Efficient Program Scheduling for Heterogeneous Multi-core Processors

Jian Chen and Lizy K. John
ECE Department, University of Texas at Austin
Austin, TX 78712, USA
chenjian@mail.utexas.edu, ljohn@ece.utexas.edu

ABSTRACT

Heterogeneous multicore processors promise high execution efficiency under diverse workloads, and program scheduling is critical in exploiting this efficiency. This paper presents a novel method to leverage the inherent characteristics of a program for scheduling decisions in heterogeneous multicore processors. The proposed method projects the core's configuration and the program's resource demand to a unified multi-dimensional space, and uses weighted Euclidean distance between these two to guide the program scheduling. The experimental results show that on average, this distance based scheduling heuristic achieves 24.5% reduction in energy delay product, 6.1% reduction in energy, and 9.1% improvement in throughput when compared with traditional hardware oblivious scheduling algorithm.

Categories and Subject Descriptors: C.1.3 [**Processor Architectures**]: Other Architecture Styles – *Hybrid Systems*

General Terms: Design, Performance

Key Words: Heterogeneous Multi-core, Energy-Delay Product, Program Scheduling

1. INTRODUCTION

Heterogeneous multicore processors (HMP) have been demonstrated to be an attractive design alternative to its homogeneous counterpart, as it has a unique advantage in improving both system throughput and execution efficiency. Although most of the existing multicore processors are homogeneous, this design paradigm leads to an inevitable dilemma. That is, replicating smaller cores compromises the throughput of the high-complexity single-threaded applications; whereas replicating larger cores sacrifices the execution efficiency of the low-complexity low-priority threads. The HMP, however, integrates cores of different types or complexities in a single chip, and hence is able to address both throughput and efficiency for various workloads by matching execution resources to each application's needs. As a result, HMP is gaining preference in both industry (e.g. IBM's CELL [12]) and academia (e.g. Core Fusion [9], TFlex [7]).

While HMP is capable of addressing diverse workload demands, it relies on an appropriate program scheduling scheme to unleash its architectural potential for energy efficient computing. The successful program scheduler must be able to find the match between programs and cores with minimum cost in performance and power, which is nontrivial and remains challenging. A straightforward scheduling policy proposed by Kumar et.al uses trial-and-error approach to find the match between programs and cores [3]. The problem with this method is that it incurs significant energy overhead in context switching [15]. Becchi et al extends Kumar's work by measuring IPC ratios between two different cores to migrate applications [17]. Essentially, this method uses pair-wise program swapping to reduce the number of tentative runs.

These existing methods rely on monitoring the exhibited performance metric during tentative runs to identify the matching program-core pair. This paper, however, attempts to leverage the fundamental relationship between the inherent program characteristics and the corresponding resource demands for program scheduling. This paper, for the first time, presents a framework that addresses three aspects of a program scheduler in the HMP context: understanding the physical configurations of the core supply, estimating the resource demands of the running applications, and identifying the program-core matching for a given criteria [1]. The proposed method projects the core configurations and the program's resource demands into a unified multi-dimensional space, where the program-core matching can be easily identified with Weighted Euclidean Distances (WED). We demonstrate that the WED is strongly correlated with EDP, hence can be used to guide program scheduling in HMPs. Compared with traditional hardware oblivious scheduling, the distance based scheduling heuristic achieves an average of 24.5% reduction in EDP, and 6.1% reduction in energy, and improves the throughput by 9.1%.

The rest of the paper is organized as follows: Section 2 summarizes the related work. Section 3 describes the framework for multidimensional program-core matching. Section 4 shows the projection functions used in the proposed framework. Section 5 presents the scheduling heuristics. Section 6 gives the setup of the experimental environment. Section 7 discusses the experimental results and Section 8 concludes the paper.

2. RELATED WORK

Prior research on program scheduling in heterogeneous systems mainly focuses on optimizing the scheduling of subtasks. The widely adopted approach is to decompose the application into several dependent subtasks, formulate the constraint graph based on the inter-subtask dependencies, and partition the graph to minimize the overall execution time [13] or the data transfer on buses [18][19]. This scheduling approach requires the performance/power of a workload be known a priori or easy to predict, which fundamentally limits its applicability. Our method, on the contrary, does not need the microarchitecture dependent performance/power data as a priori information for scheduling.

