or another information system), *application* and *hardware*. There is also an implicit OS layer that coordinates interaction of applications with hardware.

The current situation with adaptive control may be summarised as follows: applications requests resources from an OS and largely assumes that any such request is successful. When it is not, an application is aborted or delivers inadequate QoS. Also, when previously allocated resources become unavailable during an application lifetime (i.e., an application gets a lower share of core time), there is no mechanism for an OS to inform an application other than request an abrupt termination. There is little a user can do to influence system power consumption or performance other than through a simplistic mechanism of frequency governors: an OS service adjusting core frequency (and possibly switching them on and off) in a delayed reaction to core load.

In principle, *user*, *application* and *hardware* may be characterised by a finite state transition system. The number of states might be extremely large and there could be no practical way to accurately characterise all states and transitions connecting them. It is often possible, however, to give a rough approximation, capturing just few relevant aspects of overall behaviour. We call such an approximation a *meta-model* of, respectively, user, application and hardware.

We make an assumption that there is no direct causation between actions of either party. That is, either can act at will at all times. For instance, a user does not control hardware directly nor hardware may interfere with application behaviour (i.e., by altering memory). Enforcing certain causation links is essential for making the complex of the three to function efficiently. However, until the definition of an adaptive control, we assume a completely uncoordinated behaviour.

3.1 Meta-model

A meta-model is an abstraction of a system at study. By a system we understand a state transition system $(\Omega, \rightarrow, \omega)$ where Ω is the universe of states, $\rightarrow \in \Omega \rightarrow \mathscr{P}(\Omega)$ defines possible transitions, and $\omega \subseteq \Omega$ is the set of initial states.

A meta-model is a state transition system $(\Sigma, \rightsquigarrow, \sigma)$ such that there exists a relation $\rightleftharpoons \in \Omega \leftrightarrow \Sigma$ where for each abstraction step $s_1 \rightsquigarrow s_2$ there is a matching system step $s'_1 \rightarrow s'_2$ such that $s'_1 \rightleftharpoons s_1$, $s'_2 \rightleftharpoons s_2$.

It is common to call some state $s \in \Sigma$ a *mode* of a system abstracted by the meta-model. Meta-models of user, application and hardware have the same structure. They differ in the kind of annotations attached to states (modes).

Although definitions are symmetric for all the meta-models the roles they play in optimisation are not the same. For any given triplet of user, application and hardware models, the user model is the least flexible one while the hardware is the most flexible one. That is, optimisation tries to primarily reconfigure hardware and only then application while trying to meet evolving user requirements that are not under the control of an optimisation logic. To summarise,

- user meta-model is not adaptable and may be changed outside of the system boundaries at any moment;
- application meta-model evolves autonomously in step with application evolution but may also be adapted forcing an application to reconfigure;
- hardware is under the direct control of the system; it may be reconfigured at any moment; it may, however, experience unpredictable failures.

The meta-model of user remains unknown and we treat it as a block box; the meta-model of an application is defined by an application designer (we are currently working on defining and implementing Linux kernel API for this); the model of a hardware is built by a hardware or OS designer and is specific to each hardware platform.

While a meta-model itself defines which states may be reached, model annotations explain the cumulative effect of the abstracted system states on the significant system characteristics such as power or reliability. To paraphrase, a meta-model answers the question of how to (if possible at all) arrive at certain state (mode). Annotations help to understand to which state one should be trying to advance to meet some optimisation criteria. In the following section, due to space restriction we only briefly overview the kinds of model annotations specific to each model type.

3.2 Meta-models and their annotation

User meta-model annotation captures QoS requirements relevant to a given application and the context in which it is used. The principle attributes are the *power budget* stating power requirements at any given moment of time, *reliability* in terms of mean time between failures (MTBF), and *performance* stating the lower bound on hardware performance in terms of operations per second (to reflect QoS requirements for a give application).

The transitions of an application meta-model mark pivotal moments of an application evolution. These must be foreseen an defined by an application developer. Application meta-model annotations include *resource consumption*, *energy promise* (the promise of the application to not require for itself more than x watts), *reliability* in terms of MTBF, *threads* and their *core utilisation*.

