Marmot: Extraction of Fine-Grain Memory Access Profiles for real-time software

Hector Chabot, Isabelle Puaut, Thomas Carle, Hugues Cassé

Marmot: Extraction of Fine-Grain Memory Access Profiles for real-time software

Hector Chabot
Univ Rennes, Inria, CNRS, IRISA
FR
hector.chabot@inria.fr

Isabelle Puaut
Université de Rennes
FR
isabelle.puaut@irisa.fr

Thomas Carle
IRIT - University of Toulouse
FR
thomas.carle@irit.fr

Hugues Cassé
IRIT - University of Toulouse
FR
hugues.casse@irit.fr

Abstract

Enforcing deadlines in real-time systems calls for the computation of an upper-bound of the Worst-Case Execution Time (WCET) of tasks. In multi-core systems, shared-resource usage leads to *interference* between tasks running on parallel cores, resulting in additional delays in the execution time of tasks. Schedulability analysis techniques rely on *Interference-Aware WCET* of tasks (IA-WCET, WCET integrating delays resulting from interference) to safely consider these delays. Calculation of IA-WCET requires knowledge about the worst-case shared-resource usage of tasks, in the form of a *memory access profile* as far as shared memory accesses are concerned.

State-of-the-art memory profiles only provide coarse-grain information (at the level of an entire task), resulting in pessimism in IA-WCET computation. More recent solutions propose to refine the information available in memory profiles, but are still limited: they lack information about shared-resource usage of code inside loops and are unable to use contextual information, which leads to over-approximation. This paper presents Marmot, a technique that extends recent memory access profile extraction solutions for real-time software. In Marmot, tasks are split in successive *intervals*, with the worst-case resource usage of each interval described as a *distribution* instead of a single value. Experimental results show that IA-WCET computation and schedulability analysis can take advantage of the fine-grain intervals produced by Marmot to obtain more precise IA-WCET and therefore higher schedulability than coarser-grain profiles.

CCS Concepts

• **Computer systems organization → Real-time systems**; **Multicore architectures**; **Embedded systems**.

Keywords

Worst-Case Execution Time Estimation, Static Analysis, Multicore, Interference, Event Arrival Function

RTNS 2024, November 06–08, 2024, Porto, Portugal

ACM ISBN 979-8-4007-1724-6/24/11
https://doi.org/10.1145/3696355.3696360

ACM Reference Format:
Hector Chabot, Isabelle Puaut, Thomas Carle, and Hugues Cassé. 2024. Marmot: Extraction of Fine-Grain Memory Access Profiles for real-time software. In *The 32nd International Conference on Real-Time Networks and Systems (RTNS 2024), November 06–08, 2024, Porto, Portugal.* ACM, New York, NY, USA, 25 pages. https://doi.org/10.1145/3696355.3696360

1 Introduction

Tasks in real-time systems interact with the external world in a timely manner. In hard real-time systems, this translates to the need of meeting a *deadline* to avoid catastrophic consequences. This has motivated research in estimation of upper bounds of execution times (Worst-Case Execution Times, WCET) [26]. Scheduling techniques and associated schedulability tests then use the produced WCET bounds to ensure that deadlines are met, even if tasks execute up to their WCET [4].

Static WCET estimation techniques have been extensively studied in the literature, and different approaches have been designed for single-core processors [26]. However, multi-cores have made their way into these systems as they offer good processing power and low energy consumption. WCET estimation techniques for single-core platforms cannot be used unmodified, as some hardware resources in multi-core architectures (last level caches, interconnect, memory controllers) can now be shared between the different cores. Shared resource usage leads to conflicts named *interference*, which add delays in a task execution because accesses to shared resources need to be arbitrated.

Different classes of techniques, surveyed in [17], were designed to account for interference. They either *avoid interference*, through hardware of software-enforced policies (e.g. time-division multiple access bus arbitration, software-enforced memory bandwidth regulation [27], exploitation of multi-phase models such as PREM – PRedictable Execution Model – [22] that separate computation phases and memory phases) or *calculate the delay resulting from interference* using knowledge of resource usage for all tasks.

Accounting for interference requires knowledge of the usage of shared resources by tasks, which led to research on how to obtain such information. Without loss of generality, we focus in this paper on accesses to the shared memory. The term *memory access profile* will be used to denote any kind of curve, from the most simple to the most complex, giving information on the memory accesses performed by tasks.

The simplest technique is coarse-grain and produces as memory access profile a flat curve, giving the worst-case number of memory accesses for the entire task. More recent techniques extract *phases* from the binary code of tasks, and produce a memory access profile for each phase. For example, the work introduced in [24] proposes a static analysis technique that converts a program into a conditional sequence of PREM phases, each phase either performing shared memory accesses or no access at all, with the idea of avoiding contention at run-time. StAMP [7] splits the code of tasks in a sequence of consecutive *intervals*, each of them having its own WCET and worst-case number of memory accesses (WCMA). The research presented in [20] and [5] calculates the distribution of memory accesses in the execution window of an entire task.

In this paper, we introduce Marmot (Extraction of Fine-Grain Memory Access Profiles for real-time software), a static analysis tool that extracts detailed memory access profiles from the executable code of legacy software. Marmot builds upon and improves existing techniques used to statically analyze binaries in order to produce memory access profiles. Marmot uses StAMP [7] to split the code of tasks in a sequence of consecutive *intervals*. An extension of Event Arrival Curves (EAC) from [20] is then used to produce detailed information on the distribution of accesses within each interval. The contributions of Marmot are twofold:

- *Introduction of detailed per-interval WCMA curves*, through a combination of the work of StAMP [7] and EAC [20], which brings the following benefits:
  - Scheduling algorithms and corresponding schedulability analysis can operate at the interval level, resulting in reduced overall cost of interference.
  - The introduction of WCMA curves in each interval allows schedulers to further reduce interference delays.
- *Consideration of contextual information* when calculating per-interval WCMA curves. This allows to handle the situations in which the number of shared memory accesses depends on the execution context of the code snippet under analysis (first versus next occurrences of a memory access in a loop nest, call point of the code snippet).

Experimental results, conducted on the TACLeBenchmark collection [10] demonstrate that: (i) using the execution context of intervals allows to obtain more precise memory profiles than the (non-contextual) state-of-the-art EAC technique described in [20]; (ii) the number of memory accesses to be considered when calculating the *Interference-Aware WCET* (IA-WCET) of a task is significantly reduced compared to concurrent techniques; (iii) this reduced number of memory accesses allows an overall reduction of interference delays.

The paper mainly concentrates on the extraction of memory access profiles. It also shows on examples that the IA-WCET of a task, assuming the concurrent task executing on the other core is known, is smaller than when using state-of-the-art solutions. We also illustrate the benefit of using the produced memory access profiles on off-line time-triggered scheduling. Marmot was clearly designed with off-line time triggered scheduling in mind. We believe that this class of scheduling algorithm benefits the most from our approach, since at any time the set of tasks executing concurrently are known. However, the profiles produced by Marmot are not restricted to a specific scheduling algorithm. A larger set of scheduling algorithms (off-line, on-line, dynamic and static priorities) could leverage the newly-gained precision, but we consider the usage of Marmot profiles by the different classes of schedulability tests as out of scope.

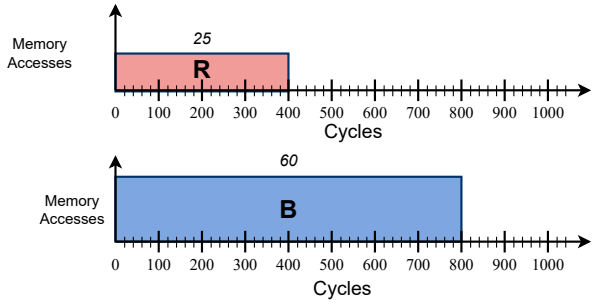


Figure 1: Memory profiles of the task set of the motivating example. Task R has a WCET of 400 cycles and performs 25 memory accesses, while task B has a WCET of 800 cycles and performs 60 memory accesses.

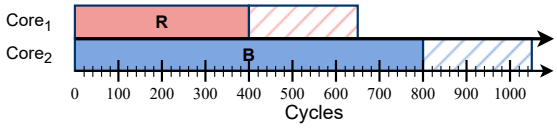


Figure 2: Possible schedule and resulting interference delays when calculating interference delays at the task level. The WCET of tasks is represented in solid colors and the additional interference delays in stripes.

The rest of this paper is organized as follows. Section 2 first gives a motivating example. Section 3 then provides background on recent techniques Marmot builds upon, and compares Marmot with related work. Section 4 presents the core of Marmot, that extracts detailed memory access profile curves from executable code. Section 5 is devoted to an experimental evaluation of Marmot. Finally, Section 6 concludes this paper.

2 Motivating Example

Let us introduce a simple example of the computation of interference delays during schedulability analysis to motivate our research, using for illustration purposes static off-line scheduling. Let us consider a system with two tasks, R and B, with their task-level WCMA depicted in Figure 1. R has a WCET of 400 cycles (in isolation, excluding interference delays) and performs 25 memory accesses, whereas B has a WCET of 800 cycles and performs 60 memory accesses. For the sake of illustration, let us assume that the contention delays' upper bound is 10 cycles when a memory access from one core may interfere with another access from another core, and that the memory controller follows a Round-Robin policy. Any access performed by a core can thus suffer from at most one contention per concurring core.

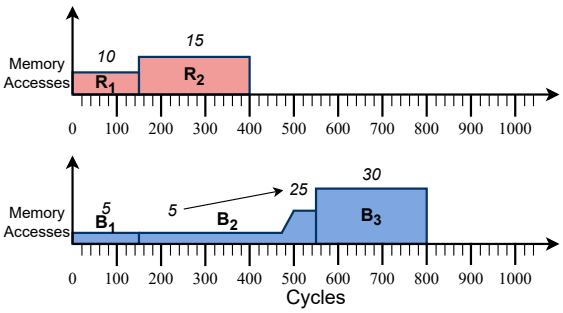


Figure 3: Memory profiles of the task set with interval separation. B's second interval's memory access count is distributed over time: it first stagnates at 5 memory accesses before rising to the final value of 25 at the end of the interval.

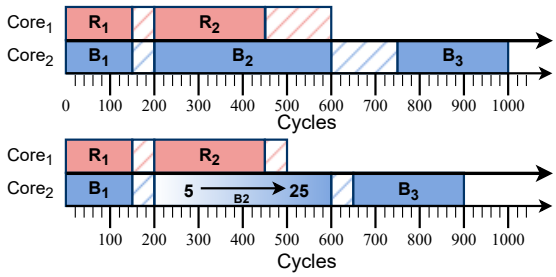


Figure 4: Possible schedule and resulting interference delays when calculating interference delays at the level of intervals. The topmost figure considers a single WCMA value per interval while the bottom-most figure account for the distribution of WCMA provided by Marmot.

In the selected example, B and R are scheduled concurrently on two cores such that there exists a period of time where both tasks run concurrently. Note that the tasks share the same starting date solely for the sake of the example, no assumptions are made about the scheduling algorithm. Every memory access from R must be considered as suffering from contentions from B and vice-versa. Thus, a maximum of 25 concurrent accesses between the two tasks has to be considered, with each access suffering from a contention. The interference delay suffered by both R and B is thus 250 cycles. Figure 2 shows the final scheduling on two cores C_1 and C_2 with the interference delays considered.

In Marmot, tasks are split in multiple consecutive intervals. Figure 3 shows the new memory profiles with intervals extracted by Marmot: R is split in two consecutive intervals (R_1 and R_2) and B in three intervals (B_1, B_2 and B_3). The curve inside B_2 will be discussed later, but for now let us assume that B_2 performs 25 accesses in the worst case. The WCET and memory access count is split between the different intervals. By scheduling intervals instead of entire tasks, calculation of interference delays can rely on the WCMA count of intervals instead of the overall task's WCMA. Figure 4 shows a scheduling example on its topmost part. R_1 and B_1 are scheduled concurrently leading to a maximum of 5 contentions for each interval in the worst case as B_1 has a WCMA of 5 and R_1 a WCMA of 10. In the same manner, we consider at most 15 contentions between R_2 and B_2. The final interval B_3 has no concurrent interval and therefore does not suffer from contentions. The final total interference delay considered for both cores is reduced to 200 cycles each.

Beyond splitting the code into consecutive intervals, Marmot further provides a distribution of the number of worst-case memory accesses for each interval instead of a single value, as depicted in Figure 3 for interval B_2. The curve inside the interval is a step function that gives for each date in the interval a conservative approximation of the maximum number of memory accesses that may have occurred on that date since the start of the interval. As a result, accesses are not accounted for sooner than they may occur. The step function naturally reaches the WCMA value before the end of the interval. This information can be leveraged to further improve the precision of the interference analysis. In our example, B_2 performs at most 5 memory accesses for most of the time of its execution, before attaining the maximum value of 25 memory accesses.

Figure 4 shows on its bottom-most part the scheduling of intervals with their interference delays when considering the distribution of worst-case memory accesses. We observe that R_2 only overlaps with the start of B_2. Marmot's memory profile ensures that no more than 5 shared memory accesses can be performed by B_2 in the period where both tasks overlap, thus the method accounts for a maximum of 5 contentions for each task in the worst case. The overall contentions count is reduced to 10, with 5 happening between R_1 and B_1 and 5 between R_2 and B_2 leading to a delay considered for both tasks of 100 cycles. This offers the earliest finish time for the task set compared to the previous solutions.

3 Background and Related Work

As stated before, the concept of multi-phase representation of tasks was introduced with PREM [22], which is often referred to as the 2-phase model, and AER ([9, 23]), also known as the 3-phase model. These models have since been successfully used in optimizing compiler methods (e.g. [11]) to suppress contentions, and in conjunction with online scheduling algorithms in order to tolerate contentions and to bound their effect [2]. Recently, a version of the multi-phase model featuring an arbitrary number of phases has been proposed simultaneously in [18], using Time Interest Points, and in [7], using Single-Entry-Single-Exit intervals.

Marmot builds upon and improves existing techniques, which are presented in this section, used to statically analyze binaries in order to produce memory access profiles. Section 3.1 first presents the Implicit Path Enumeration Technique (IPET) [16], originally used for WCET calculation but also used by EAC [20] to generate a distribution of memory accesses in the code of a task. Section 3.2 then presents two recent works (StAMP [7] and EAC [20]) used and improved by Marmot to extract memory access profiles. Details on the equation system used by EAC and based on IPET are given in Section 3.3.

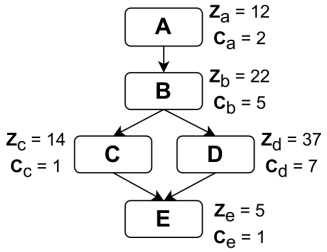


Figure 5: The CFG of a program containing an if-then-else construct. Each basic block i is annotated with its local WCET Z_i and event contribution C_i (see Section 3.3).

3.1 WCET calculation using IPET

Most WCET computation techniques rely on the Control-Flow Graph (CFG) representation of a program to perform their analysis. A CFG is a directed graph describing the possible flows of execution of a program. CFG nodes are basic blocks, defined as straight-line code sequences with no branches in except to the entry and no branches out except at the exit. Directed edges represent the control flow between basic blocks.

IPET, as originally introduced in [16], is a WCET computation technique that relies on the CFG of a program to identify its longest execution path. IPET translates the WCET calculation problem into an Integer Linear Programming (ILP) equation system, in which the integer variables to be calculated represent the execution counts of nodes and edges along the longest execution path. Linear equations represent the *constraints* on the control flows in the CFG, that stem from the structure of the code (loops, conditional constructs). The objective function to be maximized in the ILP system to calculate the WCET is the sum of the timing contributions of all nodes.

As an example, let us consider the CFG displayed in Figure 5, where each node i is annotated with its worst-case timing contribution in isolation Z_i. In the system of equations, variables n_i represent the execution counts of nodes (e.g. n_A for node A) whereas variables $e_{i,j}$ represent the execution counts of edges between nodes (e.g. $e_{A,B}$ for the edge from node A to node B).

