# Addressing Bandwidth Contention in SMT Multicores Through Scheduling


UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Josué Feliu
Julio Sahuquillo
Salvador Petit
José Duato


## 1. INTRODUCTION

**Multicore processors** are the common implementation for high performance microprocessors. The current industry trend increases the core count in each new microprocessor generation, so stressing the pressure on the **main memory bandwidth**, which conforms one of the major **performance bottlenecks** of these processors.

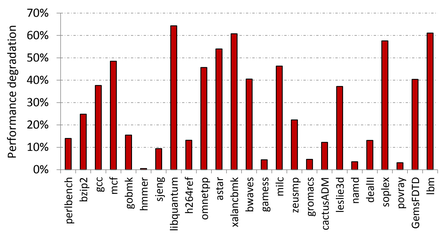
In addition, **multithreading** is also becoming the common processor implementation. The different processes running concurrently on an **SMT** core can issue instructions in the same cycle, which are executed sharing some of the core resources. Among them, the L1 cache and, particularly, the **L1 bandwidth** are **critical** for the overall processor performance.

In short, most research work regarding scheduling has tackled:
i) main memory and LLC bandwidth contention, or
ii) resource sharing on SMT cores.
However, to the best of our knowledge, this is the **first proposal** that combines both approaches to **deal with bandwidth contention in multithreaded CMPs**.

The performance evaluation is performed in an **Intel Xeon E5645 processor**, which implements **six dual-threaded SMT cores**, with private L1 and L2 caches per core and a shared LLC. The system runs a Linux distribution with kernel 3.11.4.

## 2. CAUSES OF PERFORMANCE DEGRADATION

### Main-memory bandwidth contention analysis

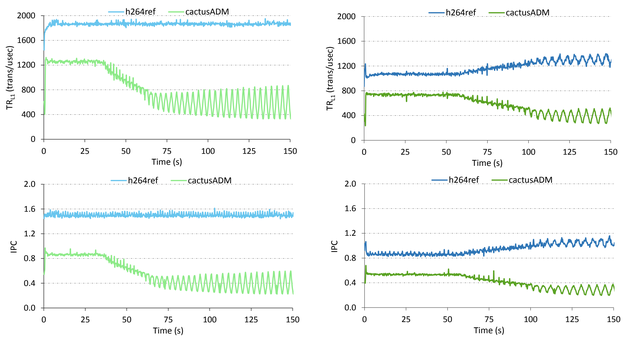



*Experiment 1:*
Each benchmark is concurrently launched with five instances of a memory-hungry microbenchmark with a standalone main memory transaction rate ($TR_{MM}$) of 55 t/usec.

*Observation 1:*
**Half of the benchmarks suffer a performance degradation above 30% due to main memory bandwidth constraints**.

Fig. 1: IPC degradation due to main memory bandwidth contention.

### L1 bandwidth contention analysis




(a) Standalone execution (b) Simultaneous execution on the same core

Fig. 2: L1 bandwidth and IPC of *h264ref* and *cactusADM*.

*Experiment 2:*
The IPC and transaction rate in L1 ($TR_{L1}$) of two benchmarks are compared when they run:
i) in stand-alone execution (Figure 2a)
ii) simultaneously on the same core (Figure 2b)

*Observation 2:*
**When the processes run simultaneously, their performance can drop near 40%**, as can be observed until second 50.

*Observation 3:*
There is a **connection between the L1 bandwidth and IPC of the co-runners** [1]. For instance, when *cactusADM* reduces its $TR_{L1}$ (implicit in its behavior), more bandwidth is available to *h264ref*, resulting in a growth of its $TR_{L1}$ and IPC.

## 3. PROPOSED SCHEDULER

The performance degradation analysis shows the suitability of using a bandwidth-aware scheduling algorithm to mitigate contention and improve performance. **The proposed algorithm consists of two main sections: process selection and process allocation**.

