Performance and Power Aware CMP Thread Allocation Modeling

[1]Yaniv Ben-Itzhak, [2]Israel Cidon, [2]Avinoam Kolodny

Electrical Engineering Department, Technion, Haifa, Israel
[1]yanivbi@tx.technion.ac.il [2]{cidon, kolodny}@ee.technion.ac.il

Abstract. We address the problem of performance and power-efficient thread allocation in a CMP. To that end, based on analytical model, we introduce a parameterized performance/power metric that can be adjusted according to a preferred tradeoff between performance and power. 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 ITA with constrained nonlinear optimization, pattern search algorithm and genetic algorithm. ITA outperforms the best of these methods by 9%, while consuming on average 0.01% and at most 2.5% of the computational effort.

Keywords: thread allocation algorithm, performance power metric, Chip Multi-Processor, CMP, coarse grain multi-threading, many core.

1 Introduction

The inherent scalability limitations of a single processor design combined with the advancement of VLSI technology drove the introduction of chip multi-processors (CMP). CMPs promise higher power efficiency, better performance and lower design complexity. However, CMP architectures introduce new challenges associated with thread level parallelism ([1],[2]). Today's CMPs carry parallelism beyond the traditional super-computing markets to notebooks, PDAs and other mobile devices running a mix of applications and involving complex energy saving requirements.

In this paper, we address the 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 are limited to performance [5]-[7] or power [8].

Our system includes a number of cores with a shared cache interconnected by a network on chip (NoC) [2] (e.g. Piranha [3] and Nahalal [4]). The CMP executes multiple multi-threaded applications and its cores perform coarse-grain multithreading. For reducing power consumption, unused cores are shut down. Varying conditions and battery levels may dynamically change the preferred balance between performance and power and dictate changing targets for the threads allocation solution.

To maximize the performance threads should be spread over many cores, which in turn increases power consumption. On the other hand, running all threads in a single core minimizes the power (since other cores are shut-down), but greatly impairs the performance. Our algorithm for thread allocation utilizes the CMP resources in a way that maximizes a performance/power metric that can be adjusted according to the preferred balance between performance and power.

Fig. 1 presents a 9-cores CMP example (each core includes a private first-level cache), connected by a NoC to a shared second-level cache. The CMP 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 at each core at a given time. We assume that such a thread runs until it incurs a miss in its private cache and then waits for the shared cache response. At that time, the core may perform context switching to another 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 thread performance. The response time is the sum of the shared cache access time and the NoC latency, which depends on the 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.

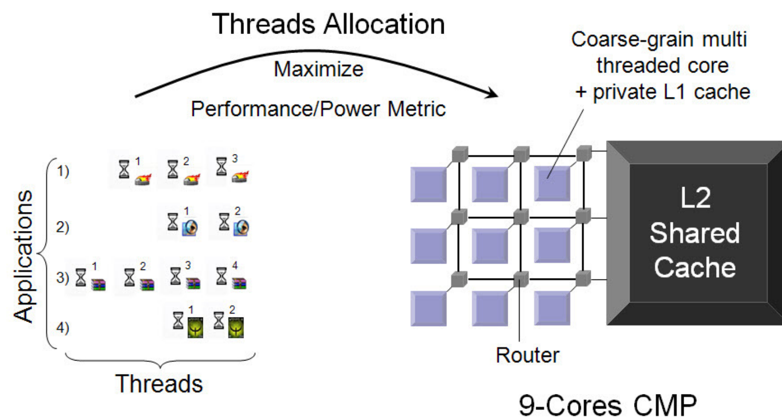


Fig. 1. Example of the thread allocation problem

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. Based on these models, a parameterized performance/power metric is defined, which can be adjusted 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 and is extended for the case of multiple applications. ITA achieves better and faster results than standard optimization algorithms.