The constraint representing the possible paths after node B is shown in Equation (1). The two outgoing edges from node B, e_{BC} and e_{BD}, are linked to the node's variable n_B. The sum operation works as an OR operator: only one of the two edges can be considered for every execution of node B.

$$e_{B,C} + e_{B,D} = n_B \tag{1}$$

In the same manner, other constraints limit the number of iterations of loops and account for function calls that alter the control flow. The objective function to be maximized on the example is given in Equation (2).

$$Maximize : 12n_A + 22n_B + 14n_C + 37n_D + 5n_E \tag{2}$$

In our example, only two possible paths can be taken in the CFG: $A-B-C-E$ or $A-B-D-E$. As D holds a bigger timing contribution than C (37 against 14), the maximal solution corresponds to path A-B-D-E, for an overall WCET of 73.

3.2 Techniques for memory access profile extraction

Marmot is based on two recent techniques that extract memory access profiles from binary codes: StAMP [7] and EAC (Event Arrival Curves) [20].

StAMP produces profiles made of consecutive *intervals*, each of them having its own WCET and worst-case number of memory accesses (WCMA). To compute its profiles, StAMP first extracts Single-Entry Single-Exit (SESE) regions in linear time [15] from the binary code of the task. SESE regions are defined as sub-graphs of the CFG that hold a unique entry and exit point where the execution flow always passes. Intervals are built as the aggregation of one or more SESE regions. The properties of SESE regions (unique entry and unique exit) guarantee that the intervals built from the SESE regions are consecutive and non-overlapping. As intervals are defined as sub-graphs of the CFG, state-of-the-art static WCET estimation tools can then be used to calculate the WCET and WCMA of each interval. The modification of WCET estimation tools to calculate WCET and WCMA are minimal, because the SESE property holds for intervals. Examples of intervals as extracted by StAMP are the two intervals of task R as depicted in Figure 3.

An Event Arrival Curve (EAC), as defined in [20], is a curve that gives the worst-case number of *events* that can occur during the execution of a task in any time window of a given length. Events, as defined in [20], are any action that can occur in the code of a task. The curve is computed using modified IPET equations: (i) the objective function in the modified IPET formulation maximizes the number of events instead of the timing as in the original IPET; (ii) the time window on which the number of events has to be evaluated is a constraint in the ILP formulation. The solving of multiple IPET equations for different timing windows allows the profile to hold a *distribution* of events in the form of a curve instead of a single value for analyzed tasks. An example of EAC is the curve of the second interval of task B in Figure 3.

Events may be any action that occurs within the code of a task, such as peripheral accesses or memory accesses. However, the IPET formulation introduced in [20] is *non-contextual*: it is assumed that the event occurs in *all* execution contexts of the task. When applied to the estimation of WCMA, the obtained curve is thus overly pessimistic in architectures with caches, because it assumes that a memory access occurs all the time. As further detailed in Section 4 Marmot reduces this pessimism through the introduction of a *contextual* IPET formulation which is an original contribution of the paper.

3.3 Equation system used to produce EACs

This Section details the equation system, inspired from IPET, defined in [20] to calculate Event Arrival Curves (EACs), with a non-contextual occurrence of events. Table 1 defines the notations used in these equations and all equations in this paper.

C_i is the event contribution from basic block i, whereas Z_i is its timing contribution. Both values are considered as constants in the IPET formulation. The total timing and event contributions of a basic block i depend on the number of times a flow enters the basic block, as expressed in Equations (3) and (4). Expression $\sum_{j \epsilon \mathcal{P}_i} e_{j,i}$

	Definitions	
Sets	$\mathcal{B}$	Set of basic blocks
	$\mathcal{P}_i$	Predecessors of basic block i
	C_i	Event contribution of basic block i
	Z_i	Timing contribution of basic block i
Variables	c_i	Event contribution of basic block i in the critical path
	z_i	Time contribution of basic block i in the critical path
	$e_{j,i}$	Execution count of edge from j to i in the critical path
	$entry\_edge_l$	Execution count of outer edge leading to l's loop header
Annotations	$first$	First context of execution of a loop
	$next$	Next context of execution of a loop
	cc_x	Context x of execution of a function

Table 1: Notations used in the equation system. Annotation can be attached to sets and variables.

in the equation expresses the number of times node i is executed, by accumulating the execution counts of preceding edges.

$$c_i = \sum_{j \epsilon \mathcal{P}_i} e_{j,i} C_i \tag{3}$$

$$z_i = \sum_{j \epsilon \mathcal{P}_i} e_{j,i} Z_i \tag{4}$$

To find a path with the worst event contribution, the equation system of EAC implicitly considers all possible sub-paths inside a CFG, in the sense any node can be a starting or ending node in the path. Specific constraints ensure that only one of the nodes is the starting and/or ending node. Other constraints, similar to the original IPET, express the possible control flows in the CFG, and specify loop bounds. For space considerations, the reader is referred to the original publication [20] for the complete list of flow constraints.

The EAC curve is computed by solving multiple systems of equations for different sizes of *timing windows* ranging from 1 cycle to the WCET of the task. Equation (5) constrains the timing of the program to be lower than the window size, by summing the individual contributions of individual basic blocks.

$$\sum_{i \epsilon \mathcal{B}} z_i \leq window\_size \tag{5}$$

Finally, the objective function aims at maximizing the event contribution of the final path found.

$$maximize : \sum_{i \epsilon \mathcal{B}} c_i \tag{6}$$

The event contribution C_i for each basic block i is context-agnostic. Accounting of the results of static cache analysis techniques (e.g. a memory reference in a loop results in a miss at its first occurrence and in hits at the following occurrences) cannot be directly integrated in EAC, except if all accesses are considered as misses, which results in very pessimistic curves. Through Marmot, we propose a contextual modeling of event counts to obtain precise profiles.

4 Marmot: Generation of Fine-Grain Memory Access Profiles

4.1 Architecture model and assumptions

Our work targets multi-core systems with shared memory (DRAM). Each core may have a local cache hierarchy (local L1 instruction and/or data caches in the simplest configuration or a more complex local cache hierarchy).

It is assumed that for every memory reference, to code or data, a static cache analysis is able to determine if the reference will be served by the core-local cache hierarchy or may be served by the shared memory, and thus may suffer from contention. More precisely, it is assumed that for every instruction, data and instruction cache analyses are led to provide a Cache Hit Miss Classification (CHMC)[1] with the following Categories [1, 13]:

- Always-Hit (AH): the reference will always result in a cache hit,
- Always-Miss (AM): the reference will always result in a cache miss,
- First-Miss (FM): the reference could neither be classified as hit or miss the first time it occurs but will result in cache hits afterwards,
- Not-Classified (NC) in all other cases.

Some data cache analysis methods can also analyze accesses performed by loops over their entire execution span [6, 25]. In this case, the number of shared memory accesses is provided for a loop instead of specific instructions inside of it. Integration of these classes of analyses in Marmot is left for future work. Marmot is still applicable when no static cache analysis exists for the core-local cache hierarchy. In such a case, all memory accesses will be classified as NC, which will inevitably lead to pessimism.

4.2 Overview of Marmot

The outcome of Marmot is a sequence of intervals, each holding a distribution of shared memory accesses, as explained below:

- *Intervals.* An interval is a subset of the task's CFG, with a single entry-point and a single exit-point. Intervals are consecutive: the exit-point of each interval is the entry-point of the next interval in the sequence of intervals. Each interval is characterized by its WCET (considered in isolation, i.e. excluding interference delays), and a distribution of shared memory accesses, as explained just after.
- *Distribution of shared memory accesses.* For each interval, the WCMA is defined as a distribution of shared memory accesses, through a non-decreasing step-function, that for any time t since the start of the interval, counts the worst-case

[1] On a multi-level cache hierarchy, only the last-level cache misses are considered as shared memory accesses.

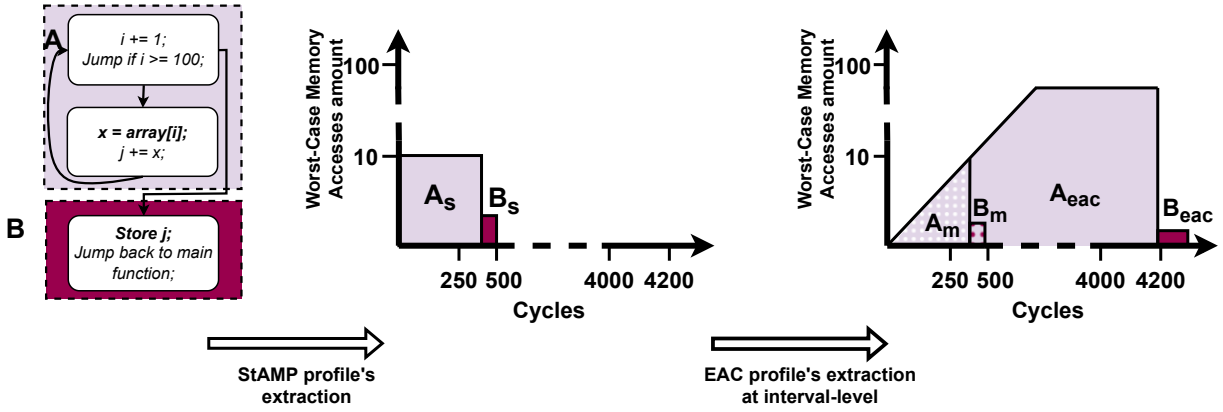


Figure 6: Marmot's steps to compute its memory access profiles from the binary code of a task.

number of memory accesses that may occur since the interval start. The last point in the curve counts the maximum number of accesses in the entire interval.

Marmot builds memory access profiles by combining StAMP's ability to decompose a CFG into intervals, and the equation system of EAC to derive a distribution of shared memory accesses for each interval. Figure 6 illustrates Marmot's workflow on a simple example, in which the task under analysis is composed of a loop that loads a new array element at each iteration, followed by a simple epilogue. Marmot proceeds in two steps:

(1) The first step consists in decomposing the CFG (depicted on the left, with memory access instructions in bold font) into intervals. This is achieved using StAMP with no modification. In the example, two intervals are found: A, in light purple, and B, in dark purple. The profile of these intervals as computed by StAMP is then a flat curve with a single WCMA value per interval (A_s and B_s in the middle of Figure 6, with s standing for StAMP).
(2) The second step refines the StAMP's profiles to obtain distributions of memory accesses in the intervals. An application of the original equation system of *EAC*'s results in the light (resp. dark) purple interval denoted A_{eac} (resp. B_{eac}) in the right-most part of Figure 6. However, simply applying the equations of *EAC* that lack of contextual information would result in pessimistic WCET and WCMA for the intervals, as each access would be considered as resulting in a cache miss. This would result in the profile A_{eac} having very large WCET and WCMA counts. To avoid this pitfall, in Marmot we extend the equation system with contextual information. The result of our example is pictured in dots in the right-most part of Figure 6, with each interval annotated m for Marmot. Note that A_m and B_m both hold the same WCET and final WCMA value as A_s and B_s, but the representation of the WCMA as a step function that slowly increases is more precise than a flat integer value over the whole interval.

As opposed to the previously mentioned techniques, the next two subsections present our own contribution that extends the EAC's equation system to account for contextual memory accesses. Section 4.3 presents the consideration of contexts for loops, while Section 4.4 presents the consideration of calling contexts. A comparatively minor and straightforward modification of EAC was to constrain the paths under consideration to start at the beginning of an interval.

4.3 Improved Loop handling in EAC equations

Static cache analysis tools provide a different number of shared memory accesses and timing contribution for instructions inside of loops depending on their *context* of execution (first occurrence versus next occurrences). A typical situation is the First-Miss (FM) classification of a *Load* instruction inside a loop, in which the instruction could neither be classified as *hit* or *miss* the first time it occurs but will result in cache *hits* afterwards.

To accurately take into consideration these *contexts*, the event contribution C_i and timing contribution Z_i of a basic block i in the equation system of the original EAC are not represented as a single value anymore. For the remainder of this section, we will refer to the two possible contexts inside loops as the *first* context and the *next* context. To implement these two contexts, we split the edge variables in two, to express the possible contexts: $e_{j,i}^{first}$ for the first occurrence and $e_{j,i}^{next}$ for the next.

$$e_{j,i} = e_{j,i}^{first} + e_{j,i}^{next} \tag{i}$$

The final event and timing contribution of nodes inside a loop (c_i and z_i) are modified in the equation system to refer to the new edges and results of static cache analysis (values of C_i and Z_i for the first and next occurrence). These contributions were defined in equation (3) and equation (4) in the original EAC equations. Their modifications are shown in equation (ii) for the event contribution and equation (iii) for the timing contribution.

$$c_i = C_i^{first} \sum_{j \in P_i} e_{j,i}^{first} + C_i^{next} \sum_{j \in P_i} e_{j,i}^{next} \tag{ii}$$

$$z_i = Z_i^{first} \sum_{j \in P_i} e_{j,i}^{first} + Z_i^{next} \sum_{j \in P_i} e_{j,i}^{next} \quad \text{(iii)}$$

Finally, the *first* occurrence is constrained by limiting the e^{first} edges in equation (iv). In this way, the *first* context is only considered once at most.

$$\sum_{j \in P_i} e_{j,i}^{first} \leq 1 \quad \text{(iv)}$$

The definition of the *first* occurrence for instructions in loops subtly differs among tools, and needs to be carefully accounted for to avoid under-approximations of the number of shared memory accesses, leading to unsafe profiles. WCET estimation tools such as Heptane [12, 13] compute a single *first* context value for instructions in nested loops. These tools can use equation (iv) as is. Other static analysis tools such as Otawa [3] provide multiple *first* contexts for instructions in nested loops. The *first* context of instructions in an inner loop is considered every time the inner loop is entered from the outer loop. To correctly use the cache classification of this class of techniques, we propose equation (v) that limits the number of first occurrences of instructions in an inner loop l to the number of times its entry edge, i.e. the edge that leads from the outer loop to l's loop header, is considered.

$$\sum_{j \in P_i} e_{j,i}^{first} \leq entry\_edge_l \quad \text{(v)}$$

4.4 Improved function call handling in EAC equations

The code of a program is usually composed of several functions, each having its own CFG. When the same function is called from different points in the program, multiple *calling contexts* for the called function become possible, and the number of accesses to shared memory may depend on the calling context. The original equation of EAC, that does not consider such contextual information, has to be modified. The modified equations are based on *virtual inlining* of called functions.

For the remainder of this section, we name *caller* the function holding the function call instruction and *callee* the target function of this call. We name *cc* the *calling context* of a function, expressed with a unique index (cc_1, cc_2 for example).

Virtual inlining operates as follows. It inlines the CFG of the callee inside the CFG of the caller in the equation system. The inlining is *virtual*, no actual CFG modification is performed. *Virtual* edges with unique context identifiers are added between the basic block holding the call instruction and the starting block of the callee, as illustrated in Figure 7. The caller function **main** holds two call sites to the callee function **foo** (A and C), that define the two calling contexts of *foo*. In the equation system, edges are added between each of them and the entry node of **foo** for each context: A leads to E in cc_1 and C leads to E in cc_2. This step is performed at the start of the equation system's generation such that the equations applied to edges of the CFG also consider the virtual edges. Equations concerning the callee's CFG are also annotated with the corresponding index and the exit nodes are linked back to the caller function. Thus, we have two distinct representations of

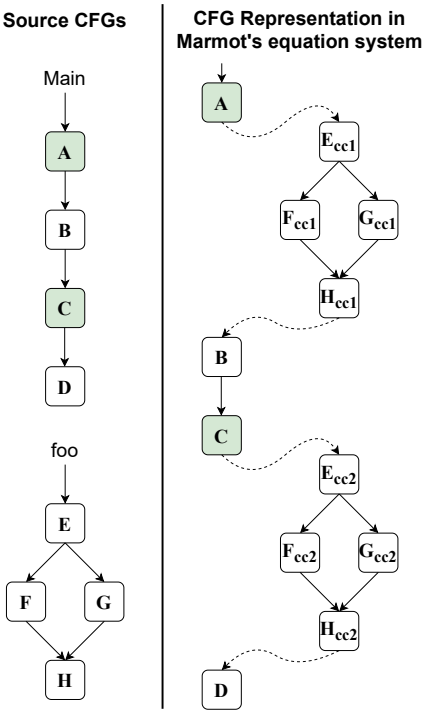


Figure 7: Virtual unrolling of a called function. The left side shows the CFG of two functions composing a program. The green basic blocks hold a call instruction to function *foo*. The right side shows the representation of these CFG inside Marmot equation's system.

foo's CFG in the equation system, one per calling context. In the same manner, functions called inside a callee's function are also inlined in the callee's CFG following a recursive approach until all call instructions are handled.

