StAMP: Static Analysis of Memory Access Profiles for Real-Time Tasks

Théo Degioanni ✉
École Normale Supérieure de Rennes, France

Isabelle Puaut ✉ 🏠 ⓘ
Univ Rennes, Inria, CNRS, IRISA, France

───── **Abstract** ─────

Accesses to shared resources in multi-core systems raise predictability issues. The delay in accessing a resource for a task executing on a core depends on concurrent resource sharing from tasks executing on the other cores. In this paper, we present StAMP, a compiler technique that splits the code of tasks into a sequence of code intervals intervals, each with a distinct worst-case memory access profile. The intervals identified by StAMP can serve as inputs to scheduling techniques for a tight calculation of worst-case delays of memory accesses. The provided information can also ease the design of mechanisms that avoid and/or control interference between tasks at run-time. An important feature of StAMP compared to related work lies in its ability to link back time intervals to unique locations in the code of tasks, allowing easy implementation of elaborate run-time decisions related to interference management.

1 Introduction

In order to guarantee timing constraints of real-time software, an upper bound of the Worst-Case Execution Time (WCET) of its sequential tasks is needed [17]. Static WCET estimation techniques provide such upper bounds (WCET *estimates*) and are well understood for single-core processors. However, multi-core architectures are now commonplace as they offer unprecedented processing power and low power consumption. Applying static WCET analysis to multi-core systems is more difficult than single-core ones since a task running on a core may suffer from interference delays caused by resource sharing with software executing on the other cores. Shared resources may be the Last-Level Cache (LLC), the memory bus or the memory controller.

Different approaches may be used to deal with interferences (see [11] for a survey). One class of techniques is to avoid interferences, by using, for instance, specific task models like PREM (PRedictable Execution Model, [13]), which separates the code of each task in a memory phase and an execution phase that does not perform any memory access. Specific scheduling techniques can then be designed to avoid the co-scheduling of phases that interfere with each other [14,18]. Another class of approaches allows interferences to occur at run-time, and leverages knowledge of the usage of shared resources to compute the resulting worst-case

interference delays [7,9]. This latter class of approaches relies on knowledge of shared resource usage of tasks executing concurrently, but surprisingly few methods provide such information at a granularity smaller than the entire task.

In this paper, we propose StAMP, a static analysis technique that divides the code of a compiled program into a *linear sequence of non-overlapping code intervals*, each with a distinct worst-case memory access profile. State-of-the-art methods for the static analysis of memory accesses [2, 12] all rely on *time-based* approaches: they divide the execution timeline into *time segments* for which they compute a worst-case number of memory accesses. In these approaches, there is no direct way to link segments back to concrete code sections. Therefore, run-time decisions to avoid or control interference have to rely on time only and are agnostic to the code location where the decision is taken. As compared with these time-based techniques, StAMP can link back its output time segments to *concrete code intervals*. This allows run-time decisions to be based not only on time but also on the code interval under execution. Such run-time decisions may, for instance, re-calculate interference based on the current progress of tasks or introduce synchronizations to avoid or minimize interference between intervals [15, 16].

Dividing the binary code of tasks into a linear sequence of code intervals is based in StAMP on a compiler technique that operates at the binary level. StAMP first divides the binary code into a tree of "well-formed" regions, called Single Entry point Single Exit point (SESE) regions [8]. The advantage of using such regions is that the entry and the exit of each region are natural frontiers for intervals, and as such natural points for introducing interference-related scheduling decisions. StAMP generates different sizes of intervals depending on the depth at which the SESE region tree is explored (deeper exploration induces finer-grain intervals). It is then possible to apply worst-case memory access analysis to each interval individually.

Traditional extraction of SESE regions is edge-centric [8], creating sections with a single entry edge and a single exit edge. We show in this paper that when using node-centric SESE regions (regions with a single entry node and a single exit node), the number of regions is more important than when using edge-centric regions. This provides fine-grain regions to scheduling strategies, in particular on code with many branches, which we believe will improve the quality of scheduling strategies.

The contributions of this paper are the following:

- We propose StAMP, a compiler technique that splits the binary code of a task into consecutive code intervals. Similarly to state-of-the-art techniques [2,12], StAMP generates worst-case memory access profiles for intervals with known WCETs. However, in contrast to [12] and [2], StAMP links back intervals to locations in the code of tasks. Moreover, the algorithmic complexity of StAMP is much lower than the one of the algorithms from [2].
- We provide an extensive experimental evaluation of StAMP, showing in particular that:
    - Interval extraction using node-centric SESE regions results in finer-grain regions than traditional edge-centric regions.
    - Controlling the depth at which the SESE region tree is explored allows us to control the size of the produced intervals.