Previous works have addressed somewhat related problems: [9] proposed thread allocation that maximizes a performance/power metric while considering process variations in the cores. However, their model is restricted to a single thread in each core while our work deals with multi-threaded cores. Furthermore, unlike [9], our work deals with shared memory architecture and considers the cache miss rate. [10] introduced 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. Section 2 presents the performance and power models and the performance/power metric. Section 3 introduces the *single application problem* and the iterative algorithm for thread allocation and extends the problem into the *multiple applications problem*. Section 4 presents 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

[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 first-level private cache and assume that all threads are independent of each other. We define δ_j as the average number of core cycles between private cache misses for thread j, i.e. $\delta_j = 1/(r_{m,j} \cdot \mathrm{m.r_j})$, where $r_{m,j}$ is the ratio of memory access instructions out of the total instruction mix and $\mathrm{m.r_j}$ is the cache miss rate for thread j. Also, we define $T^{(c)}$ as the cycles of core *c* required to satisfy such a request from the shared cache. For simplicity, at this point, we assume that $T^{(c)}$ is fixed for all threads. On average, thread j executes instructions for δ_j cycles until it incurs 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 simplicity, we assume that context switches happen only at private cache misses and ignore the thread-switching cycles and their associated power.

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, 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.

Coarse-grain multi-threaded cores utilize cache access time by running other threads. Fig. 2(b) shows 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 the other (*M-1*) threads didn't finish their execution. This case is denoted as *saturation* of the core.

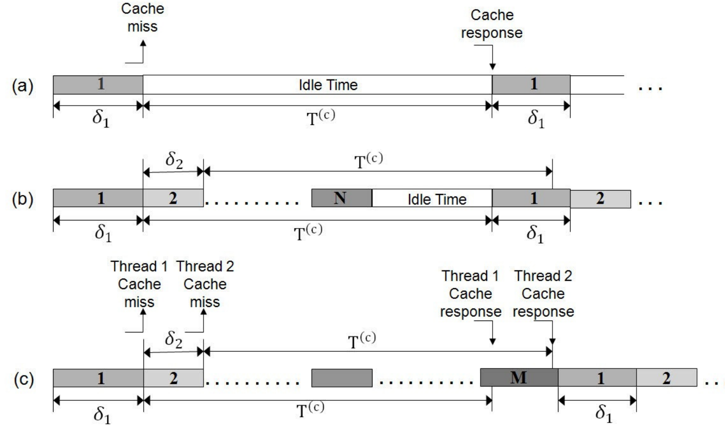


Fig. 2. A single threaded core (a) vs. multi threaded core (b,c). Saturation of the core(c).

[13] assumes that all the threads are symmetric (i.e., the threads have the same constant number of cycles between private cache misses). We extended the model for asymmetric threads. 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 running and waits for its execution until the core becomes available. Assuming that thread 1 has the largest value δ_1 among all threads, we prove in [17] 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.

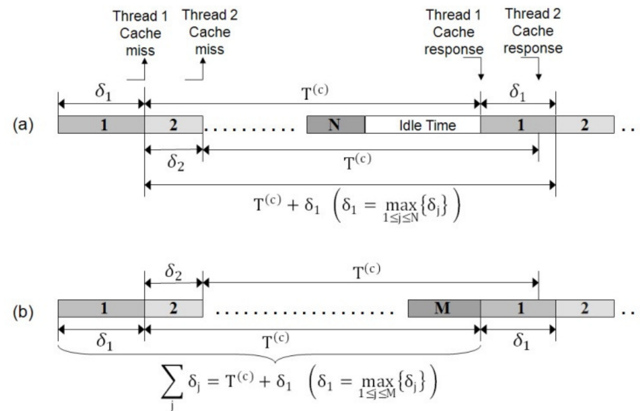


Fig. 3. (a) Steady state and the period time, (b) *saturation threshold* case

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

We define the performance of thread *j* in core *c* as the percentage of time the thread is executed and the utilization of core *c*, $\eta^{(c)}$, as the busy time percentage of the core. Therefore,

$$\mathrm{Thread_j Performane}^{(c)} = \begin{cases} \frac{\delta_j}{th_{sat}^{(c)}} & ;\sum_k \delta_k \le th_{sat}^{(c)} \\ \frac{\delta_j}{\sum_k \delta_k} & ;\sum_k \delta_k > th_{sat}^{(c)} \end{cases} \quad (1)$$

$$\eta^{(c)} = \begin{cases} \frac{\sum_k \delta_k}{th_{sat}^{(c)}} & ;\sum_k \delta_k \le th_{sat}^{(c)} \\ 1 & ;\sum_k \delta_k > th_{sat}^{(c)} \end{cases} \quad (2)$$

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

$$\mathrm{CorePower}^{(c)} = \begin{cases} \eta^{(c)} P_{active}^{(c)} + \left(1-\eta^{(c)}\right) P_{idle}^{(c)} & ;0 < \eta^{(c)} \le 1 \\ 0 & ;\ \eta^{(c)} = 0 \end{cases} \quad (3)$$

