*IRWIN AND JOAN JACOBS*
*CENTER FOR COMMUNICATION AND INFORMATION TECHNOLOGIES*

# Performance and Power Aware CMP Thread Allocation Modeling

**Yaniv Ben-Itzhak, Israel Cidon and Avinoam Kolodny**

**CCIT Report #735**
**June 2009**

***DEPARTMENT OF ELECTRICAL ENGINEERING***
***TECHNION - ISRAEL INSTITUTE OF TECHNOLOGY, HAIFA 32000, ISRAEL***

*המרכז לטכנולוגיות תקשורת ומידע*
*הפקולטה להנדסת חשמל*
*הטכניון - מכון טכנולוגי לישראל, חיפה 32000, ישראל*

CCIT REPORT #735 June 2009

# Performance and Power Aware CMP Thread Allocation Modeling

Yaniv Ben-Itzhak[1], Israel Cidon[2], Avinoam Kolodny[2]
Electrical Engineering Department, Technion, Haifa, Israel
[1]yanivbi@tx.technion.ac.il [2]{cidon, kolodny}@ee.technion.ac.il

## Abstract

CMP is the architecture of choice for addressing the performance and power requirements of future computational systems. In this paper we address the problem of mapping software threads to coarse-grain multi threaded cores. Our goal is performance and power-efficient thread allocation in a CMP. To that end, we develop a parameterized performance/power metric that is adjusted according to a preferred tradeoff between performance and power and is based on an analytical model. We introduce an iterative threshold algorithm (ITA) for allocating threads to cores in the case of a single application with symmetric threads. We extend this to a simple and efficient heuristic for the case of multiple applications. We compare the performance/power metric value of the ITA with the results of several standard optimization methods. Our algorithm outperforms the best of these methods, while consuming less than 0.05% of the computational effort. Consequently, the ITA can serve as a basis for practical thread allocation in CMP systems, taking into account the preferred tradeoff between performance and power.

## 1 Introduction

The demand for more capable microprocessors has led CPU designers to explore various types of parallelism. Many applications are suited to thread level parallelism (TLP), and multiple independent processors can be used to increase a system's overall TLP [1][2]. The combination of increased transistor count due to advanced manufacturing processes and the inherent scalability limitations of a single processor design drove the introduction of chip multi-processors (CMP). CMPs offer higher power efficiency, better performance, and lower design complexity in comparison with other ways to achieve high performance in hardware. However, CMP architectures introduce new challenges associated with massive parallelism. This is emphasized by the fact that CMPs carry parallelism beyond its traditional super-computing and server markets to desktop, notebooks, PDAs and other mobile systems running a rich mix of heterogeneous applications and involving complex power and energy saving requirements. The heterogeneous and pervasive use of CMPs raises the need for different and tunable tradeoffs between system performance and power consumption. There is a clear difference between the requirements of a fan-cooled server in a server farm and a personal laptop computer. Battery operated mobile systems have a completely different operating point as compared to grid-powered equipment. Moreover, the same mobile equipment may have different performance-power settings according to its battery charge level, the set of applications that are currently executed and its desired residual operation time. Application examples can be watching a video during a flight, making a presentation and using the mobile equipment as a video camera or as a surveillance sensor.
In this paper, we address the important problem of efficient allocation of applications and threads to cores in a CMP, taking into account the desired tradeoff between performance and power, unlike works which deal with performance only [5]-[7] or power only [8].
We examine a system where several cores with a shared cache are interconnected by a network on chip (NoC) [2], for example the Piranha CMP [3] or Nahalal [4]. The CMP based system executes several multi-threaded applications. Each core performs coarse-grain multithreading. For reduced power dissipation and battery energy saving, any unused core may be shut down. Varying

operational and battery conditions may dynamically change the preferred tradeoff between performance and power, thus dictating a different goal for the threads allocation.
In order to maximize the performance, one would maintain a light load in each core. In such a case, the threads are thinly spread over many cores that in turn increase power consumption. On the other hand, running all threads in a single core would minimize the consumed power, since all of the other cores can be shut-down, but the performance would be greatly hurt. Our iterative algorithm for thread allocation utilizes the CMP resources in a way that maximizes a parameterized performance/power metric which can be adjusted according to the preferred tradeoff between performance and power.

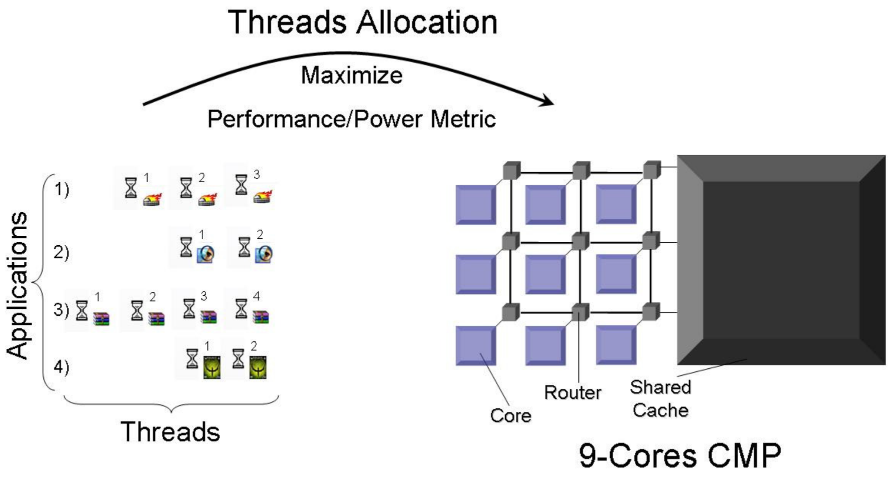


**Fig. 1 – Example of the thread allocation problem**

Fig. 1 presents an example of a 9-core CMP (each core includes a private first-level cache), connected by a NoC to a shared second-level cache. The system concurrently executes four multi-threaded applications that need to be allocated to single-pipe coarse-grain multi-threaded cores. Naturally, only a single thread can run in each core at a given time. We assume that such a thread runs until it incurs a miss in its private cache, then the thread waits for the shared cache response. At that time, the core may perform context switching to the next ready thread (i.e., a thread that already received its response from the shared cache). It is clear that the shared cache response time has a major effect on the performance of each thread. The cache response time is the sum of the shared cache access time and the NoC latency, which depends on the hop distance between the corresponding core and the shared cache. Therefore, both the number of threads allocated to each core and the location of these cores in respect to the shared cache are important for the system performance.
Our goal is to find a thread allocation that maximizes a given performance/power metric. Our performance model considers both communication delays between the cores and the shared cache, and the core performance dependence on the number of the threads it executes. Our power model considers both active and idle power of the cores (but for simplicity it does not account for the power associated with context switching). Based on these models, a parameterized performance/power metric is defined, which can be adjusted (either statically or dynamically during system operation) according to the relative importance of performance versus power consumption.
We introduce an Iterative Threshold based thread allocation Algorithm (ITA) that maximizes the performance/power metric for the case of a single application with symmetric threads. We extend it for the case of multiple applications, which achieves better and faster results than standard general purpose optimization algorithms, making it feasible to implement in practical hardware.
Previous works have addressed somewhat related problems: Ding et al. [9] proposed thread allocation that maximizes a performance/power metric while considering process variations in the cores. However, in their model each core can be assigned only one thread while our work deals with the multi-threaded cores case. Furthermore, unlike [9], our work deals with shared memory architecture and considers the cache miss rate. Chen et al. [10] introduce performance and power

aware thread allocation for NoC-based CMP, which attempts to optimize a "locality metric" of data accesses. In contrast, our work uses explicit power and performance models, and introduces an adjustable metric for thread allocation according to the relative importance of performance versus power.
The rest of this paper is structured as follows. We present the performance and power models and the performance/power metric in section 2. In section 3 we introduce the *single application problem* and the iterative algorithm for thread allocation and we extend the problem into the *multiple applications problem*. In section 4 we present numerical results for both problems and demonstrate the efficiency of the algorithm for the case of multiple applications relative to several standard optimization algorithms. Section 5 concludes the paper

## 2 Power and Performance Models

Agarwal [13] proposed a coarse-grain multi-threaded core model which we extend for our analysis. We define *N* as the number of threads executed by a multi-threaded core *c* with a private cache. We define $\delta_j$ as the average number of core cycles between private cache misses for thread *j*, i.e. $\delta_j = 1/(r_m \cdot \mathrm{m.r_j})$, where $r_m$ is the ratio of memory access instructions and $\mathrm{m.r_j}$ is the cache miss rate for thread *j*. Also, we define $T^{(c)}$ as the number of core (c) cycles required to satisfy such a request from the shared cache. [13] assumes that all the threads have the same cache miss rate (i.e., the same $\delta_j$ for all *j*). For simplicity, at this point we assume that $T^{(c)}$ is fixed for all threads. On average, thread *j* executes instructions for $\delta_j$ cycles until it hits a private cache miss, and then waits for $T^{(c)}$ cycles for the miss request to be satisfied before it can execute more instructions. For the sake of simplicity, we assume that context switches happen only at private cache misses and we ignore thread-switching cycles and the associated power overhead. Generally, as the number of threads allocated over a core increases their miss rates also increase due to the sharing effect. In this paper, for the sake of simplicity, we assume that sharing does not affect the miss rates of the threads. This assumption is reasonable in cases where the entire footprints of the threads are located on the private cache and thus hold for a relatively small number of threads per core. In order to quantify the validity range of this assumption, for each of the benchmarks presented in [16] we simulated a single thread for, measured its miss rate versus cache size by "pin tool" and obtained its footprint size. Our results show that *blackscholes* has footprint of 3kB, *fluidanimate* 7kB and *freqmine* 8kB. Therefore, such benchmarks satisfy our assumption. Future work will address the case where the cache sharing effect cannot be ignored.

