Proposal of PMCs for tracking worst-case behaviour in multicores: the case of the NGMP

Document Information

Author	Francisco J. Cazorla, Mikel Fenandez, Javier Jalle
Contributors	Luca Fossati
Reviewer	Jaume Abella
Keywords	Performance Monitoring Counters, worst case execution time

Table of Contents

1 Introduction

The main goal of this document is to define a PMC infrastructure for a NGMP-like processor that will help providing insight information about the effect of inter-task interference among corunning tasks (a.k.a. corunners). This document takes as baseline D1(1) who covered an analysis of the PMCs of the latest multicore architectures of some of the major chip providers in the market.

2 Reference architecture

The reference processor architecture we are going to use is the NGMP.

We note that the NGMP has a nominal implementation (the actual) and several proposal for implementation. We have identified in each not-fully decided design the particular NGMP implementation we consider.

2.1 Reference NGMP architecture

Our reference NGMP architecture is not the actual NGMP architecture since it is going to be upgraded with new features. The actual implementation does not implement SPLIT transactions support on the processor bus. Without it, all inter-task interferences occur on the processor bus, since it is acting as a lock for the whole system, allowing only one request to be accessing the shared memory hierarchy.

The future NGMP will have SPLIT transactions support on the processor bus, which allows up to one request per-core to be issued to the shared memory hierarchy, i.e. the L2 cache. The bus will only split on load misses, since for hits and stores the L2 is able to respond with a reasonable amount of wait states. For store misses, the L2 still responds immediately since it can handle the request by its own.

To be able to handle one request per-core on the L2 it is required a LockupFree cache (Gaisler calls it multiport, since their idea is to implement a 4-layer bus in the future with one port per core in the cache), that will allow the L2 to manage hit-under-miss and miss-under-miss, which means that when a miss is processed other requests to the L2 can be served: hits and misses. Another feature of the L2 cache is the variable latency depending on the type of request of the actual request and the previous one. Since the information about the time-window in which the previous request can affect the actual one it is not clearly stated by Gaisler, we will assume only that there is a different latency depending on the type of request, i.e, load or store.

The L2 sends the miss to the memory controller one at a time, and in case there are more than one outstanding miss, it is saved on a FIFO queue. The memory controller is not intended to serve request in parallel and will only serve one request at a time, in a close-page manner. If we consider the FIFO queue of the L2 as part of the memory controller, it behaves as a FIFO close-page memory controller that behaves almost like the round-robin processor bus that serializes the requests to the L2.

The last interesting architectural feature that affects the timing is the write-buffer present in each core. This buffer will hold store requests until they are resolved without stalling the processor. The processor will only be stalled if a store finds the write-buffer full or a load operation finds the write-buffer non-empty (to keep memory consistency). The write-buffer has a big effect on the interference accounting, since not all the interference that will happen on the different architectural components (bus, cache, memory,…) will be relevant, for instance, if core 0 is being interfered on the access to the processor bus, but it is a store inside the store-buffer, core 0 is not stalled, thus being immune to that interference.

3 Inter-Task Interference CPI stack

Our proposal moves around the concept of CPIstask. The CPI stack relates architectural components with performance counters to show which CPU component is generating stalls and hence the program to consume cycles in its execution.

A CPIstack for given core i, in the classic view, breaks up the total execution cycles, t_i, into useful cycles, u_i, and cycles stalled, s_i, or lost due to cache misses, pipeline stalls, brach-misspreddiction, etc… . In an ideal processor (IPC=1), all cycles are useful cycles, i.e. $s_i = 0$.

$$t_i = u_i + s_i$$

Our goal is identifying contention and conflicts in the access to hardware shared resources in a multicore architecture. We call those conflicts inter-task interferences. With this idea in mind, we build an inter-task interference CPI stack(*itiCPIstack*). Our goal with the *itiCPIstack* is to classify the stalled cycles that the processor consumes into the cycles doing local activities (e.g. handling a long latency FPU instruction, pipeline stalls, etc…), which we name *local stall*, l_i; and the cycles consumed in the external hardware shared resources, or *external stall*, e_i.

$$s_i = l_i + e_i$$

In the former category, local-stall cycles, we do not further subdivide stall cycles since in our case we are interesting on breaking down the effect of inter-task interferences, which only affects the external stall cycles. In a classic CPI stack local-stall cycles can be classified as FPU stalls, branch-misspreddiction, ….

In the latter category, external-stall cycles, we will classify the cycles consumed on each hardware shared resource. For each resource n of all N_R resources, we create a subcategory called inter-task interferences, int_i^n, which covers that cycles in which the processor i was stalled due to some inter-task interference activity generated by another core j and a subcategory called intrinsic latency or useful cycles, u_i^n, which covers the cycles spent actually using the resource, as it will be when running in isolation, i.e., without contention and conflicts.

$$e_i = \sum_{n=0}^{N_R-1} u_i^n + int_i^n$$

To increase the information available on the *itiCPIstack*, we classify the total interference for a given core on each resource, int_i^n, according to the core that is generating it, i.e., the interference suffered by core i on resource n because of each core j of all N_C cores is defined as $int_{i\leftarrow j}^n$, so that the total interference on a resource can be expressed as:

$$int_i^n = \sum_{j=0}^{N_C-1} int_{i\leftarrow j}^n$$

The final *itiCPIstack* is given by:

$$t_i = u_i + l_i + \sum_{n=0}^{N_R-1} \left[u_i^n + \sum_{j=0}^{N_C-1} int_{i\leftarrow j}^n \right]$$

3.1 itiCPIstack in the NGMP

To build the *itiCPIstack* we need the following terms:

1. t_i, or total execution cycles for each core i.
2. u_i, or useful core cycles for each core i.
3. l_i, or local stall cycles for each core i.
4. u_i^n, or useful cycles for each core i in each resource n.
5. $int_{i \leftarrow j}^n$, or inter-task interference cycles for each core i in each resource n and each interfering core j.

The first three terms are already available through NGMP PMCs.

1. Term 1 has its own PMC.
2. Term 2 matches with instruction count, since in a stall-free NGMP, IPC $=1$.
    a. Note that while this is true for many instructions, there are a few multi-cycle instructions, such as mult / div and the various FPU instructions.
    b. A solution to this is having a cycle counter that only counts those sycles in which the fetch stall signal is not active.
3. Term 3 can be obtained by subtracting term 2 from term 1, which gives us the stall cycles, and further substracting icache and dcache hold PMCs which will give us the stall cycles not caused by memory accesses, i.e., local stall. Note that there is an overlap between ic-hold and dc-hold, which can be used to increase the precision of this term.
4. Terms 4 and 5 need to be accounted for each resource and will require some hardware support in the form of new PMCs, since they cannot be indirectly obtained from available PMCs. In the NGMP the shared resources that we are going to consider are:
    - AMBA processor bus.
    - Shared L2 cache (or Last Level Cache, LL$).
    - Memory controller.

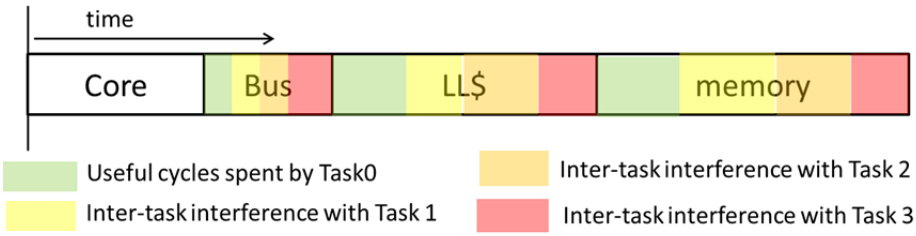

Figure *1 Detailed itiCPIstack of the NGMP*

The *itiCPIstack* will look like the one in Figure 1. The core component covers: 1) all the cycles in which the program is proceeding with no stalls, i.e. useful cycles; and 2) all the cycles in which the pipeline is stalled (no instruction can be committed) due to a local stall, for instance a FP operation blocking the processor due to its long latency.

The bus component covers the cycles in which the processor is stalled due to the processing of a request in the bus. This processing covers the actual transfer time of the request, i.e., intrinsic latency, and the inter-task interference delay.

Similarly the last-level cache (LL$) component covers the cycles in which the pipeline is stalled due to a cache request being processed. In the case of loads and

instructions they are blocking. In the case of stores, this situation happens when the store buffer is full. Also the LL$ covers the inter-task interference that corresponds to the misses caused because other cores evicted the data of the actual core.

The memory part covers the cycles actually accessing the memory, and the cycles accounting for the inter-task interference which correspond to the cycles waiting on the FIFO queue.

4 Accounting the interference on the NGMP

We envision having an inter-task interference module (*iti-module*), which is similar to a statistics unit, which concentrates the main logic required for computing the CPIstack. The *iti-module* is connected to other components that export information about some events needed by the *iti-module*. Figure 3, shows how the *iti-module* should be connected to other modules for the case of the NGMP, in which the main shared resources are the bus, the LLC and the memory controller.