Static WCET analysis tools (at least those that consider calling contexts) provide a timing and event contribution (C_i and Z_i) per calling context. With the implementation of virtual inlining, we are able to use them in our equation system. Equations (vi) and (vii) show the implementation of contexts in the timing and event contribution of basic blocks. Contributions $c_i\_cc_x$ and $z_i\_cc_x$ are related to the result of the static analysis tools and edges in the corresponding context cc_x.

$$c_i\_cc_x = C_i\_cc_x \sum_{j \in P_i} e_{j,i}\_cc_x \quad \text{(vi)}$$

$$z_i\_cc_x = Z_i\_cc_x \sum_{j \in P_i} e_{j,i}\_cc_x \quad \text{(vii)}$$

Equation (viii) shows an example of our equation system for the code of Figure 7, focusing on node A. The first line shows the addition of the virtual edge in the corresponding context cc_1. All equations concerning the CFG of the foo function are annotated with this context. Lines 2 to 5 show this for the event contribution of nodes. In the same manner, the timing contribution t of each node is expressed but not shown here for size consideration. Finally,

the representation of the edge from A to B in the original EAC equation system is removed and replaced by the addition of a virtual edge $e_{H,B}$. The last two lines show the addition of this edge in the corresponding context cc_1 with the definition of flow entering node B from the new virtual edge.

$$\begin{aligned}
&e_{A,E}_cc_1 = n_A \\
&c_{E}_cc_1 = C_{E}_cc_1 \; e_{A,E}_cc_1 \\
&c_{F}_cc_1 = C_{F}_cc_1 \; e_{E,F}_cc_1 \\
&c_{G}_cc_1 = C_{G}_cc_1 \; e_{E,G}_cc_1 \\
&c_{H}_cc_1 = C_{H}_cc_1 \; (e_{F,H}_cc_1 + e_{G,H}_cc_1) \\
&e_{H,B} = n_{H}_cc_1 \\
&n_B = e_{H,B}
\end{aligned} \tag{viii}$$

5 Experimental evaluation

After a short presentation of the experimental setup in Section 5.1, Section 5.2 demonstrates the interest of considering contexts when extracting memory access profiles. Section 5.3 shows through an example how the Marmot profiles can be used to produce schedules with less total interference delays than related techniques. Finally, Section 5.4 compares Marmot's profiles to state-of-the-art profiles (both coarse grain profiles and single WCMA value per interval profiles) on benchmarks.

5.1 Experimental setup

We run our experiments on the TACLe benchmarks collection [10], with loop bound annotations expressed in the annotation format of the static WCET analysis tool Heptane [13]. Heptane is used for interval determination and for WCET estimation at the task and interval levels.

The TACLe benchmarks collection can be divided in two categories: (i) sequential benchmarks holding more or less complicated code structure and (ii) more complex use-cases that were developed for multi-core analysis, namely the Papabench [19], Rosace [21] and debie1 [14] benchmarks. To qualify our results, we extracted profiles for all sequential benchmarks and all three use-case aforementioned which amounts to 76 benchmarks[2]

We target MIPS code with a single-layer of caches (one data cache and one instruction cache). Both caches are 2-way associative LRU caches with 64-bytes cache lines and 256 cache sets. Load and store instructions are assumed to take 1 cycle on cache hits and 50 cycles on cache misses, due to the subsequent shared memory access. The identification of memory accesses relies on the Heptane static cache analysis technique that produces a cache classification for each instruction and data access.

The data address analysis of Heptane, in case the target of a pointer cannot be determined precisely, considers that the pointing instruction may access *all* the memory, leading to combinatorial timing complexity during analysis. To produce our profiles in reasonable time, we therefore modified the data address analysis to consider a single unique *unknown address marker* for instructions accessing data through a pointer. An *unknown address marker* is an address outside of the memory range of the program under analysis. Because these addresses are outside of the memory range and are unique, they are always considered as misses in the data cache analysis, which is the worst possible case.

As benchmarks under analysis can perform up to millions of shared memory accesses, a choice was made between precision and time spent solving the equation systems. We re-used the *time granularity* parameter presented in the original EAC paper, that safely limits the number of steps in the step-function. For space consideration, we do not detail this parameter here and refer the reader to the original EAC paper. The granularity chosen for the profiles used in our experiments and shown in the Appendix is of 0.001% times the WCET. For example, a profile with a WCET of 1 millions cycles will hold one new step in its curve every 1000 cycles.

5.2 Comparison of contextual and non-contextual memory access profiles

The addition of contextual information in Marmot's equation system is motivated by the gain of precision in the resulting profiles. We emphasize the importance of this original feature by comparing *non-contextual* profiles to *contextual profiles*. Contextual profiles were obtained by using the original cache classification obtained with Heptane. On the other hand, non-contextual profiles were obtained by first modifying the cache classification obtained with Heptane to modify all FM accesses to NC, thus loosing contextual information, before applying Marmot's equation system.

We extracted contextual and non-contextual profiles for all 76 benchmarks considered in the TACLe benchmark suite. For every benchmark, we compared the cumulative WCMA and WCET values held by both profiles. On average, the WCET for non-contextual profiles is 1.2 times higher than the WCET for contextual profiles and the WCMA for non-contextual profiles is 4.5 times higher than the WCMA for contextual profiles.

Integrating contextual information in the analysis can thus significantly improve the precision of the profiles, and in turn tighten the calculation of interference delays.

5.3 Example of use of profiles produced by Marmot

This section demonstrates the interest of using Marmot's profiles for IA-WCET computation using two TaCLE benchmarks, *complex_updates* and *statemate*. This experiment was performed through a pen-and-paper direct transposition of the IA-WCET calculation technique of [8]. Moreover, we assume a time-triggered off-line scheduling algorithm for the experiments. As mentioned earlier, taking benefit of Marmot's profiles in other scheduling algorithms is left for future work.

5.3.1 Scheduling setup. Figure 8 displays the StAMP and Marmot memory access profiles extracted from *complex_updates* and *statemate*. On the left-most part (resp. right-most part), the profile obtained for *complex_updates* (resp. *statemate*) by StAMP is depicted in blue (resp. red) and the profile obtained by Marmot is depicted in light blue (resp. light red). To keep our example concise, *statemate*'s

[2]Due to the presence in the binary code of instructions not handled by Heptane, we were unable to compute a memory profile for the *susan* sequential benchmark. Heptane failed to compute the WCET of sequential benchmarks *md5* and *mpeg2*, preventing us from computing a memory access profile.

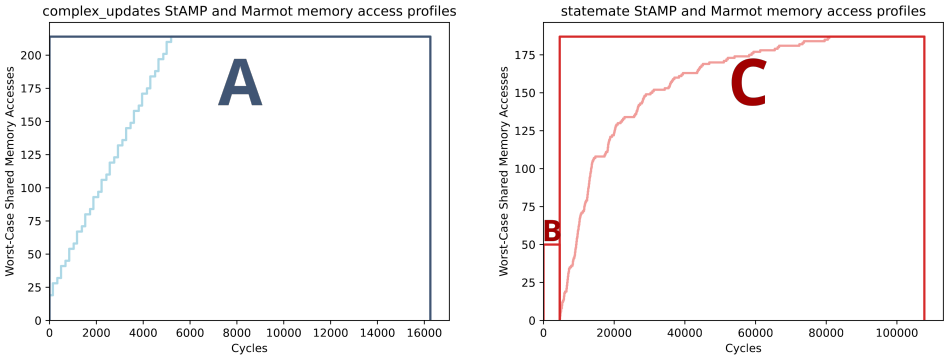


Figure 8: StAMP and Marmot's profiles for the *complex_updates* (left-most figure) and *statemate* (right-most figure) benchmark.

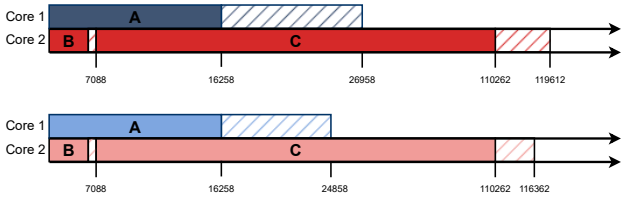


Figure 9: Schedulability analysis results when relying on StAMP's (resp. Marmot's) information for IA-WCET computation on the top-most (resp. bottom-most) part of the figure. WCET is drawn in solid color and interference delays in stripes.

memory access profile was simplified by merging intervals preceding the main interval C in a single interval B. The merging was done through the addition of the WCET and WCMA of all intervals preceding C. For the remainder of this section, the variants of the profiles obtained with Marmot (resp. StAMP) are labeled with a m (resp. a s).

We choose to schedule interval A of the *complex_updates* benchmark on a core in parallel with intervals B and C of the *statemate* benchmark on another core. In our schedule, A and B share the same starting date. Interval C, which takes place after B's execution in the flow of execution of *statemate*, is scheduled to start as soon as B's IA-WCET date is passed. We consider a simple Round-Robin arbitration policy for the shared memory, meaning every shared memory access requested by a core can suffer from at most one contention per parallel core. We consider a 50 cycles delay per contention that corresponds to the latency of a memory access in isolation.

5.3.2 Computing IA-WCET using Marmot profiles. Following the work described in [8], IA-WCET computation using Marmot's profiles is performed iteratively. We present the first step in this process when computing interference between intervals A and C.

- The initial end date of A is computed as its start date (0) plus its WCET in isolation (16258 cycles): 16258 cycles. As a result, A finishes within the execution span of C. To compute the worst case number of contentions between both tasks, we look at the worst-case number of memory accesses held by the memory access profile C_m at date 16258, here 103. Since A_m holds a value of 214 WCMA at 16258 cycles, we keep the value of 103 as the (temporary) worst-case number of contentions between both intervals.
- This value is used to update the IA-WCET of A and C. Using a penalty of 50 cycles per contention, the end of A is postponed from cycle 16258 to 21408. In the same fashion, the end date of C is postponed by 5150 cycles.
- Now that the end date of A has been postponed, we must consider the effect of accesses that C may perform between the previous and the postponed end date of A. These potential accesses can also interfere with the accesses of A, postponing its end date again. We thus have to repeat the previous steps until a fixed point is reached.

Table 2: IA-WCET computation steps by scheduling interval A alongside interval C.

Step	Total number of contentions	A_m's IA-WCET	C_m's curve WCMA	Additional contentions
1	0	16258	103	103
2	103	21408	120	17
3	120	22258	122	2
4	122	22358	122	0

Table 2 shows all the steps for the computation of the IA-WCET of A in this example. For each step of the fixed point computation, we display the total amount of contentions accounted for and the current IA-WCET (ending date) at the start of the step, the WCMA held by C_m corresponding to the current end of A and the amount of contentions that must be added to A's ending date.

5.3.3 IA-WCET comparison between profiles. Figure 9 displays on the top-most part (resp. bottom-most part) the schedulability analysis result obtained using the StAMP's (resp. Marmot's) profiles of intervals. The WCET is shown in solid color and the additional interference delay cycles in stripes.

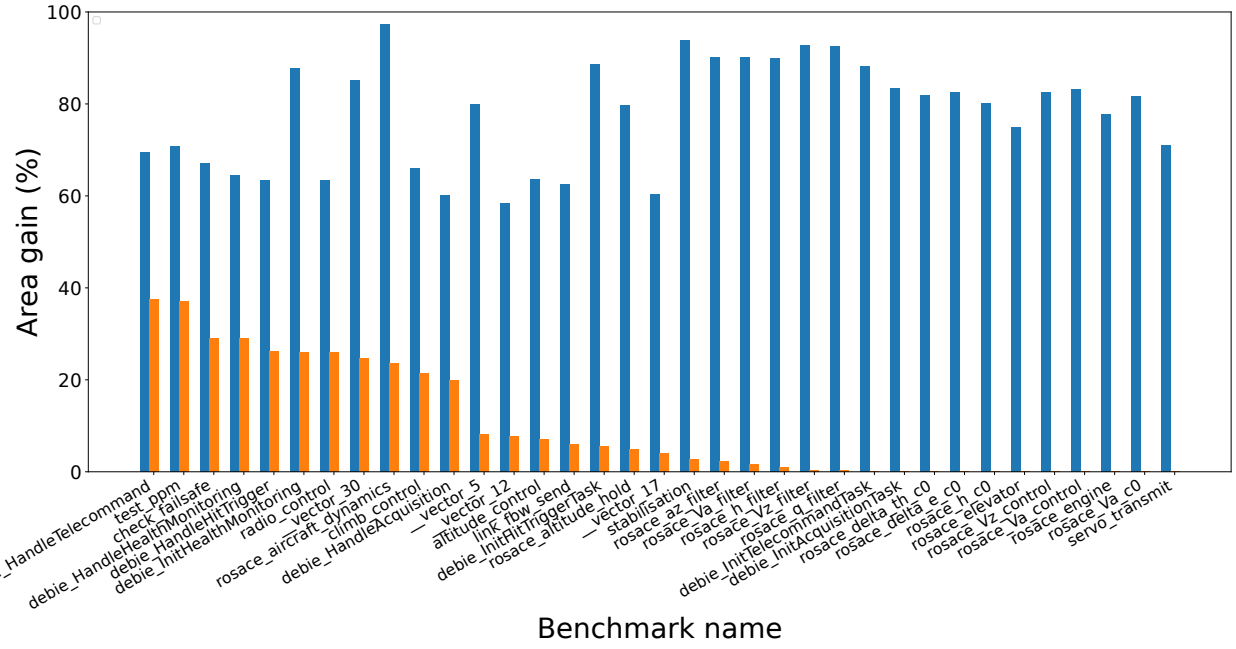


Figure 10: Area gain in percentage for each multi-core use-case benchmarks when comparing Marmot to StAMP (in orange) and coarse-grain profiles (in blue)

In the first schedule, the IA-WCET was computed using the single WCMA value held by the StAMP profile of intervals: 214 for A_s, and 187 for C_s. We account $min(214, 187) = 187$ contentions for C and $min(214, (187 + 50)) = 214$ contentions for A following the interference caused by B. The finish date of the first core is 26958 cycles and the finish date of the second core is 119612 cycles.

In the second schedule, the IA-WCET of intervals was computed using the technique presented in Section 5.3.3. Because A finishes at a date where C_m curve is lower than C_s, the worst-case number of contentions to consider can be effectively reduced. The finish date of the first core is 24858 cycles and the finish date of the second core is 116362, a reduction of 2100 cycles and 3250 cycles respectively.

Schedulability analysis can thus tighten its interference delay result when relying on Marmot's curve inside a memory access profile.

5.4 Comparison of Marmot's profiles with state-of-the-art solutions

This section is devoted to a comparison between Marmot's profiles and state-of-the art contextual profiles, for all considered TACLe benchmarks. The state-of-the-art profiles considered are coarse-grain profiles holding a single WCMA value per task and StAMP profiles holding multiple intervals per task with each interval holding a single WCMA value. All analyzed benchmarks can be found in the Appendix.

To stay independent from the scheduling algorithm, the comparison metric is only based on the difference of *shapes* of the profiles' curves. The employed metric relates to the size of the *areas* under the profile curves (the lower the better, as illustrated previously by the light blue and light red curves in Figure 8). More precisely, if a_{Marmot} represents the area under Marmot's curve and $a_{baseline}$ the area under a baseline solution, the metric we use to express the gain is the difference of area expressed in percentage:

$$gain = 1 - \frac{a_{Marmot}}{a_{baseline}}$$

Results for all tasks of the three multi-core use-case benchmarks (namely Papabench, Rosace and debie1) are shown in Figure 10. The figure shows the gain for each benchmark, when comparing Marmot to StAMP in orange and to coarse-grained profiles (i.e. profiles with a single WCMA for the entire task) in blue. For these benchmarks, we observe an average gain of 77.9% when comparing Marmot to the coarse-grain profiles and an average gain of 10.1% when compared to StAMP.

For the sequential benchmarks, we observe an average gain of 75.8% when comparing Marmot to the coarse-grain profiles and an average gain of 15.6% when compared to StAMP.