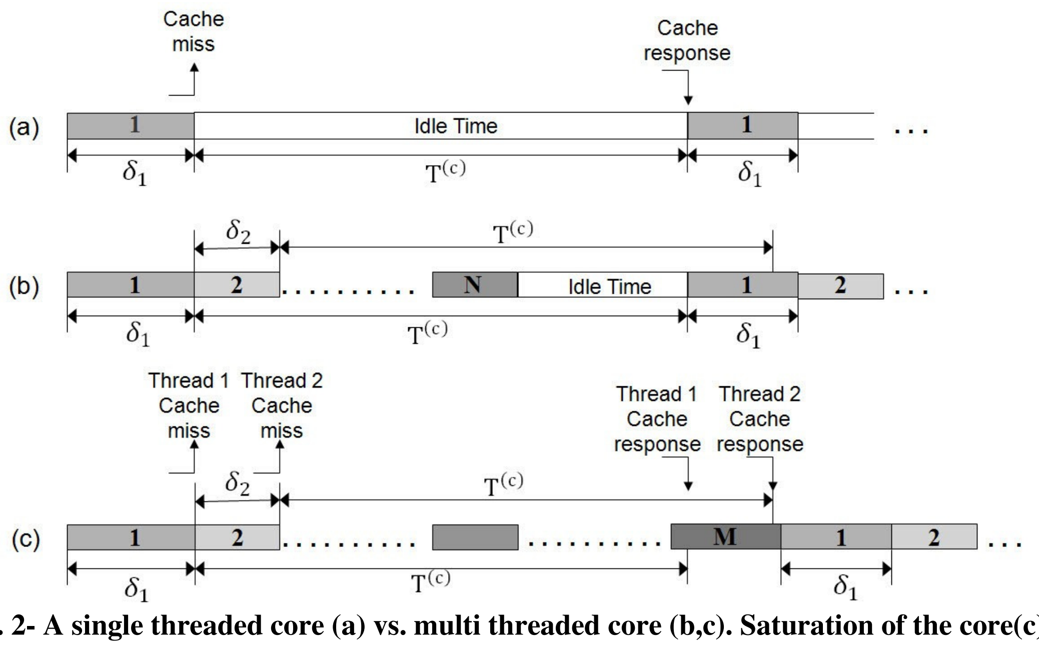


**Fig. 2- A single threaded core (a) vs. multi threaded core (b,c). Saturation of the core(c).**

Coarse-grain multi-threaded cores utilize cache access time by running other threads, which increase the overall system power efficiency (Fig. 2(b-c)). Fig. 2(b) presents that while thread 1 is waiting for its cache response, other (*N-1*) threads are executed, and the total idle period decreases compared to Fig. 2(a). Fig. 2(c) presents the case where thread 1 cannot be executed immediately when its cache response arrives, because other (*M-1*) threads didn't finish their execution. This case is denoted as *saturation* of the core.

Under the assumption that a context switch happens only at private cache misses, when threads with different cache miss rates run in a multi-threaded core, a cache response may arrive while another thread is still running, therefore the thread waits for its execution until the core becomes available. Assuming (with no loss of generality) that thread 1 has the largest value $\delta_1$ among all threads, we prove in Appendix A that the execution order of the threads eventually reach a periodic steady state, in which thread 1 runs first and all the other threads run after it with no gaps (see Fig. 3(a)). In the steady state, if the core is not saturated, then thread 1 runs immediately after its cache response arrives, and any other thread has to wait for its execution although its cache response has already arrived.

We define $\Delta \triangleq \max_{1\le j\le N}\{\delta_j\}$, as the largest value of $\delta_j$ among all $N$ threads are executed by the core. Therefore, in the unsaturated steady state, the time between executions of any thread $j$ is $T^{(c)} + \Delta$. Core saturation happens when thread 1 has to wait for execution after its cache response arrives. Therefore, *saturation threshold* is defined as $th_{sat}^{(c)} \triangleq T^{(c)} + \Delta$. Fig. 3(b) presents the *saturation threshold* case. In saturation, the time between executions of thread $j$ is $\sum_j(\delta_j)$, and it exceeds $th_{sat}^{(c)}$.

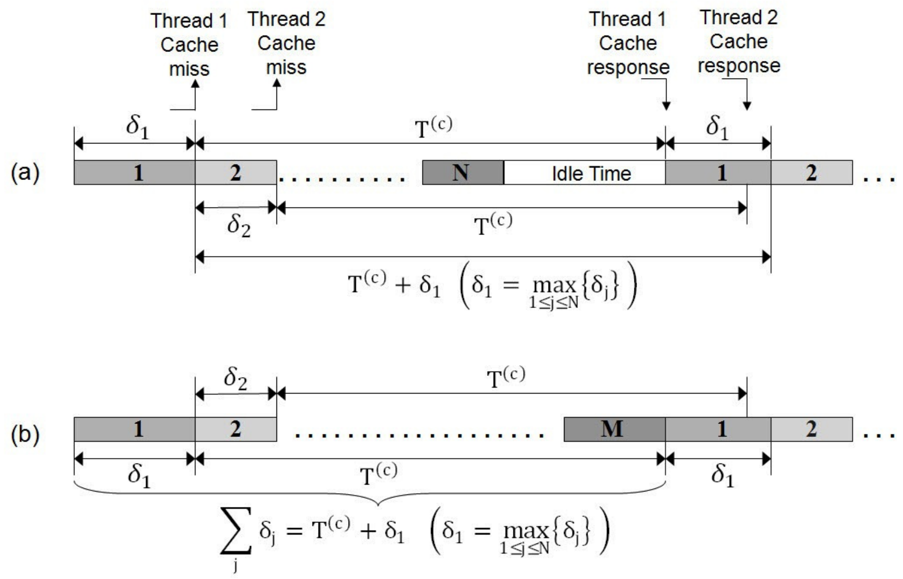

Thread 1 Cache miss
Thread 2 Cache miss
Thread 1 Cache response
Thread 2 Cache response
$\delta_1$
$T^{(c)}$
(a)
1
2
N
Idle Time
$\delta_2$
$T^{(c)} + \delta_1 \left(\delta_1 = \max_{1\le j\le N}\{\delta_j\}\right)$
(b)
M
$\sum_j \delta_j = T^{(c)} + \delta_1 \left(\delta_1 = \max_{1\le j\le N}\{\delta_j\}\right)$

**Fig. 3 – (a) Steady state and time between each thread executions, (b) *saturation threshold* case**

We define the performance of thread $j$ in core c to be its useful execution time percentage out of the achievable IPC (Instructions per Clock) and the utilization of core c, $\eta^{(c)}$, which is defined as the busy time percentage of the core. Therefore,

$$\text{Thread}_j\text{Performane}^{(c)} = \begin{Bmatrix} \frac{\delta_j}{th_{sat}^{(c)}} & ;\sum_k \delta_k \le th_{sat}^{(c)} \\ \frac{\delta_j}{\sum_k \delta_k} & ;\sum_k \delta_k > th_{sat}^{(c)} \end{Bmatrix} (2.1) \qquad \eta^{(c)} = \begin{Bmatrix} \frac{\sum_k \delta_k}{th_{sat}^{(c)}} & ;\sum_k \delta_k \le th_{sat}^{(c)} \\ 1 & ;\sum_k \delta_k > th_{sat}^{(c)} \end{Bmatrix} (2.2)$$

The utilization of a saturated core is 1.

The power consumption of core c depends on its utilization, such as:

$$\text{CorePower}^{(c)} = \begin{Bmatrix} \eta^{(c)}P_{active}^{(c)} + \left(1-\eta^{(c)}\right)P_{idle}^{(c)} & ; 0 < \eta^{(c)} \le 1 \\ 0 & ; \eta^{(c)} = 0 \end{Bmatrix} \quad (2.3)$$

$P_{active}$ is the power consumption of a fully utilized core. ($P_{active}$ depends on its core voltage supply, clock frequency and effective dynamic capacitance). In our model we also take into account a possibly lower idle power consumption, $P_{idle}$, that may results from power saving mechanisms in the processor during the idle time, such as clock gating [14]. We assume that when the core has no active threads (i.e. its utilization equals zero) it is shut down (or switched into sleep mode) so its power consumption becomes zero. Therefore, the motivation of the algorithm for thread allocation is to shut down as many cores as possible by properly utilizing all other cores in order to maximize the performance/power metric.

Our example for a NoC based CMP is depicted in Fig. 1. The system has several cores with different distances from the shared cache (1-3 hops away from the shared cache in the example of Fig. 1). The distance from a core to the shared cache affects the value of $T^{(c)}$, the number of cycles required to obtain a response from the shared cache. Assuming that each hop has a constant latency denoted by $\tau$, we get:

$$T^{(c)} = T_{cache} + h^{(c)}\tau \Rightarrow th_{sat}^{(c)} \triangleq T^{(c)} + \Delta = T_{cache} + h^{(c)}\tau + \Delta \quad (2.4)$$

Where, $h^{(c)}$ is the number of NoC hops from core c to the shared cache. Fig. 4 presents the performance per thread and core utilization as function of number of threads running in the core, for different hops distances from the shared cache. In this example we assume that the threads are symmetric (i.e., have the same cache miss rate ($\delta_j = \delta\ , \forall j$).

The performance increases as the hop distance decreases, since the time to satisfy cache misses increases with hop distances from the cores to the shared cache $\left(h^{(c)}\right)$. In addition, a core which is closer to the shared cache has a lower saturation threshold. There are cases where a saturated core has better performance than a non-saturated core. For example, the 1 hop distant core with 3 threads has a better performance than a 2 hop distance core with the same number of threads although it is saturated.

