HyperCoreX: Non-Equidistant Memory Network in a Many-core Architecture

Dmitry Khoretz[1,2], Eyal Friedman[1,2] and Ran Ginosar[2]
[1]Intel Corporation, Haifa, Israel
[2]Electrical Engineering Dept., Technion, Haifa, Israel

Abstract

HyperCore is a simple many-core architecture enabling 100-1000 cores to achieve higher performance than multicore while expending lower power and energy. It comprises on-chip shared memory (organized in many banks), simple cache-less RISC cores and a simple on-chip hardware scheduler, using a single slow clock. Thanks to sharing the cache, coherency issues are eliminated. Memory addresses are interleaved over the many banks, minimizing conflicts but requiring each core to access many banks, far and near. This study focuses on improving network to eliminate the slow-clock limitation. We study a modified HyperCoreX architecture with a non-equidistant network to memories, allowing shorter access times to nearer memory banks. Six benchmark programs were employed, representing a wide variety of inherent parallelism, address distributions, access rates and data sharing. It was shown that the non-equidistant memory in HyperCoreX, together with the resulting increase of frequency, can speed up program execution by 4-9 times relative to the equi-distant single cycle architecture, and reduce memory wait time by up to 61%.

1 Introduction

On-chip many-core architectures have been proposed as an effective high performance computing model. Suggested models include generalized, easy to resize, symmetric systems [1]–[7] with a simple MESH structure and 2D network connectivity. Such systems show good improvement in computational power and speed up over single-core processors, but they still encounter problems related to distribution of data among the cores, cache coherency, collisions and network delays. Previous research on NUMA/NUCA [20],[21] addresses multicore systems and their shared and non-shared caches, coherency problems and physical layout. That approach becomes impractical in many-core systems of many hundreds of processors due to at least quadratic complication of coherency issues. The Ultracomputer [22] is limited by an equi-distant memory; this paper addresses that limitation. More unique architectures [8] –[14] investigate different approaches to achieve greater performance from the many-core machine, but turn out to be more suitable for special purpose applications due to programming difficulties and lack of software base. Other successful many-core systems are the graphics processors (GPU), employed in general purpose calculations [15]. Features of the

many-core graphics processors are simple: cashless computation units connected to a shared memory module through a dedicated NoC.

The HyperCore architecture [16] presents a single clock system comprising many simple, cache-less RISC cores connected to single address space multi-bank memory through a dedicated network. We employ Multistage Interconnection Network (MIN, such as Omega) offering N-to-M connectivity for N processors and M memory banks. Memory latency is assumed one cycle each way in the equi-distant case and a variable number of cycles for the non-equidistant case. Bandwidth is N×W for W bits per word when non-blocking. The paper assumes on-chip SRAM memory, and the research is independent of the organization of off-chip memory (single shared cache or single shared local store). There are no cache coherency issues thanks to a single memory address space that is shared by all cores. Task distribution is controlled by a novel hardware scheduler. In order to reduce collisions, memory is divided into banks with interleaved addresses. In our simulations we consider a 32-bit address space . All round-trip memory accesses from each core to each memory bank take a constant time of two cycles if there are no conflicts. Since the HyperCore is designed as a single clock system, the clock cycle time is limited by the longest wire delay between any core and any memory bank. Wire delay seems to be a common barrier in large scale processors and chips [17][18].

The novelty of this paper consists of applying non-equidistant shared memory to the simplistic many-core architecture, introducing path latency diversity, analytical modeling of the non-uniform distance in a many-core architecture, and a new format for presentation of simulation data. Our research attempts to improve HyperCore by making it a more flexible architecture: The access time to nearby memory bank can be shorter than to a far-away bank. This approach is devised to alleviate long wire delays in large integrated circuits.

The paper contrasts the base HyperCore architecture in Sect. 2 with the modified HyperCoreX in Sect. 3. The simulator and the benchmark programs are described in Sect. 4 and 5, respectively. The simulation results are explained in Sect. 6 and the paper concludes in Sect. 7.

2 The HyperCore Architecture

The HyperCore architecture (Figure 1) comprises many cores, a shared memory divided into banks and a scheduler. The cores are simple RISC processors with no data cache. They perform tasks assigned to them by the scheduler, and report back to the scheduler when done. Small instruction caches in the cores enable efficient access to code. The cores use blocking data load and store instructions with no out-of-order execution.

The shared memory is organized in a large number of banks to enable many ports that can be accessed in parallel by the many processors. To reduce collisions, addresses are interleaved over the banks. The cores are connected to the memory banks by a many-to-many interconnection network that can serve simultaneous accesses from all cores. The network detects access conflicts contending on the same memory bank, proceeds to serve one of the requests and notifies the other cores to retry their access. The cores immediately retry a failed access. Two or

more read requests from the same address are served by a single read operation and a multicast of the same value to all requesting cores.

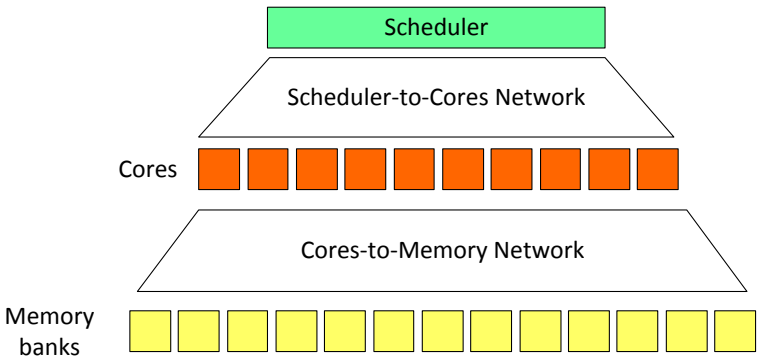

Figure 1: System architecture

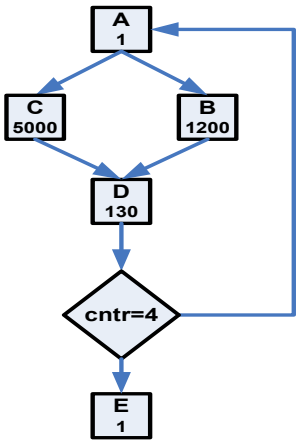

Figure 2: Basic task map

All round-trip memory accesses from each core to each memory bank take a constant time of two cycles if there are no conflicts. This memory organization is termed 'equi-distant memory' (in the rest of this paper we consider the effects of non-equidistant memories in the HyperCoreX architecture). Since the HyperCore is designed as a single clock system, the clock cycle time is limited by the longest wire delay between any core and any memory bank. A practical implementation achieves less than 500MHz on 45nm CMOS for a HyperCore comprising at least 64 cores and at least 1Mbytes of shared cache.

A hardware scheduler assigns tasks to cores for execution. When a core completes executing a task, it notifies the scheduler, which in turn assigns a new task to that processor. Task completion and allocation messages may incur certain latency, causing the cores to wait idle until the messages have made the round trip. The scheduler is capable of handling a certain number of completion messages and task allocations in each cycle; if more cores have completed their tasks in that cycle, they will remain idle for additional cycles until the scheduler manages to serve them. In this research we assume an ideal scheduler: Completion and allocation messages are delivered instantaneously with no latency, and the scheduler is capable of serving an unlimited number of completion and allocation messages at every cycle.

The programming model of the HyperCore is based on multiple sequential tasks and their inter-dependencies. The programmer defines the tasks, as well as the list of dependencies, formulated as a *task map,* a directed graph. The tasks are executed by the cores, while the task map is executed by the scheduler. Some tasks may be *duplicable*, accompanied by a *quota* that determines the number of instances that should be executed; all instances of a task are mutually independent and may be executed in parallel or in any arbitrary order. The instances are distinguishable from each other merely by their *instance number*. Ideally, the instances do not

share data and their execution time is short (fine granularity). The scheduler distributes the tasks that are eligible for execution among the available cores at that moment.