**Algorithm 1** Bandwidth-Aware Scheduler

1: Calculate:
$WK\_AVG\_TR_{MM} = \frac{\sum_{p=0}^{N}(AVG\_TR_{MM}^{p} * T^{p})}{\sum_{p=0}^{N} T^{p}} * \#CPUs$
2: while there are unfinished processes do
3: Gather $TR_{MM}$ and $TR_{L1}$ of the processes run the last quantum
4: Select the process p at the process queue head and set:
$BW_{remain} = WK\_AVG\_TR_{MM} - TR_{MM}^{p}$
$CPU_{remain} = \#CPUs - 1$
5: while # selected process < # CPUs do
6: Select the processes p that maximizes:
$FITNESS(p) = \frac{1}{\frac{BW_{Remain}}{CPU_{Remain}} - TR_{MM}^{p}}$
7: Update $BW_{Remain}$ and $CPU_{Remain}$
8: end while
9: Sort the selected processes in ascending $TR_{L1}$
10: while there are unallocated processes do
11: Select the processes $P_{head}$ and $P_{tail}$ with maximum and minimum bandwidth requirements
12: Assign $P_{head}$ and $P_{tail}$ to the same core
13: end while
14: end while

Before running a workload, the scheduler calculates the WK_AVG_TR_MM, which represents the overall target $TR_{MM}$ that should be achieved by the processes running each quantum to balance the main memory transactions over the workload execution.

Main memory and L1 bandwidth of the processes are updated each quantum using performance counters.

The **process selection** policy **deals with main memory bandwidth contention**, **selecting the** proper set of **processes that will run the following quantum with the goal of balancing the overall memory requests over the execution time** of the workload **to minimize the contention.** First, the process not executed for longer is selected to avoid process starvation. Then, the remaining processes are selected using the fitness function. This function quantifies the gap between the $TR_{MM}$ required by a given process and the average bandwidth remaining for each unallocated hardware thread.

The **process allocation** policy **assigns each selected process to a core dealing with L1 bandwidth contention.** The goal of the policy is to balance the L1 requests that each L1 cache has to serve each quantum. In this way, the L1 bandwidth contention is minimized and the performance enhanced.
Since the experimental platform implements dual-threaded cores, the processes can be easily allocated by sorting the processes according to its $TR_{L1}$ and then, reiteratively, assigning the processes with highest and lowest bandwidth utilization to the same core.

## 4. SCHEDULER EVALUATION

The devised algorithm has been implemented in a user-level scheduler, which uses: I) the **Linux system calls** to **determine the processes that will run the next quantum**, and II) the **core affinity attributes** to **determine the target core of each process**. To evaluate the performance of the proposal, a set of **ten 24-benchmark mixes** was designed. The performance of the devised algorithm is compared with that achieved by the Linux scheduler.

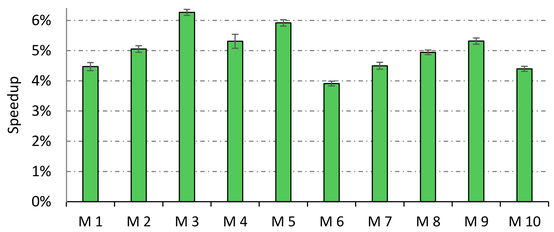



Fig. 3: Speedup of the average IPC relative to Linux.

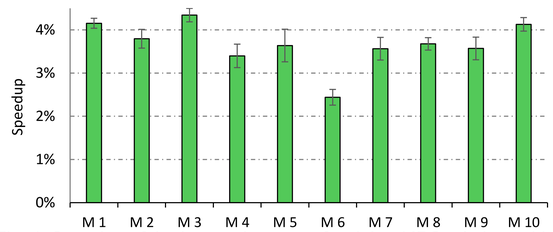



Fig. 4: Speedup of the harmonic mean of weighted IPC relative to Linux.

**Figure 3** presents the **speedups** achieved by the proposed scheduler using the **average IPC metric with respect to Linux**. **The proposed scheduler improves Linux in all the mixes**, with speedups ranging between 4% to above 6%. Since the average IPC metric measures throughput, **the results show that the devised scheduler mitigates the bandwidth contention**, which results in a significant performance increase.
**Figure 4** presents the **speedups** of the **harmonic mean of weighted IPC metric*** achieved by the proposed scheduler **with respect to Linux**. Note that **this metric considers fairness** additionally to performance since the harmonic mean tends to decrease quickly as the gap between the speedups grows. Thus, the achieved speedups, ranging from around 2.5% to above 4%, demonstrate that **the devised scheduler not only reaches greater performance, but also works fairer than the Linux scheduler.**

$^{*}H\_M\_W\_IPC = \frac{N}{\sum \frac{Single\ IPC_i}{IPC_i}}$


[1] J. Feliu, J. Sahuquillo, S. Petit, J. Duato , "*L1-Bandwidth Aware Thread Allocation in Multicore SMT Processors*", PACT 2013.



28th International Conference on Supercomputing (ICS)
Munich, Germany, 2014



GOBIERNO DE ESPAÑA MINISTERIO DE ECONOMIA Y COMPETITIVIDAD

GAP Parallel Architectures Group