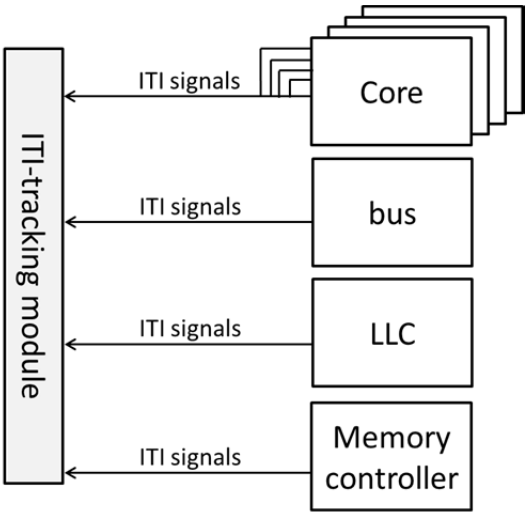

Figure 3 *iti-module.*

Each resource, except the core, will have to provide two pieces of information for each cycle: 1) the core that is using the resource, which will allow to track the useful cycles; 2) the cores that are waiting for the resource, which will allow to track if a core is interfered and in case it is so, the actual core using the resource will be assigned as the interferer.

4.1 Processor cores

Because of the existing write-buffer, not all interference and working cycles in a resource of the system have to be accounted. For instance, if a store is sent to the memory hierarchy by the write-buffer and the core is not stalled and can continue executing, all the interference and intrinsic latency cycles that the store will suffer are not going to affect the final execution cycles of the core. Our *itiCPIstack* infrastructure needs to know at any point in time is whether the processor core is stalled to be able to count the effective cycles of interference or intrinsic latency of each resource, i.e. the cycles that contribute to the final execution time. For that purpose each core has to provide a signal that says if the core is stalled or not.

A processor can be stalled because of different reasons:

- Fetch stage, in case of an instruction cache miss, cannot feed the next pipeline stages, thus stalling the pipeline.
- The same happens in the memory stage when a load instruction accessing the data cache misses.

- Also in the memory stage if a store finds the store-buffer full, stalls the pipeline.
- When the integer/floating-point pipelines need more cycles to execute an instruction, this generates backpressure that stalls the pipeline.

Several of these situations can happen at the same time, so we need to choose a criteria to define when the processor is stalled. In our case we choose to consider the stalled when the fetch stage is not able to fetch a new instruction. This will happen because of icache misses or backpressure, which can be due to *dcache* misses or pipeline stalls.

4.2 AMBA bus

One of the main conclusions in [JJQF2014] is that a single AMBA request from a given task can block other tasks' accesses to the bus. This is so, because one request can be locked by the master sending it for an unbounded period of time.
Hence, our approach to measure inter-task interferences is to measure the time that a given task waits for the other when it tries to get access to the bus. Once a master (core) is ready to send a request it raises the HBUSREQ signal in AMBA AHB. When the arbiter grants access to that master is asserts the HGRANT signal for that master and puts the master id in the HMASTER signal. For a given task A, the time $t_{grant\text{-}up}$-- $t_{req\text{-}down}$ gives the time a request from a given task T_A is waiting to get access to the bus ($t_{A\text{-}iti}$). This is the waiting time caused by inter-task interference.

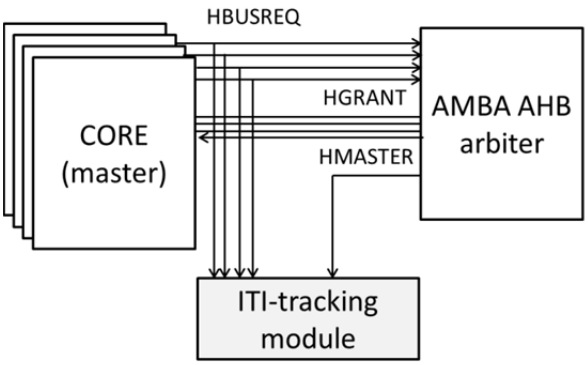

Figure 4 *Performance vs energy efficiency comparative*

We propose to send HBUSREQ signals from each master and the HMASTER signal from the arbiter to the *iti-module*. By checking these signals the *iti-module* can infer which master is using the bus, thus the useful cycles, and the time a master has been waiting for another master.

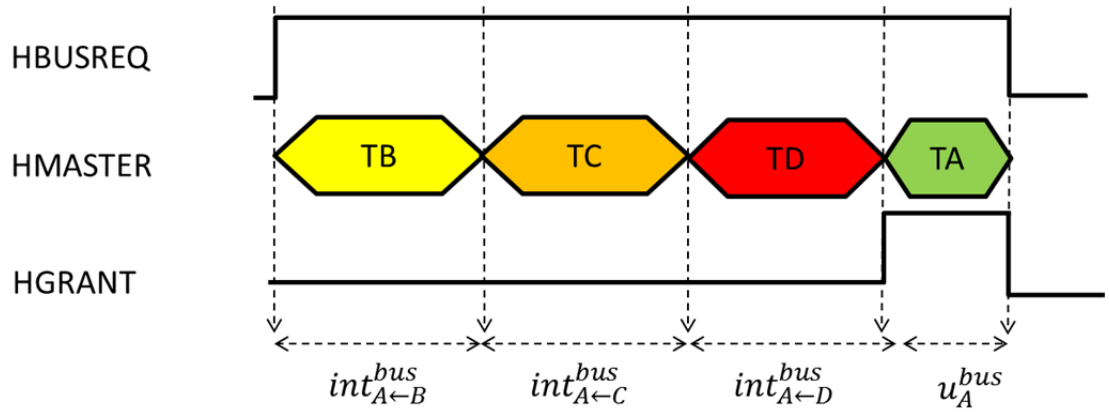

Figure 5 *Timing of the different signals for the AMBA bus.*

Figure 5 shows an example for 4 masters (cores) executing tasks T_A, T_B, T_C and T_D respectively. When T_A sends a request, the time the request signal is up and the grant signal from the arbiter is low there is inter-task interference. The *iti-module* has *NxN* registers. Each master has associated N registers, N-1 of which hold the ITI interference delay it suffers from the other N-1 masters. The last register stores the intrinsic latency to process the request.

4.3 Memory

In the case of the memory, we assume that it consists of the FIFO queue (which in fact corresponds to the L2 miss queue), the memory bus and the actual NGMP memory controller, which translates an AMBA request into DRAM commands. When a request from a core i arrives to the FIFO queue, if it is empty, it is put on the top of the FIFO and accesses the memory immediately consuming the intrinsic latency or useful cycles u_i^{mem}. Otherwise, when other requests are on the FIFO, it has to wait for them to finish since the memory bus and memory controller only accept one request at a time, i.e. the cycles on the memory will be $int_i^{mem} + u_i^{mem}$.The former addend is the time the request waits to get access to the memory controller, the latter is the time the request takes to be processed once it is granted access and it cannot be further delayed by any preceding request.

The request at the top of the FIFO queue is the one accessing the memory and the rest of the request on the FIFO queue are being interfered by the top one. In this case, each position in the FIFO queue will send a signal to the *iti-module* with the core id of the request in that position, if any. The *iti-module* will consider useful cycle in memory for core i, if that core is on the FIFO's top position. If there is any other core j on the FIFO queue, the *iti-module* will consider an interference cycle on the memory caused by core i.

This forces to propagate the core id up to the memory controller, since that knowledge is lost once you go out of the AMBA processor bus.

4.4 Shared last level cache

There are two types of inter-task interferences in the access to the cache.

- Access ITI happens when the request of a task wants to use the cache and is affected by the access of a previous request from another task. For instance, this happens due to the variable hit/miss latency that depends on the previous access.

In our case we will neglect this interference, since its effect is small compared to the content ITI explained below.

- Content ITI happens when a task suffers a miss in the access to cache due to the fact that another task evicts its data.

If the cache is partitioned using way partitioning, the task suffers only access inter-task interferences. However, we expect this to be small and we neglect its effect.

To account intrinsic latency or useful cycles in the cache for the core under analysis,i, we count the number of hits, $hits_i$, and misses, $misses_i$, when the core is stalled, by signaling hits and misses to the *iti-module*. When computing the *itiCPIstack*, the intrinsic latency or useful cycles of the cache will be computed considering the hit-penalty and the miss-penalty in the cache:

$$u_i^{L2\text{-hits}} = hits_i \times \text{hit-penalty}$$
$$u_i^{L2\text{-misses}} = misses_i \times \text{miss-penalty}$$

In case the cache is not partitioned, the content ITI has to be computed, which we call $int_{i \leftarrow j}^{L2}$. To compute that value for each core we need two things: (1) using per-cache-line owner bits that allow determining the number of inter-task evictions one task suffers due to other co-running tasks. And second, using an Auxiliary Tag Directory (ATD) that allows identifying inter-task interference misses and better approximating the cycles each task is waiting due ITI with other tasks in cache contents, which can only be exactly be derived for direct mapped caches. Since, both thing incur into a high hardware cost, we propose another two alternatives with smaller cost.

4.4.1 Non-partitioned cache: Auxiliary Tag Directory (ATD) and per-cache-line owner bits

Under this approach we add an ATD for the core under analysis. The ATD behaves equally to the tag directory of the cache, but only the core under analysis is able to modify it. The ATD then reflects the tag directory of the cache, when the core under analysis runs in isolation. This means that if a request from that core hits in the ATD and misses on the cache, it is an *iti-miss*. The ATD provides very precise information of when a task suffers an ITI content interference but provides no information about the task causing the inter-task interference. That is, the ATD allows deriving the exact amount of cycles each task waits due to inter-task interference in cache contents to other tasks but not the specific task causing it (interfering task). To derive the interference cycles, int_i^{L2}, the cache will signal when a *iti-miss* happens and when that *iti-miss* is resolved. Since only one request per core is allowed at a time, all the cycles spent in the memory and the cache will be accounted as interference.

