Extraction of memory access profiles using Time Interest Points: a comparison with StAMP

TANGUY-LEGAC Erwan
ENS Rennes
France
erwan.tanguy-legac@ens-rennes.fr

***Abstract*—Multi-core architectures suffer from predictability issues due to possible interferences when tasks try to access shared resources simultaneously. To address this, static analysis is used to extract the memory access profile of a task, describing when memory accesses may happen. In this paper, we provide an experimental comparison between the Time Interest Point framework and the StAMP method, both of them being state-of-art techniques for extracting memory access profiles in multi-core architectures. We present the Time Interest Point framework and extend and implement it within the *Heptane* software, so that it supports the same features as StAMP, resulting in a fair comparison.**

***Index Terms*—Worst-Case Execution Time Estimation, Static Analysis, Multicore, Interference**

I. INTRODUCTION

Real-time systems require strong timing guarantees to ensure their correctness. Static scheduling can be used to verify such properties of our programs, but it requires some knowledge about the tasks, such as their Worst-Case Execution Time (WCET). In this context, multi-core architectures come with an important problem: the tasks cannot be analyzed in complete isolation, and we have to consider the possible execution of tasks on other cores which may interfere with the task targeted by our analysis. These interferences may occur when multiple tasks try to simultaneously access a resource shared between multiple cores (like a shared interconnect or a memory controller). Knowing when these interferences may happen gives more information for efficient scheduling.

In this paper we consider architectures with multiple cores, each of them containing a private L1 data and instruction cache, and connected to some shared memory (which could be a higher-level cache or a memory controller) through a shared bus. Using this model, we provide an experimental comparison between two state-of-the-art techniques which produce a memory access profile of the analyzed task, giving a precise description of when a task may try to access the processor's shared resources. This profile can then be used to perform static scheduling or worst-case response time analysis [1].

The StAMP technique [3] separates the code into single-entry single-exit (SESE) regions, and uses a subset of those regions, covering the analyzed code. Each region in this subset is then analyzed individually using the Implicit Path Enumeration Technique (IPET) [6], producing the memory access profile of a task in a reasonable time while mapping the time intervals to actual code. The Time Interest Points (TIP) framework [2] is the other method producing memory access profiles of a task, which we want to compare against StAMP. It relies on execution trace enumeration to have the finest level of detail during the analysis. For this work, the TIP framework was implemented in the *Heptane* software [5] (which contains StAMP's implementation) in an attempt to provide a fair comparison between both techniques. This required extending the TIP framework with features supported by StAMP, such as a more detailed cache analysis and support of function calls (which was not detailed in the original paper). As such, the focus of this work is the implementation and extension of the TIP framework.

The rest of this paper is organized as follows. Section II presents the TIP framework and our extensions. Section III covers the memory profile extraction using the TIP framework. Section IV gives an experimental comparison between the TIP framework and StAMP.

II. EXTRACTING TIPS FROM A CFG

A. The basics of the TIP framework

The basics of the TIP framework are defined in [2]. A Time Interest Point is an instruction which may result or suffer from interferences with other tasks. In our model, these are memory references which may result in a memory access.

A memory reference is found for every instruction (because the instruction has to be loaded from memory), and additional ones can be found in memory instructions (*i.e.* instructions which perform a *load* or *store* operation). To get this information, a static cache analysis is run on the program, labeling each memory reference with a Cache Hit-Miss Classification (CHMC) for the corresponding cache. A CHMC label for a memory reference is one of the following:

- *AH* (Always Hit): a memory access will never be performed by this reference
- *AM* (Always Miss): a memory access will always be performed by this reference
- *FM* (First Miss): the reference will always result in a cache hit after its first occurrence
- *NC* (Not Classified): all other cases

Because the WCET estimation must be pessimistic, we treat *NC* labels as *AM* ones. Using this CHMC, we can define TIPs as instructions containing a memory reference which is not labeled with *AH*.

The TIP framework is based on the isolation of TIPs and abstraction of any instruction which will never perform a memory access. To have a global overview of this framework, it can be separated in 3 steps.