Figure 2 shows a simple task map example. Squares represent tasks and show the task name (A, B, C, …) and the number of required duplications. Arrows represent task dependencies. A task is eligible to run only when all its predecessors have completed. The rhombus represents a condition affecting the scheduler. It is executed only by the scheduler in zero time; there is no real code associated with this task. In this example, the condition controls task looping: The scheduler goes back to task A (for another invocation) for 4 times, and then proceeds to task E.

The HyperCore architecture is designed for high performance by enabling fine grain parallelism. Since all scheduling of tasks is done by hardware, incurring only minimal scheduling overhead time, it is feasible to create a large number of very short tasks and execute them efficiently. The shared memory model that enables multiple simultaneous accesses with minimal conflict rate also contributes to high level of parallelism. The programming model allows the programmer to ignore hardware details such as the number of processors and having to allocate tasks to specific cores, or to worry about scheduling details. Since cores are homogeneous, there are no asymmetries that complicate programming. An implementation comprising 64 cores and 2Mbytes on 45nm technology, operating at 400 MHz has already been designed by Plurality Ltd., demonstrating feasibility of this architecture and acceptable low power operation

3 Architecture of HyperCoreX

We study a modified HyperCore architecture, HyperCoreX, comprising of 256 RISC cores and 512 memory banks. Unlike the base system (Sect. 2), access time to a nearby memory bank can be shorter than to a far-away bank. The cores-to-memory network is no longer equi-distant. We first analyze the chip floorplan and examine the potential benefit of this more complex structure. We then define several possible clock frequencies and multi-cycle access to memories, in order to model performance of the new structure and operating mode. Changing the clock frequency also implies changing the manner of access to memory banks and the network bandwidth. The base (equi-distant) architecture is feasible at 400MHz, and this paper studies extending that frequency only eight -fold up to 3.2GHz, using the same chip area.

3.1 Equi-distant memory access

To simplify both the analysis and the logic design, any access to memory is timed with an integral number of clock cycles. In the equi-distant memory of the base HyperCore, the access takes two clock cycles: One cycle for travelling from core to memory, and a second cycle for the return trip. Thus, the cycle time must be sufficiently long to accommodate the longest wire delay on the chip. In the modified HyperCoreX, a faster clock is employed. Access to a nearby bank could take two cycles, while access to the farthest possible bank would take $2k$ cycles for some k. We consider four different clock frequencies: A slow frequency (‘freq1’) and twice, four times and eight times faster frequencies (freq2, freq4 and freq8, respectively). Thus, access to the farthest bank takes 2, 4, 8 and 16 cycles, respectively.

To investigate this, consider the chip floor-plan in Figure 3. In the base HyperCore equi-distant memory model, access latency is the same for all core-bank pairs, regardless of the physical distance between the initiating core and the target memory bank. Essentially, increasing the frequency in an equi-distant architecture increases only cores and memory banks frequency, but does not reduce the propagation delay over the cores-to-memory network.

The equi-distant memory access model serves as the base line for comparison to the non-equidistant memory access model, described below. In our research, to reflect the fact that memory banks are accelerated in the freq2, freq4 and freq8 models (relative to freq1), two, four and eight simultaneous accesses respectively are allowed in these frequencies. This is also done in order to decouple the memory frequency increase from the equi-distant architecture.

3.2 Non-equidistant memory access

To enable shorter access times to closer memory banks, we divide the long clock cycle time of freq1 into 2, 4 or 8 (freq2, freq4, freq8 models, respectively) and assign variable multi-cycle access times, depending on physical core-to-bank distances. A round-trip access is always measured in terms of an even number of cycles (for instance, 2, 4, 6 or 8 cycles in the freq4 model). Thus, the network becomes logically non-equidistant, enabling shorter access to memory banks that are physically closer to a core.

As we increase the system clock frequency, in addition to speeding up memory accesses according to distance, we also speed up both processor cores and memory banks. Speeding up the cores helps execute the rest of the code faster. Speeding up the memory banks enables handling the accelerated rate of accesses that may be generated by the accelerated processors and lower latency network.

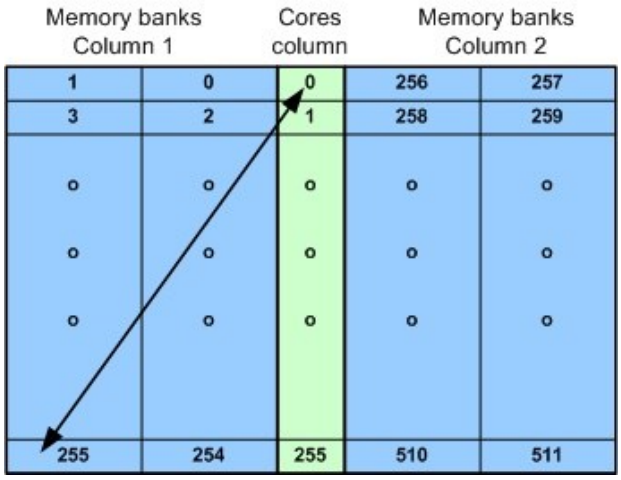

Figure 3: Floor-plan. Farthest access path is marked by an arrow

Figure 3 depicts the floorplan with indication of cores and memory banks indices. Of course, this is only one of the possible floorplan options . Other floorplan schemes may follow Nahalal [23] where the shared memories are placed in the center, surrounded by cores, and any private memories are placed outside the ring of cores. Each core-to-bank access time is placed in

an Access Time Matrix (ATM), shown in Figure 4. The ATM contains the memory access time distribution in the chip between cores and memory banks. Table 1 summarizes the average memory access times depending on frequency, assuming a uniform core/bank access distribution.

Frequency	Average Cycles per-access	Average cycles per-access relative to freq1 cycles	Average access time improvement from freq1
freq1	2	2	
freq2	3	1.5	25%
freq4	4.5	1.125	44%
freq8	7.25	0.906	55%

Table 1: Average memory access times and improvement according to frequency

Thus, the potential for memory access time improvement increases with frequency. In Sect. 6 we compare these theoretical calculations with simulation results and show that the simulations confirm the theoretical calculations.

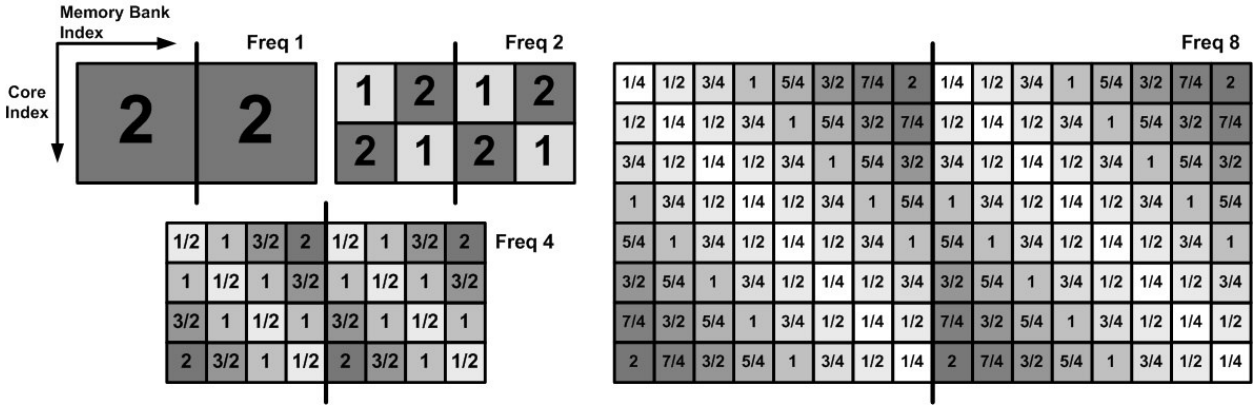

Figure 4: ATM (Access Time Matrix) for different frequencies (normalized to freq1 cycles)