P_{active} is the power consumption of a fully utilized core. 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 so its power consumption becomes zero. Therefore, the thread allocation algorithm should shut down as many cores as possible and 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. 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 τ, we get:

$$T^{(c)} = T_{cache} + h^{(c)}\tau \Rightarrow th_{sat}^{(c)} \triangleq T^{(c)} + \Delta^{(c)} = T_{cache} + h^{(c)}\tau + \Delta^{(c)} \quad (4)$$

Where, $h^{(c)}$ is the number of NoC hops from core *c* to the shared cache.

Our definition of the tradeoff between performance and power follows definitions used in logic circuit design. If E is the energy and t is the delay, [11] introduces the $E \cdot t$ and $E \cdot t^2$ metrics, extended by [12] to the metric $E \cdot t^{\alpha}$, where α becomes larger as the performance becomes more important.

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

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 performance/power metric is maximized. It can be easily shown that if $\beta > \alpha$, the optimal allocation is to allocate all threads in a single core. Therefore, similar to [12], we use $\beta = 1$ and $\alpha \geq 1$. Consequently, $\text{PPM} \triangleq \left(\frac{\text{Average Performance}^{\alpha}}{\text{Consumed Power}}\right)$.PPM stands for Performance Power Metric.

3 Problems Statements and Allocation Algorithms

3.1 The Single Application Problem

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. α – the relative importance of performance versus power consumption (as bigger as performance is more important)

Find: Optimal thread allocation, $n^{(c)}$ the number of threads are executed by core c.

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

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

Where: $\alpha \geq 1$

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

(Thread $\text{Performance}^{(c)}$ depicted by equation (6))

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

($\eta^{(c)}$ depicted by equation (7))

With the assumptions above, the *saturation threshold* of core c, $th_{sat}^{(c)}$, and the performance of each thread j running in core c can be written as:

$$th_{sat}^{(c)} = T_{cache} + h^{(c)}\tau + \delta \tag{5}$$

$$\text{Thread Performane}^{(c)} = \begin{cases} \frac{\delta}{th_{sat}^{(c)}} & ; n^{(c)} \leq \frac{th_{sat}^{(c)}}{\delta} \\ \frac{\delta}{n^{(c)}\delta} & ; n^{(c)} > \frac{th_{sat}^{(c)}}{\delta} \end{cases} \tag{6}$$

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

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

When $n^{(c)} \geq \frac{th_{sat}^{(c)}}{\delta}$, the core is saturated and achieves its maximum total performance. The core utilization $\eta^{(c)}$ is calculated by:

$$\eta^{(c)} = \begin{cases} \frac{n^{(c)}\delta}{th_{sat}^{(c)}} & ; n^{(c)} \leq \frac{th_{sat}^{(c)}}{\delta} \\ 1 & ; n^{(c)} > \frac{th_{sat}^{(c)}}{\delta} \end{cases} \quad (7)$$

In the following we develop an efficient algorithm for the *single application problem* termed Iterative Threshold Algorithm (ITA). ITA computes the (discrete) number of threads that are executed by each core.

First, we assume that $n^{(c)}$ is a continuous variable. While this formulation is not realistic, as threads cannot be split, it leads to a good approximation for a large numbers of threads. This continuous version is entitled Continuous ITA (CITA). 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 Ω_d, to be the group of all cores located *d* hops from the shared cache (.i.e., $\Omega_d = \{c \mid h^{(c)} = d\}$). The number of cores in Ω_d is $|\Omega_d|$. The algorithm starts to allocate threads in cores that belong to the closest cluster (i.e., smallest *d*) and continues to allocate threads to the closest unallocated clusters till the last iteration. In each iteration, the algorithm allocates threads to a core only if its final utilization exceeds a minimum utilization value termed MU.

The need for such a minimum utilization value stems from the following reason: When a core is brought into operation it causes a minimal increase of P_{idle} in the power consumption (see equation (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 or *m* saturated cores in exactly the *saturation threshold* and the *(m+1)*th core utilization equals MU.