The memory access profiles generated by StAMP can be used by off-line scheduling strategies to minimize interference overhead. Moreover, the fact that StAMP links back intervals to locations in the code of tasks provide useful information to take elaborate run-time decisions such as dynamic reconfiguration of schedules and re-calculation of interference delays [15, 16]. This paper focuses on calculation of memory access profiles, their use for off-line or on-line scheduling strategies is considered outside the scope of the paper.

The rest of this paper is organized as follows. The method implemented in StAMP is described in detail in Section 2. Experimental results are given in Section 3. Section 4 compares our approach to related techniques. We finally discuss the results achieved and present our future work in Section 5.

2 Estimation of memory access profiles with StAMP

After presenting the system model StAMP relies on (Section 2.1, the properties of code intervals as computed by StAMP are detailed in 2.2. SESE regions, the blocks from which code intervals are constructed, are defined in 2.3. Sections 2.4 and 2.5 then respectively present the construction of code intervals and the calculation of their worst-case number of memory accesses.

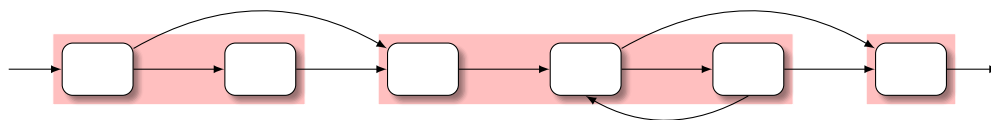### 2.1 System model and problem statement

StAMP operates of the binary code of an individual task, from static analysis of its Control Flow Graph (CFG). The target architecture may have a complex memory hierarchy (instruction and data caches). We assume that there exists a way (for instance static cache analysis as in our experimental evaluation in Section 3) to figure out if an access to a given address may result in a memory access.

The problem addressed by StAMP is the following. Given the code of a task, StAMP splits its code in consecutive code intervals (formally defined in 2.2) and for each of them calculates the worst-case number of memory accesses that may occur when executing the interval (*memory access profile*).

2.2 Code intervals

The analysis in StAMP first divides a given control-flow graph into consecutive *code intervals*, each linked to a code section, for which we individually compute the number of worst-case memory accesses. Then, WCET analysis is performed to convert the code-based segmentation into a time-based segmentation, allowing a straightforward bidirectional mapping between the two.

Picking the right abstraction to represent code intervals is critical for the correctness and effectiveness of the method. In this work, a code interval is a single-entry, single-exit sub-graph of the control-flow graph. Code intervals cover the entirety of the control-flow graph and are chained as a sequence. Figure 1 represents an example control-flow graph along with an example code interval cover for it.

Figure 1 Example control-flow graph in black with an example code interval cover of length 3 in pink. Notice how all exit edges of a given code interval point to the entry point of the next code interval.

More formally, code intervals are defined as follows (Definition 3), based on the concepts of Control Flow Graph (Definition 1) and Code Interval Cover (Definition 2).

▶ **Definition 1** (Control Flow Graph (CFG)). *A CFG is a directed connected graph $C = (V, E)$, with V the vertices of the CFG (basic blocks, straight-line sequences of instructions with no branch no out except at the exit) and E the edges (pair of nodes, subset of $V \times V$), that represent the possible control flows between basic blocks. A CFG can contain special vertices (call nodes) representing calls to another CFG. A CFG has a single entry node and a single exit node.*

▶ **Definition 2** (Code interval cover). *Let $n \in \mathbb{N}$. A code interval cover of length n is a partition $(I_k)_{k \in \{1,\ldots,n\}}$ of a CFG C such that for all $k \in \{1,\ldots,n\}$, the following properties hold:*

- *If there exists $m \in \{1,\ldots,n\}$ different from k such that there exists $(v_1, v_2) \in E$ with $v_1 \in I_k$ and $v_2 \in I_m$, then m and v_2 are unique.*
- *No such m exists if and only if I_k contains the exit node of the CFG.*

▶ **Definition 3** (Code interval). *A code interval is an element of a code interval cover.*