First, the Control Flow Graph (CFG) is extracted from the compiled program. A CFG of a program is a graph whose vertices are sequences of consecutive instructions called Basic Blocks (BB) and edges are induced from the possible flow of control (*i.e.* execution path) between basic blocks. The extraction of the CFG of a task is a built-in component of *Heptane* and is not detailed here. In practice, *Heptane* produces one CFG per function, and a program consists of a set of CFGs linked together.

From the CFG, we extract the TIPs to produce a TIP Graph. This intermediate representation is then used to enumerate all possible execution traces of our program. Each of these traces gets converted into a time segment.

Finally, the time segments are merged together to produce the memory access profile of the program. All the mentioned intermediate representations are defined in [2] and later in this paper.

The following subsections present (in execution order) each step of the extraction process: subsection II-C covers the handling of function calls, subsection II-D details how loops are processed, and subsection II-E finally describes how TIPs are extracted from a CFG.

B. TIP Graph definition

A TIP Graph is a graph extracted from the control flow graph (CFG) of a task, where each node corresponds to a TIP, and each edge represents all possible execution paths between two TIPs (that don't encounter any other TIP). The edges are labeled with their corresponding WCET, and the nodes with the number of possible memory accesses of the corresponding TIP. TIP Graphs also contain nodes which do not represent any instructions. We call such a node an *empty TIP*, because it does not perform any memory access. There are four of those empty TIPs: CFG *start* and *end*, *call* nodes and loop *head*s.

Fig. 1 gives an example of a CFG and its corresponding TIP Graph. The dashed node corresponds to the loop's head duplication (which is detailed in subsection II-D). All other nodes correspond to a sequence of TIPs contained in a basic block of the CFG.

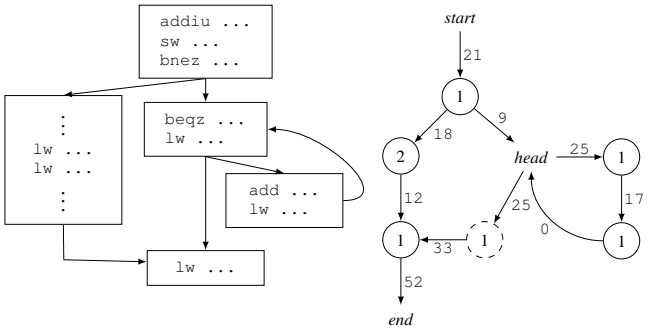

Figure 1. A CFG (left) and the corresponding TIP Graph (right)

C. Handling function calls

The cache analysis of a function may differ if we consider the context from where the function is called from. As an example, the data a function accesses may already be located in the cache if the function is called from within a loop (because of the previous iterations) whereas if the function is called from outside the loop, the cache may not be populated with the accessed data. Because of this, *Heptane* runs one cache analysis per call context for each CFG.

To make use of this contextual analysis, we create one TIP Graph per call context for each function and only take into consideration the cache analysis of the corresponding context. This is done using *virtual inlining* (we duplicate the CFGs and only keep the cache analysis labeling of the considered context).

Also, each call is represented by an empty TIP in the TIP Graph, which links to the callee TIP Graph. This TIP must be placed right after the last instruction of the basic block corresponding to the call to ensure that the edges are labeled with correct WCETs.

Fig. 2 illustrates this: the function `main` makes two different calls to the function `f`. We create two TIP Graphs for `f`, one per call context: the TIPs are extracted as explained in subsection II-E. For `f`, they are extracted by considering the corresponding context-aware cache analysis.

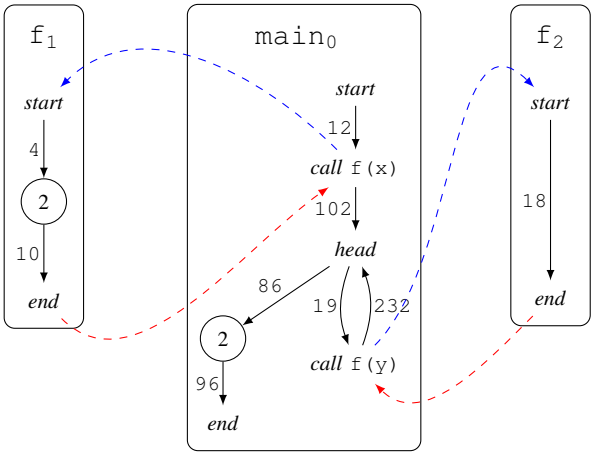

Figure 2. Representation of calls in the TIP Graph

D. Handling loops

1) Representing loops in the TIP Graph: To correctly handle loops, we have to create an empty TIP representing the head of the loop. It ensures correct labeling of the edges (the ones originally in the loop are exactly the ones in the loop after extraction). Because the loop's head gets executed one more time than the loop's body, we duplicate and add its duplicate right after the new *head*, outside of the loop.