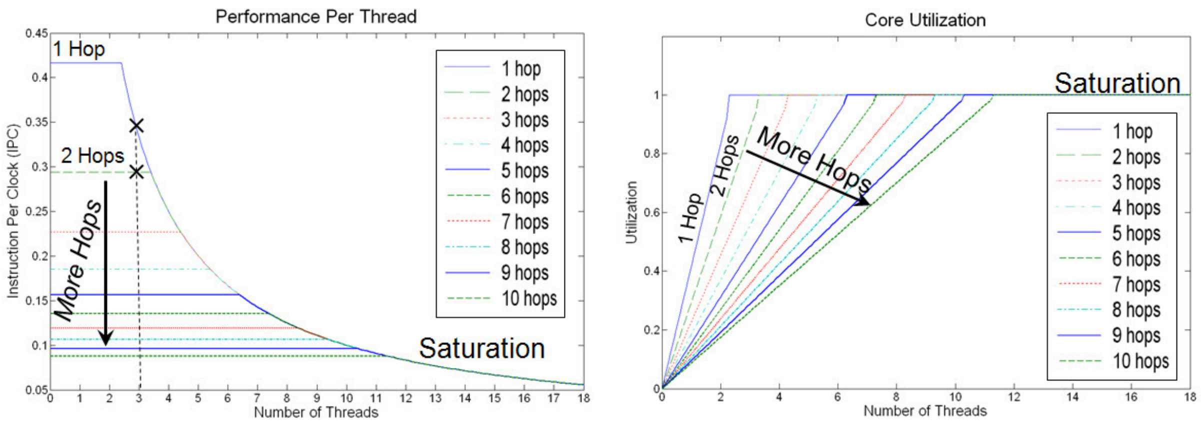


**Fig. 4 – Performance of each thread and Core Utilization vs. Number of threads $\tau$ =25 Cycles. $T_{cache}$=10 Cycles. Cache miss rate=4%.**

Our definition of the tradeoff between throughput (performance) and consumed power follows common definitions used in the field of logic circuit design. If $E$ is the energy and $t$ is the delay, Burd and Brodersen [11] introduce the $E \cdot t$ and $E \cdot t^2$ metrics, suggested for fixed-throughput application (e.g.: speech, video…) and maximum-throughput applications, respectively. P´enzes and Martin [12] extended these two metrics to the general metric $E \cdot t^{\alpha}$, where $\alpha$ is made larger as the performance becomes more significant.

The most general performance/power ratio metric is: $\left(\frac{\text{Average Performance}^{\alpha}}{\text{Consumed Power}^{\beta}}\right)$ (2.5)

Given a CMP with $M$ cores and $N$ threads, our goal is to find the optimal threads allocations which allocate each thread $j$ to core c such that the given performance/power metric is maximized. It can be easily shown that in the case $\beta > \alpha$, the optimal allocation is to allocate all threads in a single core. Therefore, similar to [12], we use the performance/power metric with $\beta = 1$ and $\alpha \geq 1$, and we define it by PPM $\triangleq \left(\frac{\text{Average Performance}^{\alpha}}{\text{Consumed Power}}\right)$.

## 3 Problems Statements and Allocation Algorithms

### 3.1 The Single Application Problem

We introduce *the single application thread allocation problem*, where a single multithreaded application is executed by a CMP with $M$ cores (each with a private cache) and a shared cache. The essence of the problem is how to allocate threads to cores, in order to optimize PPM.
For example, allocating all the threads to one of the cores and shutting down all other cores would result in low power but very low performance. In contrast to this, distributing threads to all of the cores in the system may be too wasteful in power and might incur unnecessary communication delays.

**The Single Application Problem Formulation:**

**Given**: A CMP with $M$ identical cores and a shared-cache, which executes an application with $N$ symmetric threads ($\delta_j = \delta\,, \forall j$). $h^{(c)}$ is the hop distance between core $c$ and the shared cache.
**Find**: Optimal thread allocation, $n^{(c)}$ the number of threads are executed by core $c$.

**Which:** maximizes PPM $= \left(\frac{\text{Average Performance}^{\alpha}}{\text{Consumed Power}}\right)$

**Subject to:** $\sum_{c=1}^{M} n^{(c)} = N\,; n^{(c)} \geq 0$.

**Where:** $\alpha \geq 1$

$$\text{Average Performance} = \sum_{c=1}^{M}\left(\frac{n^{(c)} \cdot \text{ThreadPerformance}^{(c)}}{N}\right)$$

(Thread Performance$^{(c)}$ depicted on equation (3.2))

$$\text{Consumed Power} = \sum_{c=1}^{M} \text{Core Power}^{(c)} = \sum_{c:n^{(c)}\neq 0} \left(\eta^{(c)} \cdot P_{active}^{(c)}\right) + \left(1-\eta^{(c)}\right) \cdot P_{idle}^{(c)}$$

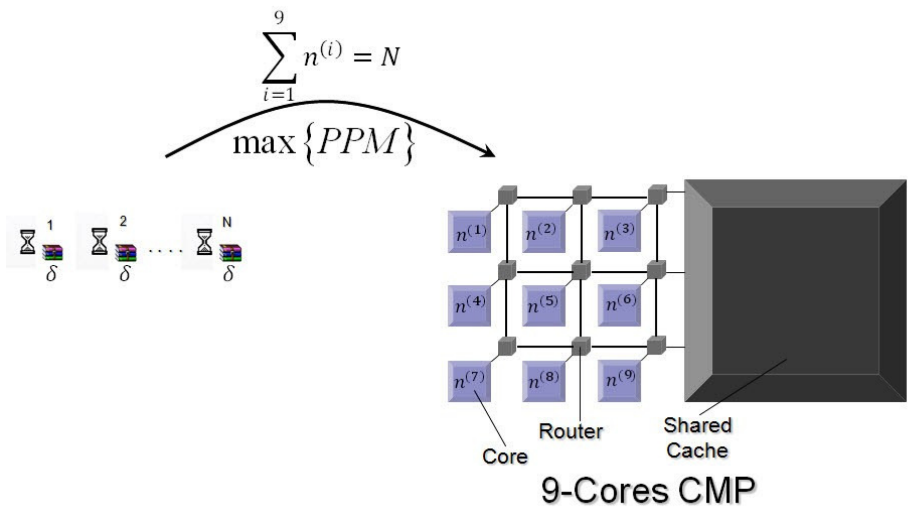


**Fig. 5 - Example of The Single Application Problem**

With the assumptions above, the saturation threshold of core c and the performance of each thread *j* running in core c can be written as:

$$th_{sat}^{(c)} = T_{cache} + h^{(c)}\tau + \delta \quad (3.1) \qquad \text{Thread Performane}^{(c)} = \begin{cases} \frac{\delta}{th_{sat}^{(c)}} & ; n^{(c)} \le \frac{th_{sat}^{(c)}}{\delta} \\ \frac{\delta}{n^{(c)}\delta} & ; n^{(c)} > \frac{th_{sat}^{(c)}}{\delta} \end{cases} (3.2)$$

Where $n^{(c)}$ indicates the number of threads in core *c*.

When $n^{(c)} < \frac{th_{sat}^{(c)}}{\delta}$, the core can execute more threads without performance reduction.

When $n^{(c)} \ge \frac{th_{sat}^{(c)}}{\delta}$, the core is saturated and achieves its maximum total performance.

The core utilization $\eta^{(c)}$ is calculated by:

$$\eta^{(c)} = \begin{cases} \frac{n^{(c)}\delta}{th_{sat}^{(c)}} & ; n^{(c)} \le \frac{th_{sat}^{(c)}}{\delta} \\ 1 & ; n^{(c)} > \frac{th_{sat}^{(c)}}{\delta} \end{cases} (3.3)$$

Our goal is to develop a low overhead and efficient algorithm for the single application problem. Since the algorithm is directed by the saturation threshold, we call it iterative threshold algorithm (ITA). ITA results in the (discrete) number of threads that need to be executed by each core.

To that end, for simplicity, we first assume that $n^{(c)}$ is a continuous variable. The continuous formulation is not realistic, as threads cannot be split. However, it leads to an intuitive formulation and solution that serves as a good approximation for large numbers of cores and threads. Hence the intermediate algorithm is entitled Continuous ITA (CITA). Of course, CITA results should be discretized in order to get the ITA results.

We define a Distance Core Cluster (in short, a cluster) of distance *d*, noted as $\Omega_d$, to be the group of all cores located *d* hops from the shared cache (.i.e., $\Omega_d = \{h^{(c)} | h^{(c)} = d\}$). The number of cores in $\Omega_d$ is $|\Omega_d|$. The algorithm starts to allocate threads in cores which belong to the closest cluster (i.e., smallest *d*) in the first iteration, continues to allocate threads to the closest unallocated clusters in each iteration till the furthest cluster (i.e., biggest *d*) in the last iteration. This is because the performance of cores which are closer to the shared cache is higher than the performance of the distant cores. In each iteration, the algorithm allocates threads to a core only if its final utilization exceeds a minimum value, which is determined according to the problem parameters. We term this minimum utilization value as MU.

The need for such a minimum utilization value stems from the following reasoning: When a core is brought to operation it causes an increase in the power consumption by at least $P_{idle}$ (see equation (2.3)), which leads to a reduction of the PPM. Therefore, an appropriate matching increase in the performance is required in order to compensate for this reduction. In Appendix B, we present that the required increase of performance is achieved when the core utilization reaches the value MU, which depends on m (the number of cores which are already operating). The value of MU which justifies operating the *(m+1)*th core is calculated by equating the loss and the gain in PPM, yielding the expression

$$\text{MU} = \left(\frac{1}{P_{active} - P_{idle}}\right) \cdot \left[P_{active}\left(\frac{m(m+MU)^{\alpha}}{m^{\alpha}} - m\right) - P_{idle}\right] (3.4)$$

MU can be approximated by: $\text{MU} \approx \frac{P_{idle}}{(\alpha-1)P_{active} + P_{idle}}$ (3.5)

It can be seen in equation (3.1) that $th_{sat}^{(c)}$ is the same for all cores which belong to $\Omega_d$ cluster, Since all these cores have the same MU value, the number of threads which achieve MU is the same too, and it takes the value $\text{MU} \cdot \frac{th_{sat}^{(c)}}{\delta}$ (3.6)

```
(1) MU=Minimum Utilization.
(2) L=N
(3) d= first Distance Core Cluster d.
(4) While (L>0)
(5)   $Y = \left\lfloor \frac{\delta \cdot L}{th_{sat}^{(c)}} \right\rfloor, c \in \Omega_d$ - Number of cores in $\Omega_d$ that execute $th_{sat}^{(c)}/\delta$ threads each.
(6)   Switch $X = \min\{|\Omega_d|, Y\}$
(7)     Case  $X = |\Omega_d|$ :
(8)           if (d is last Distance Core Cluster)
(9)                 Each Core in $\Omega_d$ executes $th_{sat}^{(c)}/\delta$ threads. The remaining threads,
                    $L - |\Omega_d| \cdot th_{sat}^{(c)}/\delta$ may be allocated in any selected cores.
(10)                L=0
(11)                else
(12)                Each Core in $\Omega_d$ executes $th_{sat}^{(c)}/\delta$ threads.
(13)                $L = L - |\Omega_d| \cdot th_{sat}^{(c)}/\delta$
(14)          end if
(15)    Case  X=Y>0:
(16)          if $\left(L - \left(\frac{th_{sat}^{(c)}}{\delta}\right) \cdot X < MU \cdot \left(\frac{th_{sat}^{(c)}}{\delta}\right)\right)$
(17)                Y cores from $\Omega_d$ execute at least $th_{sat}^{(c)}/\delta$ threads each.
(18)                $L = L - Y \cdot th_{sat}^{(c)}/\delta$
(19)                else
(20)                Any allocation such as Y+1 cores from $\Omega_d$ execute at least
                    $MU \cdot (th_{sat}^{(c)}/\delta)$ and at most $th_{sat}^{(c)}/\delta$ threads each.
(21)                L=0
(22)          end if
(23)    Case  X=0:
(24)          if $\left(L \geq MU \cdot (th_{sat}^{(c)}/\delta)\right)$
(25)                One core from $\Omega_d$ executes L threads.
(26)                elseif (d is first Distance Core Cluster)
(27)                One core from $\Omega_d$ executes the L threads
(28)                else
(29)                The threads may be allocated in any selected cores of Distance Core
                    Clusters lower than d.
(30)          end if
(31)          L=0
(32)  end switch
(33)  d=next Distance Core Cluster d.
(34)end while
```