3.3 HyperCoreX Power and Energy Considerations

While this paper does not analyze power and energy implications of either HyperCore or HyperCoreX architectures, a short discussion is provided here for completeness. On a first order of approximation, power is linearly proportional to frequency:

$$PowerAtFreqX = PowerAtFreq1 \times FrequencyFactor$$

Since benchmarks execute at varying amounts of time, the energy consumed by the system per benchmark is:

$$EnergyAtFreqX = \frac{PowerAtFreqX}{SpeedUp}$$

Where *EnergyAtFreqX* is the energy consumed by a benchmark executing on a specific HyperCoreX configuration, , *FrequencyFactor* is the frequency multiplier relative to the base configuration, *PowerAtFreqX* is the power consumption and *SpeedUp* is measured relative to the base configuration. Energy saving per benchmark can be calculated as follows:

$$EnergySavedRatio = \frac{EnergyAtFreqX}{EnergyAtFreq1} = \frac{FreqFactor}{SpeedUp}$$

In addition to these simplified calculations, other power saving features may be considered, such as gating power to idle cores and to cores waiting for memory access.

4 Simulation Environment

4.1 Definitions

Duplicable tasks have been explained in Sect. 2. A *normal task* is a duplicable task with only one instance. Tasks may be *invoked* more than once by a scheduling loop in the task map. The *i*th instance of the *j*th invocation of task *A* is marked *Ai.j*. Each core can be in one of the following states: *Busy* (assigned a task and is executing a non-memory access instruction), *Wait* (waiting for a successful Load or Store memory access to complete), *Collision* (waiting for an unsuccessful memory access to complete; a retry will be needed) and *Idle* (not assigned any task).

4.2 Simulation

We implemented an architectural simulator in Matlab. Execution traces of the benchmark programs, together with their corresponding task maps, are used by the simulator to investigate parallel executions under different architectural variations.

The traces were generated as follows. Each benchmark program is simulated first on a single core, and the resulting trace contains only two components: (a) length of execution (in cycles) for each code segment between two successive memory access operations, (b) for each memory access, the address and its type (read or write). These data items are invariant regardless of the specific parallel architecture and any collisions with other simultaneously executing tasks.

The simulator generates statistics per each clock frequency (freq1, freq2, freq4, freq8), which include cores activity (per-cycle and per-core) and memory accesses (per-memory bank and per-cycle), as shown in Sect. 6. These activities are measured in terms of the four core states (busy, wait, collision, idle).

5 Benchmarks

The benchmark programs include one synthetic program and three real applications. The synthetic ("Parallel") program presents a highly parallel case (Figure 5). The three real applications are Mandelbrot set calculation, JPEG image compression (of a 160x160 image) and a linear solver (Figure 6).

The benchmarks used (Mandelbrot, JPEG compression and Linear Solver) are taken from actual complex applications that execute on HyperCore. Other benchmark workloads will be re-programmed for many-core and investigated in the future.

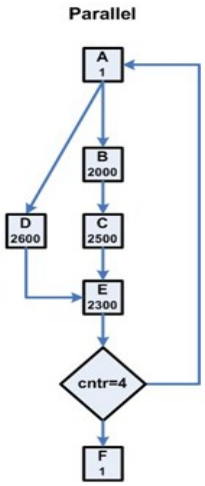

Figure 5: Task map of the synthetic Parallel program

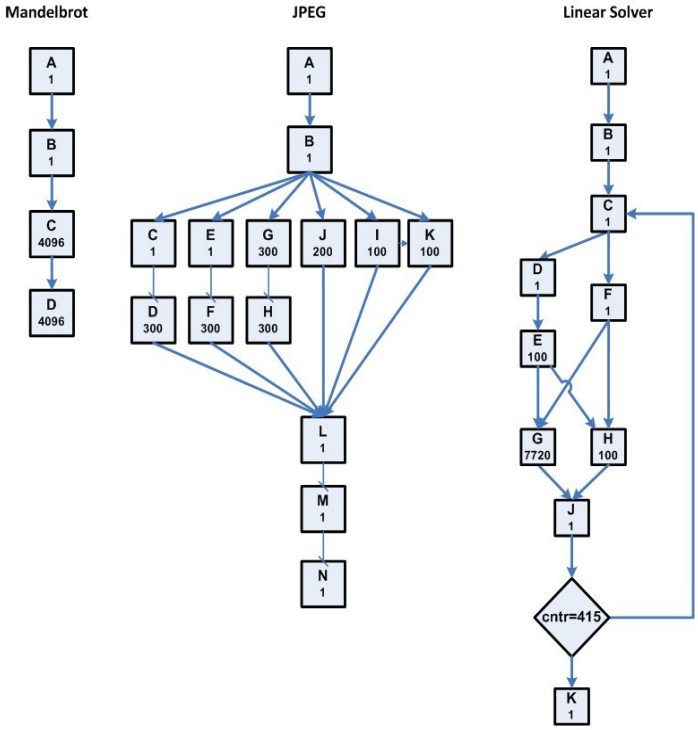

Figure 6: Task maps of the benchmark programs

6 Simulation Results

The four benchmark programs were simulated on the 256-cores, 512 memory banks equi-distant HyepCore and non-equidistant HyeprCoreX architectures using four clock frequencies. Some of the reported results are separated into the cases of equi-distant vs. non-equidistant memory.

6.1 Activity per core

In this section we observe the activity of every core during different program executions. The charts show the number of cumulative cycles each core is engaged in each type of activity: Busy, Wait, Collision and Idle. In each benchmark the four charts relate to the four frequencies

(Freq1 through Freq8 from left to right). The charts show the cumulative number of cycles of each type of activity for each core. In each chart, the x-axis is the Core indices, and the y-axis shows the cumulative activity of each core during the program, measured in cycles of the respective frequency. The Scheduler is ideal, leaving no idle cores while there are tasks available for execution and assigning tasks with priority to cores with lower indices; this is why lower indexed cores are less idle than the higher index ones.

Note that the cycles in each chart represent cycles in the respective frequency. In order to see relative program run time, we have to divide the number of cycles in the program by the frequency factor. This is shown in section 6.4 below.

6.1.1 Activity per core: Equi-Distant

Figure 7 shows activity per core. In the simulation of the equi-distant architecture, the time it takes for memory requests to propagate through the network is the same regardless of the core or memory bank locations. What really changes when increasing the frequency is the cores' internal frequency and the number of simultaneous bank accesses (ports) in the memory banks. For example, in freq2, internal core frequency is two times faster than in freq1 and each memory bank has two parallel ports for bank access. In the charts we can see that for different benchmarks there is a different distribution of activity on the cores, but it can be seen clearly that as frequency increases, the number of collisions decreases.

In the synthetic Parallel program we see an even distribution of work between cores, thanks to high number of tasks invocations and high parallelism. The share of the busy cycles in the program decreases as frequency increases and the percentage of access time (wait) increases.

Mandelbort and the linear solver are two realistic benchmark programs that behave similarly to the Parallel program, indicating that they are also highly parallel. Note that the Mandelbrot program also has a significant serial section (the idle cycles) but its relative portion is reduced when increasing frequency. The JPEG program has a large serial section. Although it is not visible in the JPEG charts directly, core zero has a very high busy cycles percentage for performing the serial task, while all other cores are idle. If this serial task is removed from the program, the result is similar to the parallel benchmarks.