Recently, Kumar et al. [3] propose a dynamic program scheduling approach based on the sampled EDP during tentative runs. Becchi et al. [17] extend this method by using the measured IPC ratios between two programs for program migrations. Gulati et al. [8] uses efficiency threshold to dynamically allocate processor for the given task. All of these methods exploit intra-program diversity, and could adapt to program phase changes. Our scheduling scheme exploits inter-program diversity and statically allocates programs to cores by analyzing inherent program characteristics.

Chen and John [6] employ fuzzy logic to calculate the program-core suitability, and use that to guide the program scheduling. However, their method is not scalable since the complexity of fuzzy logic increases exponentially as the number of characteristics increases. Our method, however, is scalable and can be easily extended to the case where the number of characteristics is four or beyond.

3. FRAMEWORK

The idea of multi-dimensional program-core matching stems from the observation that both programs and cores can be described with a set of orthogonal characteristics. By projecting these characteristics from both program side and core side to a unified multi-dimensional space, we are able to visualize the correlation between the program and the core, and simplify the program-core matching.

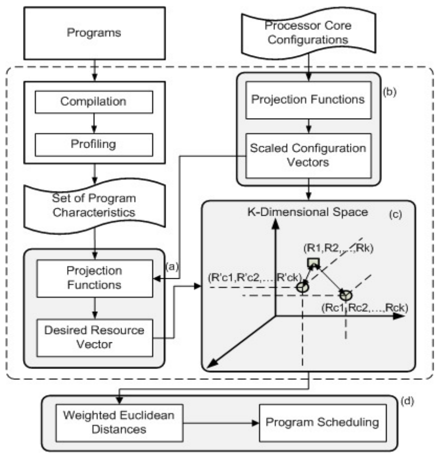


Fig 1. Framework for multidimensional program-core matching

Figure 1 shows the proposed framework for multidimensional program-core matching. The programs are first compiled and profiled to obtain K sets of inherent program characteristics: $\overrightarrow{X_1}, \overrightarrow{X_2}, \ldots, \overrightarrow{X_K}$, where $\overrightarrow{X_i} = (x_{i0}, x_{i1}, x_{i2}, \ldots, x_{in})$ is the vector that describes program characteristic i (i=1..K). Define g_i as the projection function that transforms characteristic $\overrightarrow{X_i}$ to the program's desired resource demand R_{pi}, i.e. $R_{pi} = g_i(\overrightarrow{X_i}) = g_i(x_{i0}, x_{i1}, x_{i2}, \ldots, x_{in})$ .We have the program's desired resource vector: $\overrightarrow{R_D} = (R_{p1}, R_{p2}, \ldots, R_{pK}) = \left(g_1(\overrightarrow{X_1}), g_2(\overrightarrow{X_2}), \ldots, g_K(\overrightarrow{X_K})\right)$.This vector points to the program's desired configuration node in the K-dimensional space, as shown in Figure 1 (c). On the other hand, each processor core has a configuration vector $(C_1, C_2, \ldots, C_K)$ with element C_i correspondent to the program characteristic $\overrightarrow{X_i}$ (i=1..K). Similarly, this vector can also be transformed by a set of projection functions to a scaled configuration vector $\overrightarrow{R_C} = (R_{c1}, R_{c2}, \ldots, R_{ck})$ in the K-dimensional space. Once the desired resource vector $\overrightarrow{R_D}$ and the configuration vector $\overrightarrow{R_C}$ have been projected to the same space, the distance between these two becomes the natural measurement for the degree of match between the program and the core. Specifically, larger distance leads to less compatibility between the program and the core, and hence less execution efficiency. Note that not every dimension of the vector contributes equally to the degree of match. Therefore, we use WED in the K-dimensional space as the metric for the program-core matching, as shown in equation (1):

$$D^2 = \sum_{i=1}^{k} w_i (R_{pi} - R_{ci})^2 \tag{1}$$

where w_i is the weight coefficient for the i-th dimension (i=1..K), and D is the weighted Euclidean distance. This distance could guide the program scheduler to identify the matching program-core pair.

4. PROJECTION FUNCTIONS

The projection functions are used to map program characteristics and core configurations to the unified K-dimensional space. On one hand, the projection functions need to interpret and quantify the implications of program's inherent characteristics on its hardware resource demand. On the other hand, the projection functions need to scale the raw hardware configurations in accordance to the diminishing return effect. This section gives the detailed description of the projection functions in this study.