As can be seen in Figure 10, half of the benchmarks analyzed show little to no difference between the Marmot and StAMP profiles, with gains in area ranging from 0% to 5%. This lack of gain can also be found in one out of three sequential benchmarks analyzed. Following observations on the source code, we have identified two categories of benchmarks where the code structure hinders the extraction of a refined curve:

- A first category of benchmarks loads all the required data from the shared memory once before entering a main loop where most of the execution time is spent, or data-loading instruction in the main loop performs shared memory accesses only on their first occurrence. Thus, the WCMA rapidly rises at the start of the profile before stagnating at the highest WCMA point reachable for the entirety of the profile as no

new shared memory accesses can be described. This is the case for the *insertsort* sequential benchmark, for example.
- A second category of benchmarks holds very small code size with only one or two basic blocks per interval. As in EAC, Marmot's equation system neutralizes the timing contribution of the starting and ending basic block while keeping their WCMA contribution to remain safe (see [20]). As the code is only composed of these two basic blocks, the WCMA contribution of the entire program is totally accounted for even for the smallest time window considered. This results in a profile holding a curve which instantly rises to the highest WCMA point. This is the case for 12 out of 15 benchmarks analyses from the Rosace use-case.

We believe Marmot's potential compared to other techniques lies in its ability to describe shared memory accesses during the executions of loops. For tasks where no shared memory accesses are performed inside of a loop, or where instructions in loops only perform shared memory accesses on their first occurrence, the contribution of Marmot is reduced.

6 Conclusion

This paper introduces Marmot, a static memory profile extraction technique. Compared to state-of-the-art solutions, profiles computed using Marmot are split in intervals, each holding a distribution of WCMA in the interval instead of a single value. We illustrated how the produced profiles can be leveraged to tighten interference delays during IA-WCET computation and schedulability analysis. We believe that further work should be focused on the use of Marmot's profiles for different scheduling algorithms. Offline schedulers can use the profiles to reduce the interference delays. On-line schedulers can perform new estimations of interference delays at run-time by using the *interval-to-code* correspondence of our profiles. Future work can also focus on further refining the information provided by Marmot's profiles. For now, all memory accesses performed along a path found by solving the equation system contribute to the considered WCMA. However, accesses performed at the start of these paths could be ignored after a certain point of time when we are sure they have finished.

Acknowledgments

This work was supported by a grant overseen by the French National Research Agency (ANR) as part of the CAOTIC ANR-22-CE25-0011 and JCJC MeSCAliNe ANR-21-CE25-0012 projects

References

[1] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. 1996. Cache behavior prediction by abstract interpretation. In *Static Analysis: Third International Symposium, SAS'96 Aachen, Germany, September 24–26, 1996 Proceedings 3*. Springer, 52–66.

[2] Jatin Arora, Cláudio Maia, Syed Aftab Rashid, Geoffrey Nelissen, and Eduardo Tovar. 2022. Bus-contention aware WCRT analysis for the 3-phase task model considering a work-conserving bus arbitration scheme. *J. Syst. Archit.* 122 (2022), 102345. https://doi.org/10.1016/J.SYSARC.2021.102345

[3] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. 2010. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *Software Technologies for Embedded and Ubiquitous Systems - 8th IFIP WG 10.2 International Workshop, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6399)*, Sang Lyul Min, Robert G. Pettit IV, Peter P. Puschner, and Theo Ungerer (Eds.). Springer, 35–46. https://doi.org/10.1007/978-3-642-16256-5_6

[4] Giorgio C Buttazzo. 2011. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Vol. 24. Springer Science & Business Media.

[5] Thomas Carle and Hugues Cassé. 2021. Static extraction of memory access profiles for multi-core interference analysis of real-time tasks. In *International Conference on Architecture of Computing Systems*. Springer, 19–34.

[6] Siddhartha Chatterjee, Erin Parker, Philip J Hanlon, and Alvin R Lebeck. 2001. Exact analysis of the cache behavior of nested loops. *ACM SIGPLAN Notices* 36, 5 (2001), 286–297.

[7] Théo Degioanni and Isabelle Puaut. 2022. StAMP: Static Analysis of Memory access Profiles for real-time tasks. In *WCET 2022-20th International Workshop on Worst-Case Execution Time Analysis*.

[8] Maximilien Dupont de Dinechin, Matheus Schuh, Matthieu Moy, and Claire Maiza. 2020. Scaling Up the Memory Interference Analysis for Hard Real-Time Many-Core Systems. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 330–333.

[9] Guy Durrieu, Madeleine Faugère, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and Wolfgang Puffitsch. 2014. Predictable flight management system implementation on a multicore processor. In *Embedded Real Time Software (ERTS'14)*.

[10] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. 2016. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016) (OpenAccess Series in Informatics (OASIcs), Vol. 55)*, Martin Schoeberl (Ed.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:10.

[11] Björn Forsberg, Marco Solieri, Marko Bertogna, Luca Benini, and Andrea Marongiu. 2021. The predictable execution model in practice: Compiling real applications for cots hardware. *ACM Transactions on Embedded Computing Systems (TECS)* 20, 5 (2021), 1–25.

[12] Damien Hardy and Isabelle Puaut. 2011. WCET analysis of instruction cache hierarchies. *J. Syst. Archit.* 57, 7 (2011), 677–694.

[13] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. 2017. The heptane static worst-case execution time estimation tool. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[14] Niklas Holsti, Thomas Langbacka, and Sami Saarinen. 2000. Using a worst-case execution time tool for real-time verification of the DEBIE software. *EUROPEAN SPACE AGENCY-PUBLICATIONS-ESA SP* 457 (2000), 307–312.

[15] Richard Johnson, David Pearson, and Keshav Pingali. 1994. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. 171–185.

[16] Yau-Tsun Steven Li and Sharad Malik. 1995. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*. 88–98.

[17] Claire Maiza, Hamza Rihani, Juan M. Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. 2019. A Survey of Timing Verification Techniques for Multi-Core Real-Time Systems. *ACM Comput. Surv.* 52, 3 (2019). https://doi.org/10.1145/3323212

[18] Rémi Meunier, Thomas Carle, and Thierry Monteil. 2022. Correctness and Efficiency Criteria for the Multi-Phase Task Model. In *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*. 16326.

[19] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne De Michiel. 2006. Papabench: a free real-time benchmark. In *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)(2006)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

[20] Dominic Oehlert, Selma Saidi, and Heiko Falk. 2018. Compiler-based extraction of event arrival functions for real-time systems analysis. In *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[21] Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard, and Pierre Siron. 2014. The ROSACE case study: From simulink specification to multi/many-core execution. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 309–318.

[22] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. 2011. A predictable execution model for COTS-based embedded systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 269–279.

[23] Benjamin Rouxel, Stefanos Skalistis, Steven Derrien, and Isabelle Puaut. 2019. Hiding communication delays in contention-free execution for spm-based multi-core architectures. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[24] Muhammad R. Soliman and Rodolfo Pellizzoni. 2019. PREM-Based Optimal Task Segmentation Under Fixed Priority Scheduling. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 133)*, Sophie Quinton (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 4:1–4:23. https://doi.org/10.4230/LIPIcs.ECRTS.2019.4

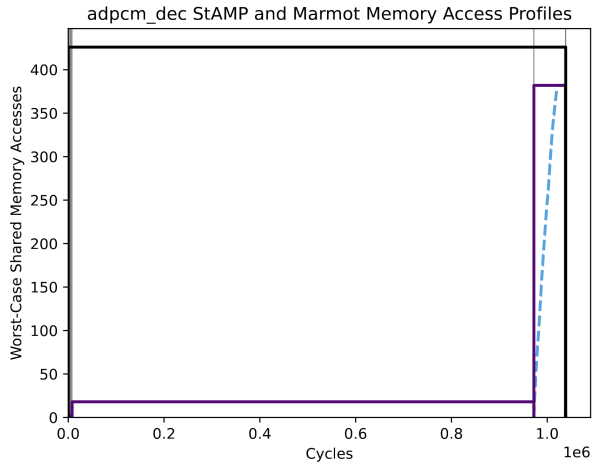
[25] Jan Staschulat and Rolf Ernst. 2006. Worst case timing analysis of input dependent data cache behavior. In *18th Euromicro Conference on Real-Time Systems (ECRTS'06)*. IEEE, 10–pp.

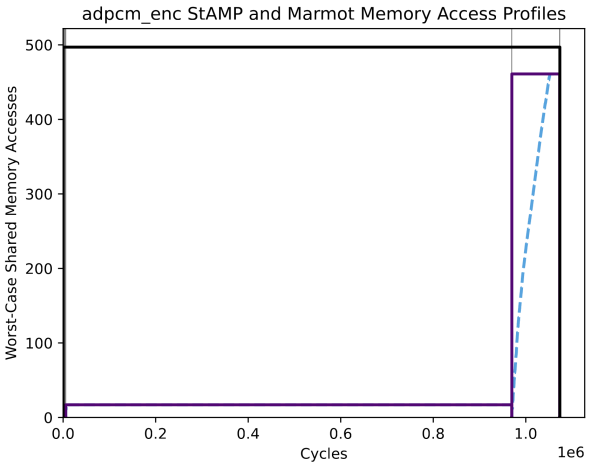
[26] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. 2008. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 3 (2008), 1–53.

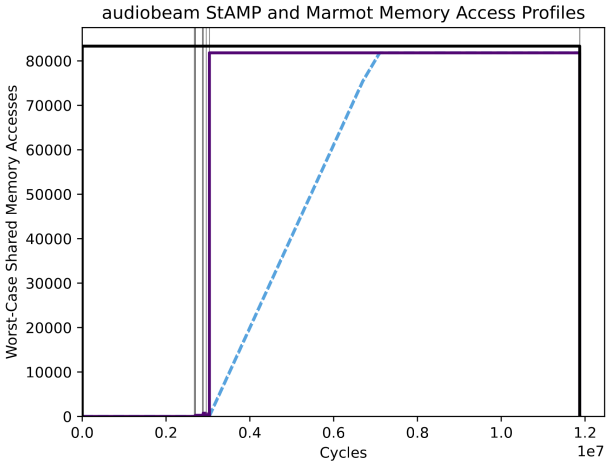
[27] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. 2013. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 55–64.

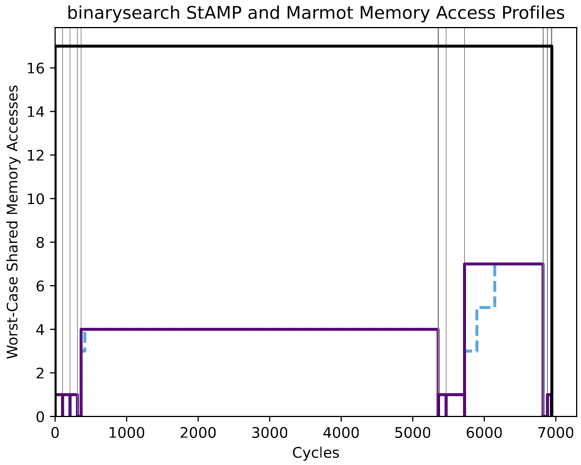
A Appendix

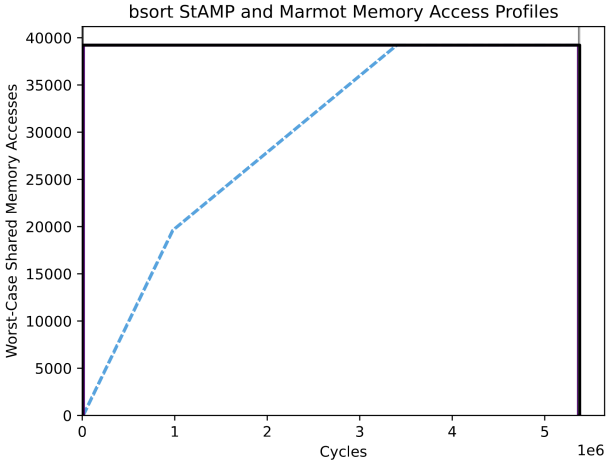
Benchmarks are presented as follows: the coarse-grain profile is drawn with a black solid line, the StAMP profile is drawn with a purple solid line and Marmot's profile is drawn with a light-blue dashed line. Finally, vertical grey lines denote the points of passage between intervals which are produced in StAMP and Marmot.

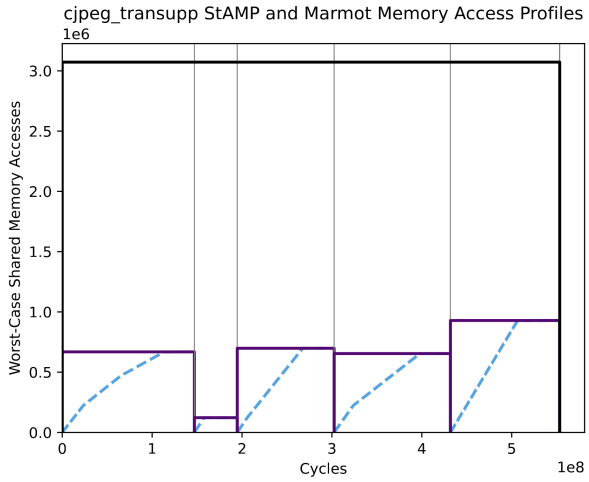

adpcm_dec StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
1e6

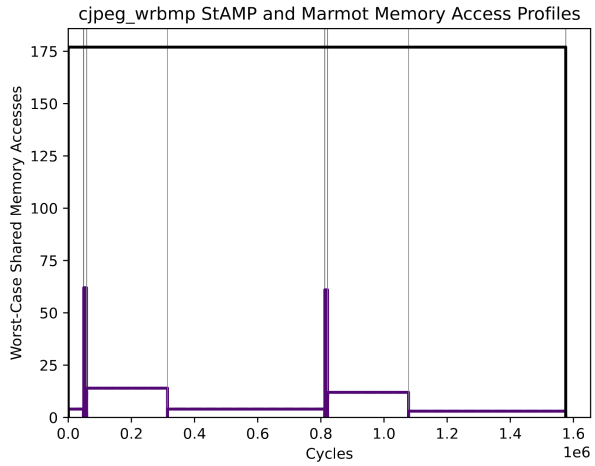

adpcm_enc StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
1e6

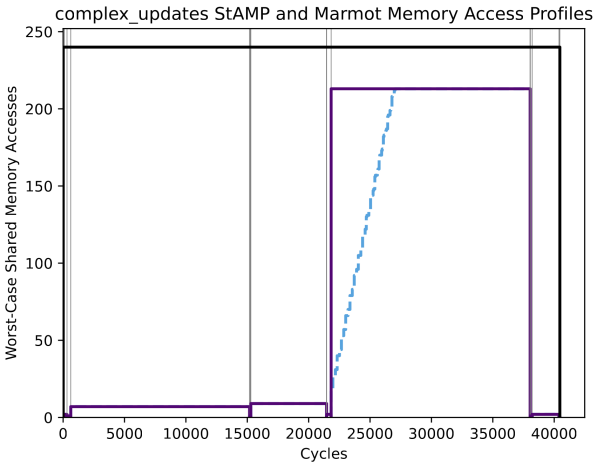

audiobeam StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
1e7

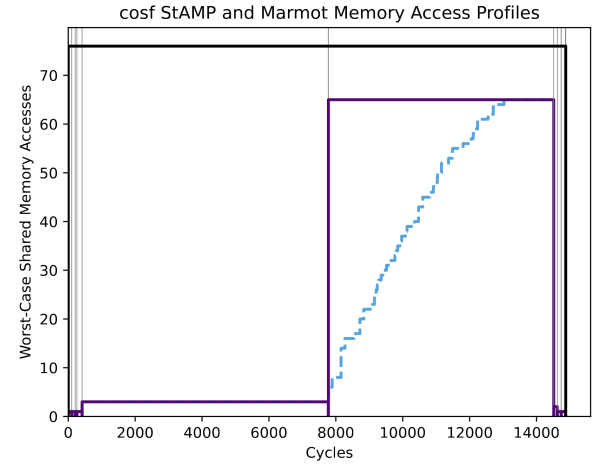

binarysearch StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

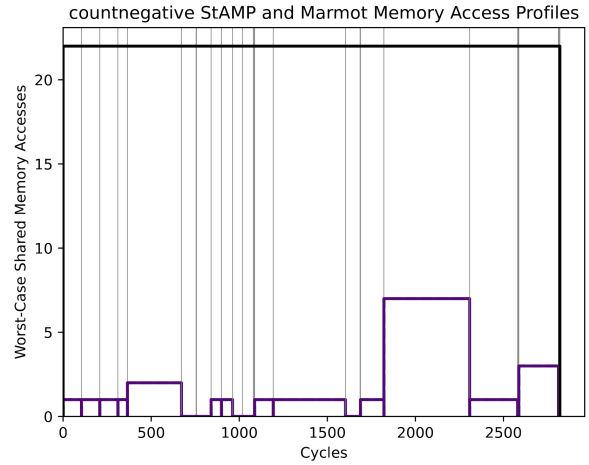

bsort StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
1e6

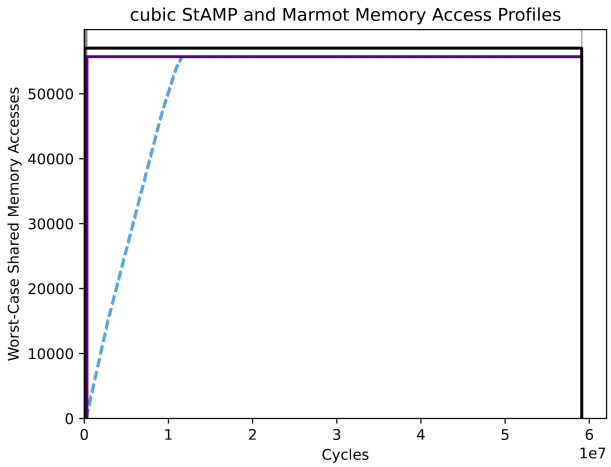

cjpeg_transupp StAMP and Marmot Memory Access Profiles
1e6
Worst-Case Shared Memory Accesses
Cycles
1e8