This results in the following equation:

$$\underbrace{\frac{\left(\frac{m}{\text{Amount of Threads}}\right)^{\alpha}}{m \cdot P_{active}}}_{\text{PPM of the first case}} = \underbrace{\frac{\left(\frac{m+\text{MU}}{\text{Amount of Threads}}\right)^{\alpha}}{m \cdot P_{active} + (P_{active} - P_{idle}) \cdot MU + P_{idle}}}_{\text{PPM of the second case}} \quad (8)$$

$$\Rightarrow \text{MU} = \left(\frac{1}{P_{active} - P_{idle}}\right) \cdot \left[P_{active}\left(\frac{m(m+MU)^{\alpha}}{m^{\alpha}} - m\right) - P_{idle}\right] \quad (9)$$

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:

$$\text{MU} \approx \frac{P_{idle}}{(\alpha - 1)P_{active} + P_{idle}} \quad (10)$$

Fig. 4 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 for $th_{sat}^{(c)}/\delta$ threads in the second

case is due to the increase of power by P_{idle} (i.e. turning on the second core). 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.

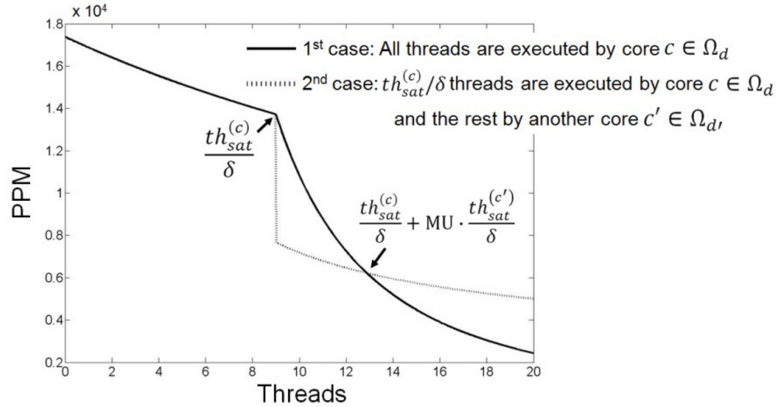


Fig. 4. PPM for the 2 cases using two cores

Generally, MU depends on *m*, number of already operating cores, (equation (9)) however MU value changes by no more than 8% as *m* increases. Therefore, the approximated MU of equation (10) is sufficient and offers a fast calculation alternative for decreasing the allocation algorithm overhead. As α increases, the weight of the performance metric increases that in turn decreases the value of MU. 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.

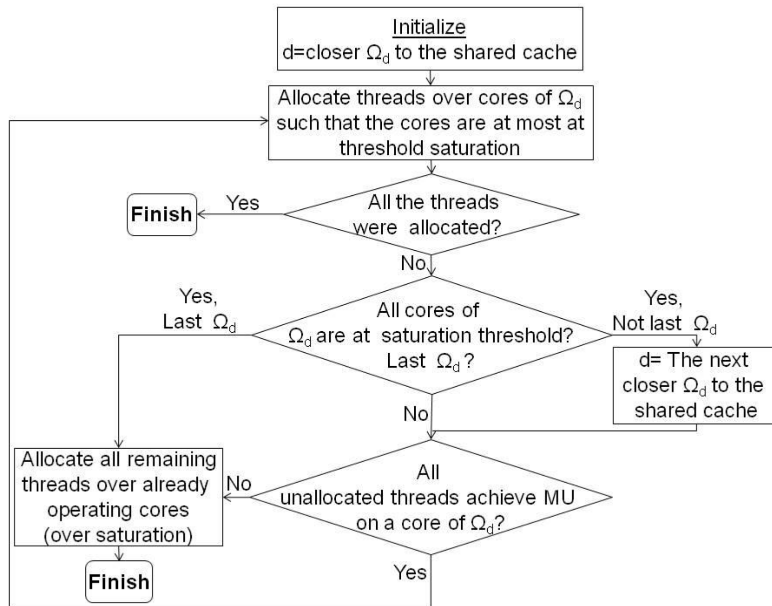


Fig. 5. Concise flow chart of CITA for *single application problem*

Fig. 5 depicts a flow chart of CITA for the *single application problem*. The concept of CITA is to allocate threads cluster by cluster, starting from the cluster which is closest to the cache. On each cluster CITA allocates threads core by core until at most threshold saturation of the cores. Once a core is saturated, another core is assigned only if its utilization would be at least MU. If not, the remaining threads may be allocated in any of the operating cores .A detailed pseudo code is presented in [17].