Now we have the total amount of interference suffered, we need to distribute it through the interferers or cores. Finding the precise interfering task is complex for a set associative cache, where data allocated to a given memory location has a unique set in which it can be located but *W* different ways. In this case, the core that evicts the actual cache-line is not the only interferer, but also all the cores that accessed that line and modified the LRU state. Every time a core is evicting a cache-line, is generating a LRU accesses to every *W* cache-line owners in that cache-set. Also we cannot know if the evicted data is going to be reused again in the future or not.

To be able to distribute the total interference to each interfering core, we will count the inter-task evictions that each core generates. For that purpose, we add $\log_2 Nc$ bits to each cache line. These bits provide the *id* of the task that allocated that cache line. We further have Nc x Nc registers storing inter-task LRU accesses in the cache. When a core *j* suffers a miss, it reads the core id, eg. *i,* of all the cache-lines in the cache-set where the evicted cache-line is. The intertask interference counter of the task whose LRU state has been modified by the eviction is then incremented: $accesses_{i \leftarrow j}$.

The resulting interference will be distributed according to the number of inter-task evictions generated:

$$u_i^{L2} = u_i^{L2\text{-hits}} + u_i^{L2\text{-misses}}$$
$$int_{i \leftarrow j}^{L2} = int_i^{L2} \frac{accesses_{i \leftarrow j}}{\sum_{\substack{k=0 \\ k \neq i}}^{N_C - 1} accesses_{i \leftarrow k}}$$

With this, we already have all the necessary elements to calculate the *itiCPIstack*.

4.4.2 Non-partitioned cache: Auxiliary Tag Directory (ATD)

As in the previous approach, we still have the ATD, but we don't have the per-cache-line owner bits. In this case, the ATD provides the int_i^{L2} term and to distribute it, we count the evictions that each core generates $evict_i$. It is clear that the number of evictions coincides with the number of misses. The resulting interference will be distributed according to the number of evictions generated:

$$int_{i \leftarrow j}^{L2} = int_i^{L2} \frac{evict_j}{\sum_{\substack{k=0 \\ k \neq i}}^{N_C - 1} evict_k}$$
$$u_i^{L2} = u_i^{L2\text{-hits}} + u_i^{L2\text{-misses}}$$

$$evict_j = misses_j$$

4.4.3 Non-partitioned cache: Per-cache-line owner bits

In this case, we still have the per-cache-line owner bits, but we don't have the ATD. Putting an ATD can have a hardware cost similar to increasing the cache size, since we are adding an entirely new tag directory, which can easily be close to half of the actual cache area. If we don't have an ATD, we cannot identify *iti-misses,* thus, we cannot know if the cycles spent in the cache and the memory are caused by a normal miss or an *iti-miss.* Because of this, we cannot breakdown the time spent in the L2 cache and in the memory controller. So every cycle assigned to the memory or to misses in the L2, will be collapsed into the interference term, only hits in the L2 can be considered as useful cycles, since misses can potentially be *iti-misses,* being part of the interference in that case:

$$int_i^{L2+mem} = u_i^{L2\text{-misses}} + (int_i^{mem} + u_i^{mem})$$

The resulting interference will be distributed according to the number of LRU accesses generated:

$$int_{i\leftarrow j}^{L2+mem} = int_i^{L2+mem} \frac{accesses_{i\leftarrow j}}{\sum_{\substack{k=0 \\ k\neq i}}^{N_C-1} accesses_{i\leftarrow k}}$$

$$u_i^{L2+mem} = u_i^{L2\text{-hits}} + int_{i\leftarrow i}^{L2+mem}$$

4.4.4 Non-partitioned cache: minimum support

In this case, neither the ATD nor the per-cache-line owner bits are present. In this case we need to distribute the int_i^{L2+mem} term through the different interfering cores we use the number of evictions $evict_i$. The resulting interference will be distributed according to the number of evictions generated:

$$int_{i\leftarrow j}^{L2+mem} = int_i^{L2+mem} \frac{evict_j}{\sum_{k=0}^{N_C-1} evict_k}$$

$$u_i^{L2+mem} = u_i^{L2\text{-hits}} + int_{i\leftarrow i}^{L2+mem}$$

5 Results

In order to test the proposed performance monitoring counters, we have modeled the behavior of the NGMP on a hardware simulator based on SoCLib. In this simulator we have implemented all the NGMP characteristics described in Section 2. For time constraints we have not evaluated all the PMC proposals made in Section 4 but a subset of them.
In particular for the proposals for the L2 cache we have evaluated the one in subsection 4.4.4, no ATD and no core ID. In this proposal, there is no breakdown between L2 and memory cycles, as it is impossible to determine interference in L2 without an ATD.

5.1 Experimental setup

To exercise the implemented PMCs we have run several executions of different combinations of benchmarks. We have used two types of benchmarks:

- Control applications, which receive interference.
- Payload applications, which cause interference.

In our setup, we have one control application running in a core, and three payload applications running in the other cores. We have collected the data generated by the PMCs regarding the control application.

As control applications we have used the well-known EEMBC Automotive benchmarks, and the Attitude and Orbit Control System (AOCS) from the EagleEye project [EAGLEEYE].
As payload applications we have used the ESA “On board data processing” (OBDP) application as well as set of microbenchmarks designed to stress the shared resources differently [MULTIOS]. The microbenchmarks used for the experimentation are the following:

- *L2-full*: The microbenchmark executes almost 100% load instructions. Its memory footprint is 200KB, so when executing 3 instances of this microbenchmark, it will cause a very high amount of L2 misses.
- *L2-half*: The microbenchmark executes almost 100% load instructions. Its memory footprint is 40KB, so when executing 3 instances of this microbenchmark, it may cause a high amount of L2 misses. It will affect memory intensive control applications, while control applications with small memory footprint will not have a very noticeable increase of L2 misses.
- *st*: The microbenchmark executes almost 100% store instructions. Its memory footprint is small, but it stresses the bus in higher degree than FULL and HALF microbenchmarks, because the L2 latency for writes is 0.
- *OBDP*: European Space Agency’s “On board data processing application”.

Payload benchmarks are always executed in groups of 3, which are run concurrently in 3 separate cores. We have defined the following sets of payload benchmarks:

- *full*: it is composed of three L2-full benchmarks.
- *half*: it is composed of three L2-half benchmarks.
- *st*: it is composed of three ST benchmarks.

- *OBDP*: it is composed of three OBDP benchmarks.
- *mix*: it is composed of one L2-full, one L2-half, and one OBDP benchmark.

5.2 Benchmark characterization

In order to provide insight in the results obtained, we have characterized each of the control benchmarks. We have decided to use the number of bus accesses per kilo-instruction (BAKI) as a metric to determine the likeliness of the benchmark to become interfered by the payload benchmarks. In the following table we present the result of this characterization:

	Benchmark	BAKI
EEMBC Automotive	a2time	204
	Aifftr	170
	Aifirf	271
	Aiifft	174
	Basefp	233
	Bitmnp	70
	Cacheb	330
	Canrdr	239
	Idctrn	135
	Iirflt	254
	Matrix	88
	Pntrch	146
	Puwmod	235
	Rspeed	272
	Tblook	282
	Ttspkr	302
ESA	AOCS	158

To provide an intuition of how intensively the payload benchmarks access the bus, we have also modeled this for them:

	Benchmark	BAKI
ESA	OBDP	93
Micro-benchmarks	ST	347
	FULL	1376
	HALF	834

The payload micro-benchmarks require an explanation to undertand their BAKI values:

- ST presents a low BAKI because it alternates store and nop instructions, hence, the maximum BAKI would be 500.
- FULL presents a BAKI higher than 1000, which may look surprising as there is more than one bus access per instruction. This is because FULL is an unrolled loop where every instruction is a load. The unrolled nature of the loop causes some instruction cache misses, in addition to the data cache misses that the benchmark is supposed to cause.
- HALF is similar to FULL, but less misses occur.

5.3 Results

In this sections the results obtained using the proposed PMC infrastructure are presented. The results are drawn as a cycle-stack. A cycle-stask is the same as a CPI stack, the cycle count by the number of instructions executed by the program. Cycle tasks are derived for each control benchmark in isolation and with payload (interfering) benchmarks, in the four scenarios described in section 5.1.

Different subsections will be used to present the results; first we will show the results for particularly interesting cases, where we can see a very high amount of interference detected in different components of the chip. Then, the rest of results are presented.

The figures in the following subsections present the cycle stack of the control benchmark, with time (number of CPU cycles) in the Y axis, and different payload configurations in the X axis (being ISOL the cycle stack for the execution of the control benchmark in isolation). Each graph comes with a legend showing different places where CPU cycles are spent:

- *Core-run*: the core is fetching an instruction this cycle
- *Loc-stall*: the core is running, but not fetching an instruction. This is caused by instruction requiring more than 1 execution cycle and stalling the pipeline. This stall is intrinsic to the core, i.e not caused by other cores and unavoidable.
- *Bus-run*: the core is accessing the bus. This time is spent in the bus regardless of interferences.
- *bus1*, *bus2*, *bus3*: cycles spent waiting for the bus due to interferences caused by cores 1, 2, or 3.
- *L2/MC-run:* cycles spent in the L2 or memory caused by the control benchmark itself.
- *L2/MC1*, *L2/MC3*, *L2/MC3*: cycles spent in the L2 or memory controller due to interference caused by cores 1, 2, or 3.

5.3.1 High bus usage: EEMBC cacheb

Cacheb is the control benchmark with the highest BAKI (330) of all the control applications used in the experiment.