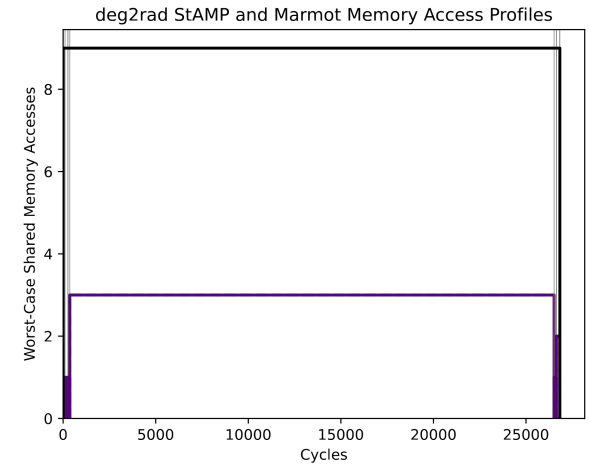

cjpeg_wrbmp StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
1e6

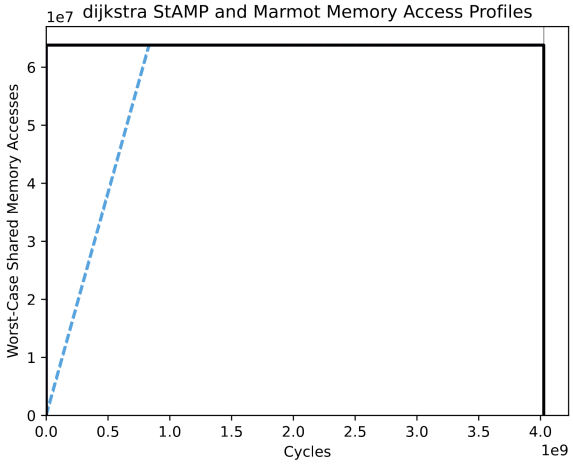

complex_updates StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

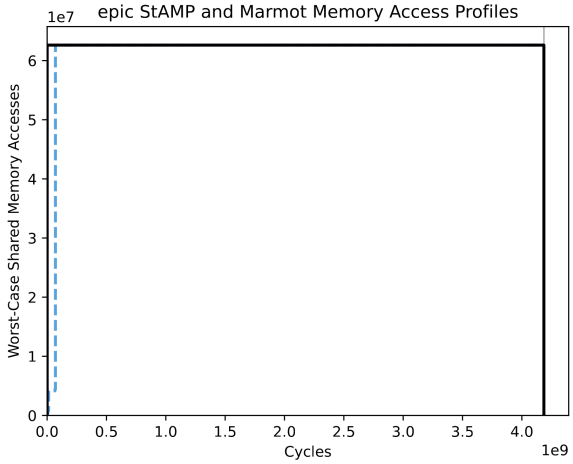

cosf StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

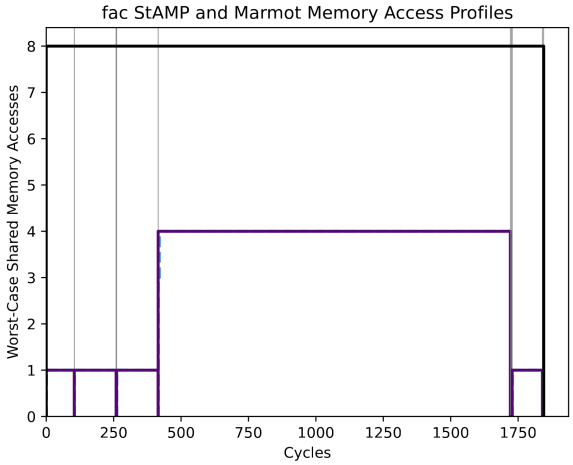

countnegative StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

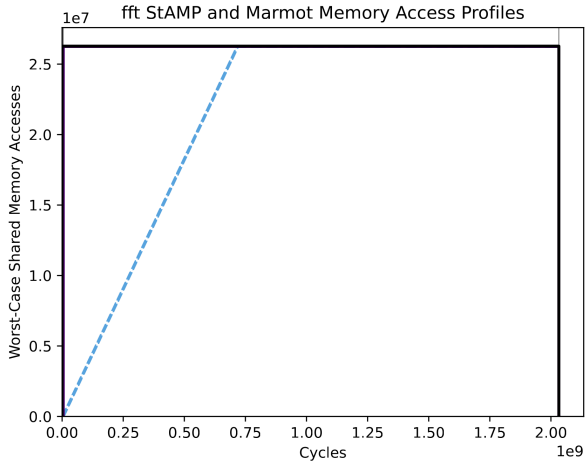

cubic StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
1e7

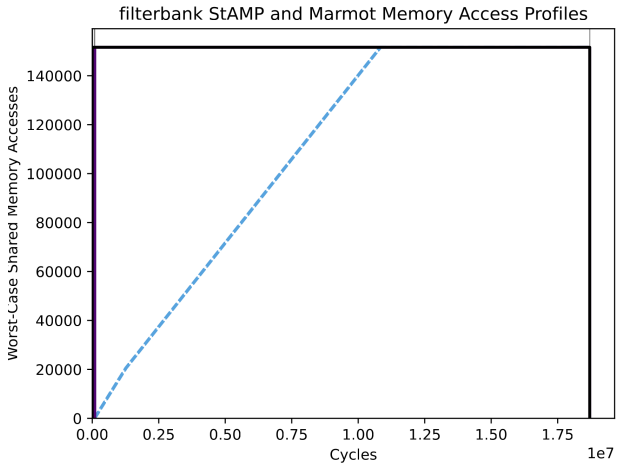

deg2rad StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

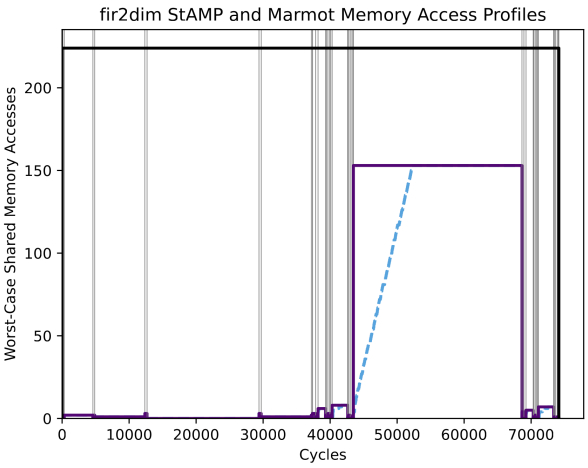

dijkstra StAMP and Marmot Memory Access Profiles
1e7
Worst-Case Shared Memory Accesses
Cycles
1e9

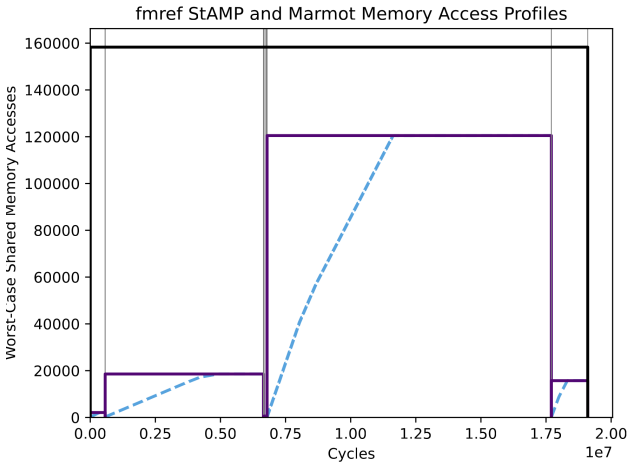

epic StAMP and Marmot Memory Access Profiles
1e7
Worst-Case Shared Memory Accesses
Cycles
1e9

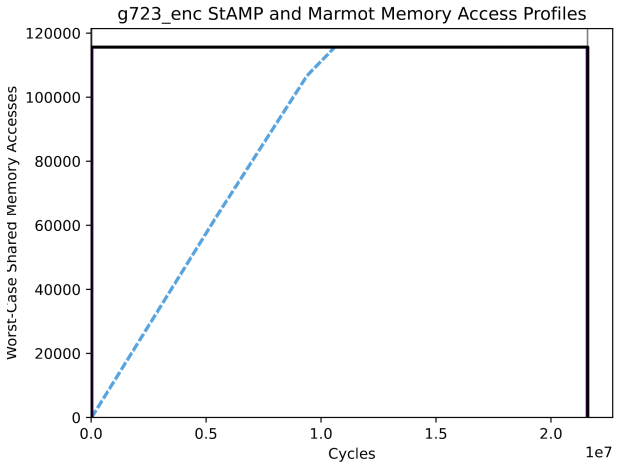

fac StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

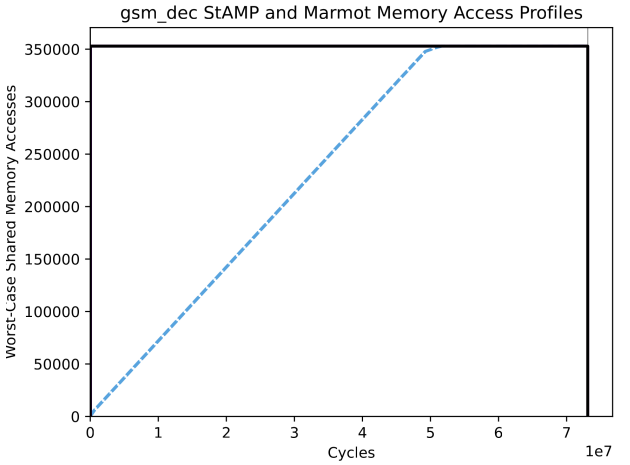

fft StAMP and Marmot Memory Access Profiles
1e7
Worst-Case Shared Memory Accesses
Cycles
1e9