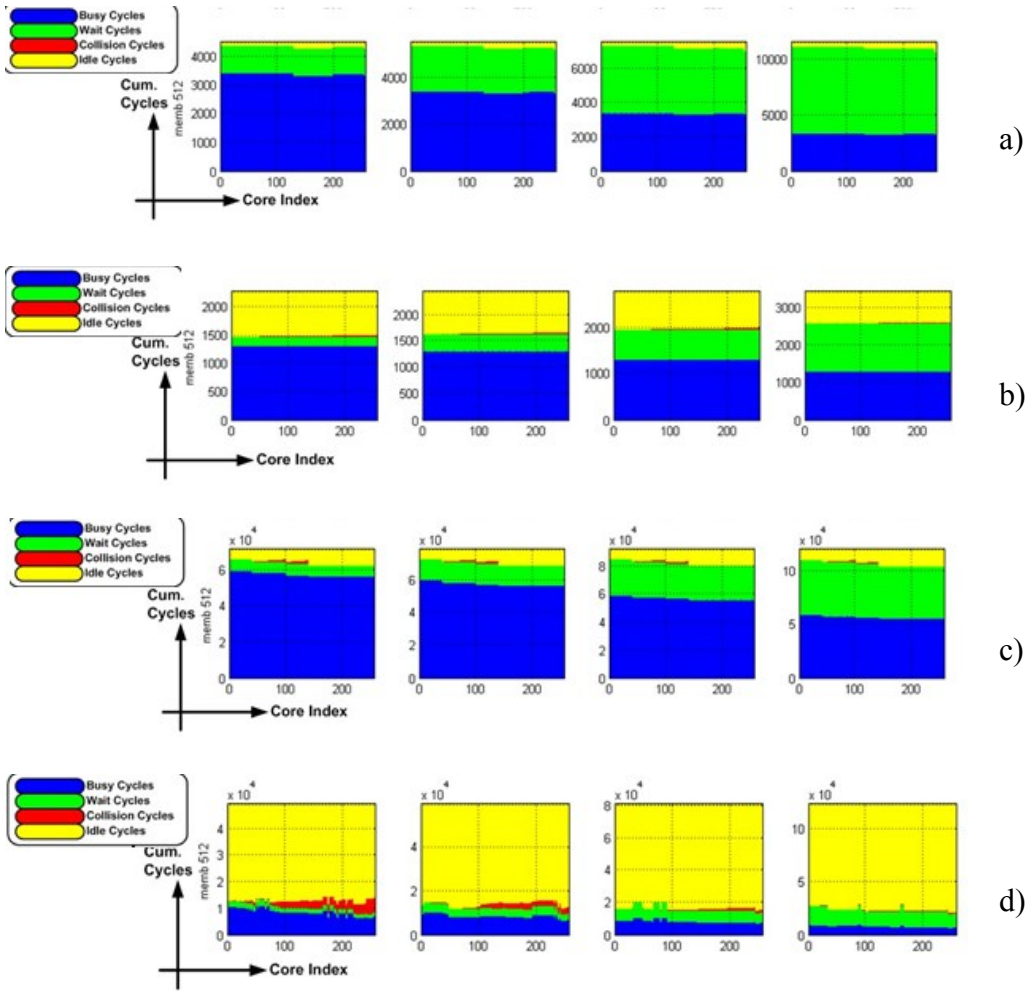

Figure 7: Activity per-Core on equi-distant architecture: (a) Parallel, (b) Mandelbrot, (c) Linear solver, (d) JPEG

6.1.2 Activity per core: Non-Equidistant

In the simulation of the non-equidistant HypreCoreX architecture (Figure 8), increase in frequency affects cores-to-memory banks access time and makes our system resemble a NUMA (Non Uniform Memory Access) system [19]. When comparing the charts in this section to the equi-distant cases above, we observe reduction of total run time ranging from 30% to 55%. Based on these charts, we can reach the same conclusion as from the simulation of the equi-distant architecture. Collisions decrease when frequency increases for all benchmarks. More interesting here is to see the difference between the equi and non-equidistant architecture simulations and to extract the direct contribution of non-equidistant approach in our particular system. Such comparisons are discussed below in section 6.5. While the total number of busy cycles is the same in both sets, the charts in this section show that the percentage of busy cycles is larger, indicating that relatively less time is spent waiting for memory accesses.

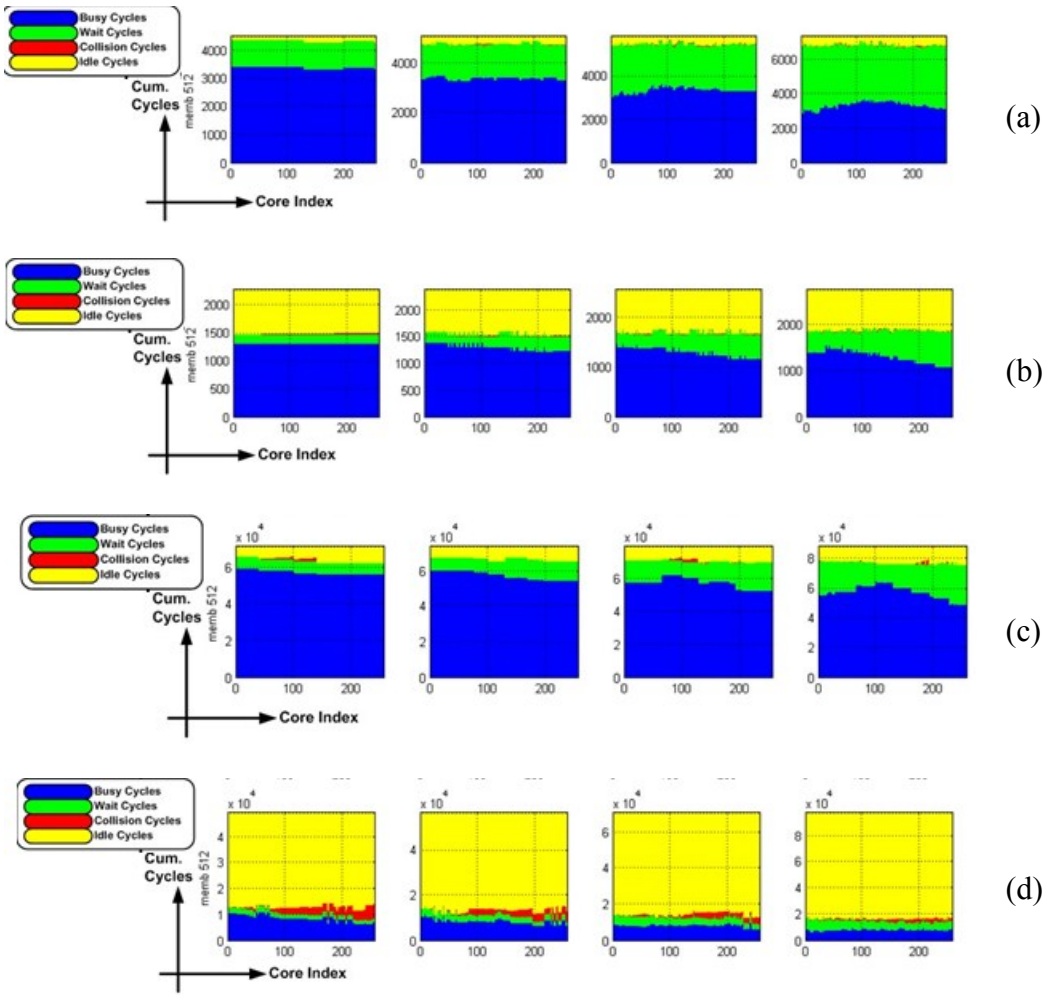

Figure 8: Activity per-Core in the non-equidistant architecture: (a) Parallel, (b) Mandelbrot, (c) Linear solver, (d) JPEG

6.2 Collisions per cycle

This section discusses the total number of collisions per cycle, enabling us to focus on the temporal view of collisions in the memory banks. This way we can clearly see the influence of the frequency on memory collisions in the various benchmarks. Clearly, collisions happen in bursts reflecting different phases of the benchmark programs.