2) Loop peeling: To provide a fair comparison with StAMP, support of First-Miss (*FM*) labeling by the cache analysis is required. These labels are encountered on instructions in a loop. They are mainly handled during the trace enumeration step presented in section III, but a partial support of them is presented in this subsection. This partial support is based on what is called *loop peeling*: the unrolling the first iteration of each loop in the CFG. Fig. 3 illustrates this process.

It should be noted that loop peeling produces an exponential number of new basic blocs in the CFG when encountering nested loops, which increases memory usage. In practice, this is still reasonable because loops are nested only up to a small and manageable depth.

Once the first iteration of a loop is unrolled, we want to check whether an instruction labeled as *FM* can be relabeled to *AH*. This is possible if all execution paths to the given instruction contain its duplicate in the unrolled

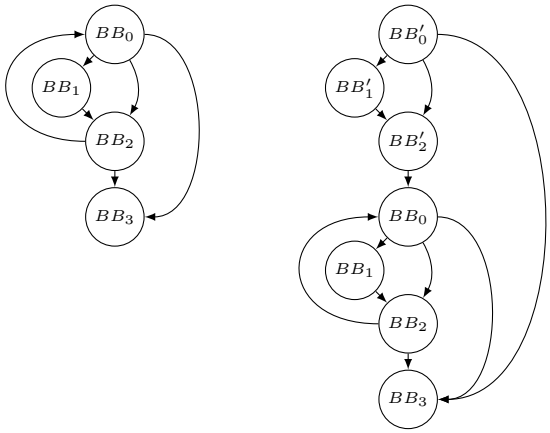

Figure 3. Example of loop peeling: original loop on the left, peeled loop on the right

iteration, which is checked by performing a dominator analysis after the peeling of the loops. In the example of Fig. 3, in the peeled loop, BB_0 and BB_2 are dominated by BB'_0 and BB'_2. If they contain an instruction with a *FM* label, it is relabeled with *AH* and the duplicated instruction in the dominating BB gets relabeled with *AM*. This reduces the number of TIPs in the loops once the TIP Graph is extracted. However, loop peeling is optional and disabling it still produces a valid TIP Graph, but a less precise one.

3) Addressing the edge cases of loop peeling: The reason loop peeling is optional is because it limits over-approximations but is not enough on its own. StAMP uses IPET and counts the number of executions of each BB along the *worst-case-execution-path* and uses the WCET calculation of the first execution of this block and the next ones accordingly. Loop peeling does not cover all cases and results in over-estimations. The example of Fig. 3 illustrates this: BB_1 is not dominated by BB'_1, so we can not relabel any of its instructions labeled *FM* using our dominator analysis.

To overcome this, we will have to keep a set of visited TIPs during the trace enumeration presented in section III. This makes the loop peeling step optional, as it does not improve the accuracy of the produced memory access profile. However, it can speed up the process by reducing the number of visited TIPs.

E. Extraction of TIPs

Finally, we can extract the TIPs of each CFG and build the corresponding TIP Graphs (remember that we create one TIP Graph per context for each CFG). This is done by iterating over the set of instructions of each basic block in the CFG, and extracting the TIPs from

them. Instructions which are not TIPs get abstracted and are only represented by the WCET of edges which correspond to an execution flow going through the instruction. Whenever we encounter two or more consecutive TIPs, we merge them into a single one to reduce the empirical complexity of the TIP Graph, and label the resulting TIP with the sum of possible memory accesses of each TIP.