A direct consequence of Definition 2 is that a code interval cover always covers the entirety of the CFG, as it is a partition. A node of the CFG is thus always part of exactly one code interval. Another consequence is that the exit edges of a given code interval always enter the same code interval, as m in the definition is unique, and only the interval containing the exit node does not have exiting edges. A code interval cover is thus always a *chain* of code intervals. Note that a code interval cover always exists for a given control-flow graph, as the single-element partition of the control-flow graph itself is a valid code interval cover.
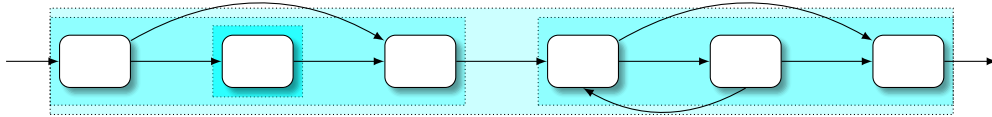
2.3 SESE regions

The single-entry and single-exit nature of code intervals invites us to formally introduce and use the notion of Single Entry Single Exit (SESE) regions. SESE regions may be edge-centric as originally introduced in [8] or node-centric.

▶ **Definition 4** (Edge-centric SESE region). *An edge-centric SESE region of a CFG $C = (V, E)$ is a subset $R \subseteq V$ for which there exists $e_{in} \in E$ and $e_{out} \in E$ such that:*
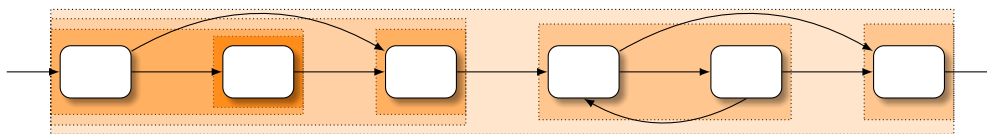- *For all $(v_1, v_2) \in E$, if $v_2 \in R$ then either $v_1 \in R$ or $(v_1, v_2) = e_{in}$.*
- *For all $(v_1, v_2) \in E$, if $v_1 \in R$ then either $v_2 \in R$ or $(v_1, v_2) = e_{out}$.*

Previous work has shown that it is possible to generate edge-centric SESE regions in a tree arrangement in linear time [8]. A parent-child relationship in the tree indicates that the child region is completely nested in its parent, and detected regions never partially overlap. An example of edge-centric tree-arranged SESE regions (from the CFG of Figure 1) is represented in Figure 2.

Figure 2 Control-flow graph from Figure 1 with stacked non-overlapping edge-centric SESE regions in blue (darker is deeper in the SESE region tree).

As a side-product of our approach, we propose another way to define SESE regions. Instead of defining frontiers of regions as entry and exit *edges*, we focus on entry and exit *nodes*. Similarly to edge-centric regions, node-centric regions can be arranged in an inclusion tree. An example of node-centric tree-arranged SESE regions (from the CFG of Figure 1) is represented in Figure 3.

Figure 3 Control-flow graph from Figure 1 with stacked non-overlapping node-centric SESE regions in orange (darker is deeper in the SESE region tree).

▶ **Definition 5** (Node-centric SESE region). *A node-centric SESE region of a CFG $C = (V, E)$ is a subset $R \subseteq V$ for which there exists $v_{in} \in V$ and $v_{out} \in V$ such that:*
- *For all $(v_1, v_2) \in E$, if $v_2 \in R$ then either $v_1 \in R$ or $v_1 = v_{in}$.*
- *For all $(v_1, v_2) \in E$, if $v_1 \in R$ then either $v_2 \in R$ or $v_2 = v_{out}$.*

Node-centric regions are extracted from the CFG of a program using Algorithm 1. A node-centric SESE region is canonically represented by its pair (v_{in}, v_{out}).

Algorithm 1 Algorithm to compute node-centric SESE regions from a control-flow graph C. Returns a set of regions that are represented as their pair (v_{in}, v_{out}). This algorithm has a worst-case time complexity of $O(n^3)$ where n is the number of CFG nodes in a function.

```
1: function COMPUTENODESESE(C)
2:     compute dominators and post-dominators of C
3:     regions := {}                                    ▷ Generate all regions (including overlap).
4:     for each node start ∈ V do                                              ▷ C = (V, E)
5:         for each dominator end of start do
6:             if start is a post-dominator of end then
7:                 inside := nodes in region (start, end) ignoring back edges
8:                 if all back edges in region (start, end) link two nodes of inside then
9:                     regions.push((start, end))
10:                end if
11:            end if
12:        end for
13:    end for
14:    overlapping_regions := {}                              ▷ Remove overlapping regions.
15:    for each r₁ in regions do
16:        for each r₂ in regions do
17:            if r₁ ∩ r₂ ≠ ∅ and r₁ ⊄ r₂ and r₂ ⊄ r₁ then
18:                overlapping_regions.insert(r₁)
19:                overlapping_regions.insert(r₂)
20:            end if
21:        end for
22:    end for
23:    return regions ∖ overlapping_regions
24: end function
```