The meta-model of hardware characterises hardware operational modes in terms of power, performance, reliability and longevity. Hardware meta-models annotations are *resource availability*, *power budget* (may have negative power values to signify inflow of energy i.e., due to battery charging), per-core MTBF *reliability* metric, and per core *ageing* metric. We use the following configurable definition of hardware meta-model transition system:

$$(\mathbf{F} \times \mathbf{V} \times \mathbf{C}, \hookrightarrow, _)$$

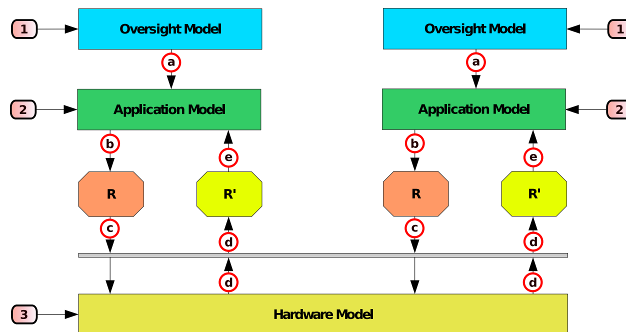

Figure 4: Major directions of control flow within and into the system.

where $\mathbf{F} \in \mathbf{C} \to \mathbb{R}$, $\mathbf{V} \in \mathbf{C} \to \mathbb{R}$ and $\mathbf{C}$ are frequency, voltage and core count domains. Set $\mathbf{C}$ enumerates all the available cores. A state transition $\hookrightarrow$ may achieve some or all of the following: (1) alter cores frequency; (2) alter cores voltage; (3) enable/disable cores.

At all times, hardware must remain in the region of reliable operation (Fig. 2). Depending upon application parallelism and core utilisation properties, hardware platform would adjust voltage and frequency, and disable cores to offer maximum performance using strategies outlined in Section 2.

3.3 Optimisation problem

Recall that the overall meta-model of a system is a Cartesian product of user, application and hardware meta-models. The objective of optimisation is to restrict the overall meta-model to a subset of models exhibiting desired behaviour.

Assume U, A and H are the user, application and hardware meta-models and $f \in T \times \mathsf{U} \times \mathsf{A} \times \mathsf{H} \to \mathbb{R}$ is a reward function judging on a specific configuration at a given time; T is a temporal domain. The goal of optimisation is formulated as

$$\arg\max_{s} f(t,s) \qquad s \subseteq \mathsf{U} \times \mathsf{A} \times \mathsf{H},\ \ t \in T$$

One way to identify promising configurations s is to introduce coordination between the meta-models.

3.4 Adaptive control

We briefly discuss the information flow introduced by an adaptive controller. Assume a system is currently in an optimum configuration. There are several ways it can be taken out of this configuration (see Fig. 4): (1) a change in a user meta-model induces application reaction; (2) an application evolution is reflected in the corresponding application meta-model state evolution; (3) detection of a hardware fault causes hardware reconfiguration.

Fig. 4 illustrates the main events within the system: (a) an application model adapts to user requirements (the current hardware configuration is not taken into the account); (b) resource request reflects the new application configuration; in doing so it disregards current hardware state and simply goes to most fitting state reachable from a current state; (c) hardware collates resource requests to compute a hardware reconfiguration step; there is a time window during which requests are collected but not acted upon; (d) updated resources reflecting actual resource allocation are issued to each application; (e) an application adapts to new hardware configuration; current user requirements are taken into the account.

Most of the activity happens in application reconfiguration steps (a), (e) and the hardware reconfiguration (c). At step (a), an application tries to minimize any mismatch between user requirements and the current application configuration. At this point, the application is free to request any hardware configuration. At step (c), the management layer computes a hardware configuration satisfying to the greatest possible extent hardware models computed at step (b). Generally, there will be some mismatch between actual and requested hardware configurations. Step (e) tries to find an alternative application configuration that will satisfy the allocated hardware model and minimise the mismatch with the user requirements.

Events (a) - (e) are the steps of a *three-partite protocol*. We are working on a formal model of this protocol and are going to implement a number of adaptation algorithms.

References

[1] N. Hardavellas et al., Toward Dark Silicon in Servers, IEEE Micro, 31(4)

[2] N. Goulding-Hotta et al., The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future, IEEE Micro, 31(2), 2011.

[3] S. Fuller, L. Millett, Computing Performance: Game Over or Next Level?, Computer, 44(1), 31-38, 2011.

[4] H. Esmaeilzadeh et al., Dark Silicon and the End of Multicore Scaling, IEEE Micro, 32(3), 2012.

[5] A. Branover et al., AMD Fusion APU: Llano, IEEE Micro, 32(2), 2012.

[6] R. McGowen et al., Power and Temperature Control on a 90-nm Itanium Family Processor, IEEE JSSC, 41(1), 229-237, 2006.

[7] H.-Y. McCreary et al. EnergyScale for IBM POWER6 microprocessor-based systems, IBM JRD, 51(6), 775-786, 2007.

[8] A. Baz et al, Self-timed SRAM for Energy Harvesting Systems, Journal of Low Power Electronics 2011, 7(2), 274-284.

[9] J. Mistry, Leakage Power Minimisation Techniques for Embedded Processors, PhD Thesis, University of Southampton, 2013.