III. Extracting memory access profiles

From the TIP Graphs of a task, we extract its memory access profile. A memory access profile is defined in [3] as a sequence of pairs $(wcet, wcma)$ representing a time-sequence of code intervals of maximum duration $wcet$ in which at most $wcma$ memory access can happen. The extraction of the memory access profile presented in [2] is done in three steps, which are detailed in the following subsections: first, all possible execution traces are enumerated using the TIP Graphs. Then, using this intermediate representation, we extract a memory access profile for each trace which is named Time Segment. Lastly, all the Time Segments are merged together to obtain the task's memory access profile. A parameter named $delta$ controls the allowed size of time intervals which do not contain any memory accesses: if such a time interval with a duration less than $delta$ exists, it gets merged with its surroundings time intervals. All these steps are detailed in the following subsections.

A. Trace enumeration

The execution traces are enumerated by using the algorithm presented in [2], which we extended to support *FM* (First Miss) labeling by the cache analysis as explained in II-D3, and function calls as described in II-C. To ensure termination of the algorithm, the number of iterations for each loop must be bounded. In the *Heptane* software, this is done manually.

The algorithm works by traversing the TIP Graph. When a new TIP is encountered, we add a Trace Node to the current trace. A Trace Node consists of a reference to the newly encountered TIP and the WCET at which this TIP has been encountered. A Trace is a sequence of such Trace Nodes. When the TIP end is encountered, we know that the current trace is finished, and we can proceed to with next one.

When a call node is reached, the traces of the callee TIP Graph are recursively enumerated (or retrieved from a previously completed enumeration) and inserted into the current trace.

When a node with an instruction labeled by *FM* is encountered, the algorithm checks if the same instruction was already visited in the current trace. If it has, the instruction is treated as being labeled with *AH* (Always Hit) by the cache analysis.

Fig. 4 gives an example of such enumerated timed execution traces. Note that for readability, the TIPs are not represented in the example, only the corresponding number of memory accesses are. The memory latency is set to 99, and the WCET for an instruction is set to 1.

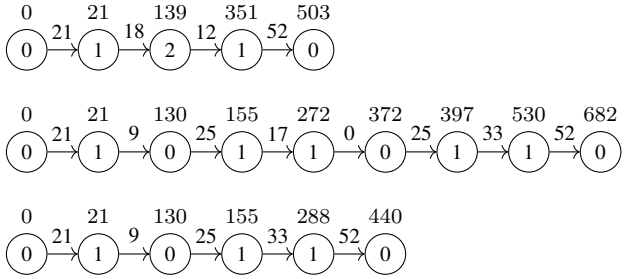

Figure 4. 3 enumerated traces from the TIP Graph in Fig. 1

B. Time segment conversion

This step converts each trace into a Time Segment. A Time Segment is an intermediate representation which abstracts the TIPs themselves. It is made of a sequence of Segments. A Segment is a triplet $(start, end, accesses)$ of integers where $start$ and end are the start and end times of the Time Segment, and $accesses$ is the number of memory accesses occurring during the time interval in the trace it was converted from.

The conversion from a Trace to a Time Segment is straightforward: for each Trace Node, we create a Segment accounting for the corresponding TIP of the Trace Node (with the WCET at which the TIP has been encountered as $start$, end set to account for the WCET of the TIP itself, and $accesses$ set to the number of memory accesses made by the TIP); then we create another Segment representing the edge between the current Trace Node and the next one ($start$ set to the previously created Time Segment's end, end set to the next Trace Node's WCET and $accesses$ set to 0). Fig. 5 shows the Time Segments converted from the traces in Fig. 4.

C. Time segment fusion

The last step is the to fuse all the Time Segments together. To do this, we first merge them one by one into one Time Segment. To merge two Time Segments together, we run through every pair of Segments of the two Time Segments and intersect the corresponding time intervals while merging the memory accesses. To limit over-estimations, we use a map to compute the new

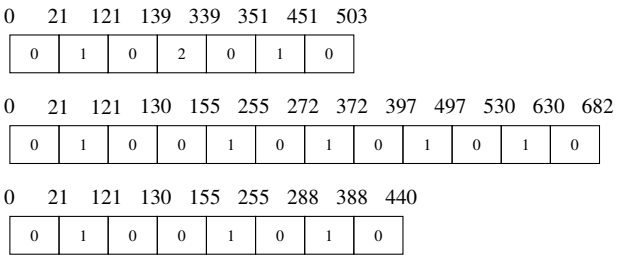

Figure 5. Time Segments of the traces in Fig. 4

number of memory accesses in the intersected Segment: each access is assigned to its original Time Segment.