**CITA for the single application allocation problem**

In each iteration *d,* the algorithm calculates, $Y$, number of cores $(c \in \Omega_d)$ that can execute $th_{sat}^{(c)}/\delta$ threads each (Line 5). The following three cases are possible. In the first case, there are too many threads to allocate in the current cluster $(Y \geq |\Omega_d|)$, therefore, each core $(c \in \Omega_d)$ in the current cluster executes $th_{sat}^{(c)}/\delta$ threads (Lines 12-13), if it is not the last cluster, the remaining threads will be handled in the next iteration, except the last cluster case, then the remaining threads can be allocated in any selected cores (Lines 9-10). Second, some of the current cluster cores $(c \in \Omega_d)$ execute $th_{sat}^{(c)}/\delta$ threads each $(0 < Y < |\Omega_d|)$. If the number of remaining threads is more than

MU $\cdot\, th_{sat}^{(c)}/\delta$, all the threads are allocated in $Y+1$ cores of the current cluster, such as each core executes at least MU $\cdot\, th_{sat}^{(c)}/\delta$ and at most $th_{sat}^{(c)}/\delta$ and threads (Line 20). If not, the remaining threads are allocated in the next iteration (Lines 17-18). Third, there are small numbers of threads such that no core ($c \in \Omega_d$) of the current cluster can execute $th_{sat}^{(c)}/\delta$ threads ($Y=0$). If the number of threads is above MU $\cdot\, th_{sat}^{(c)}/\delta$ or it's the first iteration, the threads are allocated over a single core of the current cluster (Lines 25-27). If not, the threads may be allocated in any selected cores from clusters which are closer to the cache than the current one (Line 29).

## 3.2 The Multiple Applications Problem

The following introduces *the multiple applications thread allocation problem*:

**Multiple Applications Problem Formulation:**

**Given**: A CMP with *M* cores and a shared-cache, which execute *P* multi-threaded applications with $N_i$ $(1 \le i \le P)$ symmetric threads in each application, respectively (i.e., the threads of application *i* have a constant number of cycles between private cache misses, $\delta_i$). The hop distance between core c and the shared cache is $h^{(c)}$.

**Find**: Optimal allocation of threads to cores, $n_i^{(c)}$ the number of threads of application $i$ are executed by core *c*.

**Which:** maximizes $\text{PPM} = \left(\frac{\text{Average Performance}^{\alpha}}{\text{Consumed Power}}\right)$

**Subject to:** $\sum_{c=1}^{M} n_i^{(c)} = N_i \,, n_i^{(c)} \ge 0 \,, 1 \le c \le M, \;\; 1 \le i \le P.$

**Where:** $\alpha \ge 1$

$$\text{Average Performance} = \frac{\sum_{i=1}^{P}\sum_{c=1}^{M}\left(n_i^{(c)} \cdot \text{Thread}_{\text{i}}\text{Performance}^{(c)}\right)}{\sum_{i=1}^{P} N_i}$$

(Thread$_\text{i}$Performance$^{(c)}$ depicted on equation (3.8))

$$\text{Consumed Power} = \sum_{c:\sum_{i=1}^{P} n_i^{(c)} \neq 0} \left(\eta^{(c)} \cdot P_{active}^{(c)}\right) + \left(1-\eta^{(c)}\right) \cdot P_{idle}^{(c)}$$

The saturation threshold, thread performance and utilization for each core c in this case are:

$th_{sat}^{(c)} = T_{cache} + h^{(c)}\tau + \max_j \left\{\delta_j \middle| n_j^{(c)} \neq 0\right\}$(3.7)

$$\text{Thread}_{\text{i}}\text{Performane}^{(c)} = \begin{Bmatrix} \frac{\delta_i}{th_{sat}^{(c)}} & ;\sum_{j=1}^{P}\left(n_j^{(c)}\delta_j\right) \le th_{sat}^{(c)} \\ \frac{\delta_i}{n^{(c)}\cdot\delta} & ;\sum_{j=1}^{P}\left(n_j^{(c)}\delta_j\right) > th_{sat}^{(c)} \end{Bmatrix} \quad (3.8)$$

$$\eta^{(c)} = \begin{Bmatrix} \frac{\sum_{j=1}^{P} n_j^{(c)}\delta_j}{th_{sat}^{(c)}} & ;\sum_{j=1}^{P}\left(n_j^{(c)}\delta_j\right) \le th_{sat}^{(c)} \\ 1 & ;\sum_{j=1}^{P}\left(n_j^{(c)}\delta_j\right) > th_{sat}^{(c)} \end{Bmatrix} \quad (3.9)$$

As in the single application case, we first use the continuous thread allocation approximation. In Appendix B we calculate the minimum required utilization in order to justify operation of a core, MU, in a similar way as in the single application problem.

Unlike CITA for a single application, which is based on the fact that $th_{sat}^{(c)}$ is the same for all cores $c \in \Omega_d$, in the multiple-application problem, $th_{sat}^{(c)}$ depend both on the hop distance to the shared cache and on the application with the lowest cache miss rate allocated in core c $\left(\max_j \left\{\delta_j \middle| n_j^{(c)} \neq 0\right\}\right)$. Therefore, in each iteration, CITA for the multiple applications problem allocates threads in a single core unlike CITA for the single application which allocates threads simultaneously in all cores of cluster *d*. The solution space of the multiple application allocation problem is very large and increases exponentially with *M*, *P* and $N_i$. It can be shown that the optimization problem is not convex. CITA allocates applications according to their cache miss rate such that applications with a high miss rate are allocated to cores which are close to the cache. The algorithm allocates threads core by core, starting from the cores which are closest to the cache. Once a core is saturated, another core is assigned only if its utilization would be at least MU.

```
(1)  Core=1; Application=1 // Core & Application index
(2)  $Sort(\delta_i)$ ; $1 \le i \le P$ // Sort from the higher cache miss (i=1) rate to the lowest (i=P)
(3)  $L_i = N_i$ ; $1 \le i \le P$
(4)  $\text{load} = \sum_{i=1}^{P} N_i \delta_i$
(5)  Calculate MU
(6)  while (load>0)
(7)      $th_{sat} = T_{cache} + h^{(Core)}\tau + \delta_{Application}$ /* Calculate the saturation threshold of current
                                                          core according to the current application. */
(8)      if (Core<M)
(9)          $th_{sat,max} = T_{cache} + h^{(Core+1)}\tau + \max_{1 \le i \le P}\{\delta_i\}$
(10)     end if
(11)     if $(L_i\delta_i + \sum_{i=1}^{P} n_i^{(Core)}\delta_i \ge th_{sat})$ /* Does the current application threads achieve
                                                          saturation in the current core? */
(12)         $L_{Application} = L_{Application} - \left(th_{sat} - \frac{\sum_{i=1}^{P} n_i^{(Core)}\delta_i}{\delta_{Application}}\right)$ // Allocate threads of current
(13)         $n_{Application}^{(Core)} = th_{sat} - \frac{\sum_{i=1}^{P} n_i^{(Core)}\delta_i}{\delta_{Application}}$ /* application such that the current
                                                          core is saturated. */
(14)         if $\left(Core < M \;\&\; \frac{\sum_{i=1}^{P} L_i\delta_i}{th_{sat,max}} \ge MU\right)$ /* Does the remaining threads achieve MU in the
                                                          next core ? */
(15)             Core=Core+1 // If yes, Increment the core index.
(16)         else
(17)             $n_i^{(Core)} = n_i^{(Core)} + L_i$ ; $\forall i: L_i \ne 0$ /* If not, the current core is over saturated and
(18)             $L_i = 0$ ; $\forall i$                  executes all the remaining threads.*/
(19)         end if
(20)     else
(21)         $n_{Application}^{(Core)} = L_{Application}$ // Allocate all current application on the current core.
(22)         $L_{Application} = 0$
(23)         $Application = Application + 1$
(24)     end if
(25)     $\text{load} = \sum_{i=1}^{P} N_i \delta_i$
(26) end while
```

**CITA for multiple application allocation**

As mentioned, CITA produces a continuous $n_i^{(c)}$, which is not a final solution to our thread allocation problem, as threads cannot be split. Therefore we need to convert every vector allocation of application $i$, $n_i^{(c)}$, to a discrete vector. This process converts the CITA results to the ITA results. We propose an example of a method to convert the CITA results into a discrete allocation (i.e., ITA) by the following iterative discretization method, the method result example denoted by $m_i^{(c)}$.

$$\forall i\text{: } m_i^{(1)} = \text{round}\left(n_i^{(1)}\right),\ m_i^{(c)} = \text{round}\left(\text{round}\left(\sum_{k=1}^{c} n_i^{(k)}\right) - \sum_{k=1}^{c-1} m_i^{(k)}\right);\ 2 \le c \le M \quad (3.10)$$

The iterative algorithm above starts from the first core and in every iteration it produces the discrete value of the threads are executed by the next core, it equals to rounded value of the accumulated error between the continuous and the discrete vectors, as described in equation (3.10). This method is used for histogram specification [15]. Of course, there are more methods which can used in order to convert the CITA results into the ITA results.