4.1 Projection Functions for Core Configurations

Generally speaking, hardware resources suffer from the diminishing return effect, that is, the additional hardware resource yields less than proportional increase in the marginal benefit. This diminishing return effect has its implication on the core configuration projection: the spacing between adjacent configurations decreases as the value of the configuration increases. To capture this effect, we use the reciprocal function to scale the inter-configuration distance, which is formulated as follows:

$$R_X(i) = c\left(\frac{1}{X_{min}} - \frac{1}{X_i}\right) \tag{2}$$

where X_{min} is the minimum value of the configuration X among the cores, X_i is the value of configuration X of core i (i=1..n), and c is a normalizing factor. While the proposed framework supports K (K≥3) different hardware configurations, this paper only examines three configurations, i.e., issue width, branch predictor size and L1 data cache size.

4.2 Projection Functions for Programs Characteristics

In accordance with the hardware configurations, this paper investigates three important program characteristics: instruction level parallelism (ILP), branch predictability, and data locality. The projection functions for these characteristics are used to identify the desired issue width, branch predictor size, and data cache size.

4.2.1. Desired Issue Width

The desired issue width represents the pipeline width that a processor core needs to efficiently exploit the amount of ILP in the program. We use the instruction dependency distance [5] to capture the program's ILP. Typically, for a given dependency distance distribution, the more instructions with long dependency distance, the more ILP the program has, and hence the desired issue width is larger. Therefore, to obtain the desired issue width, we cluster the instructions into four groups with respect to their dependency distances, i.e., group 1 with distance of 1, group 2 with distance of 2-3, group 3 with distance of 4-7, and group 4 with distance of 8 and beyond. Let $X_{dep,i}$ represent the percentage of instructions whose dependency distance falls in group i (i=1..4). We have the program's dependency distance vector $(X_{dep,1}, X_{dep,2}, X_{dep,3}, X_{dep,4})$. Each element in this vector has its most suitable issue width, that is, 1-way issue for $X_{dep,1}$, 2-way for $X_{dep,2}$, and so on. Therefore, the mass center (or the weighted average) of the distribution indicates on average the issue width demand of the program, hence:

$$R_{issue} = \frac{\sum_{i=1}^{4} X_{dep,i} * W_i}{\sum_{i=1}^{4} X_{dep,i}} \tag{3}$$

where W_i, i=1..4, are the projected coordinates in the issue width dimension of the space, representing the issue width from 1 to 8.

4.2.2. Desired Branch Predictor Size

The desired branch predictor size represents the branch predictor size that a processor core needs to efficiently exploit branch predictability in the program. We use branch transition rate [14] to capture the branch predictability of the program. Generally speaking, the branch instructions with extremely low or extremely high transition rate are easy to predict, and as the transition rate approaches 50%, branches become harder to predict. Based on this observation, we evenly divide the transition rates into 10 buckets: [0, 0.1], [0.1, 0.2],..,[0.9, 1.0]. Let $X_{br,i}$ be the amount of branch instructions whose transition rate falls in the bucket i (i=1..10). We have the branch transition rate vector $(X_{br,1}, X_{br,2}, \ldots, X_{br,10})$. Since the branch instructions in the buckets [0.4, 0.5] and [0.5, 0.6] are the hardest to predict, they are associated with the largest branch predictor. The branch instructions in the buckets [0.3, 0.4] and [0.6, 0.7] are relatively easier to predict, and hence are associated with a smaller branch predictor. Same applies in buckets [0.2, 0.3] and [0.7, 0.8], and buckets [0.1, 0.2] and [0.8, 0.9]. Therefore, the mass center of the transition rate distribution indicates on average the demand of branch predictor size, hence:

$$R_{branch} = \frac{(B_1*(X_{br,2}+X_{br,9})+B_2*(X_{br,3}+X_{br,8})+B_3*(X_{br,4}+X_{br,7})+B_4*w*(X_{br,5}+X_{br,6}))}{\sum_{i=2}^{4} X_{br,i}+\sum_{i=7}^{9} X_{br,i}+w*\sum_{i=5}^{6} X_{br,i}} \quad (4)$$

where B_i, i=1..4, are the coordinates in the branch predictor dimension of the space ($B_1 < B_2 < B_3 < B_4$). Buckets [0, 0.1] and [0.9, 1] are not considered because branch instructions in this range are very easy to predict, and even the smallest branch predictor in this study would be more than enough for them. The parameter w is employed to tune the weight of the largest branch predictor, and is set to $\alpha\times P_{cond}$. The parameter α is an empirically determined value, and is in proportional to the instruction issue width. It is used to keep track of the fact that as the issue width gets wider the branch misprediction penalty also increases, and hence a larger branch predictor with higher prediction accuracy is more desirable. P_{cond} is the percentage of the conditional branches in the instruction mix. A large P_{cond} may lead to a large number of hard-to-predict branches; hence the weight of large branch predictor should be high.