To produce the memory access profile, the Segments of the new Time Segment must be fused together. This is done with two passes over the Time Segment: the first one fuses the consecutive Segments which do not perform any memory access and fuses the consecutive Segments which do perform at least one memory access. The second pass fuses every Segment which does not perform any memory access and has a duration smaller than $delta$ with its neighbours.

The resulting Time Segment is the memory access profile, if we consider the maximum number assigned to each time interval.

Fig. 6 show the result of merging and fusing the first and last Time Segment of Fig. 5. Each resulting Segment has two number in it: the one above is the number of memory accesses performed by the first Time Segment in this interval, and the one below is the number of memory accesses performed by the third Time Segment in this interval. The Time Segment at the bottom of the figure represents the memory access profile.

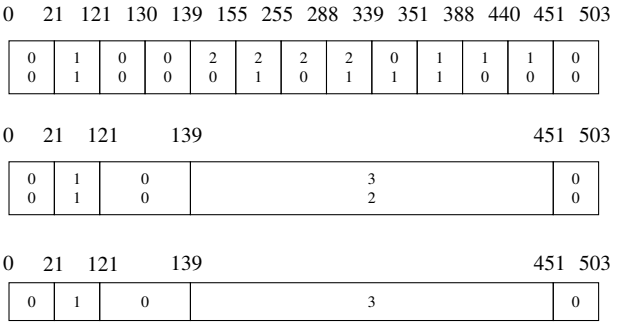

Figure 6. Intersection (above), fusion (middle) and extracted profile (below) of Time Segments 1 and 3 of Fig. 5

IV. Comparison of TIP and StAMP

To compare both methods, we used the Mälardalen benchmarks [4] compiled to the MIPS target architecture. The loop bounds are manually set.

The SESE regions for StAMP are extracted using unlimited $fuel$ and an edge-centric definition. Ideally, the node-centric definition should be used, because it has been proved in [3] that this results in more distinct intervals. We had to use the edge-centric definition because on some benchmarks, StAMP's implementation results in different total WCET calculations, and the ones obtained using the edge-centric definition are smaller.

For the TIPs framework, two profiles are extracted for each benchmark: one with the parameter $delta$ set to 0, and one with the parameter $delta$ set to the maximum duration of the time intervals not containing any memory access.

Fig. 7 shows three memory access profiles side-by-side extracted using both methods.

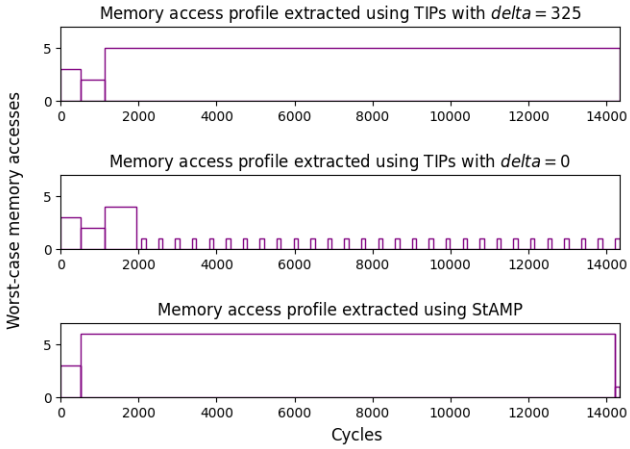

Figure 7. Memory profiles of benchmark `fibcall`

As it can be seen, the TIP framework produces more detailed memory access profiles than StAMP: the provided example shows that the memory accesses are divided in 38 separate intervals by the TIP framework but only 3 of those intervals appear when using StAMP. However, the TIP framework over-estimates the total number of memory accesses: in the example depicted by Fig. 7, StAMP counts at most 10 memory accesses whereas the TIP framework counts at most 38 memory accesses. This is because it considers all memory accesses of all execution traces, even if they are mutually exclusive.