3.2 The Multiple Applications Problem

Given: A CMP with *M* cores and a shared-cache, which execute *P* multi-threaded applications with N_i $(1 \leq i \leq P)$ symmetric threads in each application, respectively (i.e., the threads of application i have a constant number of cycles between private cache misses, δ_i). The hop distance between core *c* and the shared cache is $h^{(c)}$. α – the relative importance of performance versus power consumption (as bigger as performance is more important)

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: $\sum_{c=1}^{M} n_i^{(c)} = N_i \,, n_i^{(c)} \geq 0 \,;\; 1 \leq c \leq M, \;\; 1 \leq i \leq P.$

Where: $\alpha \geq 1$

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

($\text{Thread}_{\text{i}}\text{Performance}^{(\text{c})}$ depicted by equation (12))

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

($\eta^{(c)}$ depicted by equation (13))

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

$$th_{sat}^{(c)} = T_{cache} + h^{(c)}\tau + \max_j\left\{\delta_j \middle| n_j^{(c)} \neq 0\right\} \tag{11}$$

$$\text{Thread}_{\text{i}}\text{Performane}^{(\text{c})} = \begin{Bmatrix} \frac{\delta_i}{th_{sat}^{(c)}} & ; \sum_{j=1}^{P}\left(n_j^{(c)}\delta_j\right) \leq th_{sat}^{(c)} \\ \frac{\delta_i}{n^{(c)} \cdot \delta} & ; \sum_{j=1}^{P}\left(n_j^{(c)}\delta_j\right) > th_{sat}^{(c)} \end{Bmatrix} \tag{12}$$

$$\eta^{(c)} = \begin{Bmatrix} \frac{\sum_{j=1}^{P} n_j^{(c)}\delta_j}{th_{sat}^{(c)}} & ; \sum_{j=1}^{P}\left(n_j^{(c)}\delta_j\right) \leq th_{sat}^{(c)} \\ 1 & ; \sum_{j=1}^{P}\left(n_j^{(c)}\delta_j\right) > th_{sat}^{(c)} \end{Bmatrix} \tag{13}$$

As in the single application case, we first use the continuous thread allocation approximation. We calculate the minimum utilization required in order to justify operation of a core, MU, in a similar way as in the *single application problem* (section 3.1).

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* $\left(\max_j\left\{\delta_j \middle| n_j^{(c)} \neq 0\right\}\right)$ (equation (11)). Therefore, unlike the single application case, in each iteration, CITA for the *multiple applications problem* allocates threads in a single core. The solution space of the *multiple application problem* is very large and increases exponentially with M, P and N_i. It can be shown that the optimization problem is not convex.

Fig. 6 depicts a flow chart of CITA for the *multiple applications problem*. CITA concept is to allocate applications according to their cache miss rate such that applications with a high miss rate are allocated to cores which are close to the shared 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. A detailed pseudo code is presented in [17].

The approach of CITA to allocate applications with higher miss rate closer to the shared cache is in order to maximize the average performance. The higher the miss rate of the threads, the bigger is the benefit in performance caused by allocating them closer to the shared cache (see equations (11) & (12)). Moreover, $th_{sat}^{(c)}$ is affected by the lowest miss rate among all threads which are executed by the core *c* (see equation (11)). CITA minimizes its value and thus maximizes performance by minimizing the differences of the miss rates of threads which are executed by each core.

Also, although our model doesn`t consider the power consumption of the NoC, this approach minimizes it also. Threads with higher miss rate cause higher load on the NoC. Therefore, reducing the number of NoC hops between those threads to the shared cache results in minimum of the NoC power consumption.

3.3 Discretization of CITA Result

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 allocation vector 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 $m_i^{(c)}$.

$$\begin{gathered}\forall i:\ m_i^{(1)} = \mathrm{round}\left(n_i^{(1)}\right) \\ m_i^{(c)} = \mathrm{round}\left(\mathrm{round}\left(\sum_{k=1}^{c} n_i^{(k)}\right) - \sum_{k=1}^{c-1} m_i^{(k)}\right);\, 2 \le c \le M\end{gathered} \tag{14}$$

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 (14) . This method is used for histogram specification [15]. Of course, there are other possible methods to convert the CITA results into the ITA results.