6.2.1 Collisions per cycle: Equi-distant

We observe here that increasing the frequency reduces the number of collisions in general, but in the linear solver benchmark, for example, this reduction is moderate. Note that in the equi-distant case, much of the collision reduction is thanks to the multi-ported banks.

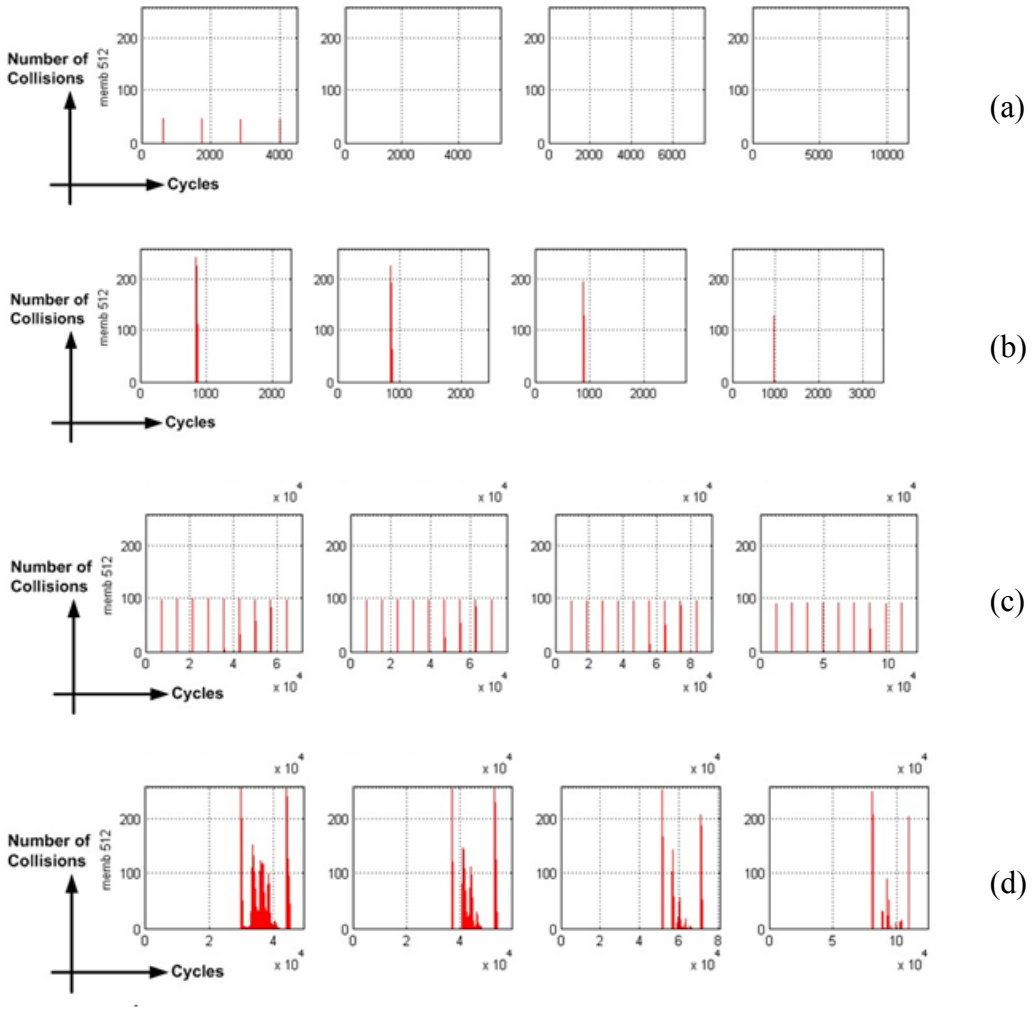

Figure 9: Number Collisions per-Cycle in the equi-distant architecture: (a) Parallel, (b) Mandelbrot, (c) Linear solver, (d) JPEG

6.2.2 Collisions per cycle: Non-equidistant

In the non-equidistant simulations (Figure 10) the collisions have higher temporal dispersion than in the equi-distant simulations, which are more bursty. This is thanks to the non-equidistant network, which disperses the cores-to-memory access times. In comparison to the collisions in the equi-distant section above, here we can clearly see major reduction of collisions with frequency increase for all benchmarks.

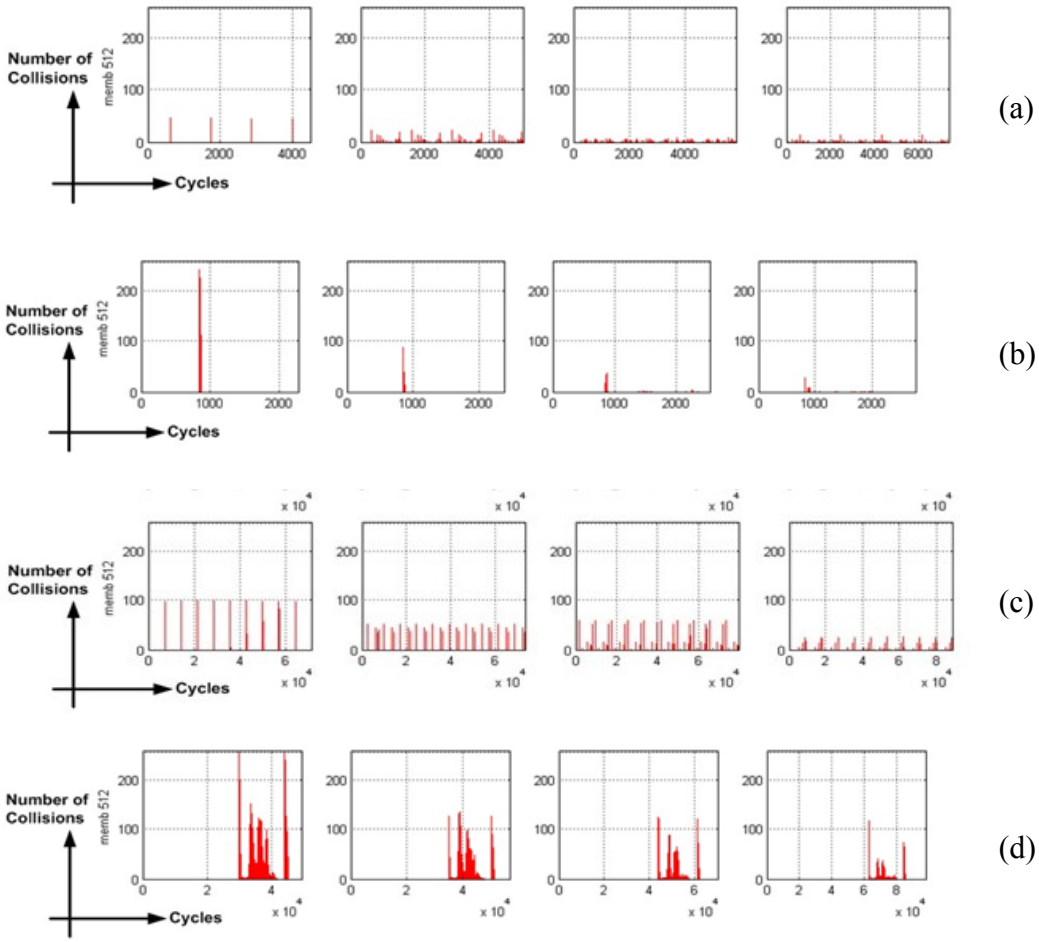

Figure 10: Number Collisions per-Cycle in the non-equidistant architecture: (a) Parallel, (b) Mandelbrot, (c) Linear solver, (d) JPEG

6.3 Average Core activity

Figure 11 presents the cumulative core activity in the non-equidistant architecture simulation averaged over all 256 cores. In each chart, time in each of the activities (busy, wait, collision, idle) is plotted as bars vs. frequency. In contrast with all charts above, the performance here is presented in execution time rather than in cycles, allowing comparison of execution of different frequencies on the same scale. Note that the time scale is not the same in all charts. Clearly, the higher the frequency, the lower is the execution time.