## 4 Numerical Results

### 4.1 Single Application Results

Fig. 6 presents several results of CITA and ITA (i.e., discretization of CITA results) for different numbers of threads in a single application problem. The CMP in this example includes three cores with 1, 2 and 3 hop distances to the shared cache, respectively, $P_{idle}$ and $P_{active}$ values were selected to be in the range of PowerPC440 and MIPS power consumption specifications. It can be seen that when there are 6 or 11 threads to allocate (Fig. 6(a,c)), the CITA results do not conform to discrete values and the final results are derived by finding discrete allocations close to the CITA results. In both cases the discrete allocations are within 2-4% from the CITA results. The discretized results for all cases are also the optimal allocation (as computed by an exhaustive evaluation of all possible allocations). In the 9 thread case (Fig. 6(b)) the CITA, ITA and exhaustive search results are identical (and optimal). Note that in this case although the first two cores are saturated, the third core is shut down, because there are not enough threads to execute at least $\text{MU} \cdot th_{sat}^{(3)}/\delta$ by it, while the first two cores executed $th_{sat}^{(1)}/\delta$ and $th_{sat}^{(2)}/\delta$ respectively.

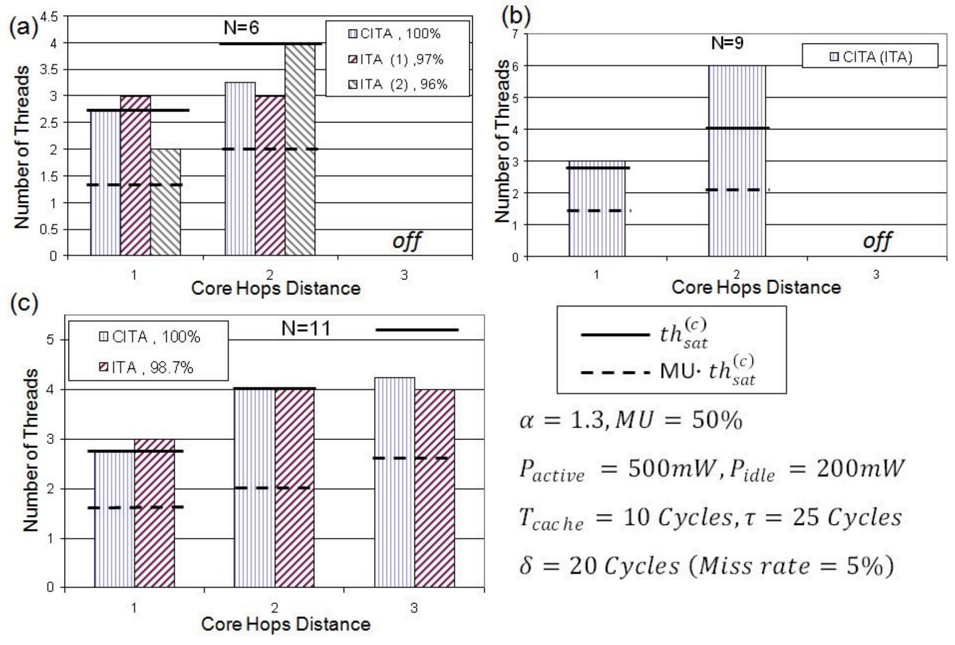
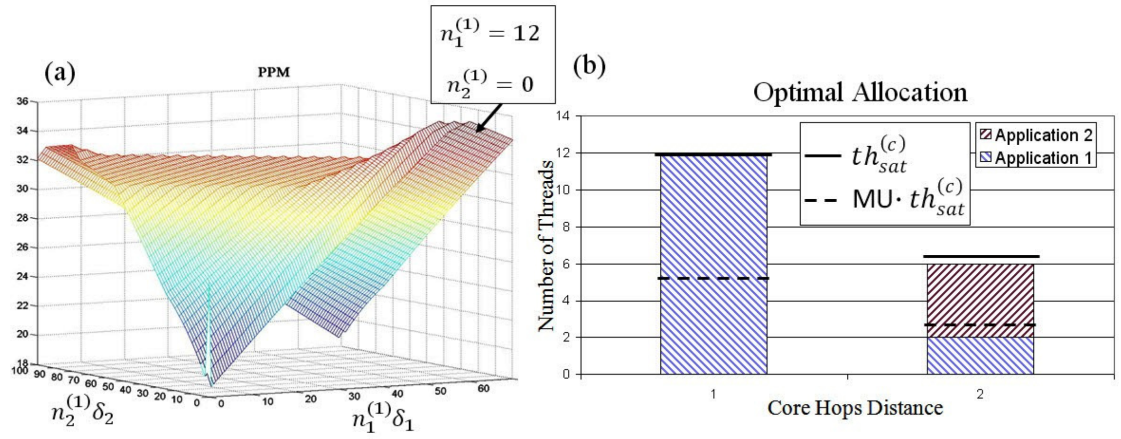


**Fig. 6 - Single Application Results Example**

### 4.2 Two Cores and Two Applications Results



$\alpha = 1.5$ , MU $= 44\%$, $P_{active} = 500mW, P_{idle} = 200mW, T_{cache} = 20$ Cycles,

$\tau = 35$ Cycles , $(N_1, \delta_1) = (14{,}5)$ , $(N_2, \delta_2) = (4{,}25)$

**Fig. 7 - 2 Cores and 2 Applications Example**

Fig. 7(a) presents the PPM value for a variable number of threads in the first core, for the case of two cores and two applications. The ITA results described in Fig. 7(b), Threads of application 1 (that has a higher cache miss rate) are allocated to the core that is closer to the cache until this core is saturated. The remaining threads are allocated to the second core as its utilization is higher than MU.

### 4.3 Multiple Applications Results

In order to evaluate our multiple-application solution, we compare CITA results and run-time with several general-purpose optimization algorithms. The optimization algorithms used are: a constrained nonlinear optimization, a pattern search algorithm and a genetic algorithm (all included in Matlab library). The optimizations algorithms were also executed for the continuous version of the problem and were not discretisized.

Table 1 presents the ratio between the results of CITA and the best result among all the optimization algorithms mentioned above, for all cases in the range of 2-8 cores and 2-8 applications. Each case was executed for 200 random instances of the problem and in each case the CMP and application parameters were selected according to the distributions included in Table 1. On average, CITA outperforms the best optimization algorithms by 5%.

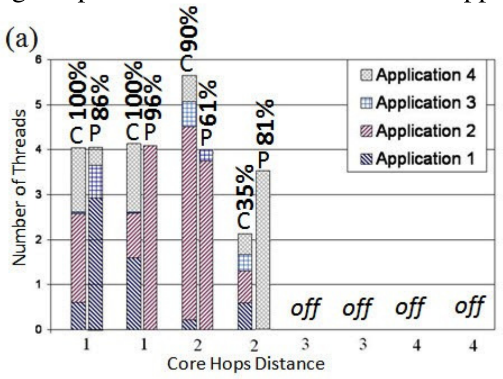
| | | Cores | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 2 | | 3 | | 4 | | 5 | | 6 | | 7 | | 8 | |
| | | Mean | Std | Mean | Std | Mean | Std | Mean | Std | Mean | Std | Mean | Std | Mean | Std |
| Applications | 2 | 99.13 | 5.47 | 103.67 | 11.46 | 109.81 | 19.68 | 112.11 | 16.40 | 118.43 | 22.77 | 126.52 | 26.98 | 131.43 | 30.04 |
| | 3 | 99.92 | 3.17 | 101.96 | 5.83 | 103.50 | 8.50 | 106.61 | 12.72 | 110.42 | 15.19 | 116.27 | 22.64 | 124.44 | 33.84 |
| | 4 | 100.06 | 1.10 | 100.26 | 2.17 | 101.64 | 6.04 | 103.43 | 8.89 | 105.13 | 10.81 | 109.77 | 17.45 | 113.13 | 29.87 |
| | 5 | 99.93 | 0.56 | 100.30 | 2.21 | 100.69 | 3.48 | 101.86 | 6.13 | 102.86 | 7.01 | 106.03 | 13.57 | 106.79 | 12.49 |
| | 6 | 99.97 | 0.01 | 100.09 | 1.12 | 100.09 | 0.85 | 100.25 | 1.44 | 101.60 | 4.47 | 102.88 | 9.15 | 103.96 | 9.26 |
| | 7 | 99.97 | 0.02 | 100.02 | 0.38 | 100.10 | 1.16 | 100.28 | 1.57 | 100.47 | 2.57 | 101.43 | 4.84 | 101.96 | 6.37 |
| | 8 | 99.97 | 0.02 | 99.98 | 0.02 | 99.97 | 0.01 | 100.05 | 0.65 | 100.24 | 1.66 | 100.82 | 3.88 | 101.00 | 4.17 |

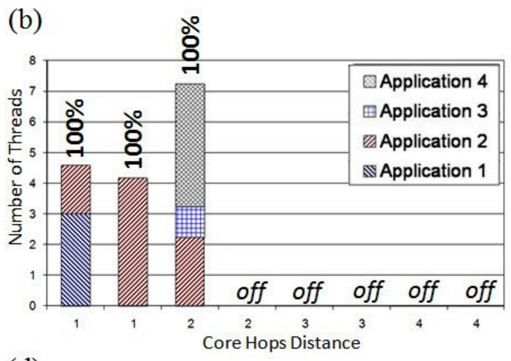
$T_{cache} \sim U(10{,}30), \tau \sim U(10{,}40), h^{(c)} \sim U(1{,}8), \delta_i \sim U(1{,}40), N_i \sim U(1{,}25), \alpha \sim U(1{,}6)$

$P_{active} = 500mW, P_{idle} = 200mW$

**Table 1 – Mean and Standard Deviation of $PPM_{CITA}/\max\{PPM_{optimization\ methods}\}$**

Fig. 8 demonstrates by an example the efficiency and the low computational overhead of CITA. For a CMP with 8 cores and 4 applications, we calculated the allocation using CITA, a constrained nonlinear optimization, a pattern search algorithm and a genetic algorithm and compared them to a naïve approach which obtains a uniform utilization of the cores. CITA achieves the highest PPM percentage improvement relative to the naïve approach, with the fastest execution time.