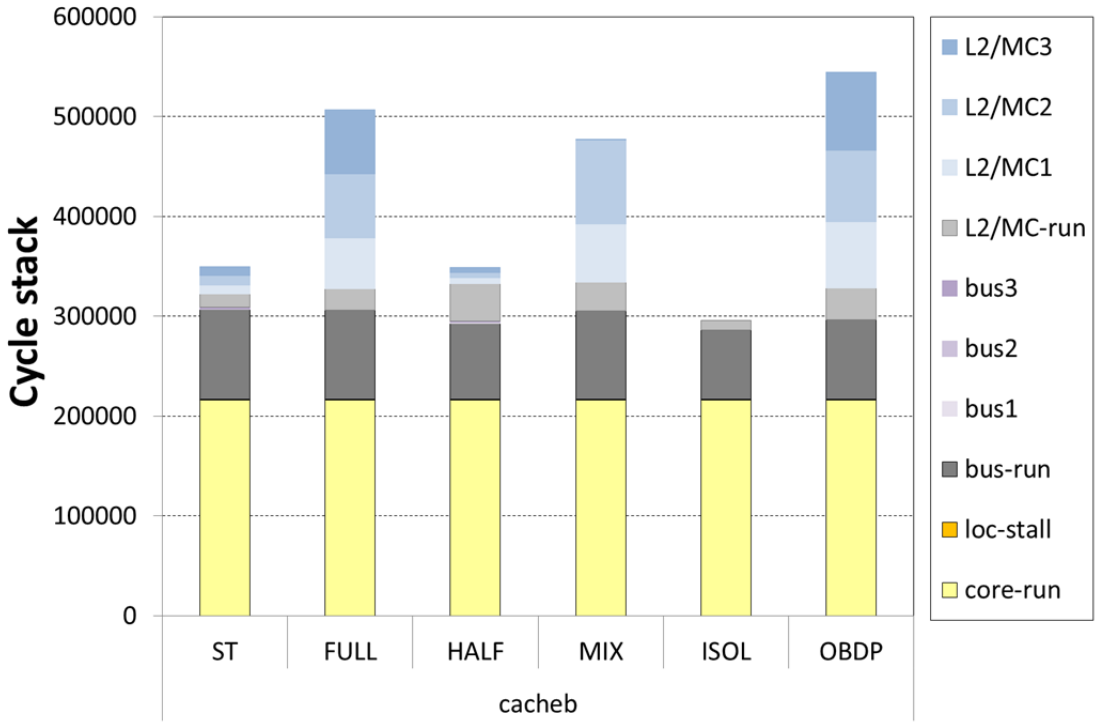

Figure 6 *Cacheb cycle stack.*

As we can see in Figure 6, the core spends a considerable amount of time stalled waiting for the bus, even when running in isolation. This is expected because of the high BAKI of the benchmark, and confirms it is a suitable metric to measure shared resource use.
We can see that the amount of time spent waiting for the L2 or memory is very small for the execution in isolation. This happens because the accesses to the bus are mainly stores that do not block the core (thanks to the write buffer). We can see that the amount of interference detected in the L2 and memory controller is greatly increased by the presence of contenders, especially FULL and OBDP, which systematically miss the L2. This interference is not found in the bus thanks to the bus split.
The slowdown caused by the payload benchmarks is 46% (in the case of OBDP).

5.3.2 Low bus usage: EEMBC bitmnp

Bitmnp is the control benchmark with the lowest BAKI (77) of all the control applications used in the experiment.

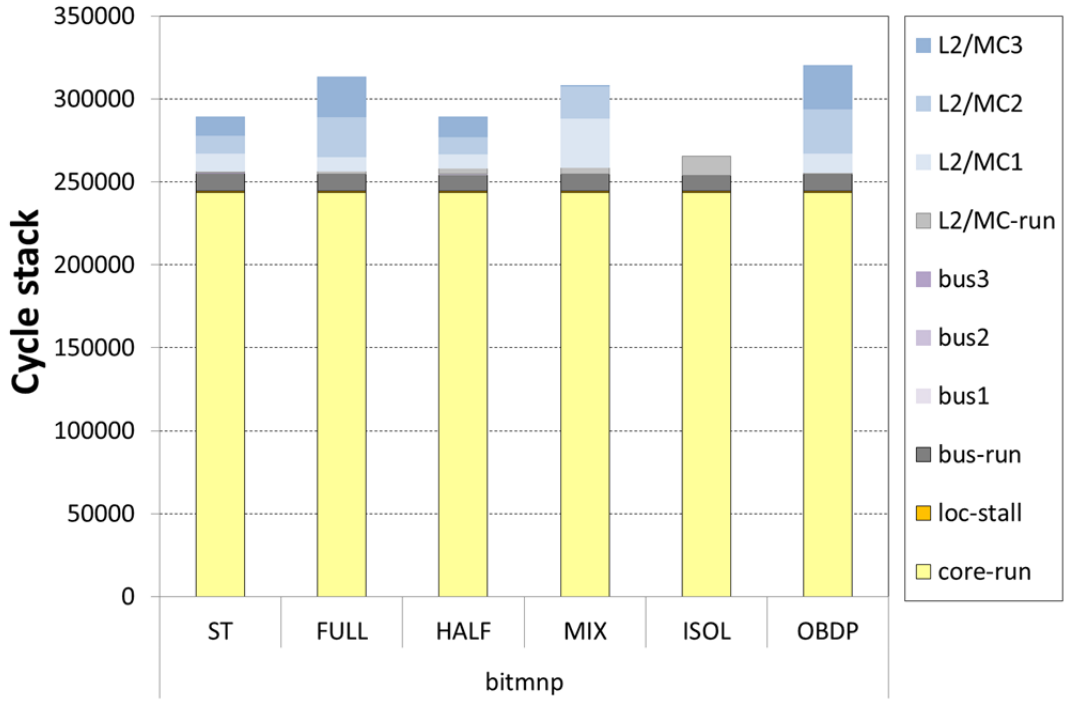

Figure 7 *Bitmnp cycle stack.*

As we can see in Figure 7, the core spends a small time stalled waiting for the bus, solving a L2 miss, or accessing the memory controller. This small amount of time is spent in all cases, including the execution in isolation.
In the cycle stack we can see that the benchmark is not sensitive at all to interference in the . We can also see that some interference is detected in the L2 and memory controller, which is the main cause of slowdown (17%).
We can also see that the amount of time spent waiting for L2 or MCis very small for the execution in isolation. This happens because the accesses to the bus are mainly stores that do not block the core (thanks to the write buffer).

5.3.3 Sensitive to L2 and MC interference: ESA AOCS and EEMBC cacheb

AOCS and cacheb are the control benchmark which present the highest sensitivity to interference in the L2 and MC.

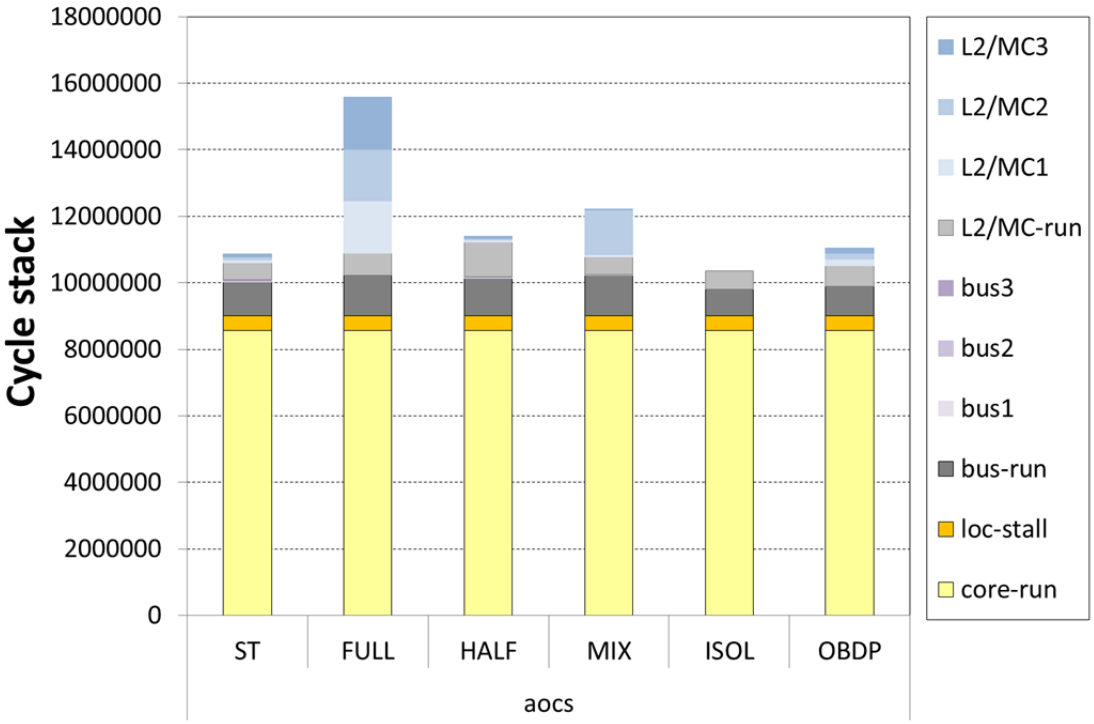

Figure 8 *AOCS cycle stack.*

In Figure 8 we can see a very high amount of interference in the L2 and MC. In the case of the FULL payload configuration, the amount of time spent accessing L2/MC caused by this interference is 10 times bigger than the amount of time that was spent in isolation. This interference is the main contribution to the 34% slowdown detected.