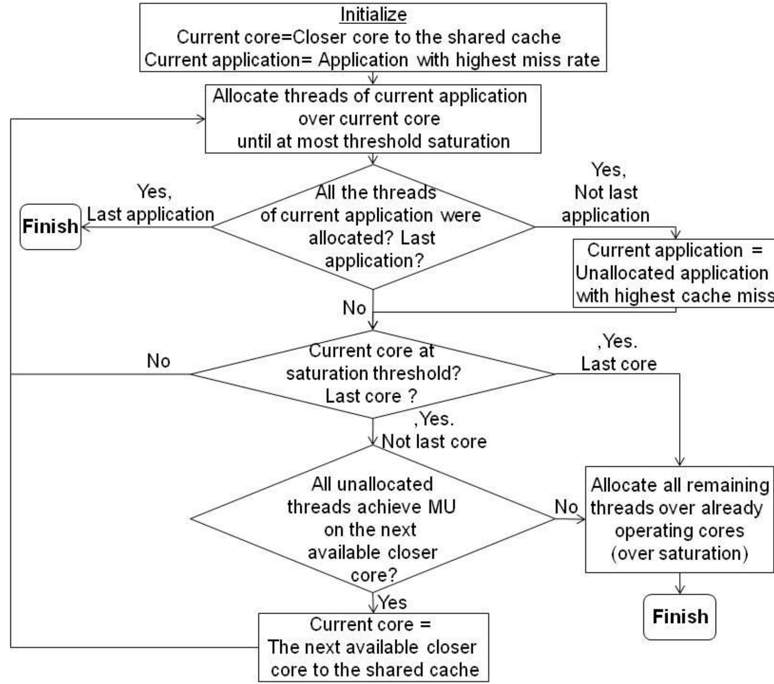


Fig. 6. Concise flow chart of CITA for *multiple applications problem*

4 Numerical Results

4.1 Single Application Results

Fig. 7 presents several results of CITA and ITA 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. 7(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 (computed over all possible allocations). In the 9 threads case (Fig. 7(b)) the CITA, ITA and optimal results are identical. In this case although the first two cores are saturated, the third core is shut down since there are not enough threads to execute at least $\mathrm{MU} \cdot th_{sat}^{(3)}/\delta$ threads by it, while the first two cores execute $th_{sat}^{(1)}/\delta$ and $th_{sat}^{(2)}/\delta$ threads respectively.

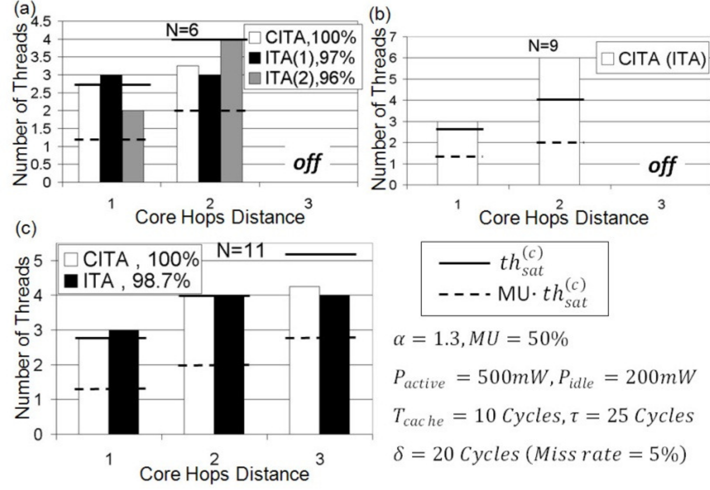
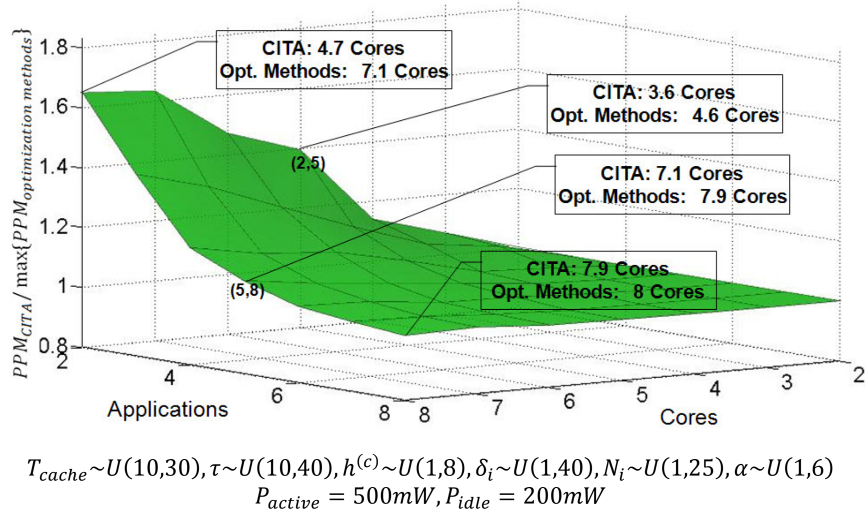