(c) $\alpha = 1.2, MU = 66\%$

$P_{active} = 500mW, P_{idle} = 200mW$

$T_{cache} = 26\ Cycles, \tau = 43\ Cycles$

$(N_1, \delta_1) = (3,19), (N_2, \delta_2) = (8,22)$

$(N_3, \delta_3) = (1,25), (N_4, \delta_4) = (4,33)$

(d)

| | Uniform Allocation Improvment | Execution Lines |
|---|---|---|
| CITA | 206% | 446 |
| Constrained Nonlinear Optimization | 137% | 522217 |
| Pattern Search Algorithm | 140% | 5258275 |
| Genetic Algorithm | 165% | 957164 |

**Fig. 8 – (a) Legend: C- Constrained Nonlinear Optimization Result, P- Pattern Search Result**

Fig. 8(a) presents the allocation results and cores utilizations according to the constrained nonlinear optimization and pattern search algorithm. The genetic algorithm allocation result allocates all applications to execute in the first core. Fig. 8(b) presents the allocation results and cores utilizations according to CITA, where application 1 which has the highest cache miss rate is allocated to the core closest to the shared cache and application 4 which has the lowest cache miss rate is allocated to the remote core. Fig. 8(d) presents the PPM improvement relative to the naïve approach, and the number of execution lines required for the allocation calculation in Matlab workspace. It can be seen that CITA achieves the highest improvement relative to the naïve approach with the lowest lines of execution.

## *4.4 Discretization Results*

ITA results are drived by the discretization of CITA results. We use the discretization method described by equation (3.10). Table 2 presents the ratio between ITA results and CITA results, for all cases in the range of 2-8 cores and 2-8 applications. Each case was executed for 10,000 random instances of the problem and in each case the CMP and application parameters were selected according to the distributions included in Table 2. On average, discretization of CITA results reduces the value by 5%.

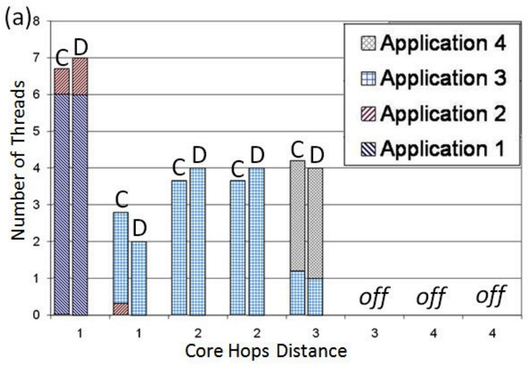
| | | Cores | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 2 | | 3 | | 4 | | 5 | | 6 | | 7 | | 8 | |
| | | Mean | Std | Mean | Std | Mean | Std | Mean | Std | Mean | Std | Mean | Std | Mean | Std |
| Applications | 2 | 96.72 | 5.81 | 95.14 | 5.86 | 94.16 | 6.04 | 93.41 | 6.36 | 92.59 | 6.79 | 92.02 | 7.05 | 91.48 | 7.31 |
| | 3 | 97.13 | 5.01 | 95.61 | 4.98 | 94.67 | 5.11 | 93.81 | 5.33 | 93.12 | 5.59 | 92.47 | 5.86 | 91.92 | 6.10 |
| | 4 | 97.41 | 4.63 | 96.00 | 4.48 | 95.06 | 4.68 | 94.39 | 4.71 | 93.82 | 4.85 | 93.29 | 5.02 | 92.73 | 5.18 |
| | 5 | 97.74 | 4.10 | 96.42 | 4.05 | 95.55 | 4.16 | 94.90 | 4.24 | 94.32 | 4.32 | 93.89 | 4.48 | 93.46 | 4.56 |
| | 6 | 97.83 | 3.81 | 96.68 | 3.82 | 95.82 | 3.90 | 95.34 | 3.84 | 94.89 | 3.95 | 94.36 | 4.05 | 93.92 | 4.20 |
| | 7 | 98.05 | 3.55 | 96.95 | 3.51 | 96.19 | 3.62 | 95.63 | 3.59 | 95.19 | 3.66 | 94.81 | 3.68 | 94.38 | 3.90 |
| | 8 | 98.12 | 3.41 | 97.06 | 3.42 | 96.41 | 3.38 | 95.92 | 3.35 | 95.52 | 3.37 | 95.06 | 3.56 | 94.72 | 3.68 |

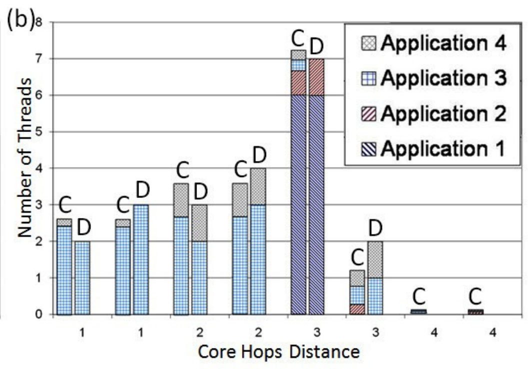
$T_{cache} \sim U(10,30), \tau \sim U(10,40), h^{(c)} \sim U(1,8), \delta_i \sim U(1,40), N_i \sim U(1,25), \alpha \sim U(1,6)$

$P_{active} = 500mW, P_{idle} = 200mW$

**Table 2 - Mean and Standard Deviation of $\mathbf{PPM_{ITA}/PPM_{CITA}}$**

Fig. 9(a) and Fig. 9(b) present an example of CITA and Constrained Nonlinear optimization algorithm result discretization respectively. Fig. 9(d) presents the CITA improvement relative to the optimization algorithms continuous results and also the ITA improvement relative to the optimization algorithms discrete results. The relative improvement doesn`t decrement by much due to the discretization and offers a realistic and efficient thread allocation.





(c)

$\alpha = 1.3, MU = 58\%$

$P_{active} = 500mW, P_{idle} = 200mW$

$T_{cache} = 20\ Cycles, \tau = 35\ Cycles$

$(N_1, \delta_1) = (6,10), (N_2, \delta_2) = (1,15)$

$(N_3, \delta_3) = (11,34), (N_4, \delta_4) = (3,37)$

(d)

| | PPM Continuous Improvement | PPM Discrete Improvement |
|---|---|---|
| Constrained Nonlinear Optimization | 123.4% | 121.5% |
| Pattern Search Algorithm | 134.5% | 130.3% |
| Genetic Algorithm | 156.4% | 150.2% |

**Fig. 9 – Example of results discretization**
**(a,b) Legend: C-Continuous Result, D-Discretization of the continuous Result**

## 5 Conclusion

Assignment of threads to cores in a CMP according to desired performance/power tradeoffs was accomplished by a computationally-efficient algorithm (ITA) which achieves close to optimal results. ITA is guided by the characteristics of the problem, such as core saturation threshold, core idle power, NoC hop delay, and the relative importance of performance versus power. The simple approach derived in this paper can be extended in several ways for implementation in practical CMP systems.

## References

[1] K. Olukotun and L. Hammond, "The future of microprocessors", Queue 3, pages 26-29, September 2005.

[2] L. Spracklen and S.G. Abraham, "Chip Multithreading: Opportunities and Challenges", High- Performance Computer Architecture, pages 248-252, Feb 2005 .

[3] L.A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets and B. Verghese, "Piranha: a scalable architecture based on single-chip multiprocessing", SIGARCH Comput. Archit. News, pages 282-293, May 2000.

[4] Z. Guz, I. Keidar, A. Kolodny and U. Weiser, "Nahalal: Memory Organization for Chip Multiprocessors", IEEEComputer Architecture Letters, vol.6(1), May 2007

[5] C. McCann, R. Vaswani, and J. Zahojan, “A dynamic processor allocation policy for multiprogrammed sharedmemory multiprocessors”, ACM Trans. Comput, May 1993.

[6] A. Fedorova, M. Seltzer, C. Small and D. Nussbaum, "Performance of multithreaded chip multiprocessors and implications for operating system design", Proc. 2005 USENIX Technical Conference, pages 395–398, 2005.

[7] S. Kim, D. Chandra and Y. Solihin, "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture", In Proc. of the Intl. Conf. on Parallel Architecture and Compilation Techniques 2004. pages 111-122, September 2004.

[8] C. Yang, J. Chen and T. Kuo, "An Approximation Algorithm for Energy-Efficient Scheduling on A Chip Multiprocessor", IEEE Computer Society, pages 468-473, 2005.

[9] Y. Ding, M. Kandemir, M.J. Irwin and P. Raghavan, "Adapting Application Mapping to Systematic Within-Die Process Variations on Chip Multiprocessors", 2009

[10] G. Chen, F. Li, S.W. Son and M. Kandemir, "Application mapping for chip Multiprocessors", ACM, pages 620-625, 2008.

[11] T. Burd and R.W. Brodersen, “Energy efficient CMOS microprocessor design”, Proc. 28th Hawaii Int’l Conf. on System Sciences, pages 288-297, January 1995.

[12] P.I. Penzes and A.J. Martin, “Energy-delay efficiency of VLSI computations”, in Proc. Great Lakes Symp.VLSI, pages 104–111, April 2002.

[13] A. Agarwal, "Performance tradeoffs in multithreaded processors", IEEE Transactions on Parallel and Distributed Systems vol.3 no.5, pages 525-539, September 1992.

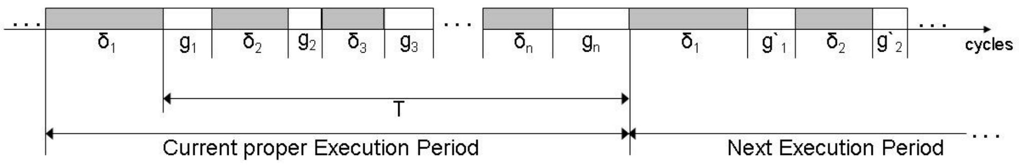
[14] L. Benini, A. Bogliolo and G. De Micheli, "A survey of design techniques for system-level dynamic power management", Kluwer Academic Publishers, pages 231-248, 2002.

[15] C.R. Gonzalez and R.E. Woods, "Digital image processing", pages 128-138, third edition, 2008.

[16] C. Bienia, S. Kumar, J.P. Singh and K. Li, "The PARSEC benchmark suite: characterization and architectural implications", In Proceedings of the 17th international Conference on Parallel Architectures and Compilation Techniques, pages 72-81, October 2008.