4.2.3. Desired L1 Data Cache

Similarly, the desire L1 data cache is the data cache size that a processor core needs to efficiently exploit the data locality of the program. This paper uses Mattson's stack distance distribution [16] to measure the program's data locality. Let H_i $(i = 1..4)$ be the possible L1 data cache sizes with $H_1 < H_2 < H_3 < H_4$, and let$X_{stk,i}$ $(i = 1..4)$ be the number of accesses whose stack distance is within the range of $[0, H_1]$, $(H_1, H_2]$, $(H_2, H_3]$, $(H_3, H_4]$ respectively. Note that $X_{stk,1}$ can be most efficiently hold by the cache with size H_1, and $X_{stk,2}$ can be most efficiently hold by the cache with size H_2, and so on. Similarly, the desired L1data cache size of the program can be calculated as:

$$R_{cache} = \frac{\sum_{i=1}^{4} X_{stk,i}*H_i'}{\sum_{i=1}^{4} X_{stk,i}} \quad (5)$$

where H_i' represents the scaled coordinates for the data cache size $H_i, i = 1..4$.

Since the issue width has the highest impact on the program-core matching, followed by cache size and branch predictor size, the highest weight is assigned to the issue width dimension, with the second highest to the cache size dimension, and the lowest to the branch predictor dimension. In this paper, we empirically assign the weights to be 0.8, 0.2, and 0.1 to demonstrate the concept.

5. SCHEDULING HEURISTICS

The WED represents the match between the program's desired resource demand and the core's hardware resource supply. The smaller the distance is, the better the match is, and hence the higher the execution efficiency would be. Therefore, the distance can be treated as a proxy of the program's execution efficiency on a certain core. Given that, the optimum scheduler shall minimize the total distance of the scheduled program-core pairs. However, such scheme is NP-complete (O(n!) with a naïve implementation), and becomes impractical for large number of programs.

```
Let $P_j$ be the j-th program in the program queue (j=1..M).
Let $C_i$ be the processor core i. (i=1..N)
for j ( 1 .. M)
    for i ( 1.. N)
        if ($C_i$ is available && min_dist > distance($P_j$,$C_i$))
            min_dist = distance($P_j$,$C_i$);
            k=i;
        end if
    end for
    schedule $P_j$ to $C_k$;
    mark $C_k$ as unavailable;
end for
```

Fig 2.Pseudo code of the distance based scheduling heuristic.

This section presents a scalable scheduling heuristic based on the distance. As shown in Figure 2, this heuristic schedules the program on a first-come, first-served (FCFS) basis, and for each program, it allocates the available core with the minimum distance to that program. The complexity of this scheme is the same as the complexity of finding the minimum distance, which is O(n). In addition, this paper also examines the following scheduling algorithms for comparison:

Hardware Oblivious Scheduling (baseline): This scheduling scheme is unaware of the hardware substrate, and schedules the program on a FCFS basis. Specifically, the first in program queue is schedule to core 1, the second to core 2, and so on.

Min EDP Scheduling: This scheduling method attempts to schedule the programs such that the overall EDP is minimized. It assumes that the EDP of each program-core pair is known a priori, and hence sets the ***best case*** scenario the overall EDP.

Max EDP Scheduling: This scheduling method attempts to maximize the overall EDP. It also assumes the EDP of each program-core pair is known a priori, and provides the ***worst case*** scenario in the overall EDP for comparison.

6. EXPERIMENTAL SETUP

To validate the WED model, we vary the instruction issue width, L1 data cache size and branch predictor size of an out-of-order superscalar processor. The configuration options of these parameters are shown in Table I. To evaluate the proposed scheduling heuristics, we create a hypothetical heterogeneous single-ISA quad-core processor. The detailed configurations of these cores are listed in Table II. Like Cortex-A9 multicore processor [2], each core is an out-of-order processor, and there is no on-chip L2 cache. Other parameters not shown in the table are chosen in a way that the design of the core is balanced.