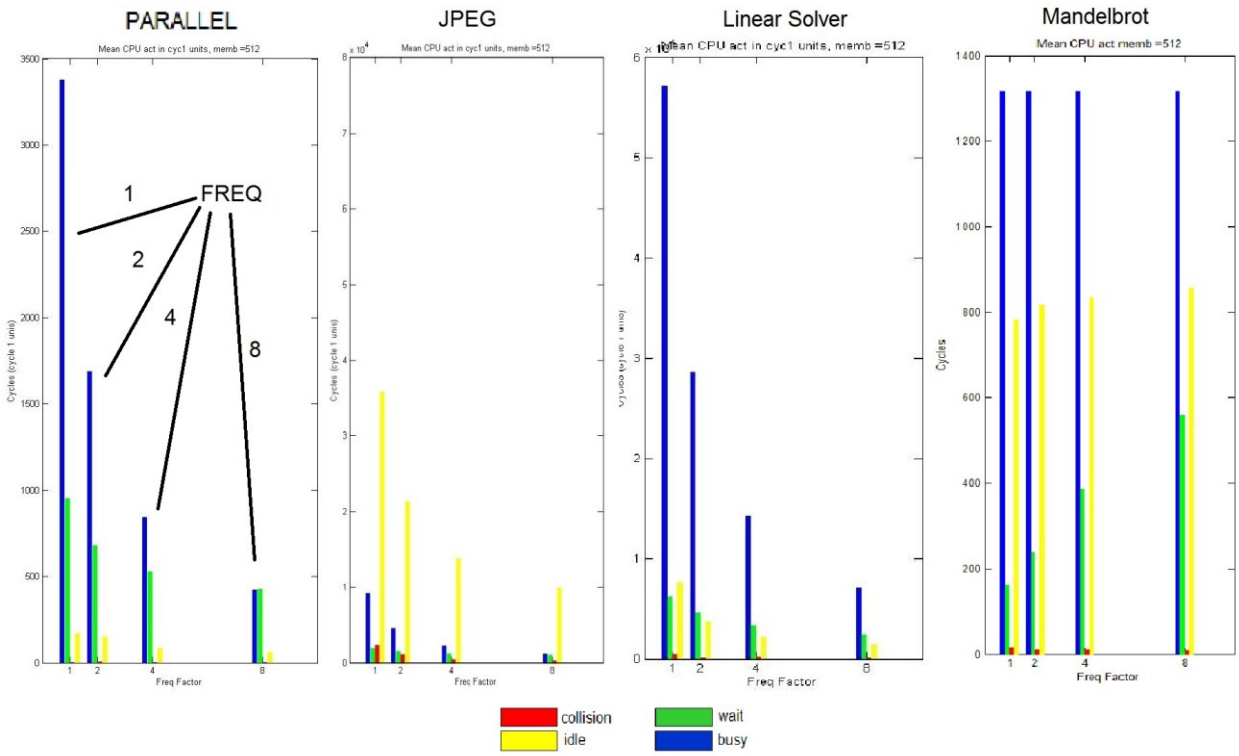

Figure 11: Average core activity

6.4 Total Wait Time

Wait time is an indicator of the dependence of network efficiency on frequency. Figure 12 shows the memory access wait time (in units of freq1 cycle time) for each frequency factor in the non-equidistant architecture. The memory access time improvement in the non-equidistant case is clearly visible here, improving memory access time in the range 45%-61%. The improvement is more significant in highly parallel programs, like the linear solver. In the JPEG program, which has a long serial section, the improvement is smaller.

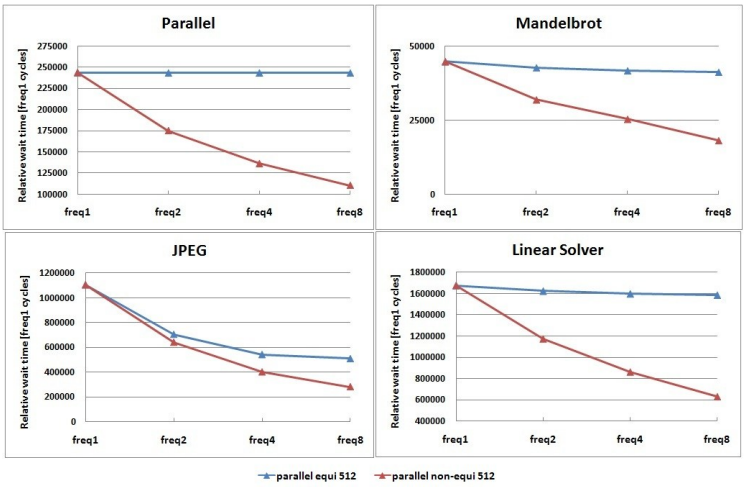

Figure 12: Benchmarks relative memory access wait time (equi-distant vs. non-equidistant)

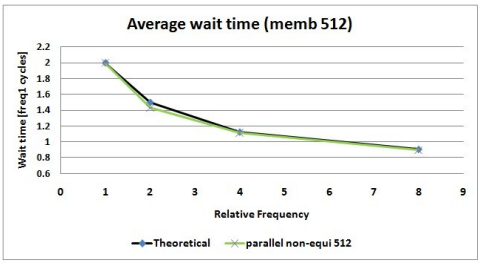

Figure 13: Average wait time comparison

Figure 13 shows the average memory access time (excluding collisions) for the synthetic Parallel benchmark and the theoretical average of the non-equidistant architecture. We can see that the result of the benchmark is in accordance with the theoretical calculations (section 3.2).

6.5 Equi-distant versus Non-equidistant Wait time comparison

Figure 14 shows the wait time in the equi-distant and non-equidistant simulations in each frequency with 512 memory banks. The four bars in each architecture represent the four frequency factor values from left to right: freq1, freq2, freq4, freq8. The figure emphasizes the difference in access time between the equi-distant and non-equidistant architectures. We can also conclude that the more parallel the benchmark, the greater contribution the non-equidistant approach gives. Note that the maximal reduction of wait time in non-equidistant memory relative to the equi-distant case is 61% in the case of the linear solver.