Algorithm 1 is based on the well-know concepts of dominators and post dominators used in compilers[1]. It is divided in two loops: one generating all node-centric SESE regions, that could possibly overlap, and one filtering out overlapping regions to enforce the inclusion property among regions. The second loop operates as follows. Consider two regions $r_1 = (n_1, m_1)$ and $r_2 = (n_2, m_2)$ that overlap without inclusion. We can consider without loss of generality that

---

[1] A node d *dominates* a node n if every path from the entry node of the CFG to n must go through d. A node d *post-dominates* a node n if every path from n to the exit node of the CFG must go through d

n_2 is also a node of r_1. Therefore, r_1 and r_2 contain three node-centric SESE regions that do not overlap: (n_1, n_2), (n_2, m_1) and (m_1, m_2). As these three regions cover the exact same nodes as r_1 and r_2, r_1 and r_2 can be filtered out.

2.4   Computing code interval covers

Code interval covers can be computed through a depth-first traversal over the SESE tree. The concept of SESE tree is defined in Definition 6. Whether the kind of regions used is edge-centric or node-centric does not influence the definition.

▶ **Definition 6** (SESE tree). *A SESE tree has two constructors:*
- BASICBLOCK(n) *containing a control-flow graph with node n alone.*
- REGION(r, *children*) *containing a SESE region r and a non-empty set of SESE tree children. The two following properties must also hold:*
  - *Any CFG node covered by an element of children must be covered by r.*
  - *Any CFG node in r must be covered by exactly one element of children.*

From this tree definition, we derive Algorithm 2 to generate a code interval cover.

■ **Algorithm 2** Computes a control-flow ordered code interval cover, taking as parameters a SESE tree and a fuel amount. The code interval cover is represented as a sequence of SESE trees exactly covering the nodes of the code interval. The fuel parameter controls the exploration depth: the higher the fuel, the more fine-grain the cover.

```
 1: function CODEINTERVALCOVER(tree, fuel)
 2:     if tree is a BASICBLOCK(n) then
 3:         if n is a call node to callee then
 4:             return [tree] + CODEINTERVALCOVER(callee, fuel)
 5:         else if n has up to one control-flow successor then
 6:             return [tree]
 7:         else
 8:             return UNCHAINABLE
 9:         end if
10:     else if tree is a REGION(r, children) then
11:         if fuel = 0 then
12:             return [tree]
13:         else                            ▷ Build CFG ordering of SESE children of tree, if possible.
14:             result_intervals := [ ]
15:             child := control-flow entry of children
16:             while child ≠ null do
17:                 child_result := CODEINTERVALCOVER(child, fuel − 1)
18:                 if child_result = UNCHAINABLE then
19:                     return [tree]
20:                 end if
21:                 result_intervals := result_intervals + child_result
22:                 child := any control-flow successor of child in children
23:             end while
24:             return result_intervals
25:         end if
26:     end if
27: end function
```

This algorithm takes as parameters a SESE tree *tree* and an amount of *fuel* modeling the maximum depth of the recursive exploration of *tree* (each recursive call consumes one unit of *fuel*), and returns a sequence of SESE trees that will each represent a code interval. The

depth-first traversal matches on the two constructors of SESE trees recursively, reducing *fuel* on each recursive call. In the BASICBLOCK case, we either forward the construction to a called CFG if there is one by unfolding the CFG of the callee, or check whether the node can be part of a chain (returning UNCHAINABLE if not). In the REGION case, we either end the exploration if there is no fuel left, or recursively call the construction function on the SESE children of the region, in control-flow order. Note that if one of the children cannot be chained, we abort breaking the region in parts and simply return the region itself, as we could not output a more precise code interval. The worst-case time complexity of the algorithm is linear with respect to the number of SESE tree nodes, which in the general case is equivalent to quadratic with respect to the number of CFG nodes in a function.