Table I. Configuration Options

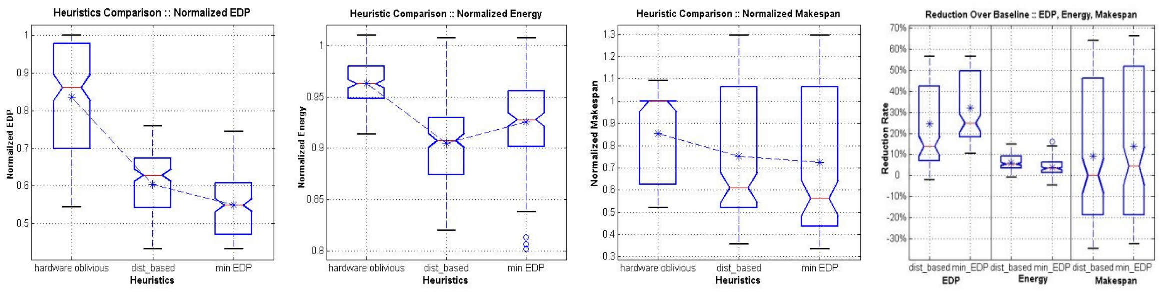
Items	Configuration Options
Issue Width	single-issue, 2-issue, 4-issue, 8-issue
L1 D-Cache	8KB, 4-way, block size 64byte, 16KB, 4-way, block size 64byte, 32KB, 4-way, block size 64byte, 64KB, 4-way, block size 64byte
Branch Predictor	1K Gshare, 2K Gshare, 4K Gshare, 8K Gshare

Table II. Configurations of Each Core

Items	Configurations
Core 1	Out-of-order, single-issue, Gshare(1k), 8KB 4-way L1d-cache 64byte, 16k 2-way i-cache 64byte
Core 2	Out-of-order, 2-issue, Gshare(2k), 8 KB 4-way L1 d-cache 64byte, 16k 2-way i-cache 64byte
Core 3	Out-of-order, 2-issue, Ghsare(2k), 64 KB 4-way L1 d-cache 64byte, 16k 2-way i-cache 64byte
Core 4	Out-of-order, 4-issue, Gshare(4k), 16 KB 4-way L1 d-cache 64byte, 16k 2-way i-cache 64byte

The workload of the experiment is composed of benchmark programs from Mibench, MediaBench and SPEC CPU2000. Each program is compiled to Alpha-ISA with peak configurations. In order to count in the effect of different input datasets, we use the test/training input datasets to profile the programs, and use the profiled characteristics to schedule the programs with other input datasets. In this study, we assume the workloads are independent with each other, and we do not consider the hardware mechanisms for core-level communication because these mechanisms are symmetric and have similar impact on independent workloads.

We use extensively modified SimProfile from Simplescalar tools [10] to profile programs and collect the aforementioned program characteristics. We employ Wattch [11] to obtain the performance and power data for each benchmark program. To demonstrate the effectiveness of the framework, we evaluate three aspects of the



(a) (b) (c) (d)

Fig3. EDP, energy and makespan comparison between different scheduling heuristics. The asterisk stands for the sample average.

proposed scheduling scheme: EDP, energy, and makespan. EDP takes into account both energy and speed, and is widely adopted as a metric for the execution efficiency. Makespan is the time between the start and finish of a group of programs, and is used as the metric for throughput [8].

7. RESULTS & ANALYSIS

To demonstrate that the WED is the appropriate proxy for the program's execution efficiency on a certain core, we need to show the correlation between EDP and WED. To do that, we measure the EDPs of each program on each of the 64 processor cores, and calculate the distance of every program-core pair according to the proposed framework. Table III shows the Pearson's correlation coefficient for each benchmark program included in this study. Note that for each program, we use one set of input for EDP measurement and a different set of input for program profiling. Therefore, the coefficients in Table III not only demonstrate that the WED is strongly correlated with EDP, but also show that the input set has negligible influence on the program's inherent characteristics.