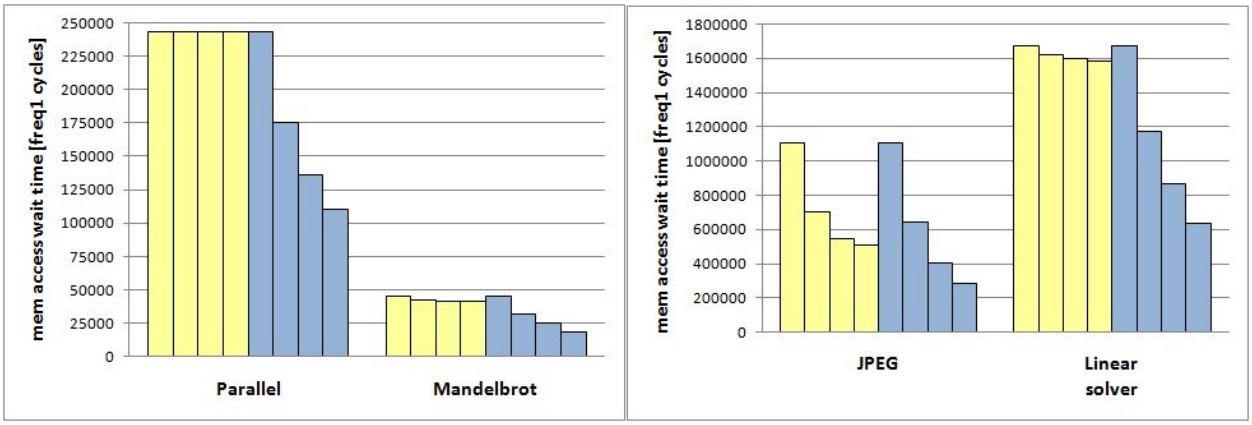

Figure 14: Memory access wait time comparison with 512 memory banks (each set of 4 bars corresponds to 4 frequency levels)

6.6 Speed Up

In this section we compare the benchmarks execution speed-up relative to the base configuration of freq1. Figure 15 shows the speed up for the non-equidistant architecture with 512 memory banks. We see that substantial speed up is achieved for all types of benchmarks thanks to the increase in frequency. The most speed up is achieved in the Mandelbrot program, probably thanks to having shared variables, while the JPEG program gains less speed-up than others, most likely due to its large serial sections.

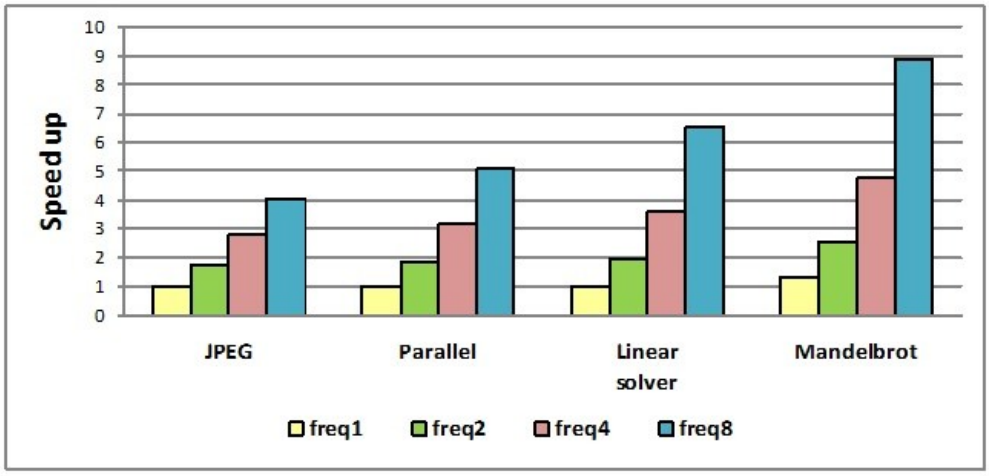

Figure 15: Benchmarks Speed-Up (non-equi-distance) with 512 memory banks

6.7 Differential memory access Speed-Up contribution

System speed up comes from different factors that roughly can be divided into three parts: core internal speed up, non-equidistant memory access (path diversity) and memory banks speed up.

In this section we identify the contribution of the following factors on speed up: Network non-equi-distant architecture and memory banks frequency. This analysis distinguishes these factors from the obvious contribution of increasing the cores frequency.

The factors are calculated according to the following equations. For Network non-equidistant architecture speed-up: Cycles saved = (wait cycles, non-equi) – (wait cycles, equi), relative to freq1 time units. For memory banks frequency speed-up: Cycles saved = (wait cycles at freq1) – (wait cycles at freqx), equi-distant, relative to freq1 time units.

Figure 16 shows the differential speed-up contribution of the two factors mentioned above. The contribution of the non-equidistant architecture definitely stands out as the most effective, especially in high frequency. We can also observe that the more parallel the program, the more the non-equidistant architecture is effective, while the memory banks frequency has more effect in the less parallel programs (JPEG program).

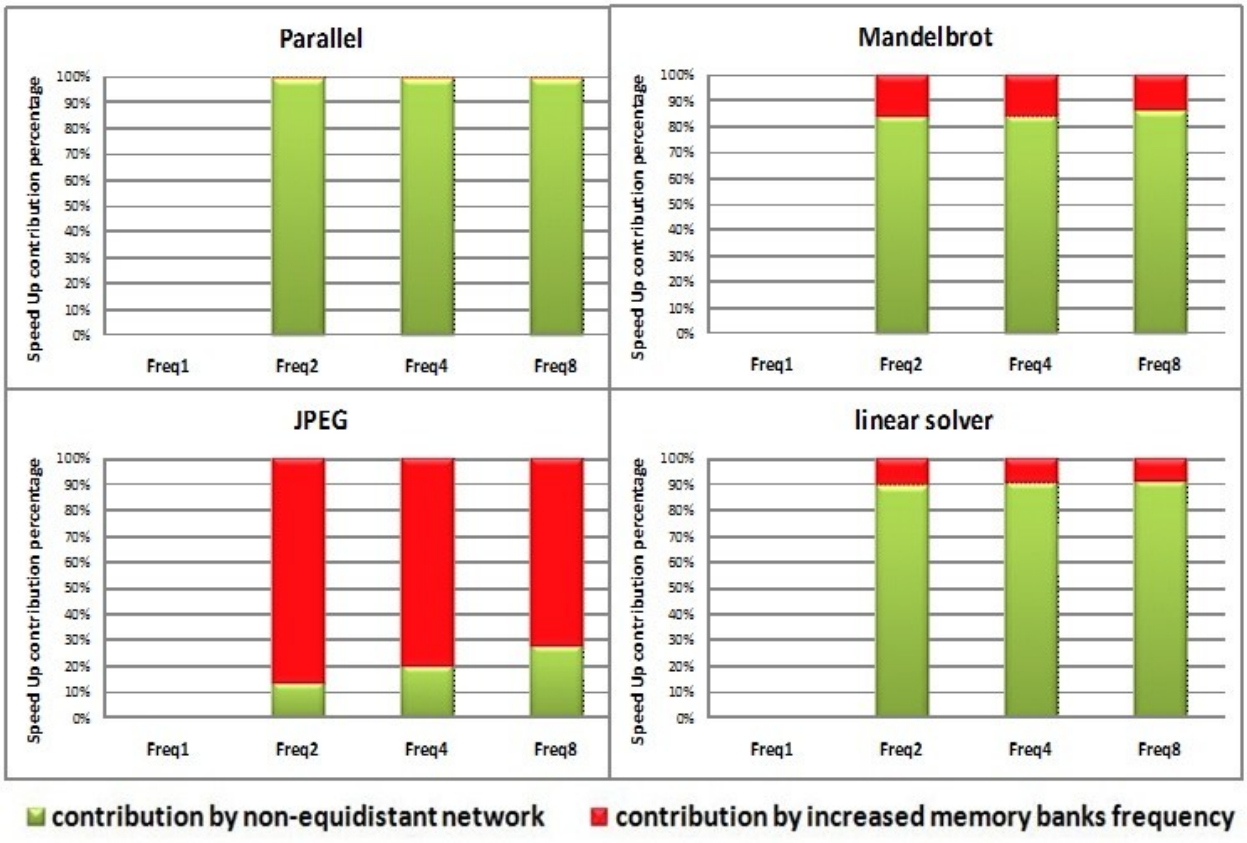

Figure 16: Differential speed up contribution (non-equidistant)

6.8 Network efficiency calculation

Here we give an illustration of the network efficiency, which is calculated by:

(successful memory access time)/(total memory access time).