Cacheb, already discussed in subsection 5.3.1 is very sensitive to memory controller interference. As we explained in the aforementioned subsection, this is caused by a very high BAKI and low reuse of L2 data, which triggers L2 misses and memory accesses

5.3.4 Sensitive to bus interference: EEMBC matrix and basefp

As we used an NGMP simulator with bus split, we have found that interference in the bus is negligible. To illustrate this, we present two cases:

- EEMBC matrix benchmark, the one suffering the highest increase in bus use relative to its execution in isolation. The bus usage is increased by 120%, but this barely accounts for 2% of the total execution time of the benchmark.
- EEMBC basefp, which is the benchmark which has to wait for bus interferences for the longest amount of cycles, relative to the total execution time of the benchmark. The benchmark is interfered in the bus for 6% of the execution time (vs ST).

In both cases, the time accessing the bus is so small that it barely is noticeable in the graph. Below, a graph for EEMBC matrix is presented.

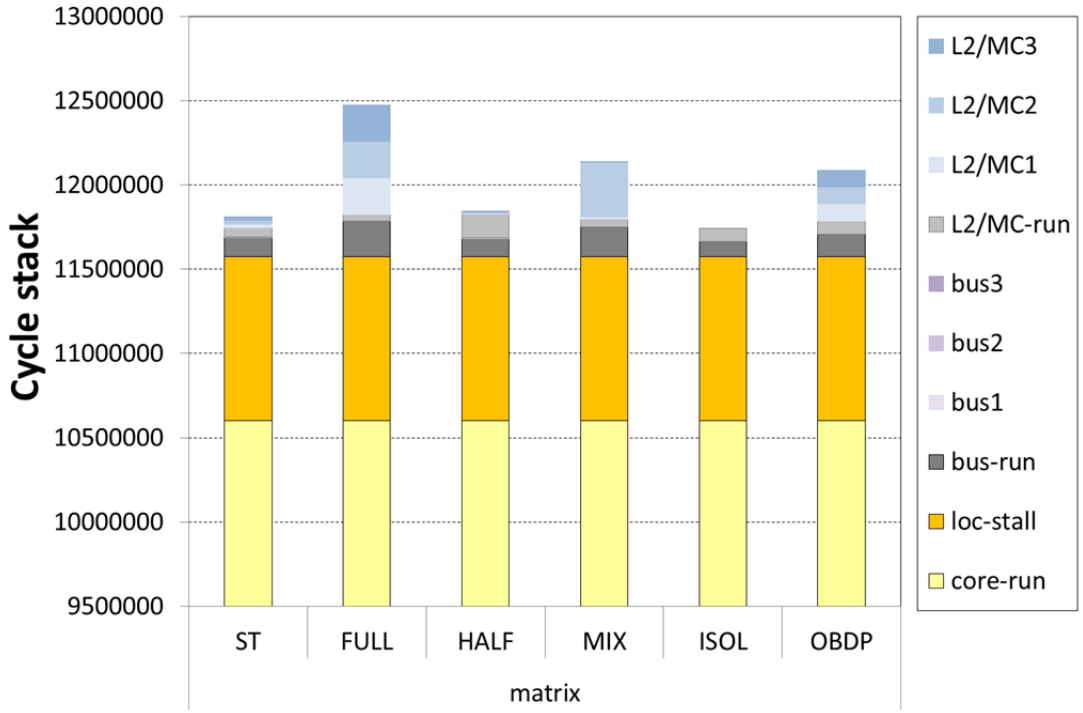

Figure 9 *matrix cycle stack.*

In Figure 9 we can see that the amount of interferences in the bus is not even noticeable in the graph. As a side note, matrix is the benchmark with highest loc-stall, caused by the high number of floating point instructions executed by the core.

In Figure 10 we can see a noticeable amount of bus interferences when running the Basefp benchmark vs the ST.

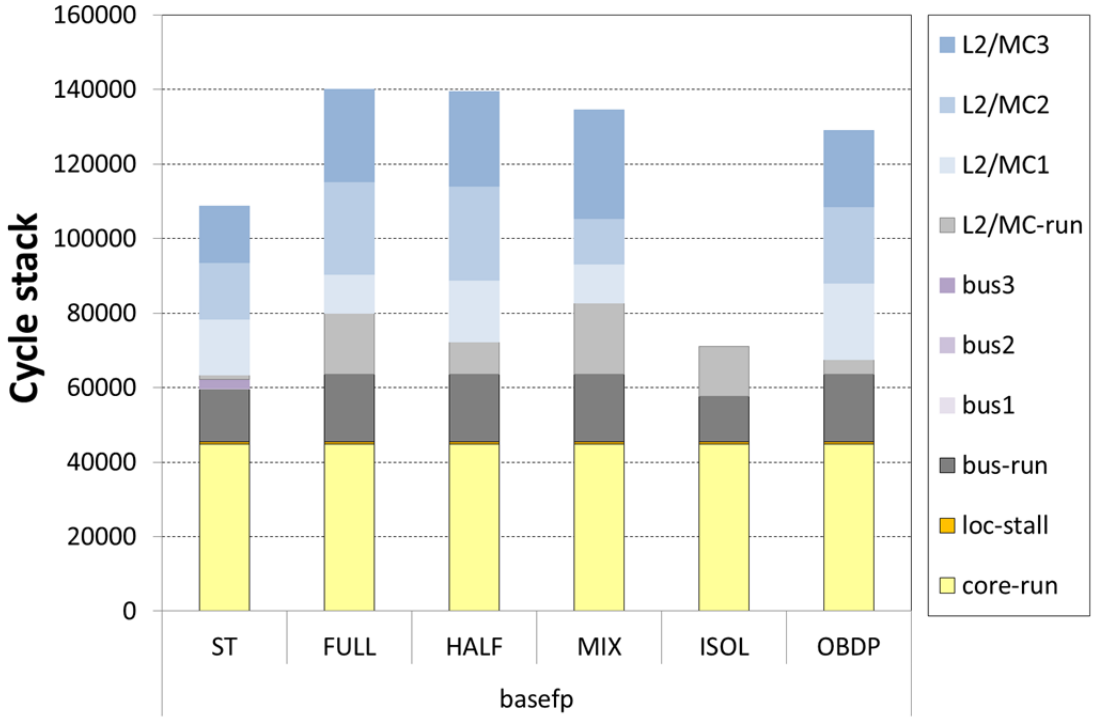

Figure 10 *Basefp cycle stack.*

5.3.5 Highest impact on execution time: EEMBC basefp

Basefp is the control benchmark which experiences the highest impact on its execution time caused by inter task interferences.

As it can be seen in Figure 10, the benchmark is slowed down a 49% in the case of FULL. This slowdown is mainly caused by interferences in the memory controller. This benchmark also has a high BAKI (233) and probably does a very low L2 data reuse, a behavior that is similar to cacheb explained in section 5.3.1.

5.3.6 Other benchmarks

Other EEMBC benchmarks with less remarkable characteristic behavior have been used as control benchmarks. In this subsection their cycle stacks are presented for completeness.