Table III. Correlation Coefficient between EDP and Distance

Benchmarks	Coeff.	Benchmarks	Coeff.
susan	0.91544	ghostscript	0.95708
qsort	0.94244	epic	0.94086
fft	0.81584	g271encode	0.89022
rawdaudio	0.83496	mcf	0.87527
blowfish	0.93837	gcc-166	0.80145
bitcnts	0.81135	gcc-expr	0.88296
mpeg2decode	0.95841	gcc-scilab	0.80598
cjpeg	0.94807	twolf	0.81261

To evaluate the proposed program scheduling schemes, we select programs with different characteristics from the benchmarks in table III to compose 150 diverse program combinations, with each one containing 4 programs. Figure 3 shows the boxplots [4] of the scheduling results of these workloads. According to these boxplots, the proposed distance based scheduling has statistically significant improvement over the baseline scheduling with an average of 24.5% reduction in EDP, 6.1% reduction in energy, and 9.1% reduction in makespan. Compared with min EDP scheduling, the distance based scheduling achieves higher reduction in energy (6.1% vs 3.9%), but less reduction in makespan (9.1% vs 13.8%). However, the difference in makespan reduction is not statistically significant because the notches of the two boxplots overlap, as shown in figure 3 (d). Overall, the distance based scheduling achieves a good balance between throughput and energy.

8. CONCLUSION

This paper presents a framework of multidimensional program-core matching for program scheduling in heterogeneous multiprocessors. The proposed method projects the core's configuration and the program's desired resource demand to a unified multi-dimensional space, and uses weighted Euclidean distance between these two to guide the program scheduling. The experimental results show that the distance based scheduling heuristic achieves an average of 24.5% reduction in EDP, 6.1% reduction in energy, and 9.1% improvement in throughput when compared with traditional hardware oblivious scheduling algorithm.

References

[1] F. A. Bower, et al. "The Impact of Dynamically Heterogeneous Multicore Processors on Thread Scheduling", *IEEE Micro, pp 17-25, May 2008.*

[2] The ARM Cortex-A9 Processor, the ARM white paper. http://www.arm.com/pdfs/ARMCortexA-9Processors.pdf

[3] R. Kumar, et al, "Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction", *Micro-36, pp 81-92, Dec. 2003.*

[4] Yoav Benjamini. "Opening the Box of a Boxplot". *The American Statistician. Vol 42 (4), pp257–262, Nov. 1988.*

[5] A. Phansalkar, et al, "Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites," *ISPASS'05, pp 10-20, Mar.2005*

[6] J. Chen and L. K. John,"Energy aware program scheduling in a heterogeneous multicore system", *IISWC'08, pp.1-9, Sept. 2008.*

[7] Changkyu Kim, et.al., "Composable Lightweight Processors," *Micro-40, pp.381-394, Dec. 2007*

[8] D.P.Gulati et al., "Multitasking Workload Scheduling on Flexible Core Chip Multiprocessors", *PACT'08, pp187-196, Oct. 2008.*

[9] E. İpek, et al., "Core Fusion: Accommodating software diversity in chip multiprocessors". *ISCA-34, pp 186-197, June 2007.*

[10] D. Burger and T. M. Austin, "The simplescalar tool set version 3.02" , http://www.simplescalar.com/

[11] David Brooks, et al. "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations", *ISCA-27, pp 83-94, June, 2000*

[12] H.P Hofstee, "Power efficient processor architecture and the CELL processor", *HPCA-11. pp. 258-262, Feb. 2005*

[13] M.Maheswaran and H.J.Siegel, "A Dynamic Matching and Scheduling Algorithm for Heterogeneous Computing Systems", *Proc. Heterogeneous Computing Workshop, pp. 57-69, 1998*

[14] M.Haungs, et.al, "Branch transition rate: a new metric for improved branch classification analysis", *HPCA-6, pp.241-250, Feb.2000*

[15] C. S. Ballapuram, A. Sharif and H. S. Lee, "Exploiting Access Semantics and Program Behavior to Reduce Snoop Power in Chip Multiprocessors", *ASPLOS XIII, pp 60-69, Mar 2008*

[16] R.L.M. et al. "Evaluation Techniques for Storage Hierarchies". *IBM Systems Journal, pp 78-117. 1970*

[17] M. Becchi, Patrick Crowley, "Dynamic thread Assignment on Heterogeneous Multi-processor Architectures*", Computing Frontiers 2006, pp 29-40, May 2006*

[18] J. Liu, et al., "Power-aware scheduling under timing constraints for mission-critical embedded systems*", DAC-38, pp840-845, July, 2001.*

[19] M. Ruggiero, et al. "Communication aware system-on-chip allocation and scheduling framework for stream-oriented multi-processor", *DATE, pp3-8, Apr.2006.*