Fig. 7. Single application results example

4.2 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 from Matlab library). The optimizations algorithms were executed for the continuous version of the problem and were not discretisized.



$T_{cache}\sim U(10{,}30), \tau\sim U(10{,}40), h^{(c)}\sim U(1{,}8), \delta_i\sim U(1{,}40), N_i\sim U(1{,}25), \alpha\sim U(1{,}6)$
$P_{active} = 500mW, P_{idle} = 200mW$

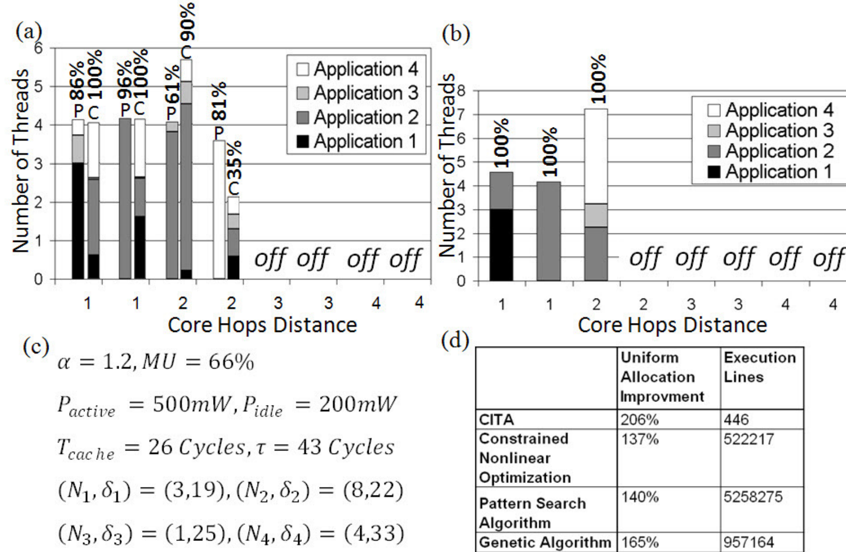
Fig. 8. $\boldsymbol{PPM_{CITA}/\max\{PPM_{optimization\ methods}\}}$

Fig. 8 presents the ratio between the results of CITA and the best result among all the optimization algorithms, 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 Fig. 8. On average, CITA outperforms the best optimization algorithms by 9%.

Fig. 8 presents also the average number of cores operating by CITA and by the best of the optimization algorithms for several cases. It can be seen that the PPM ratio is higher as the difference between those values is higher. This occurs as the CMP load is lower (i.e. there are more cores and less applications). On those cases CITA operates on average significantly fewer cores relative to the optimization algorithms.

Fig. 9 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.



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

Fig. 9. (a) Legend: C- Constrained nonlinear optimization result, P- Pattern search result

Fig. 9(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. 9(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. 9(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.

4.3 Discretization Results

ITA results are drived by the discretization of CITA results. We use the discretization method described by equation (14) . Fig. 10 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 Fig. 8. On average, discretization of CITA results reduces the PPM value by 5%.

Fig. 11(a) and Fig. 11(b) present an example of CITA and constrained nonlinear optimization algorithm result discretization respectively. Fig. 11(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.