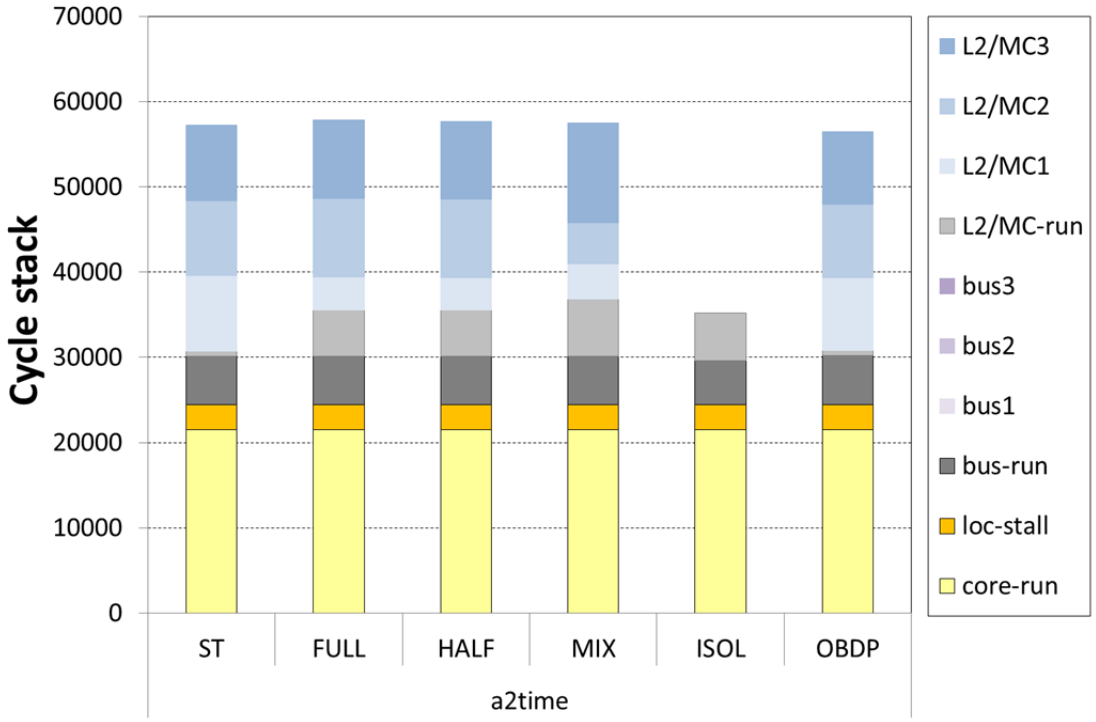

Figure 11 *A2time cycle stack.*

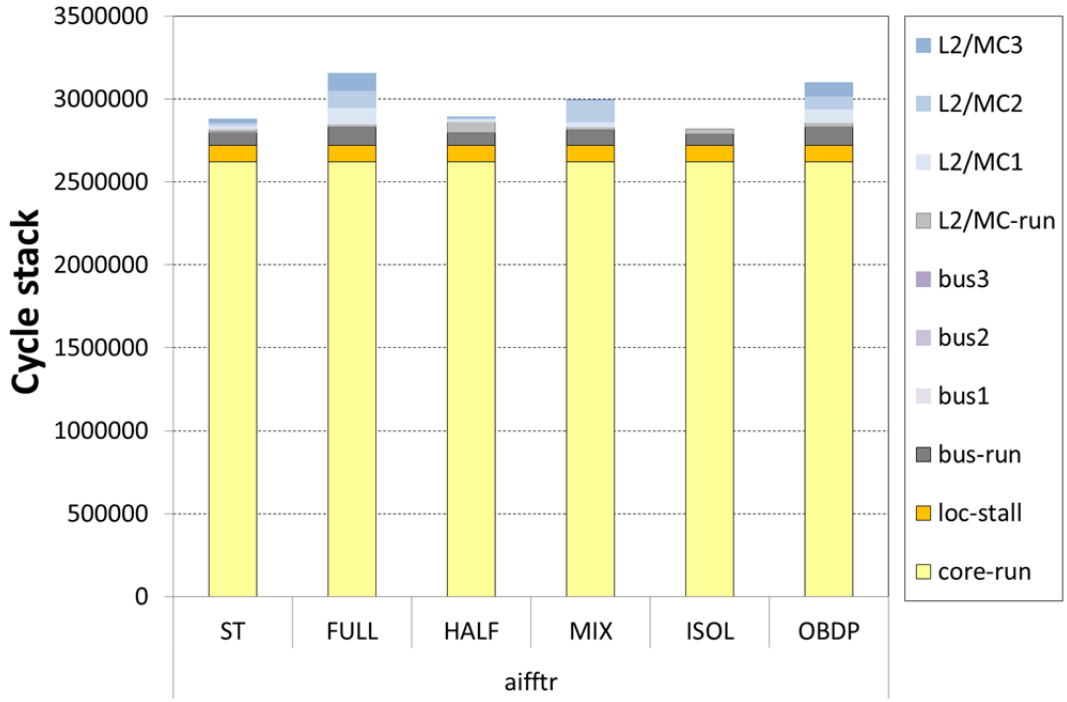

Figure 12 *Aifftr cycle stack.*

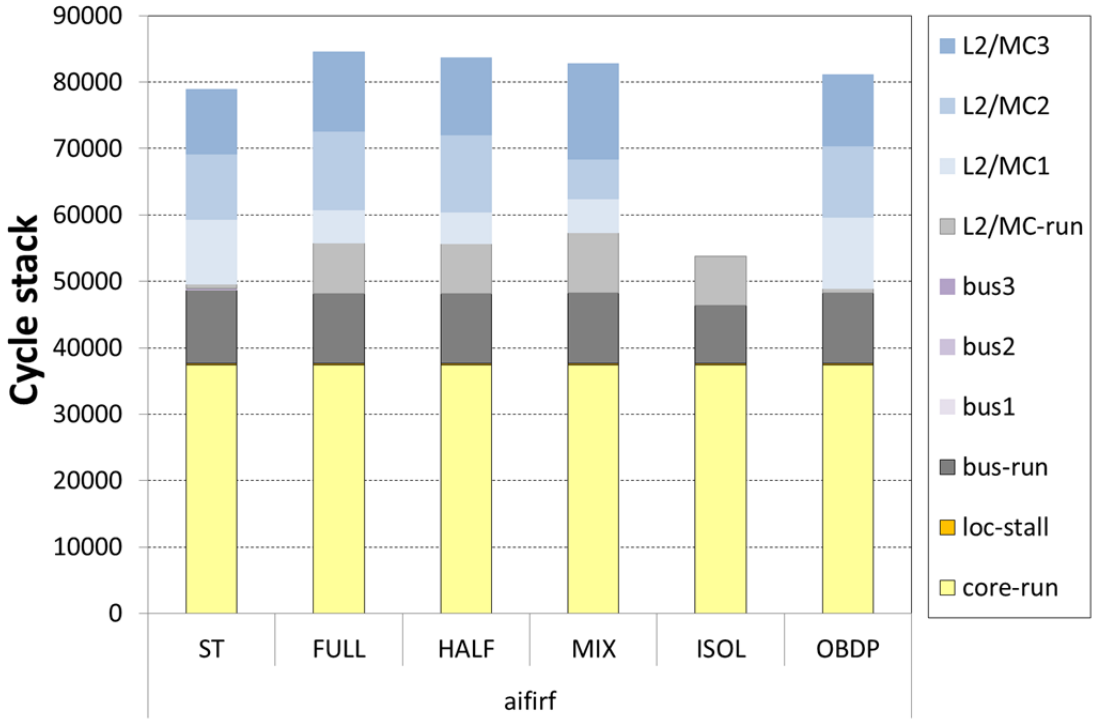

Figure 13 *Aifirf cycle stack.*

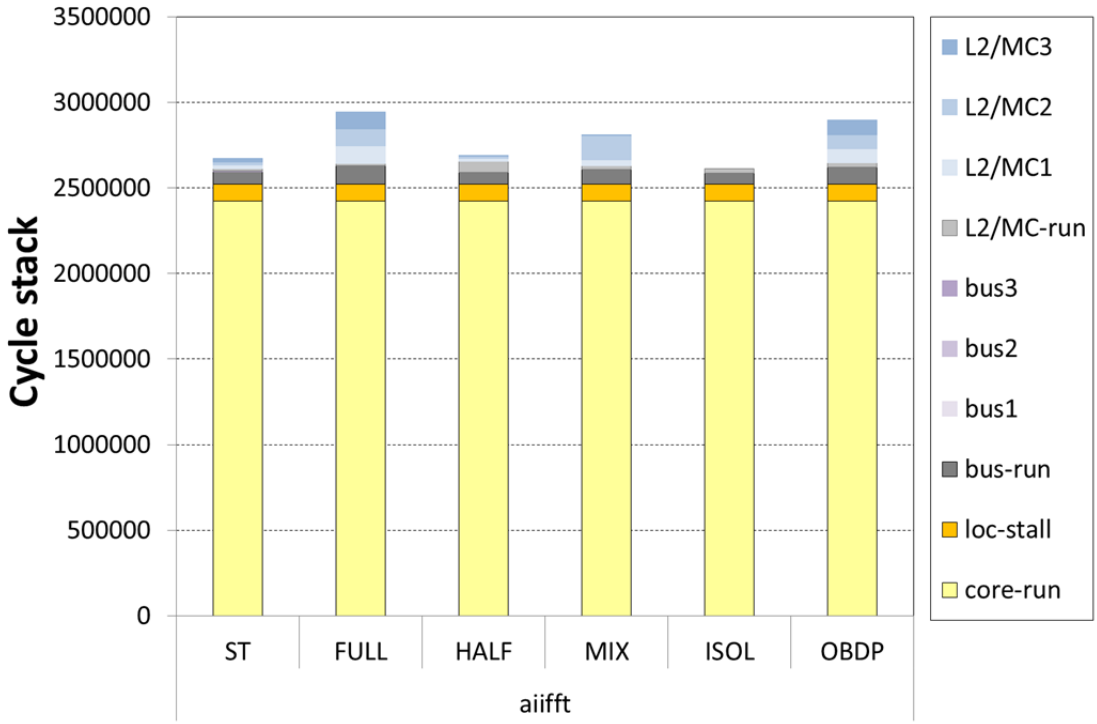

Figure 14 *Aiifft cycle stack.*

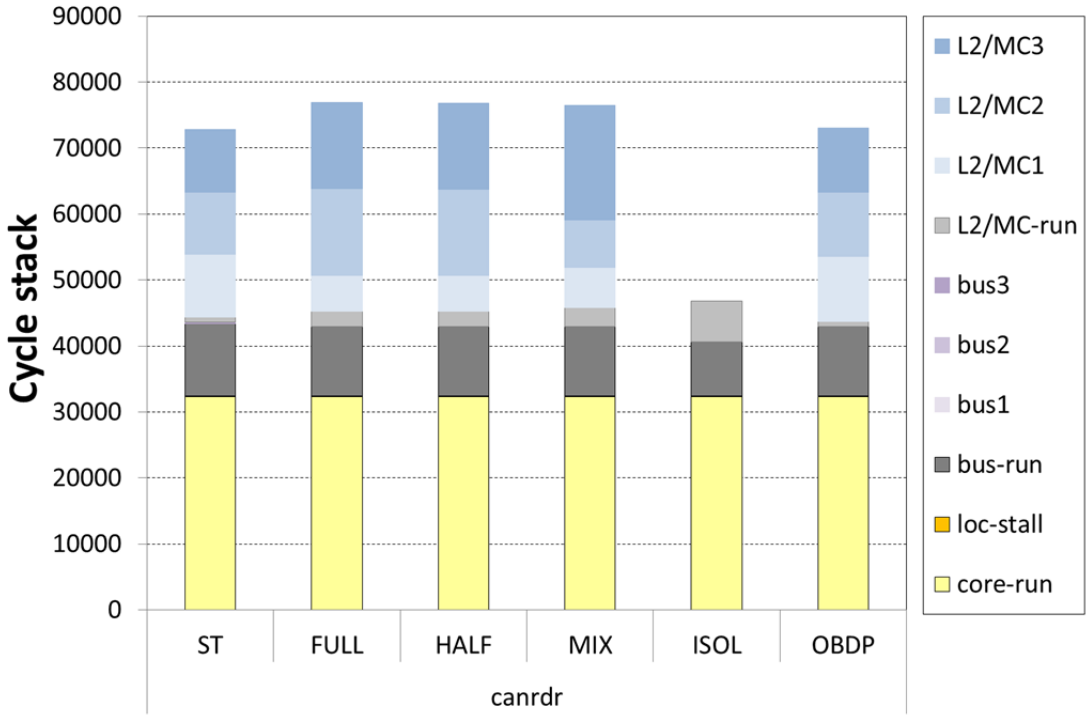

Figure 15 *Canrdr cycle stack.*

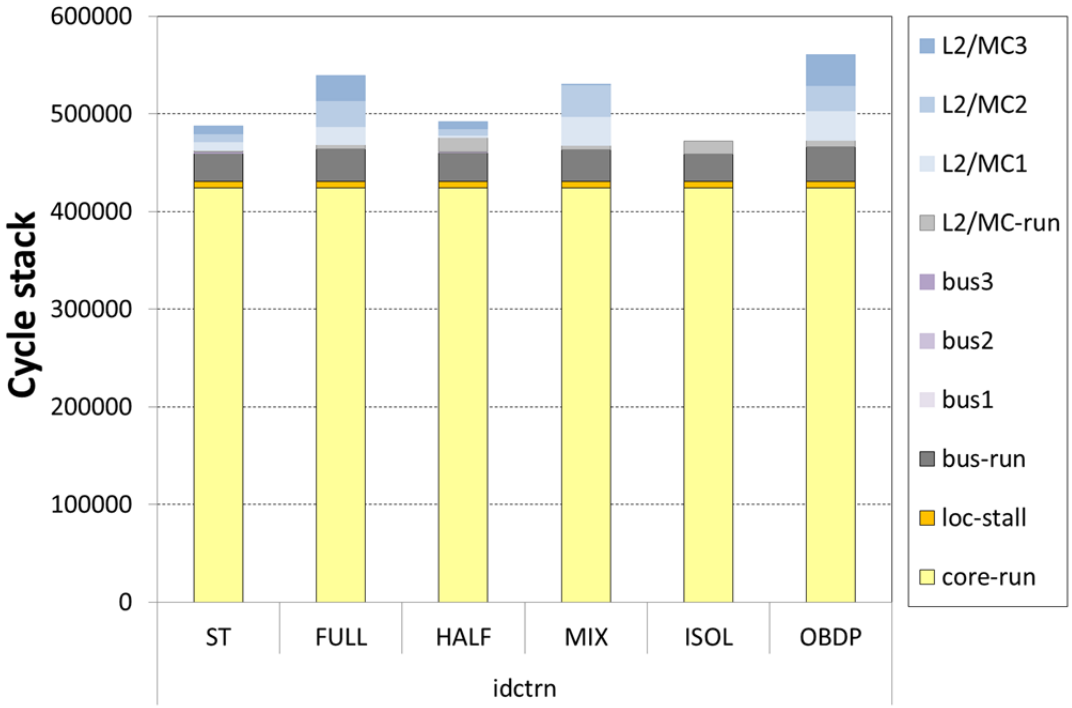

Figure 16 *Idctrn cycle stack.*

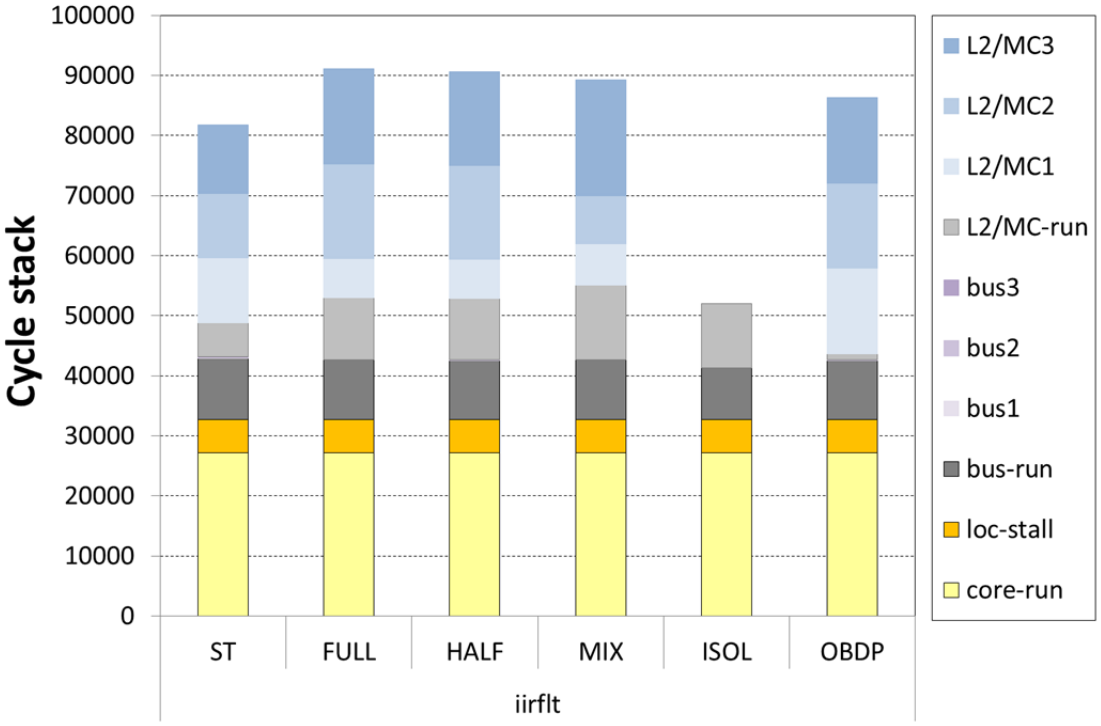