To fix this over-estimation, the parameter $delta$ can be modified. Intervals with a duration less than $delta$ in which no memory accesses occur are fused with their neighbours. This reduces the number of intervals and limits the addition of memory accesses produced by

Table I
COMPARISON OF EXTRACTED PROFILES BY STAMP AND THE TIPS FRAMEWORK

	StAMP		Time Interest Points ($delta = 0$)		Time Interest Points ($delta = w$[1])	
Benchmark	Intervals	Accesses	Intervals	Accesses	Intervals	Accesses
bs	3	18	4	20	3	18
expint	2	26	2	26	2	26
fibcall	3	10	32	38	3	10
insertsort	2	21	3	21	3	21
jfdctint	6	114	80	270	2	114
lcdnum	2	20	2	20	2	20
select	4	39	3	39	3	39
simple	1	5	1	5	1	5
sqrt	3	34	4	38	2	34

[1] w is the size of the largest time interval not containing any memory access

mutually exclusive execution traces. Table I presents an overall comparison of the produced memory accesses profiles. The profiles can be found in Appendix A of this paper.

The last difference is not shown by the extracted profiles and is their usability. What we compute is the WCET of each memory access, nothing guarantees that they cannot happen before this date. By dividing the program in SESE regions, StAMP makes it possible to link the start of an interval to a specific instruction. This makes it possible to easily enforce the WCET of the given interval, ensuring that the memory accesses happen when we expect and not before. On the opposite, this seems much more complicated with intervals extracted by the TIP framework. It may be possible, but to this date, we did not find any practical solution to this problem.

V. CONCLUSION AND FUTURE WORKS

In this paper, we extended the TIP framework with support for function calls and contextual cache analysis, as well as First-Miss labels support. We presented the memory access profile extraction process, and finally, we compared the TIP framework with the state-of-the-art StAMP method.

We left for future works the automatic selection of parameter $delta$ to produce the most precise memory access profiles, or the creation of a different fusion algorithm.

As discussed in section IV, practical use of the produced memory access profile and the enforcing of the WCET to keep the profiles correct is also left for future works.

Another improvement could be the use of Best-Case Execution Times to increase the knowledge about the memory accesses and refine the memory profiles.

REFERENCES

[1] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. 3rd. Springer Publishing Company, Incorporated, 2011. ISBN: 1461406757.

[2] Thomas Carle and Hugues Cassé. "Static extraction of memory access profiles for multi-core interference analysis of real-time tasks". In: *Architecture of Computing Systems: 34th International Conference, ARCS 2021, Virtual Event, June 7–8, 2021, Proceedings 34*. Springer. 2021, pp. 19–34. DOI: 10.48550/arXiv.2103.17082.

[3] Théo Degioanni and Isabelle Puaut. "StAMP: Static Analysis of Memory access Profiles for real-time tasks". In: *WCET 2022-20th International Workshop on Worst-Case Execution Time Analysis*. 2022. DOI: 10.4230/OASIcs.WCET.2022.1.

[4] Jan Gustafsson et al. "The Mälardalen WCET benchmarks: Past, present and future". In: *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2010. DOI: 10.4230/OASIcs.WCET.2010.136.

[5] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. "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 für Informatik. 2017. DOI: 10.4230/OASIcs.WCET.2017.8.

[6] Yau-Tsun Steven Li and Sharad Malik. "Performance analysis of embedded software using implicit path enumeration". In: vol. 30. 11. New York, NY, USA: Association for Computing Machinery, 1995. DOI: 10.1145/216633.216666.