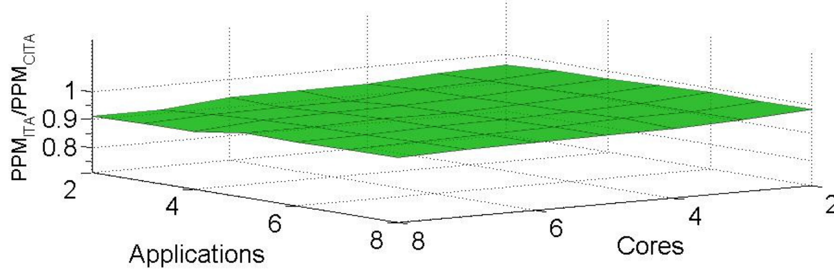
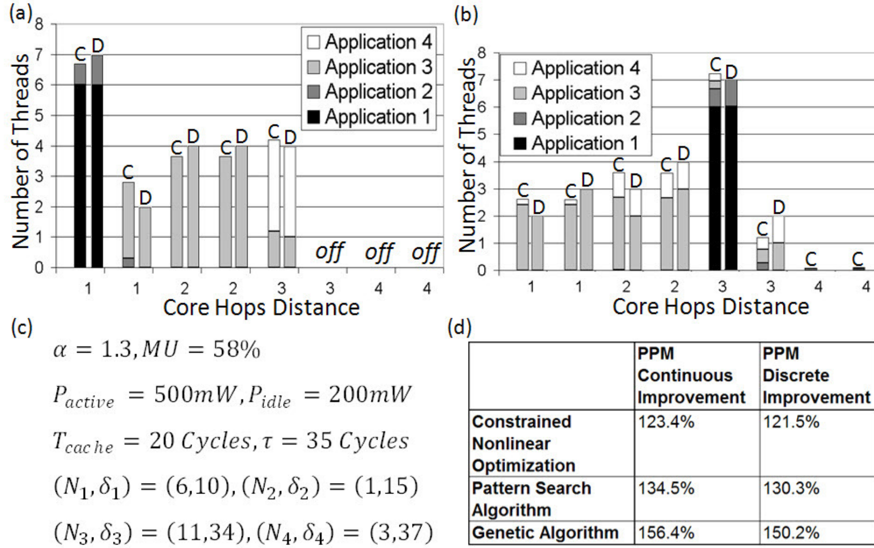


Fig. 10. PPM_{ITA}/PPM_{CITA}



	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. 11. Example of results discretization
Legend: C-Continuous result, D-Discretization of the continuous result

5 Conclusions

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.

References

[1] Olukotun, K., Hammond, L.: The future of microprocessors. Queue 3, pp. 26--29 (2005)

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

[3] Barroso, L.A., Gharachorloo, K., McNamara, R., Nowatzyk, A., Qadeer, S., Sano, B., Smith, S., Stets, R., Verghese B.: Piranha: a scalable architecture based on single-chip Multiprocessing. ACM SIGARCH Computer Architecture News, pp. 282--293 (2000)

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

[5] McCann, C., Vaswani, R., Zahojan, J.: A dynamic processor allocation policy for multi programmed shared memory multiprocessors. ACM Transactions on Computer Systems (1993)

[6] Fedorova, A., Seltzer, M., Small, C., Nussbaum, D.: Performance of multithreaded chip multiprocessors and implications for operating system design. In: USENIX 2005 Annual Technical Conference, pp. 395--398 (2005)

[7] Kim, S., Chandra, D., Solihin, Y.: Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In: 13th International Conference on Parallel Architecture and Compilation Techniques, pp. 111--122 (2004)

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

[9] Ding, Y., Kandemir, M., Irwin, M.J., Raghavan P.: Adapting Application Mapping to Systematic Within-Die Process Variations on Chip Multiprocessors. LNCS, vol. 5409, pp. 231--247. Springer, Heidelberg (2009)

[10] Chen, G., Li, F., Son, S.W., Kandemir, M.: Application mapping for chip Multiprocessors. In: Proceedings of the 45th Annual Design Automation Conference, pp. 620--625 (2008)

[11] Burd, T., Brodersen, R.W.: Energy efficient CMOS microprocessor design. In: 28th Hawaii International Conference on System Sciences, pp. 288--297 (1995)

[12] Penzes, P.I., Martin A.J.: Energy-delay efficiency of VLSI computations. In: Proceedings of the 12th ACM Great Lakes Symposium on VLSI, pp. 104--111 (2002)

[13] Agarwal, A.: Performance tradeoffs in multithreaded processors, IEEE Transactions on Parallel and Distributed Systems, vol.3 no.5, pp. 525--539 (1992)

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

[15] Gonzalez, C.R., Woods, R.E.: Digital image processing. pp. 128--138, third edition (2008)

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

[17] Ben-Itzhak, Y., Cidon, I., Kolodny A.: Performance and Power Aware CMP Thread Allocation Modeling. Technical Report, CCIT #735 (2009)