In Figure 17 below we can observe how increasing the frequency and creating a non-equidistant architecture contributes to collisions reduction in memory accesses, especially in highly collided programs, like JPEG. In the Parallel program, for example, due to low number of collisions even in Freq1, we do not see any significant improvement. The increased frequency also reduces collision penalty.

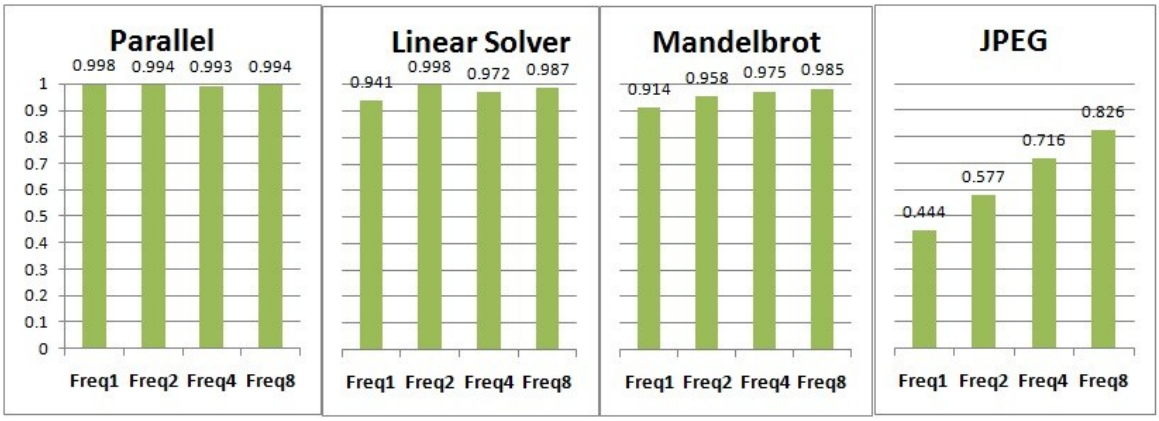

Figure 17: Network efficiency (non-equi-distant)

7 Conclusions and Future research

This paper studies the benefits of non-equidistant memory in the HyperCoreX architecture. The immediate benefit is the reduction of memory access time, as shown in sections 3.2, 6.1 and 6.5. The results clearly show that access time percentage in the equi-distant simulations always increases considerably with frequency increase, because the network is equi-distant and allows no improvement in access-time. In the non-equidistant simulations, access-time percentage increases mildly and sometimes even decreases with frequency increase. Further benefit is derived from the fact that the non-equidistant architecture provides path latency diversity between cores and memory banks. This feature has several advantages which reduce the probability of collisions, further improving speed-up. This is well demonstrated in the graphs of collisions per cycle in section 6.2.2.

The path latency diversity causes a time-division-multiplexing effect, which is very important in data-cacheless many-core architectures. Frequently during program execution, because of task dependencies, many cores begin executing instances of the same duplicable task at the same cycle. Consequently, because the cores are homogeneous, they also reach a memory instruction at the same cycle. Thus, several cores might start a write access to the same memory bank at the same cycle, which causes a collision in an equi-distant architecture. In the non-equidistant architecture, in contrast, thanks to path latency diversity, there might not be a collision at all when employing higher frequencies.

The other effect of path latency diversity is the fine granularity of core activity. Because of the time-division-multiplexing effect, cores that have started executing the same duplicable task at the same cycle are no longer in cycle-by-cycle synchronization within the task instances after the memory access, so their memory accesses are timed more randomly, thus reducing collisions. In addition, the cores finish their tasks out of sync, so that new tasks begin out of sync. This reduces collisions even further.

The simulated parameters for measuring speed-up are memory bank frequency and non-equidistant network (section 6.7). The non-equidistant network was found to be the greatest contributor to performance, not only shortening memory access time, but also shortening collision penalty time. In a highly collided program, speed-up is larger for fast frequencies than for a parallel program, because of core-to-bank path latency diversity (low frequencies have larger wait time penalty).

Possible future research may address the following questions. First, the effects of a non-ideal scheduler should be investigated. Next, while in this research we studied only synchronous networks with an integral number of clock cycles per access, asynchronous access networks may provide finer resolution and possibly improved performance. Various modes of network implementations, such as different topologies, may be considered. Last, it may be advantageous to employ a non-power-of-two number of memory banks, to further reduce conflict probabilities.

8 Acknowledgements

This research was supported in part by a research grant from Intel Corporation. The graphical help of Mrs. Friedman is greatly appreciated. Plurality Ltd. provided traces for three benchmark programs.

9 References

[1] A. Agarwal, The Tile processor: A 64-core multicore for embedded processing, HPEC Workshop, 2007.
[2] J. Kuskin, D. Ofelt, The Stanford flash multiprocessor, ISCA, 1998
[3] Agarwal et al., The MIT Alewife Machine: Architecture and Performance, ISCA, 1995
[4] Agarwal et al., The MIT Alewife Machine: A large-scale distributed-memory multiprocessor, *in* Scalable shared memory multiprocessors, M. Dubois and S.S. Thakkar (eds.), 1991.
[5] DR Fan, N Yuan, JC Zhang et al., Godson-T: An Efficient Many-Core Architecture for Parallel Program Executions, J. Comp. Sci. Tech., 2009,
[6] J. del Cuvillo, W. Zhu, Landing OpenMP on Cyclops64: An Efficient Mapping of OpenMP to a ManyCore System on a Chip, ACM Computing Surveys, 2006.
[7] S.L. Scott, Synchronization and Communication in the T3E Multiprocessor, ASPLOS-VII, 1996.
[8] V. Baumgarte and G. Ehlers, PACT XPP—A Self-Reconfigurable Data Processing Architecture, J. Supercomputing, 2003.
[9] D Lenoski and J Laudon, The Stanford DASH multiprocessor, IEEE Computer, 1992.
[10] D. Towner and G. Panesar, Debugging and Verification of Parallel Systems — the picoChipWay, Communicating Process Architectures, 2004.

[11] L. Seiler, D, Carmean, Larrabee: A Many-Core x86 Architecture for Visual Computing, ACM Trans. Graphics, 27(3), 2008
[12] S. Y. Han et al., A Massively Parallel Multithreaded Architecture: DAVRID, Proc. Int. Conf. Computer Design, 1994.
[13] J. Kim, Y. Nah and S. Han, Hybrid Multithreaded Architecture with Symmetric Multiprocessors. TR Seoul Nat. Univ., 2007.
[14] Reinhardt et al. Decoupled Hardware Support for Distributed Shared Memory. ISCA 1996.
[15] I. Buck and T. Foley, Brook for GPUs: Stream Computing on Graphics Hardware, ACM SIGGRAPH, 2004.
[16] Plurality, Ltd., HyperCore Architecture, http://www.plurality.com
[17] M. B. Taylor and W. Lee, Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams, ISCA, 2004
[18] Agarwal, Limits on Interconnection Network Performance. IEEE Tras. Par. Dist. Syst. 1991.
[19] Johnson and Agarwal, The Impact of Communication Locality on Large-Scale Multiprocessor Performance, 1992, ACM SIGARCH Computer Architecture News.
[20] Changkyu Kim and Doug Burger, An adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches, 2002, ACM ISBN
[21] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar, Distance Associativity for High-Performance Energy-Efficient Non-Uniform Cache Architectures, 2003, IEEE, 36th International Symposium on Microarchitecture
[22] Allan Gottlieb and Ralph Grishman, The NYU Ultracomputer – Designing a MIMD, Shared-Memory Parallel Machine, 1982, IEEE, Computer Society.
[23] Z. Guz, I. Keidar, A. Kolodny, and U. Weiser, "Utilizing shared data in chip Multiprocessors with the Nahalal architecture", the 20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), special track on Hardware and Software Techniques to Improve the Programmability of Multicore Machines, June 2008.