## Appendix A – Proof of execution order in steady state

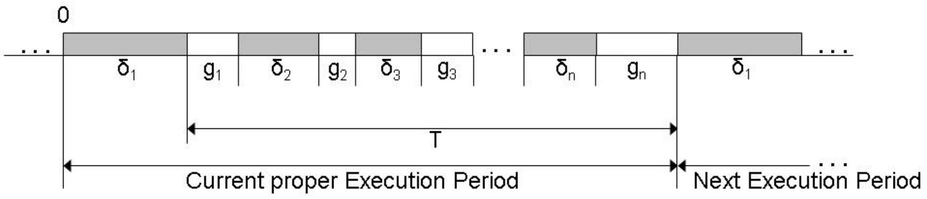
Assume that threads *1,..,n* are executed by a core, and $\delta_1 > \delta_i \;\; \forall i = 2, \dots, n \,; T > \sum_{i=2}^{n} \delta_i$. Where $\delta_j$ is the number of core cycles between private cache misses for thread *j*. (i.e. $\delta_j = 1/(r_m \cdot \mathrm{m.r_j})$, where $r_m$ is the ratio of memory access instructions and $\mathrm{m.r_j}$ is the cache miss rate for thread *j*).
We define an Execution Period as the time from start of thread *1* execution till next start of thread *1* execution. An Execution Period is proper if all threads *1,..,n* are executed during this Execution Period.
We define $g_i$ as the idle time between the executions of thread *i* and thread *i+1* at the current Execution Period. Similarly, we define $g_i'$ as the idle time between thread *i* and thread *i+1* at the next Execution Period.



We assume that each thread *j* executes useful instructions for $\delta_j$ cycles until it suffers a private cache miss. After *T* cycles the cache answer returns. When a thread receives the cache answer it will be scheduled immediately after all threads that were scheduled before it.

**Lemma 1:**
Assume that threads *1,..,n* are executed by a core, and $\delta_1 > \delta_i \;\; \forall i = 2, \dots, n \,; T > \sum_{i=2}^{n} \delta_i$.
For any given proper Execution Period, thread *1* is executed again after $T + \delta_1$ cycles and the rest of threads are executed again after $T + \delta_1$ cycles or less. Consequently, the next Execution Period is proper as well.

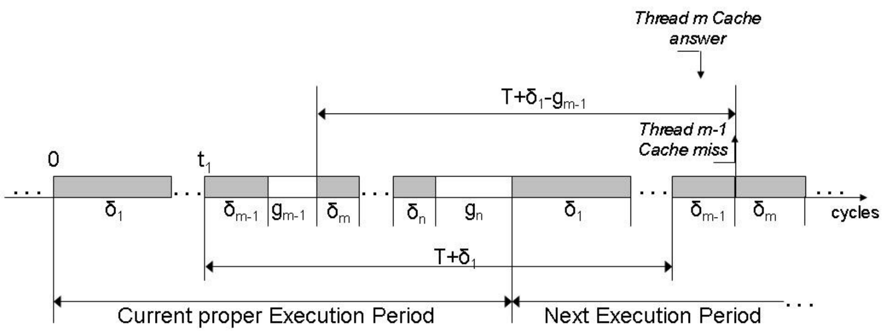
**Proof:**
Given $T > \sum_{i=2}^{n} \delta_i$ and that the current Execution Period is proper then from definition, thread *1* is executed after $T + \delta_1$ cycles.



The proof for threads *2,…,n* is by induction.

The base of the induction is thread *2*. If thread *2* is executed immediately after it receives its cache answer, then thread *2* will be executed again after $T + \delta_2$ cycles (which is smaller than $T + \delta_1$). Otherwise, thread *2* waits until thread *1* next execution suffers a cache miss that implies that thread *2* is executed again after
$(2\delta_1 + T) - (\delta_1 + g_1) = \delta_1 + T - g_1 \le \delta_1 + T.$

In the induction step we need to prove that if the lemma holds for *i=m-1* it also holds for *i=m*. Similarly to the base case, if thread m is executed immediately after it receives its cache answer, then thread *m* is executed after $T + \delta_m$ cycles ($< T + \delta_1$). Otherwise, thread *m* waits for execution until thread *m-1* next execution suffers a cache miss, therefore thread *m* will be executed again after
$(t_1 + \delta_1 + T + \delta_{m-1}) - (t_1 + \delta_{m-1} + g_{m-1}) = \delta_1 + T - g_{m-1} \le T + \delta_1$



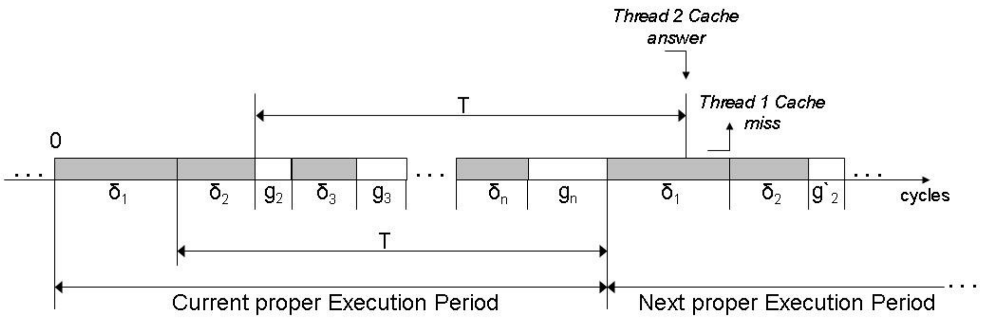
Since thread *1* is executed again after $T + \delta_1$ cycles and each other thread is executed again after $T + \delta_1$ cycles or less, if the current Execution Period is proper then the next Execution Period is proper too.

**Lemma 2:**

Assume that threads *1,..,*$n$ are executed by a core, and $\delta_1 > \delta_i \ \forall i = 2, \dots, n\,; T > \sum_{i=2}^{n} \delta_i$, then, for any $a \le n-1$, if $\forall i{:}\, 1 \le i \le a \le n-1 \ \ g_i = 0$ at the current proper Execution Period then at the next Execution Period $\forall i{:}\, 1 \le i \le a \le n-1 \ \ g_i'$=0.

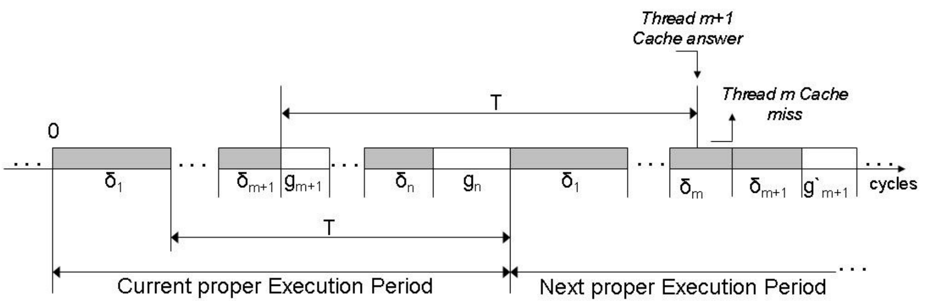
**Proof:**

By induction on *a*. The base for the induction is *a*=1, $g_1$=0.



We show that the cache answer of the second thread at the next Execution Period arrives after the start of the next execution of the first thread but before the end of its execution. Therefore the execution of the second thread cannot precede the execution of the first thread and will start immediately after it ends. Subtracting the time of the cache miss of thread *1* at the next Execution Period from the time of cache answer for thread *2* at the next Execution Period results in:
$(\delta_2 + \delta_1 + T) - (\delta_1 + T + \delta_1) = \delta_2 - \delta_1 < 0.$
Consequently, at the next Execution Period, thread *2* cache answer arrives before thread *1* suffers a cache miss. Therefore, Thread *2* is executed immediately after thread *1* cache miss, therefore $g_i'$=0.

In the induction step we need to prove that if the lemma holds for *a=m-1* it also holds for *a=m*.
If $g_i'$=0 for $1 \le i \le m \le n-1, g_1 = \dots = g_m$=0, by the induction hypothesis we can conclude that $g_1' = \dots = g_{m-1}'$=0.



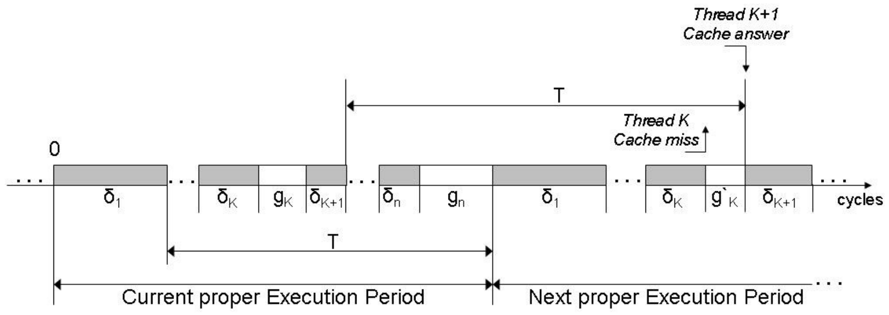
Subtracting the time of the cache miss of thread $m$ at the next Execution Period from the time of cache answer for thread *m+1* at the next Execution Period results in:

$$\left(\sum_{k=1}^{m+1}\delta_k + T\right) - \left(\delta_1 + T + \sum_{k=1}^{m}\delta_k\right) = \delta_{m+1} - \delta_1 < 0$$

Consequently, at the next Execution Period, thread *m+1* cache answer arrives before thread *m* cache miss. Therefore, thread *m+1* is executed immediately after thread m cache miss, therefore we get: $g'_m$=0.

**Lemma 3:**
Assume that threads *1,..,n* is executed by a core, and $\delta_1 > \delta_i\ \ \forall i = 2, \dots, n\,; T > \sum_{i=2}^{n}\delta_i$.
Assume that at the current proper Execution Period $\exists i$ such that $1 \le i \le n-1 : g_i \ne 0$.
Define $K \triangleq \min_{1\le i\le n-1}\{i | g_i \ne 0\}$. Then, $g'_K = g_K - \min\{g_K, \delta_1 - \delta_{K+1}\}$. In other words, $g_K$ the current idle period between thread $K$ to thread *K+1*, will be decreased at the next Execution Period by $\min\{g_K, \delta_1 - \delta_{K+1}\}$.