APPENDIX A: EXTRACTED MEMORY ACCESS PROFILES

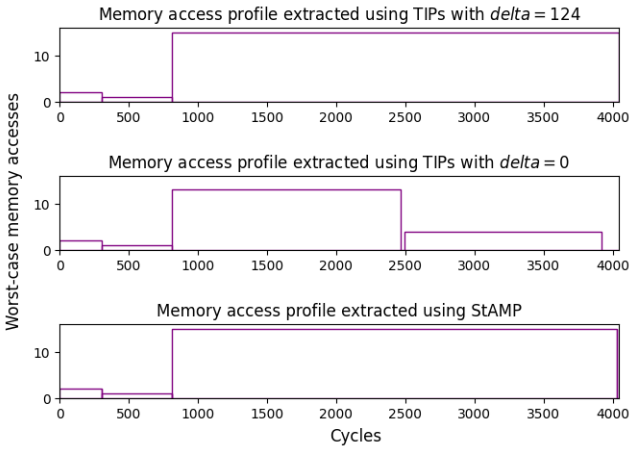

Figure A.1. Memory profiles of benchmark `bs`

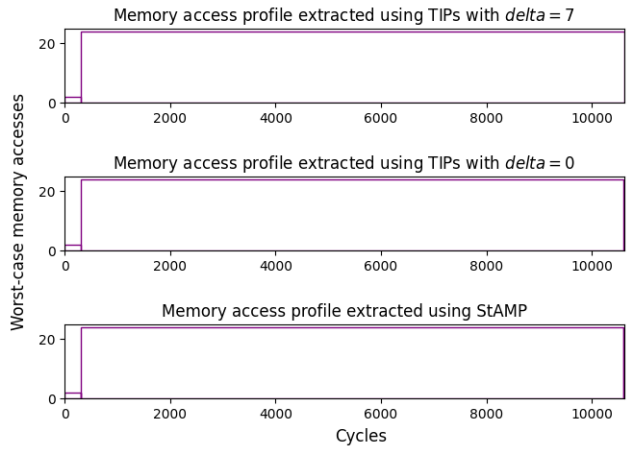

Figure A.2. Memory profiles of benchmark `expint`

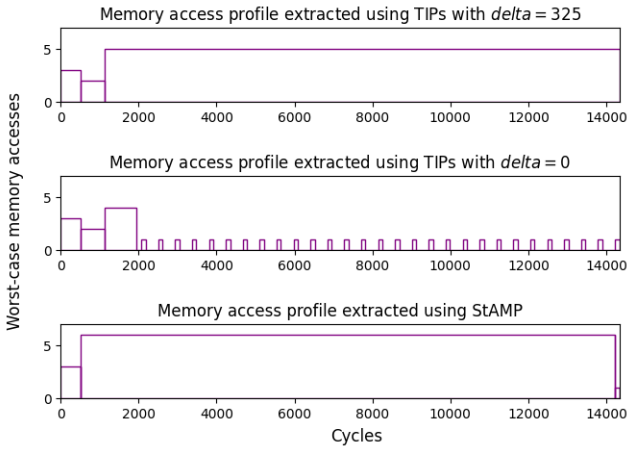

Figure A.3. Memory profiles of benchmark `fibcall`

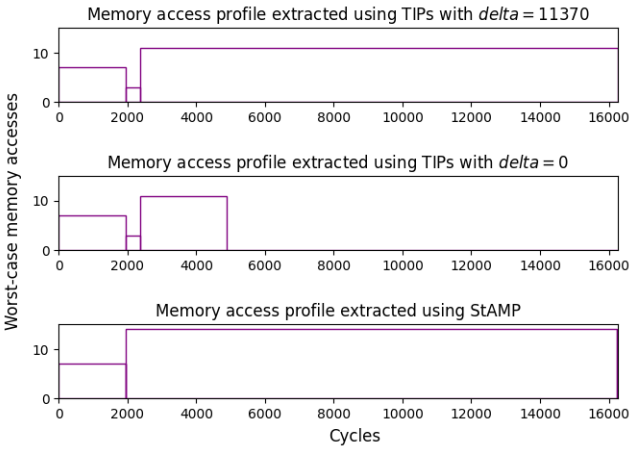

Figure A.4. Memory profiles of benchmark `insertsort`

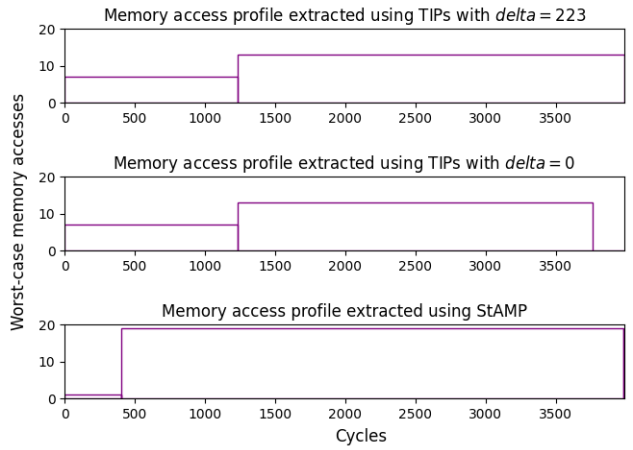