2.5 Computing memory access profiles

Once the control-flow graph is divided into code intervals, we compute the worst-case number of memory accesses (WCMA) for each interval by largely relying on standard Implicit Path Enumeration Technique (IPET) analysis [10], both for estimating: (i) the (partial) WCET of code intervals; (ii) their worst-case number of memory accesses (WCMA).

To compute the WCET of each code interval, we constrain to zero the WCET value of each basic block outside the code interval under analysis and outside any of the functions it could call (recursively). The WCMA of each code interval is estimated in a similar way, by setting to zero the WCMA of each basic block outside the interval under analysis. This partial WCET/WCMA calculation is straightforward due to the single-entry single-exit feature of code intervals.

Detecting if a load/store instruction may result in memory access depends on the presence of instruction/data caches in the architecture under analysis. The experimental evaluation of StAMP uses an architecture with instruction and data caches, thus not all load/store instructions result in memory accesses, as explained in Section 3.1.

Then, the only step left is to create a memory access profile out of the code intervals.

▶ **Definition 7** (Memory access profile). *A memory access profile is a sequence of pairs* $(wcet, wcma)$ *representing a time-sequence of code intervals of maximum duration wcet in which at worst wcma memory accesses can happen.*

Figure 4 Example memory access profile (`minver`, node-centric SESE regions, non-limiting large value of fuel parameter.

As code intervals are chained in a sequence, it is possible to create a memory access profile by mapping code intervals in the produced sequence of code intervals to their pair $(wcet, wcma)$. An example of a memory access profile is given in Figure 4. The x-axis represents the code intervals with their WCET in cycles, while the y-axis represents the corresponding WCMA.

3   Experimental evaluation

In this section, we present the details of our implementation and discuss the benefits and limitations of StAMP through experiments.

3.1   Implementation of StAMP and experimental setup

StAMP was implemented within the Heptane WCET analysis platform [6]. In order to evaluate the quality of the generated memory access profiles, we ran StAMP on the benchmarks provided by Heptane (C code of the Mälardalen benchmarks [5] with loop bounds annotated using the Heptane format). The target architecture used is based on MIPS with a single layer of data cache and instruction cache. Both caches are 2-associative LRU caches with 32-bytes cache lines and 32 sets. Heptane provides a built-in instruction cache, data cache and address analysis. Every access not classified as *always-hit* by the cache analysis of Heptane is assumed to perform a memory access. This allows obtaining the Worst-Case number of Memory Access (WCMA) of each basic block. Heptane additionally provides IPET analysis for WCET computation, that was modified to compute the worst-case number of memory accesses of intervals. As such, most of the implementation work was to compute the code interval covers.

3.2   Memory access profile results

We ran StAMP on all benchmarks with a very large value for the fuel parameter (simply termed *unlimited fuel* hereafter). All the produced profiles are provided as supplementary material that can be downloaded from [4]. An example generated profile is given in Figure 4. Note that some intervals are only a couple of cycles long and are thus not visible on the graphical representation. While this graphical representation of memory access profiles is useful to compare StAMP with state-of-the-art methods, it does not translate all the capabilities of StAMP. Indeed, each block in the memory profile corresponds to a code interval. This allows mapping back each block's WCET and WCMA information (in the time domain) to a code interval (in the code domain). This is especially useful for schedulers which operate at run-time when the current position in the code is also known.

3.3   Granularity using edge-centric versus node-centric SESE regions
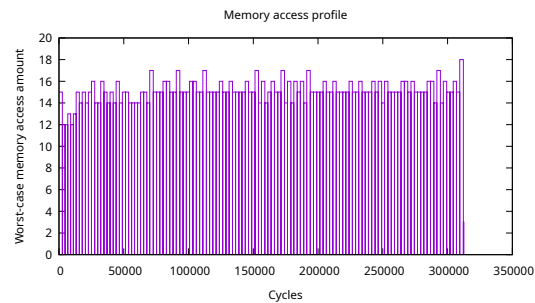
StAMP is generic over the flavor of SESE regions (edge-centric vs. node-centric) used to compute code intervals. Table 1 compares the length of code interval covers with edge-centric and node-centric regions.

With unlimited fuel, node-centric SESE regions systematically outperform edge-centric SESE regions in terms of length of the generated cover (the larger the number of intervals, the more precise the information provided to the scheduler). This can be explained by noticing that any edge-centric SESE region with $e_{in} = (s,t)$ and $e_{out} = (s',t')$ induces a node-centric SESE region with $v_{in} = t$ and $v_{out} = t'$. As such, there are always more node-centric SESE regions than edge-centric SESE regions.