**Proof:**
From Lemma 1 it is clear that $g'_i = 0 : 1 \le i < K$. We turn to calculate $g'_K$.



By subtracting the time of the cache answer for thread *K+1* at the next Execution Period from the time of the cache miss of thread *K* at the next Execution Period, we get:

$$g'_K = \left(\sum_{i=1}^{K}\delta_i + g_K + \delta_{K+1} + T\right) - \left(\delta_1 + T + \sum_{i=1}^{K}\delta_i\right) = \max\left\{g_K + \underbrace{(\delta_{K+1} - \delta_1)}_{<0}, 0\right\}$$

If at the next Execution Period thread *K+1* cache answer arrives before thread *K* cache miss $g'_K$= 0. Therefore, $g'_K$ decrease at the next Execution Period by $\min\{g_K, \delta_1 - \delta_{K+1}\}$.

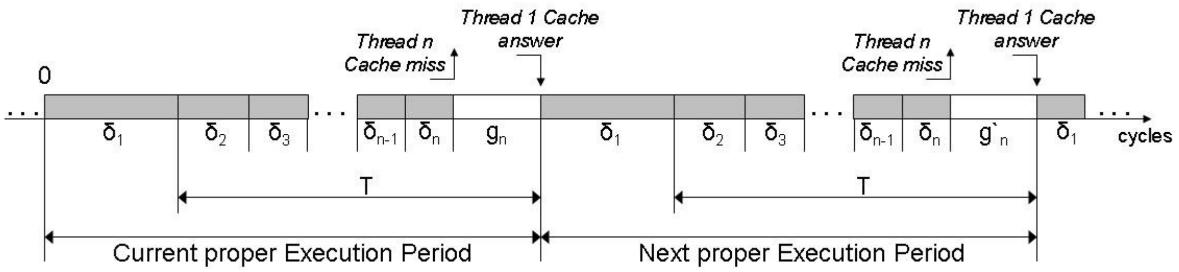
**Lemma 4:**
Assume that threads *1,..,n* are executed by a core, and $\delta_1 > \delta_i\ \ \forall i = 2, \dots, n\,; T > \sum_{i=2}^{n}\delta_i$. If at the current Execution Period $\exists i$ such that $1 \le i \le n-1 : g_i \ne 0$, and define $K \triangleq \min_{1\le i\le n-1}\{i | g_i \ne 0\}$, then $g_K$, the idle period between the executions of thread $K$ and thread *K+1* becomes zero after a finite number of Execution Periods.

**Proof:**
According to Lemma 3, $g_K$ decreases by $\min\{g_K, \delta_1 - \delta_{K+1}\}$. Therefore, $g_K$ decreases to zero after $\left\lceil \frac{g_K}{\delta_1 - \delta_{K+1}} \right\rceil$ Execution Periods.

**Theorem 1:**
Assume that threads *1,..,n* are executed by a core, and $\delta_1 > \delta_i \ \forall i = 2, \dots, n\,; T > \sum_{i=2}^{n} \delta_i$.
Given a proper Execution Period, then after a finite number of Execution Periods, for all $i$, $1 \le i \le n-1\,, g_i = 0$, and each thread is executed every $T + \delta_1$ cycles.

**Proof:**
Following lemmas 2, 3, and 4.

Given a proper Execution Period, then after a finite number of Execution Periods, the Execution Period looks the following:



$$g_n = (\delta_1 + T) - \left(\sum_{i=1}^{n} \delta_i\right) = T - \sum_{i=2}^{n} \delta_i > 0$$

By subtracting the time of the cache answer of thread *1* at the next Execution Period from the time of cache miss for thread n at the next Execution Period, we get:

$$g'_n = 2(\delta_1 + T) - \left(\delta_1 + T + \sum_{i=1}^{n} \delta_i\right) = T - \sum_{i=2}^{n} \delta_i > 0$$

Therefore, $g_i = g'_i$ for $1 \le i \le n$, and it is a steady state.
In steady state thread *1* is executed every $T + \delta_1$, and threads *j* in the range *2…n* is executed every: $\left(\delta_1 + T + \sum_{i=1}^{j-1} \delta_i\right) - \left(\sum_{i=1}^{j-1} \delta_i\right) = T + \delta_1$, also.

## Appendix B – Minimum Utilization (MU) Calculation

The MU calculation is conducted by the following methodology. When a core is brought into operation it causes a minimal increase of $P_{idle}$ in the power consumption (see equation (2.3)) that results in a reduction of the PPM. Therefore, in order to justify this new core operation an appropriate minimal increase in the performance metric is required.
In order to compute MU, we compare the PPM value of two cases. The two cases are either $m$ over-saturated cores (where the over saturation is MU divided by $m$) or $m$ saturated cores in exactly the saturation threshold and the *(m+1)*th core utilization equals MU.
This results in the following equation.

$$\underbrace{\frac{\left(\frac{m}{\text{Amount of Threads}}\right)^{\alpha}}{m\cdot P_{active}}}_{\substack{\text{PPM of the first case:}\\ \text{threads are executed by}\\ \text{m saturated cores}}} = \underbrace{\frac{\left(\frac{m+\text{MU}}{\text{Amount of Threads}}\right)^{\alpha}}{m\cdot P_{active}+(P_{active}-P_{idle})\cdot MU+P_{idle}}}_{\substack{\text{PPM of the second case:}\\ \text{m cores in exacty the threshold saturation}\\ \text{and the (m+1)th core utilization equals MU}}} \quad (4.1)$$

$$\Rightarrow \text{MU} = \left(\frac{1}{P_{active}-P_{idle}}\right)\cdot\left[P_{active}\left(\frac{m(m+MU)^{\alpha}}{m^{\alpha}}-m\right)-P_{idle}\right] \quad (4.2)$$

Using Taylor series approximation, we get: $(m+\text{MU})^{\alpha} \approx m^{\alpha}+\alpha m^{\alpha-1}\text{MU}+O(\text{MU}^2)$

and finally: $\text{MU} \approx \frac{P_{idle}}{(\alpha-1)P_{active}+P_{idle}}$ (4.3)

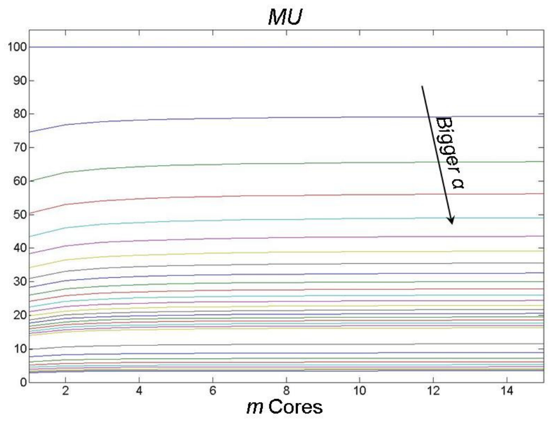


**Fig. 10 –MU for different α values (1-11) vs. m**

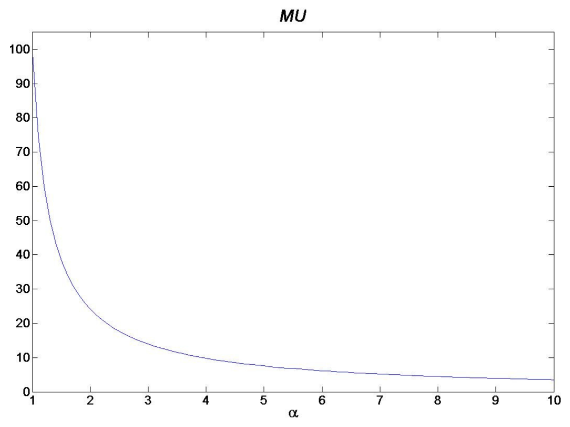


**Fig. 11 – MU versus α**

Fig. 10 presents the results of solving equation (4.2) for different values of *m* (1-15) and $\alpha$ (1-11). The MU values changes by no more than 5% as m increases. Therefore, the approximated MU of equation (4.3) is sufficient and offers a fast calculation alternative for decreasing the allocation algorithm overhead. As $\alpha$ increases, the weight of the performance metric increases that in turn decreases the value of MU (see equation (4.3)). This means that the algorithm brings cores into operation at a lower utilization and therefore increases the performance at the expense of increasing the power (Fig. 11)

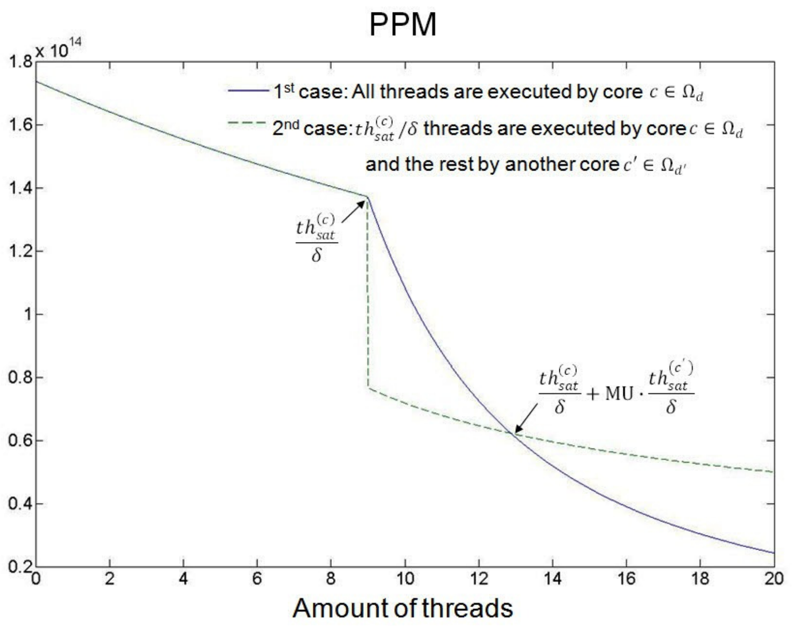


**Fig. 12 - PPM for 2 options using two cores**

Fig. 12 presents the PPM of the two cases in a CMP with two cores (c and c`) and symmetric threads. The steep drop in the PPM value in $th_{sat}^{(c\prime)}/\delta$ is due to the increase of power by $P_{idle}$ (i.e. turning on the second core) and also due to the performance differences in the case of different cores (see also Fig. 4).

When the second core utilization equals to MU, the PPM values of the two cases are equals. Therefore, the second core is brought to operation only if its utilization is at least MU.