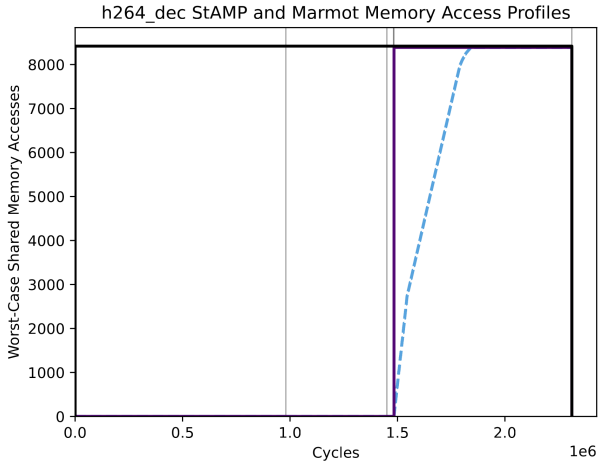

filterbank StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
1e7

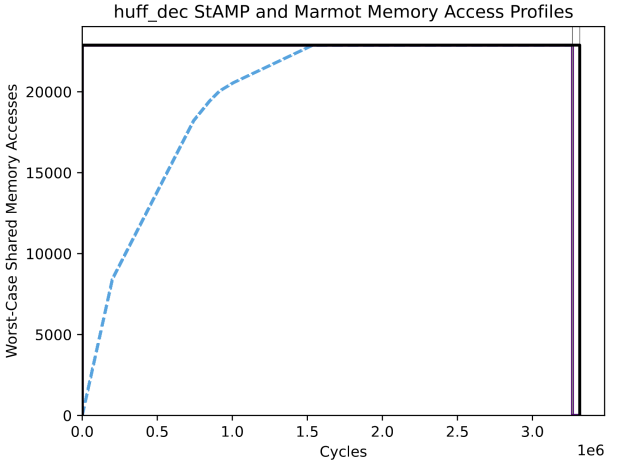

fir2dim StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

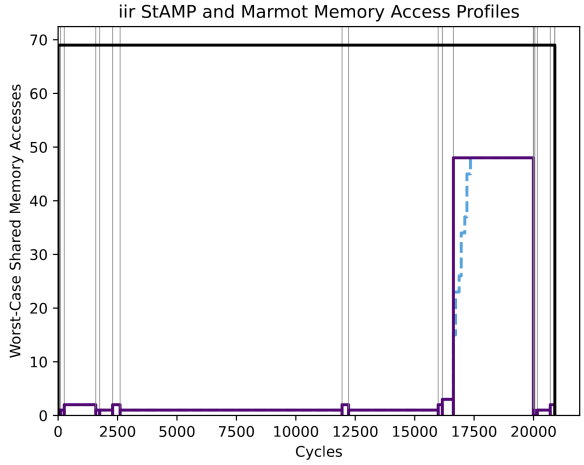

fmref StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
1e7

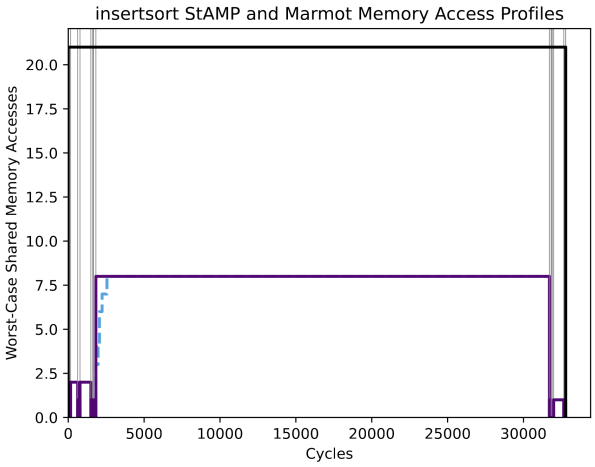

g723_enc StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
1e7

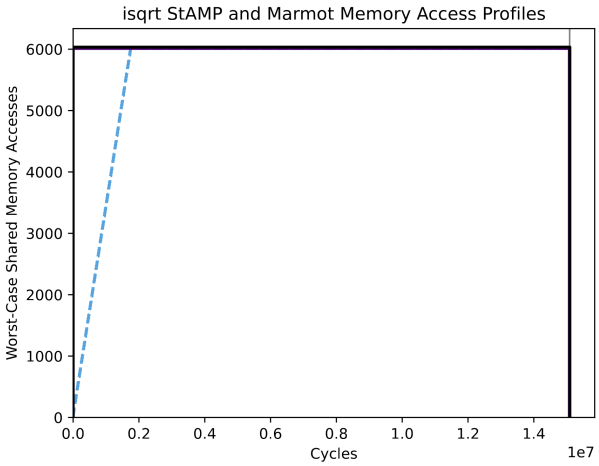

gsm_dec StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
1e7

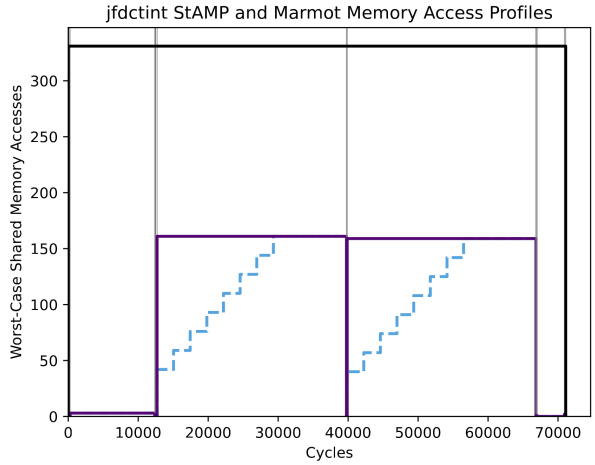

h264_dec StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
1e6

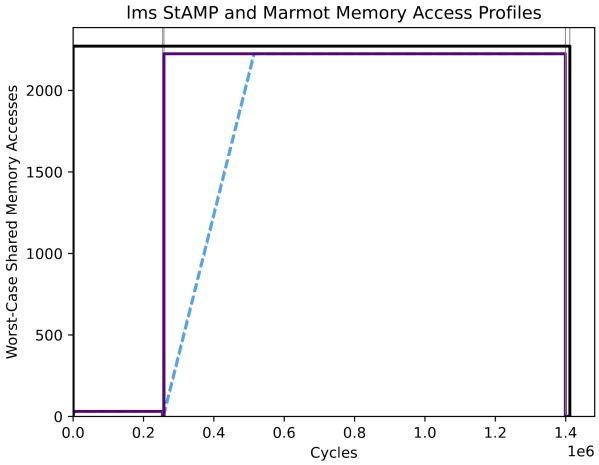

huff_dec StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
1e6

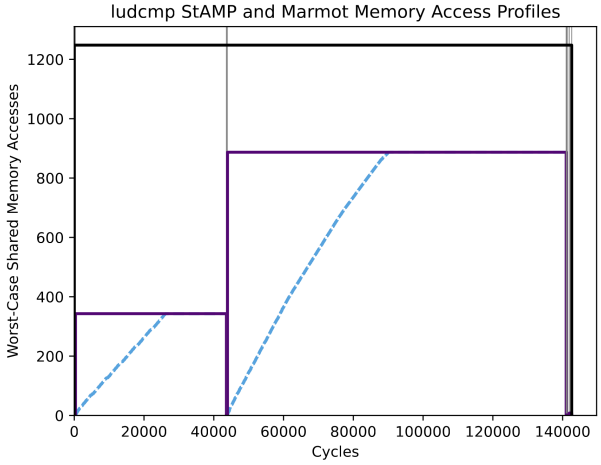

iir StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

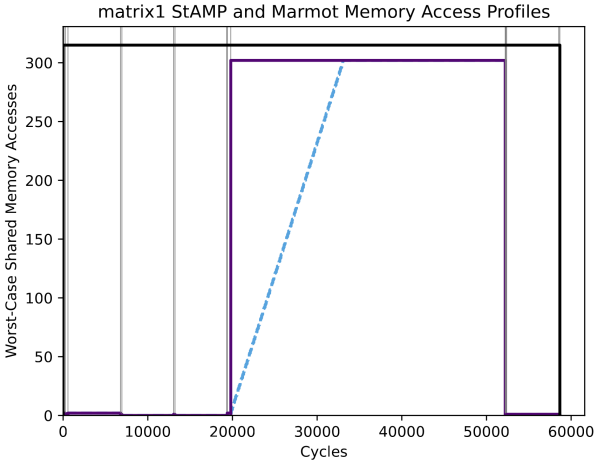

insertsort StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

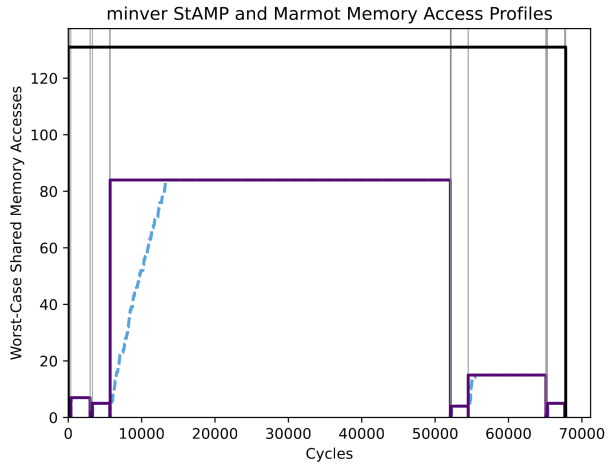

isqrt StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
1e7

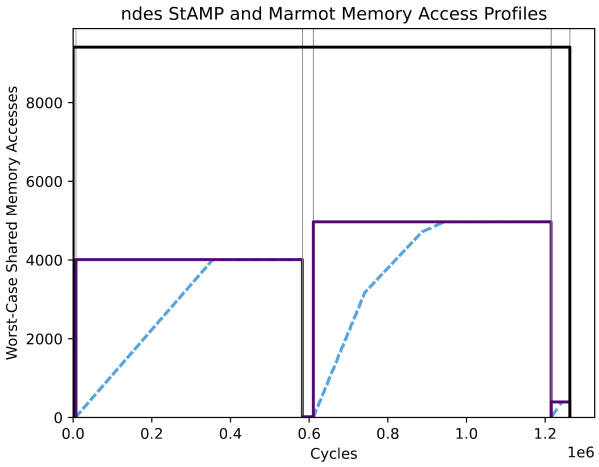

jfdctint StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

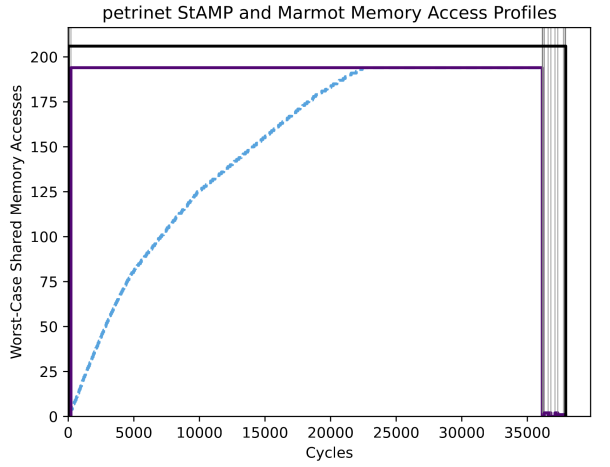

lms StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
1e6

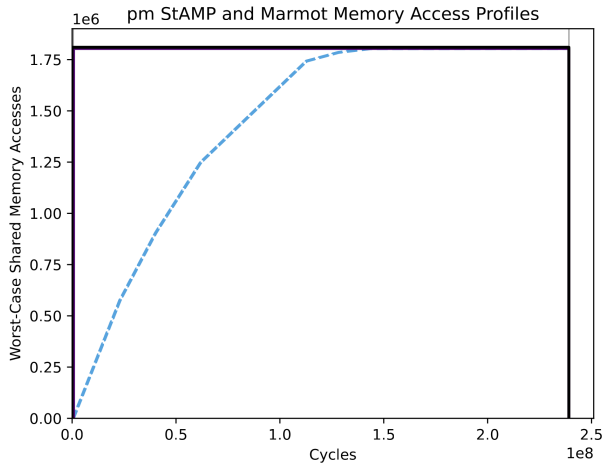

ludcmp StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

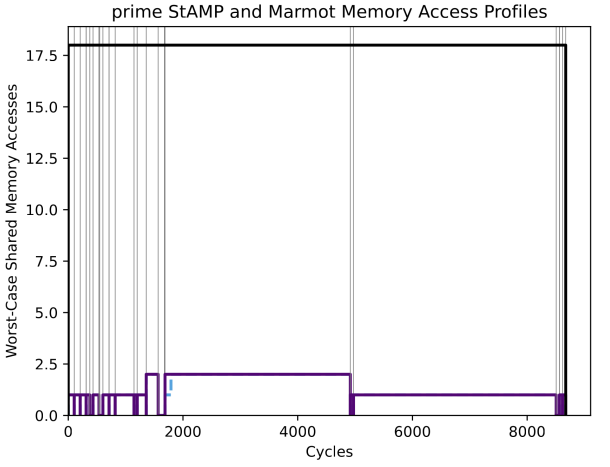

matrix1 StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

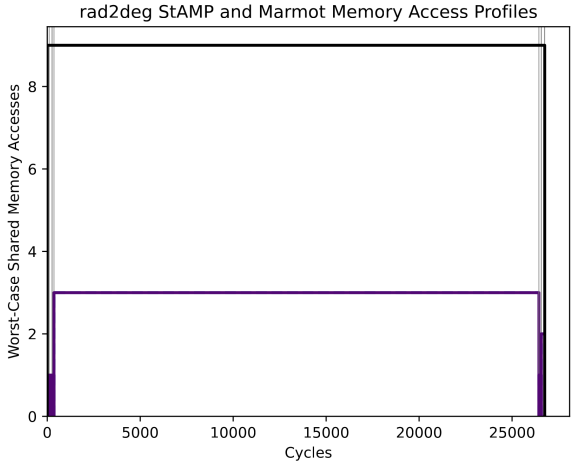

minver StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

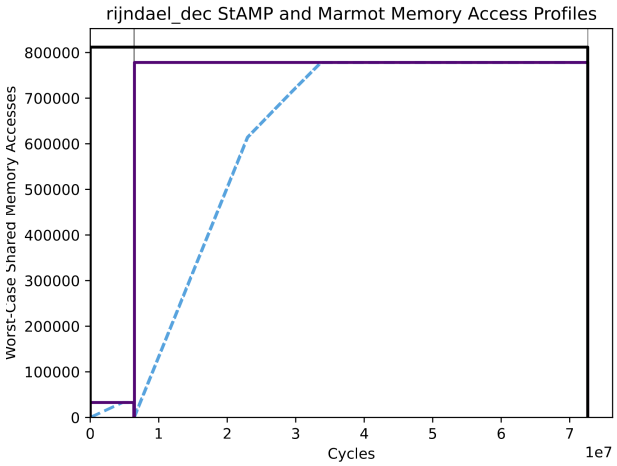

ndes StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
1e6

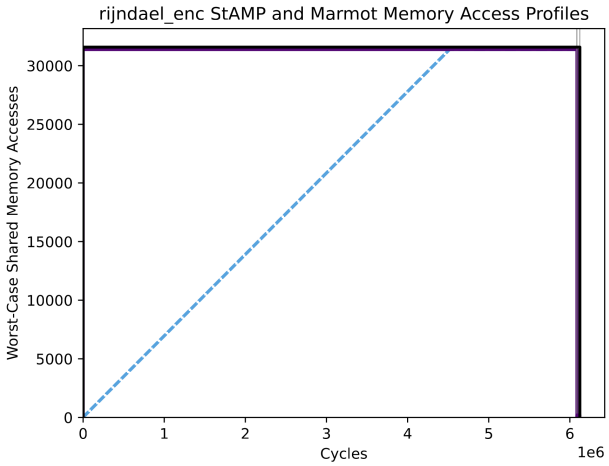

petrinet StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

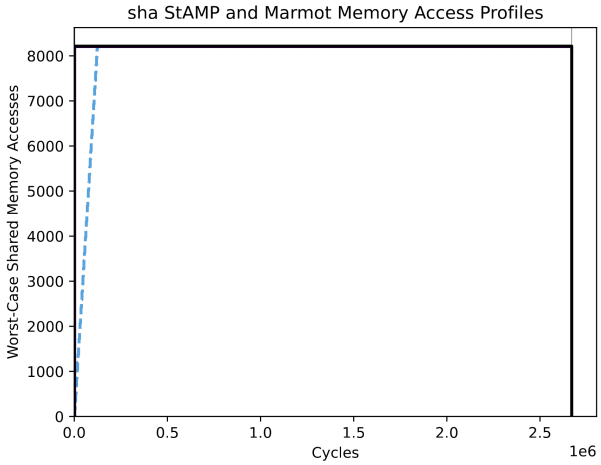

pm StAMP and Marmot Memory Access Profiles
1e6
Worst-Case Shared Memory Accesses
Cycles
1e8