Figure 17 *Iirflt cycle stack.*

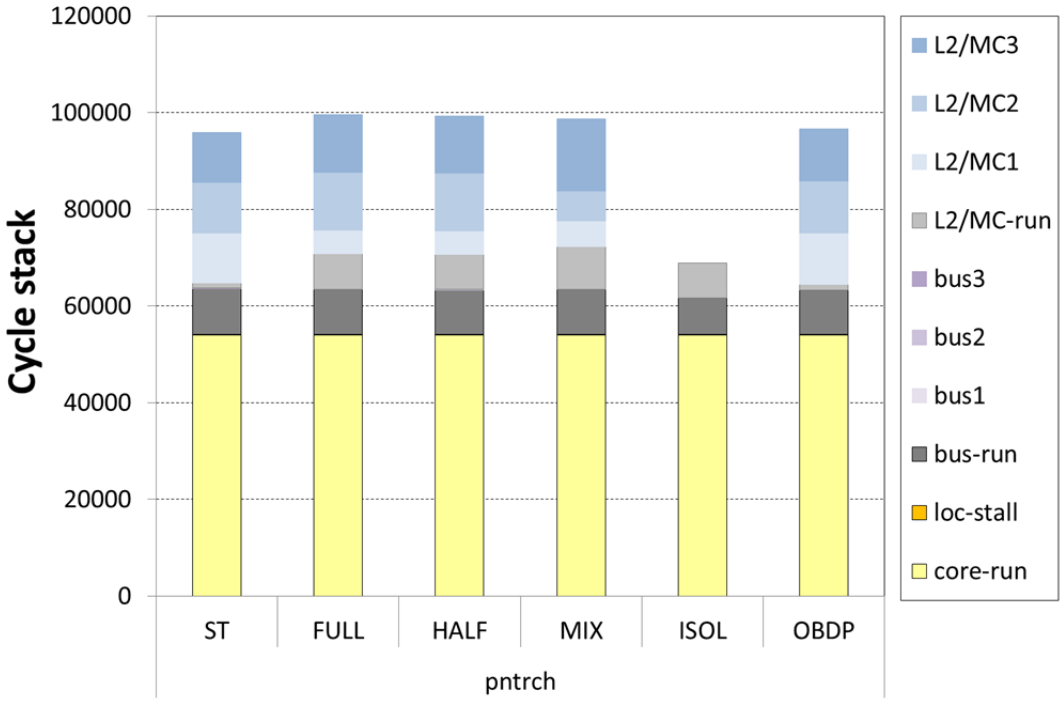

Figure 18 *pntrch cycle stack.*

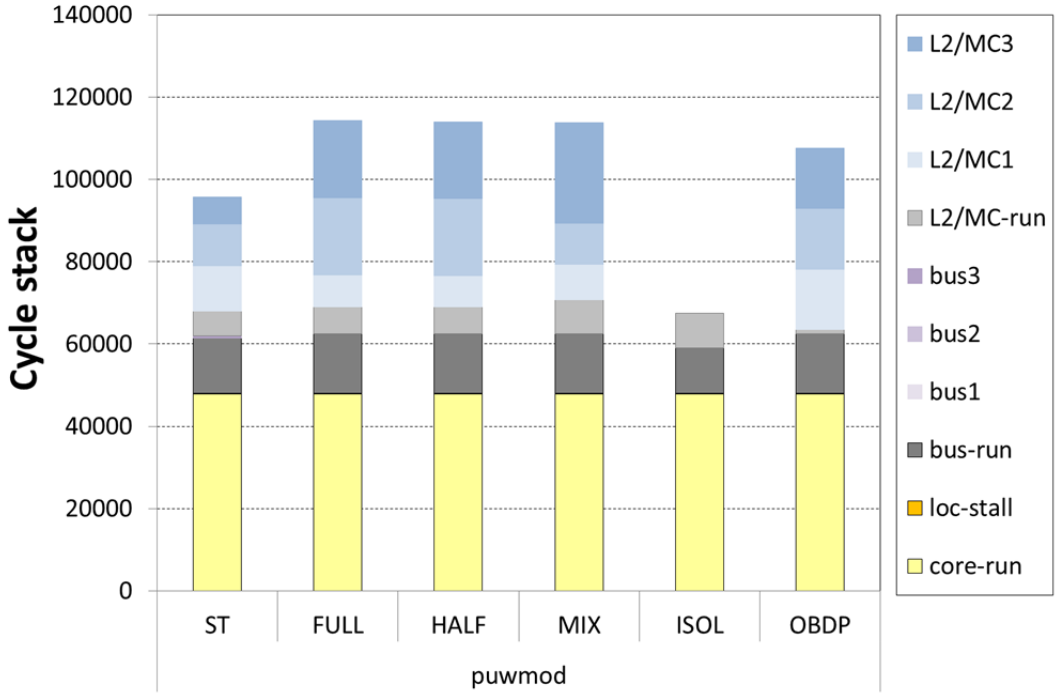

Figure 19 P*uwmod cycle stack.*

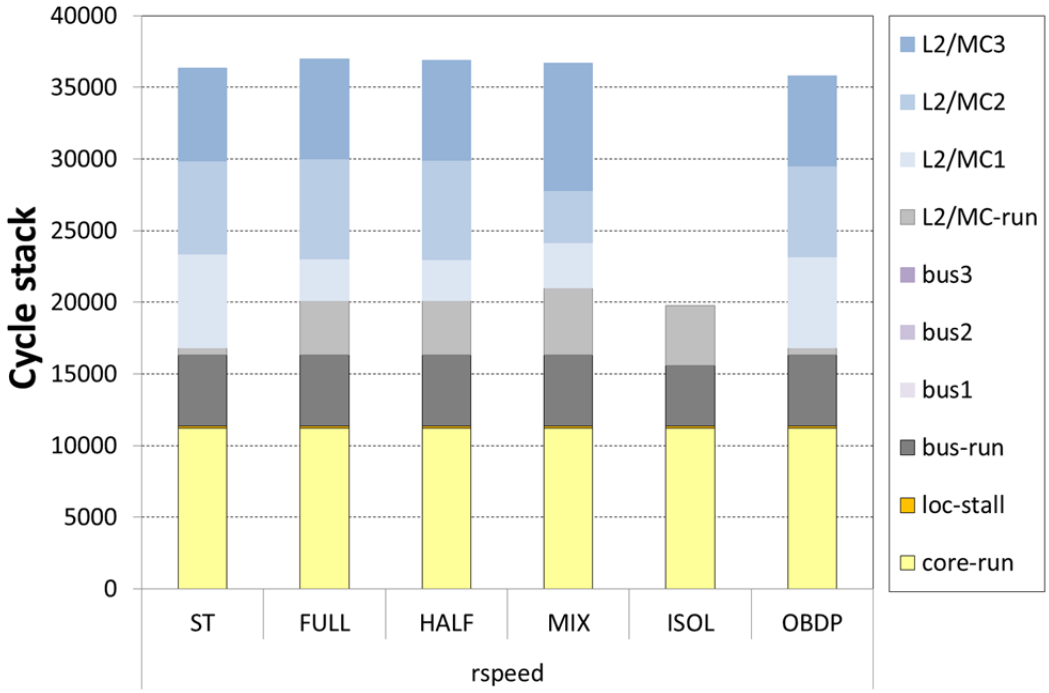

Figure 20 *Rspeed cycle stack.*

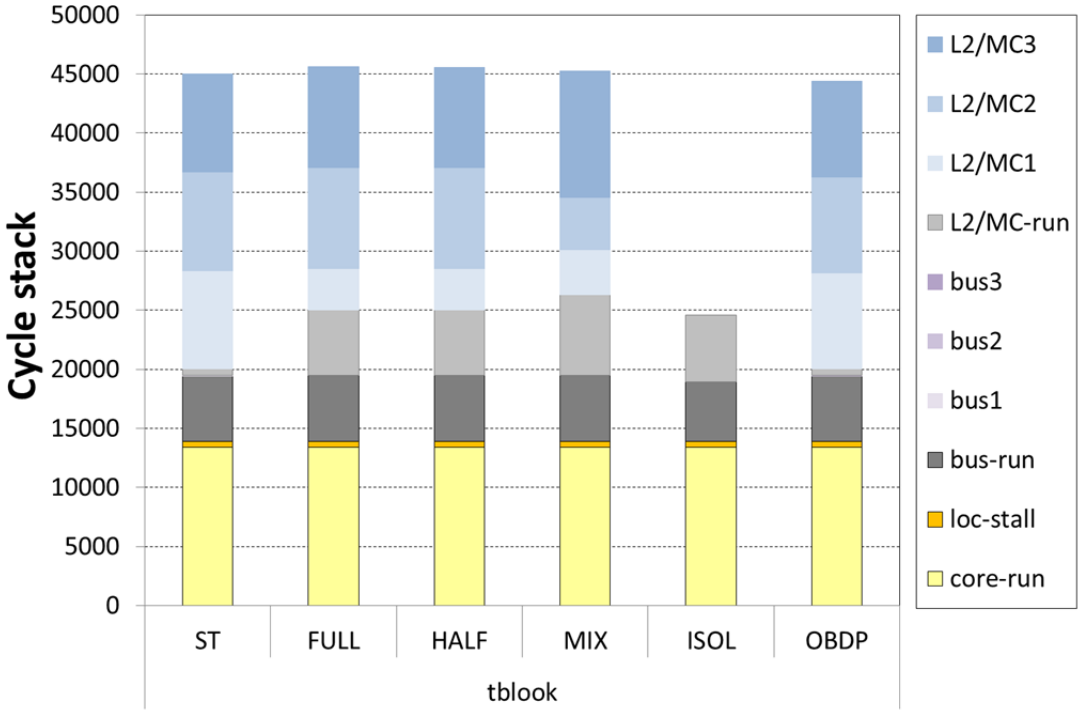

Figure 21 *Tblook cycle stack.*

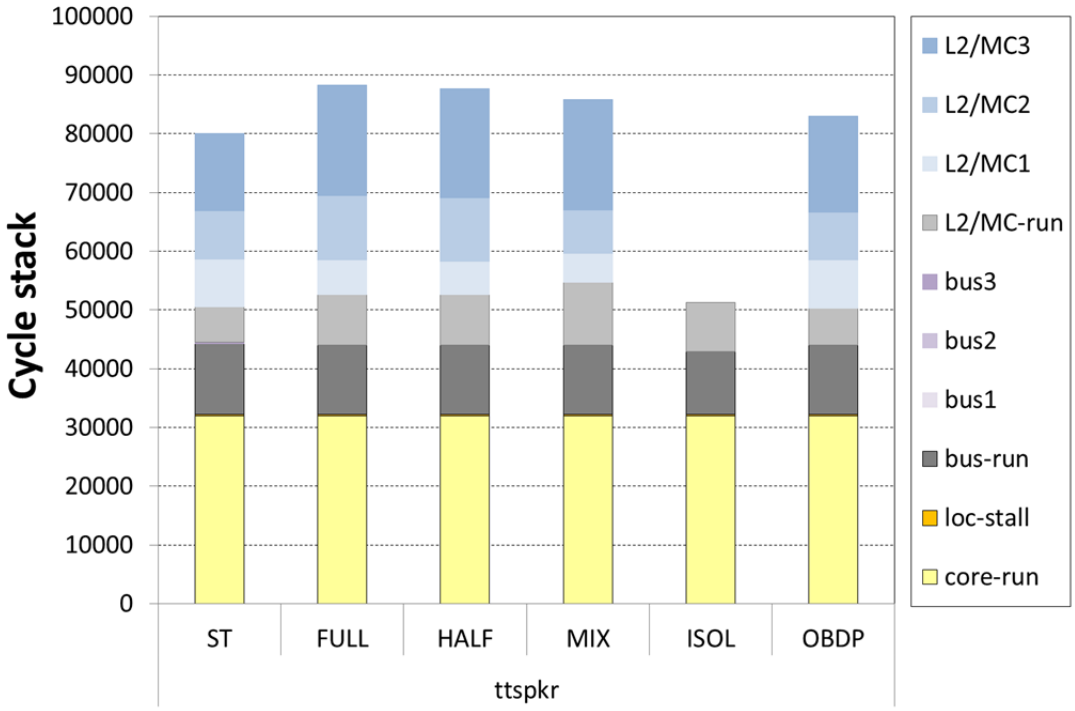

Figure 22 *Ttspkr cycle stack.*

References

NGMP	Quad Core LEON4 SPARC V8 Processor LEON4-NGMP-QUADLION Data Sheet and User's Manual
JJQF2014	Javier Jalle, Jaume Abella, Eduardo Quinones, Luca Fossati, Marco Zulianello and Francisco J. Cazorla. *AHRB: A High-Performance Time-Composable AMBA AHB Bus* In The 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2014). Berlin, Germany. April 2014.
MULTIOS	RFQ- 3-13153/10/NL/JK Multicore OS Benchmark
EAGLEEYE	Bos, Victor, *EagleEye: Evolution towards Time and Space Partitioning.* Software & Data Systems Division Final Presentation Days, 2013 ESA