Figure A.5. Memory profiles of benchmark `lcdnum`

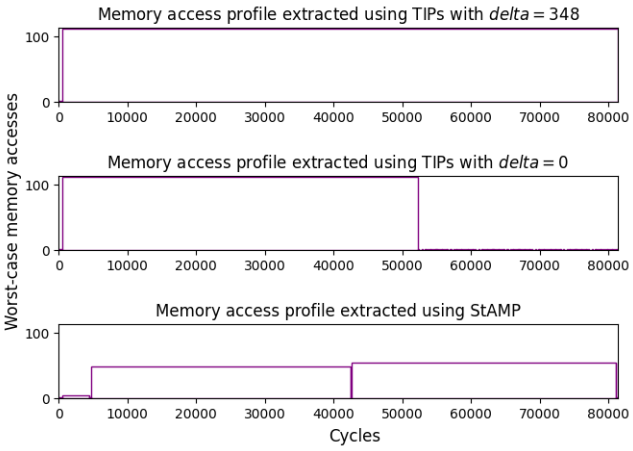

Figure A.6. Memory profiles of benchmark `jfdctint`

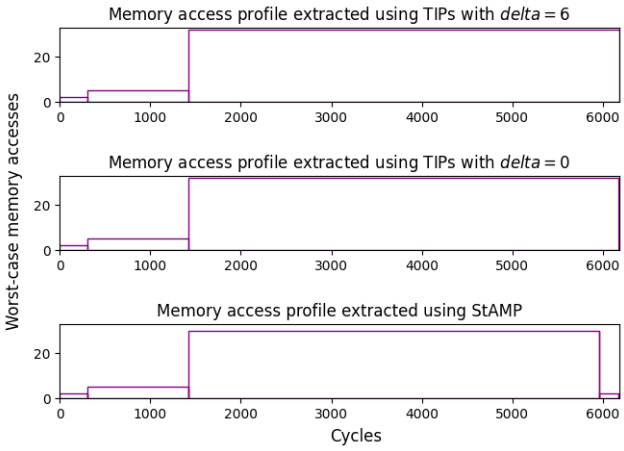

Figure A.7. Memory profiles of benchmark `select`

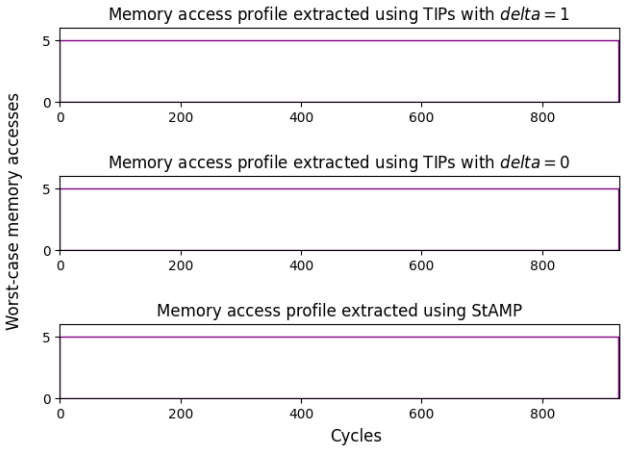

Figure A.8. Memory profiles of benchmark `simple`

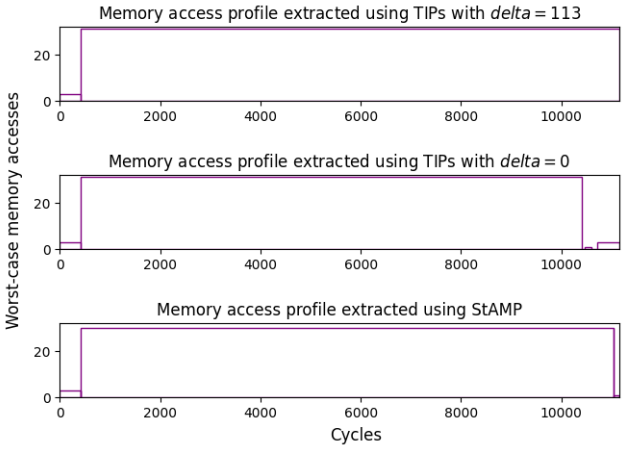

Figure A.9. Memory profiles of benchmark `sqrt`