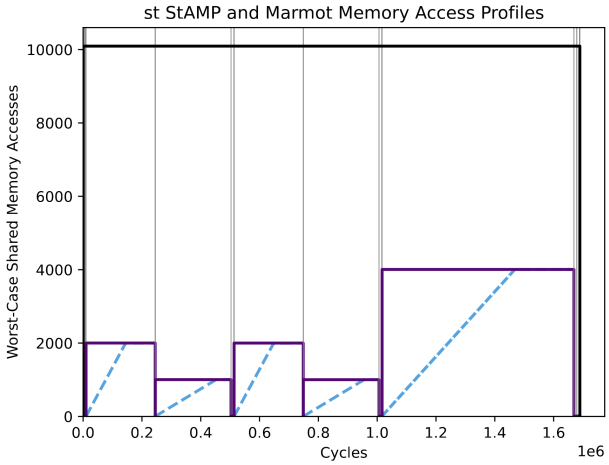

prime StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

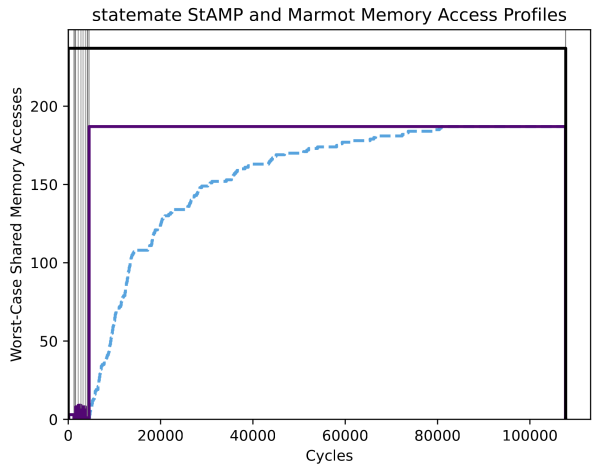

rad2deg StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

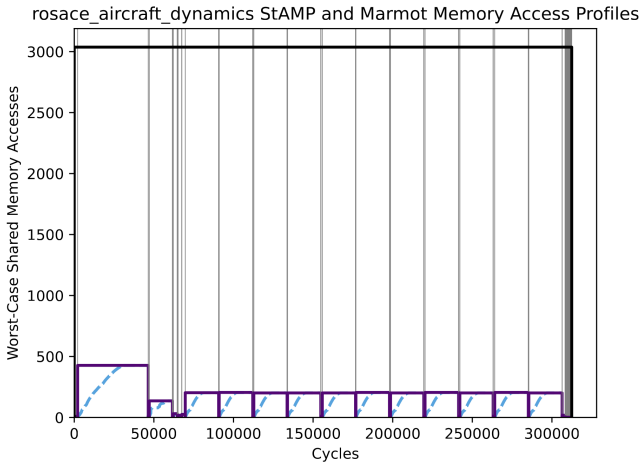

rijndael_dec StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
1e7

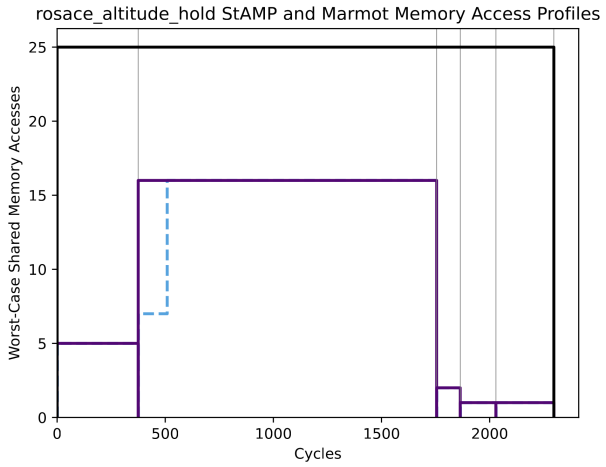

rijndael_enc StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
1e6

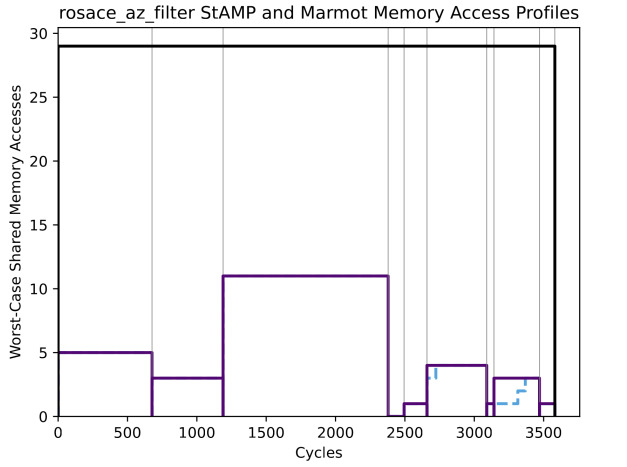

sha StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
1e6

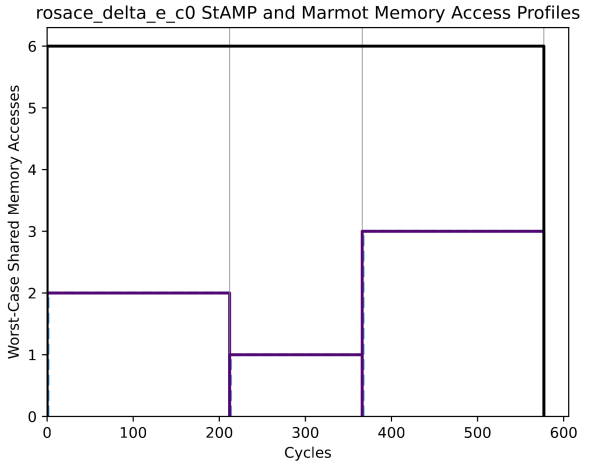

st StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
1e6

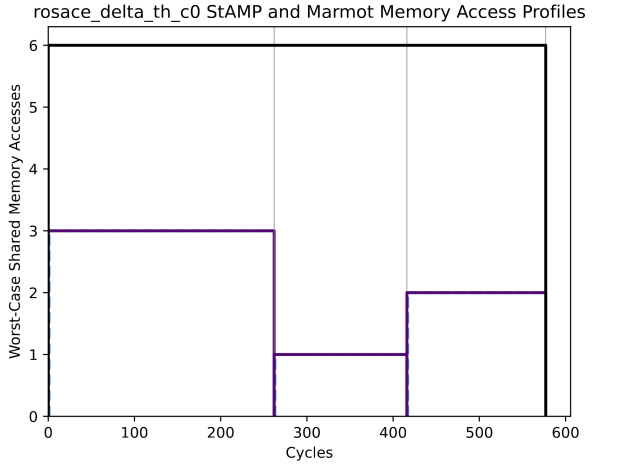

statemate StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

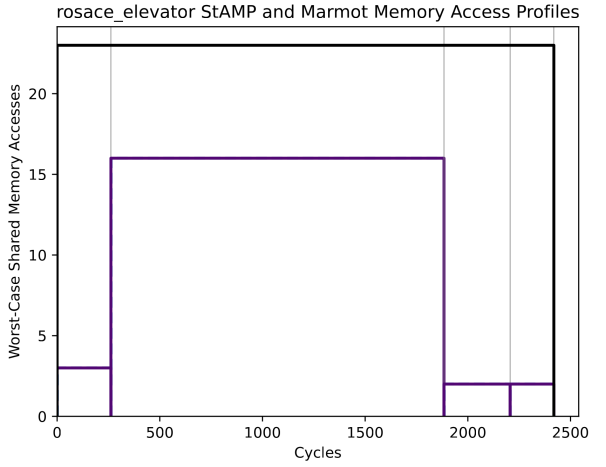

rosace_aircraft_dynamics StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
0
500
1000
1500
2000
2500
3000
0
50000
100000
150000
200000
250000
300000

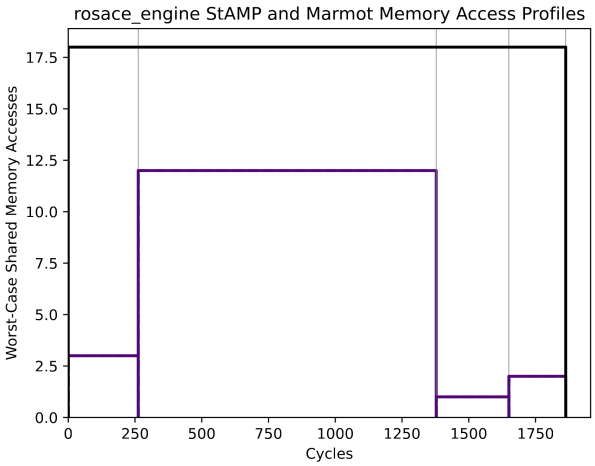

rosace_altitude_hold StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
0
5
10
15
20
25
0
500
1000
1500
2000

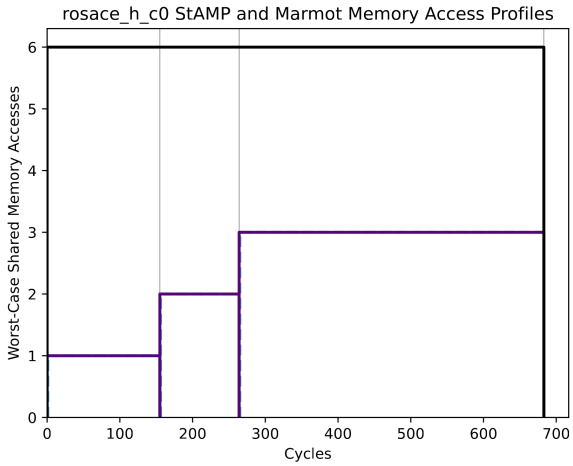

rosace_az_filter StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
0
5
10
15
20
25
30
0
500
1000
1500
2000
2500
3000
3500

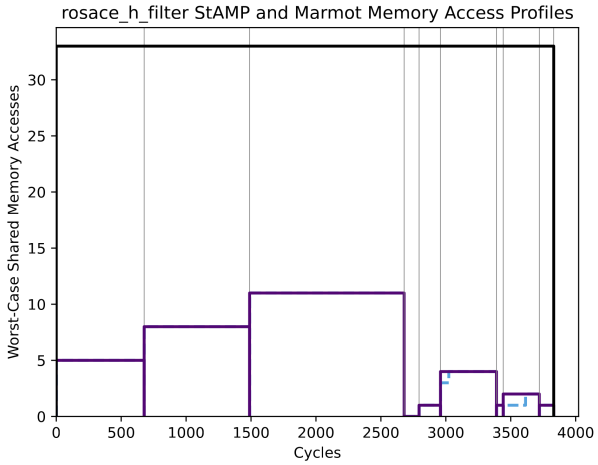

rosace_delta_e_c0 StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
0
1
2
3
4
5
6
0
100
200
300
400
500
600

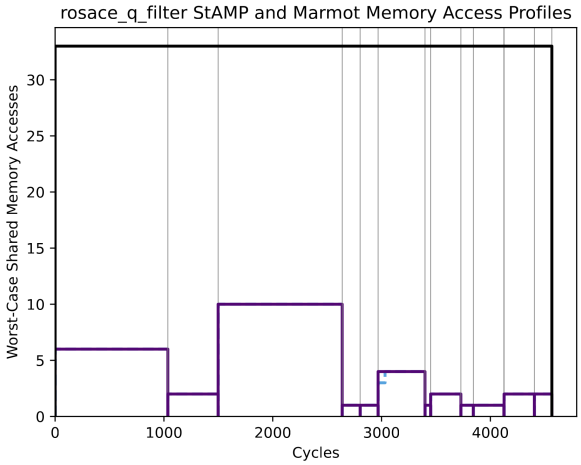

rosace_delta_th_c0 StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
0
1
2
3
4
5
6
0
100
200
300
400
500
600

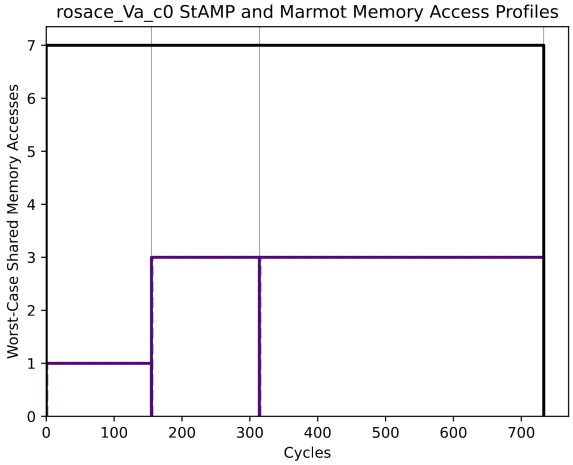

rosace_elevator StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
0
5
10
15
20
0
500
1000
1500
2000
2500

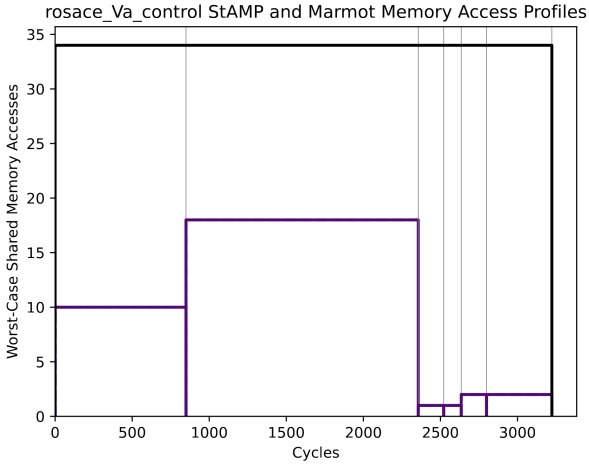

rosace_engine StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

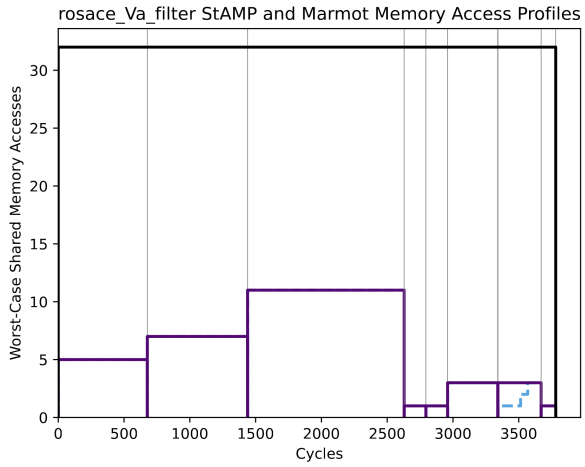

rosace_h_c0 StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

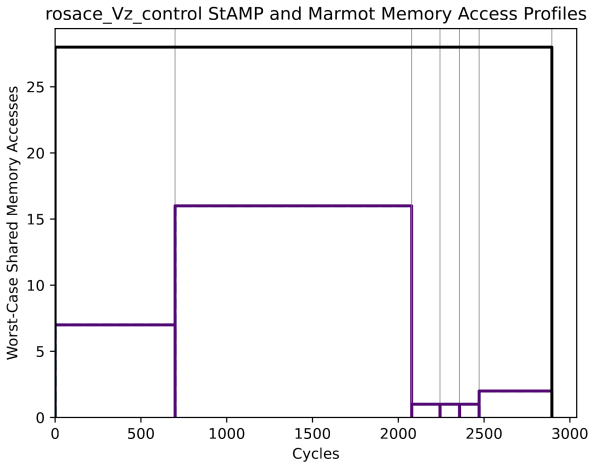

rosace_h_filter StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

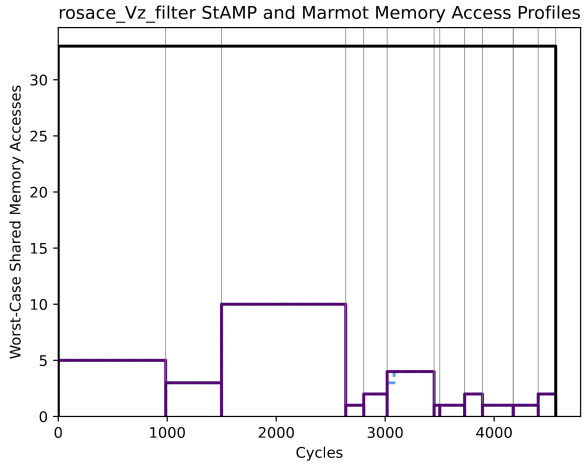

rosace_q_filter StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

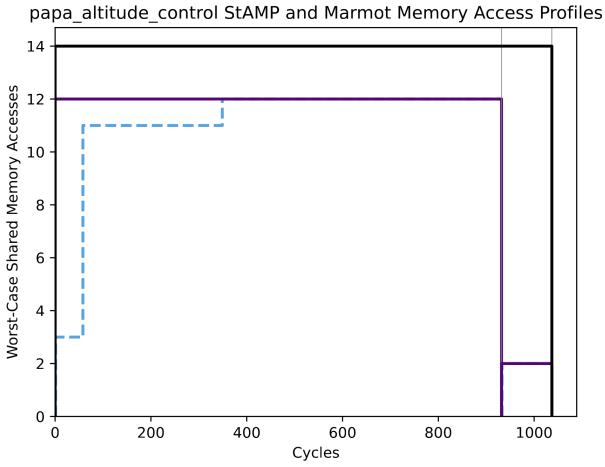

rosace_Va_c0 StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

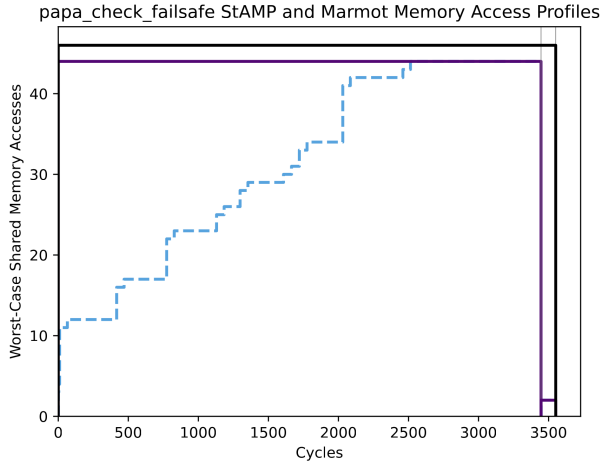

rosace_Va_control StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

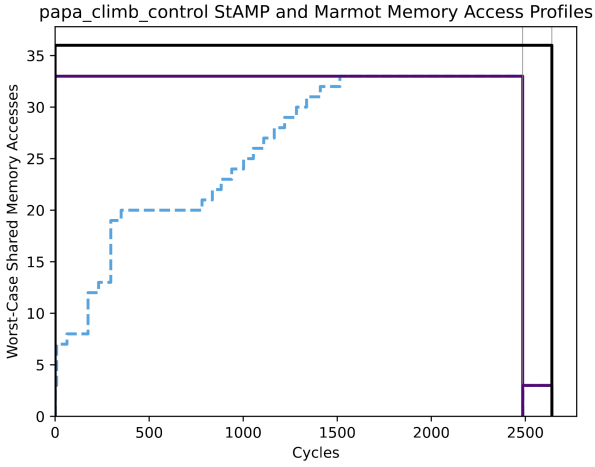

rosace_Va_filter StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
0
5
10
15
20
25
30
0
500
1000
1500
2000
2500
3000
3500

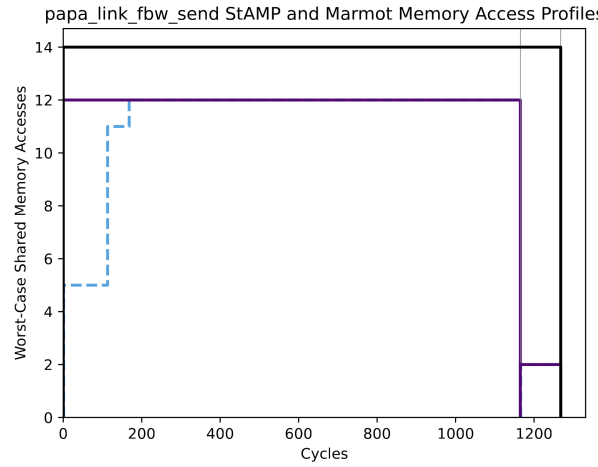

rosace_Vz_control StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
0
5
10
15
20
25
0
500
1000
1500
2000
2500
3000

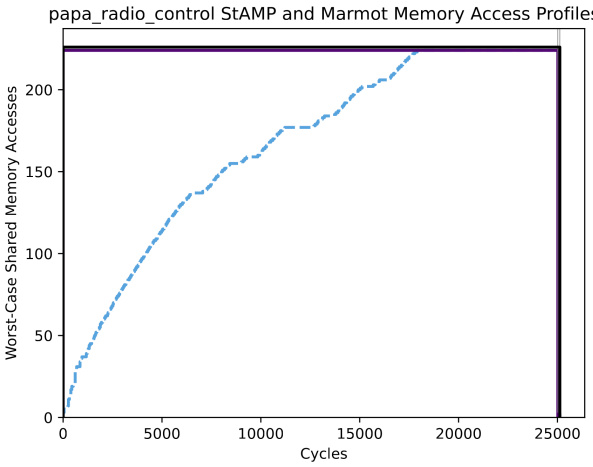

rosace_Vz_filter StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
0
5
10
15
20
25
30
0
1000
2000
3000
4000

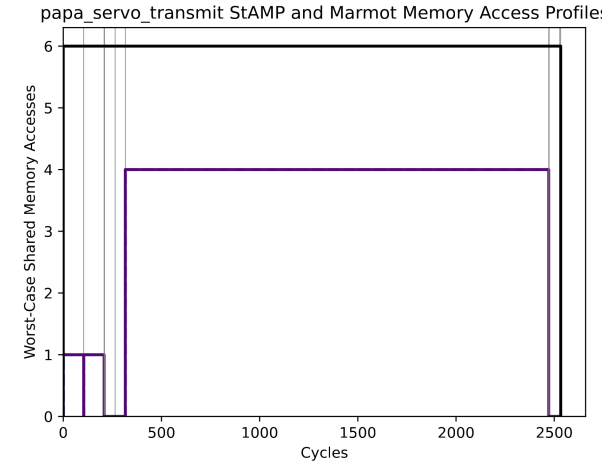

papa_altitude_control StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
0
2
4
6
8
10
12
14
0
200
400
600
800
1000

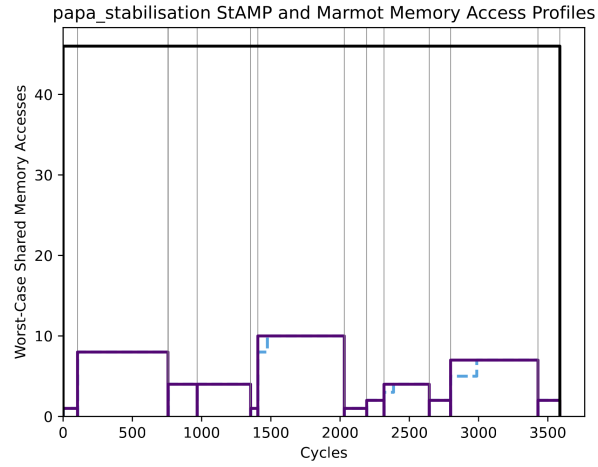

papa_check_failsafe StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
0
10
20
30
40
0
500
1000
1500
2000
2500
3000
3500

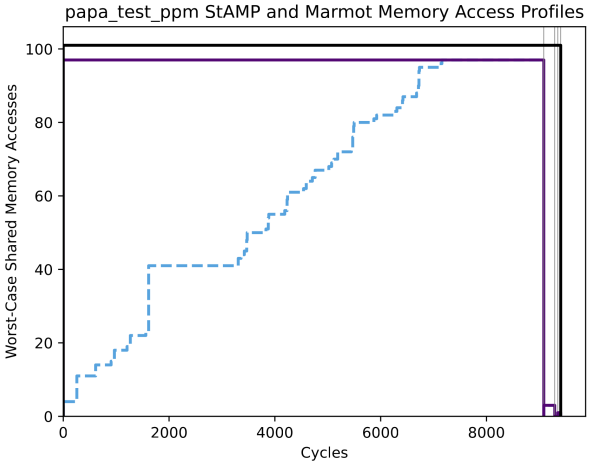

papa_climb_control StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
0
5
10
15
20
25
30
35
0
500
1000
1500
2000
2500

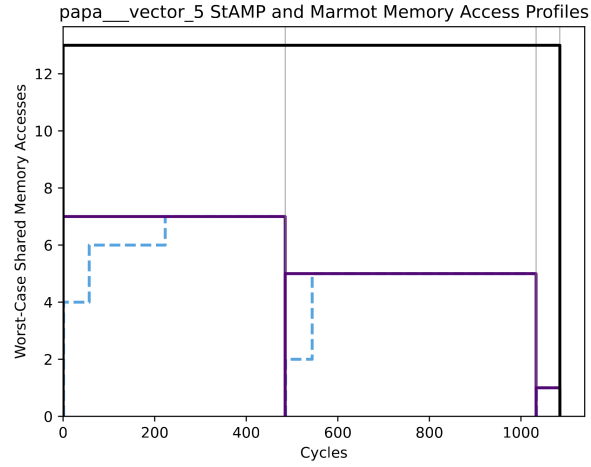

papa_link_fbw_send StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
0
2
4
6
8
10
12
14
0
200
400
600
800
1000
1200

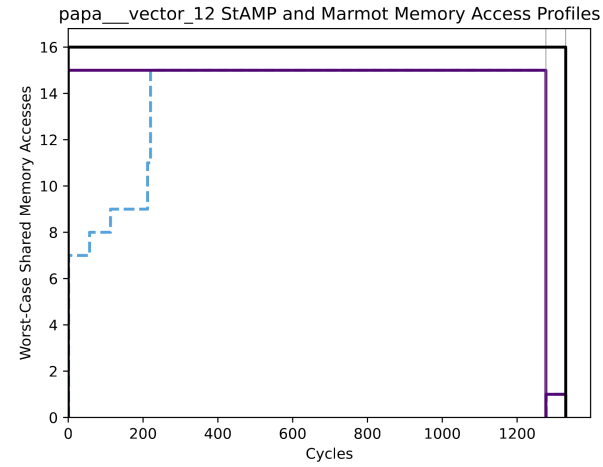

papa_radio_control StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
0
50
100
150
200
0
5000
10000
15000
20000
25000

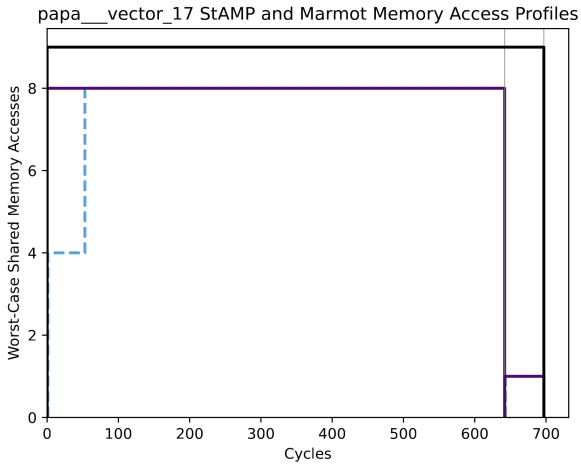

papa_servo_transmit StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
0
1
2
3
4
5
6
0
500
1000
1500
2000
2500

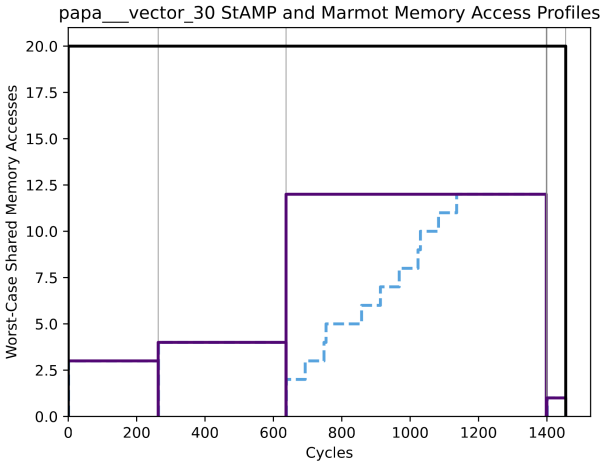

papa_stabilisation StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
0
10
20
30
40
0
500
1000
1500
2000
2500
3000
3500

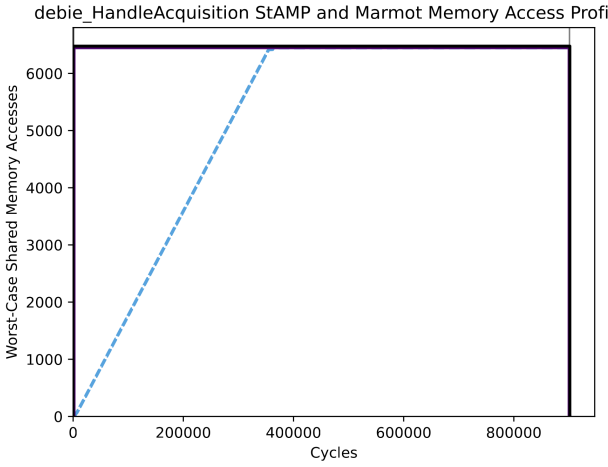

papa_test_ppm StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
0
20
40
60
80
100
0
2000
4000
6000
8000

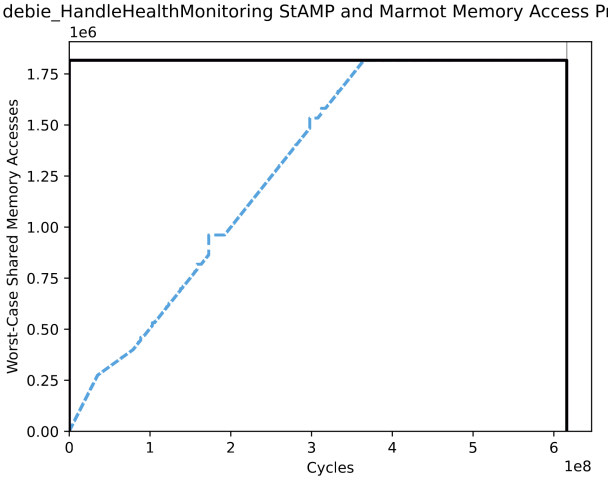

papa___vector_5 StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
0
2
4
6
8
10
12
0
200
400
600
800
1000

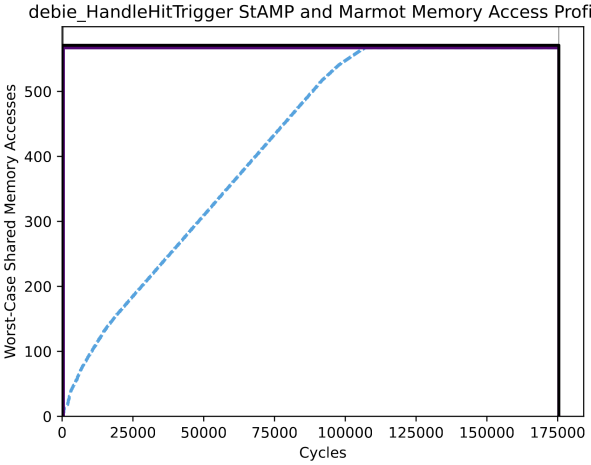

papa___vector_12 StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

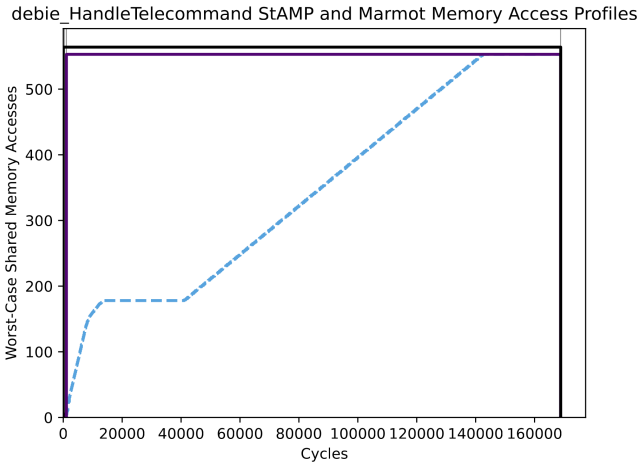

papa___vector_17 StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

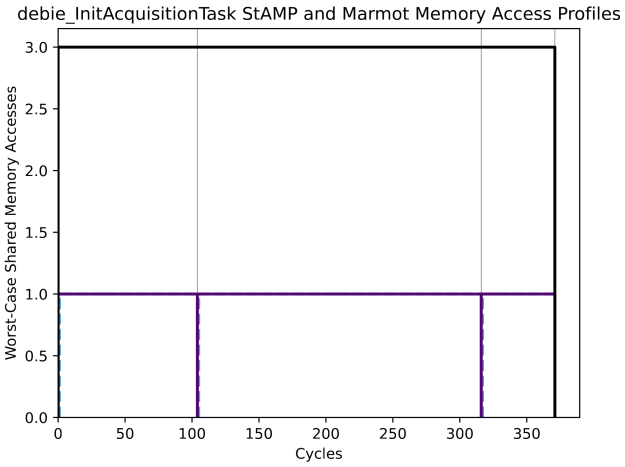

papa___vector_30 StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

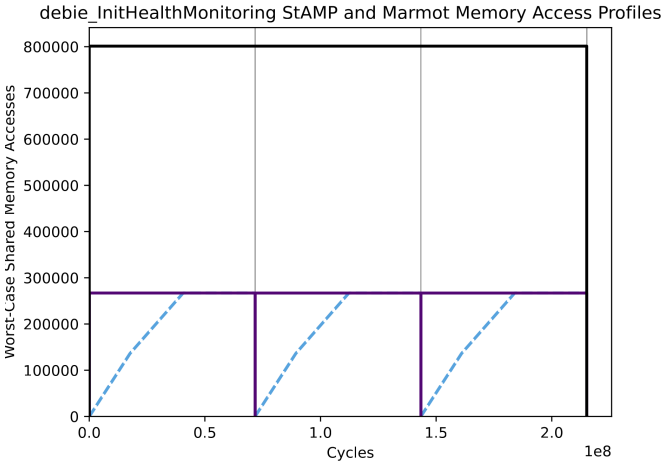

debie_HandleAcquisition StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles

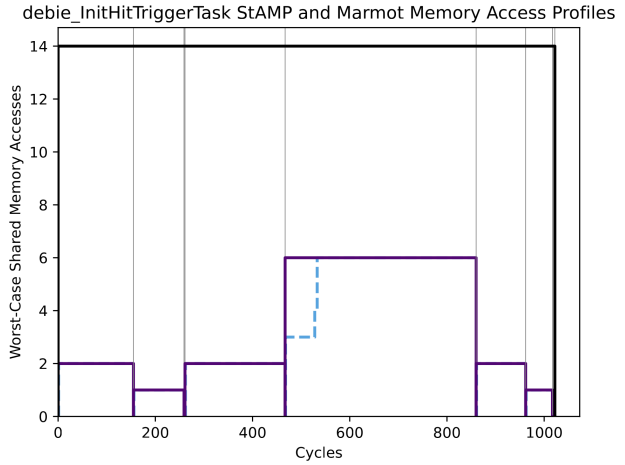

debie_HandleHealthMonitoring StAMP and Marmot Memory Access Profiles
1e6
Worst-Case Shared Memory Accesses
Cycles
1e8

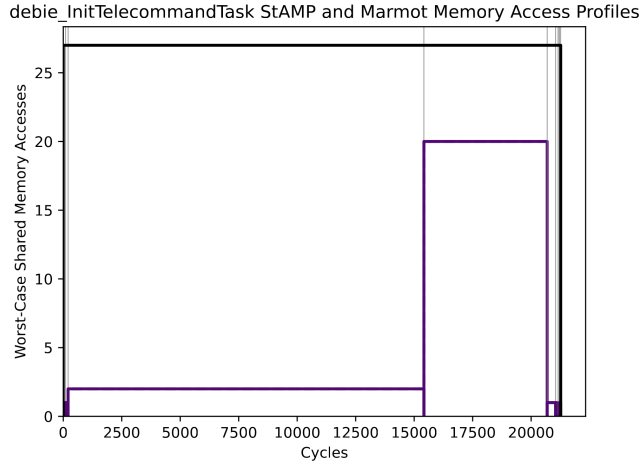

debie_HandleHitTrigger StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles


debie_HandleTelecommand StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
0
100
200
300
400
500
0
20000
40000
60000
80000
100000
120000
140000
160000


debie_InitAcquisitionTask StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0
50
100
150
200
250
300
350


debie_InitHealthMonitoring StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
1e8
0
100000
200000
300000
400000
500000
600000
700000
800000
0.0
0.5
1.0
1.5
2.0


debie_InitHitTriggerTask StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
0
2
4
6
8
10
12
14
0
200
400
600
800
1000


debie_InitTelecommandTask StAMP and Marmot Memory Access Profiles
Worst-Case Shared Memory Accesses
Cycles
0
5
10
15
20
25
0
2500
5000
7500
10000
12500
15000
17500
20000