One of the most interesting results is observed on benchmark `nsichneu`, that features many `if` statements. In this benchmark, edge-centric SESE regions generate only a single code interval, while node-centric SESE regions generate 127 code intervals. The node flavor of StAMP results in particularly fine-grain intervals, as shown in Figure 5. In this example, the large number of intervals may increase the complexity of off-line scheduling strategies. However, merging intervals is straightforward, because they form a sequence that covers all the code.

Table 1 Length (number of intervals) of code interval covers (CIC) (node vs edge-centric regions, unlimited fuel).

	CIC length			CIC length	
Benchmark	**Edge**	**Node**	**Benchmark**	**Edge**	**Node**
ud	11	**12**	select	**5**	**5**
insertsort	**3**	**3**	ns	3	**5**
sqrt	3	**4**	minver	**17**	**17**
matmult	**17**	**17**	crc	5	**15**
fibcall	3	**5**	minmax	1	**3**
fft	7	**9**	simple	1	**4**
cover	**13**	**13**	ludcmp	5	**6**
expint	3	**4**	qurt	7	**10**
jfdctint	**9**	**9**	bs	**5**	**5**
statemate	9	**17**	bsort100	7	**9**
lcdnum	**3**	**3**	nsichneu	1	**127**

Figure 5 Memory access profile for benchmark `nsichneu` (node-centric regions, unlimited fuel).
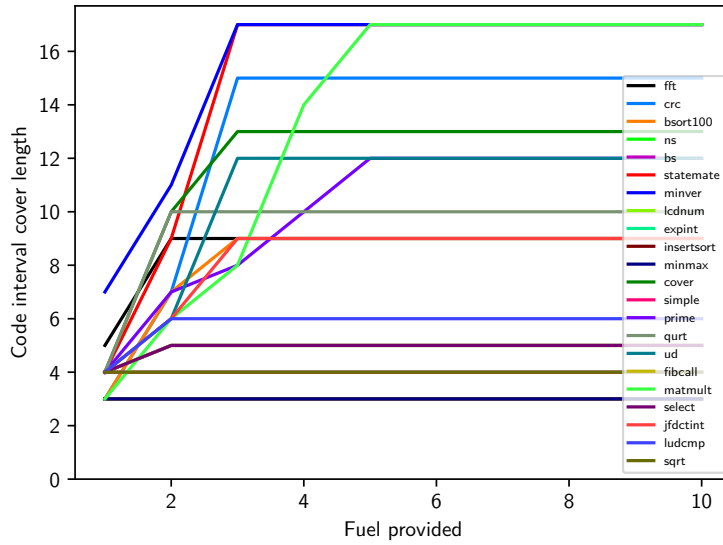
3.4 Controlling the granularity of memory access profiles

The longer the generated code interval sequence length, the more accurate the interference calculation, but the more complex the scheduling, which motivates the need to control the length of the generated sequences. This control is provided in StAMP by the *fuel* parameter, which commands how deeply the SESE tree is traversed. Figure 6 illustrates the effect of varying the amount of fuel on the sequence length for node-centric regions. Evolution of `nsichneu` is not drawn for clarity as the curve goes very high. Its behavior is however similar to other benchmarks: after a few levels of recursion, it remains stable.

Augmenting the amount of fuel increases the generated code interval cover length. However, this control is limited, as SESE trees in practice are not very deep in our benchmarks. As seen in Figure 6, none of our benchmarks benefit from a value of fuel higher than five. Instead, granularity can be reduced by merging consecutive code intervals as their union forms a new code interval. This alternative method further allows control over the size of code intervals.
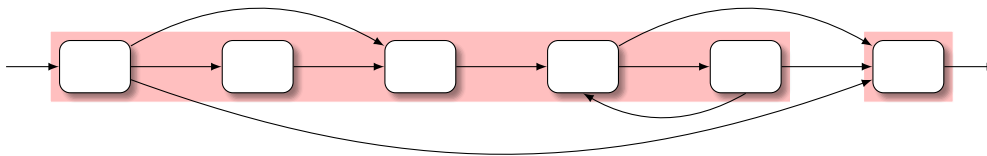
3.5 Limitation of code interval cover computation

The analysis in StAMP in some benchmarks does not go deep in the SESE tree no matter the amount of fuel, and the number of intervals detected is rather small. This phenomenon illustrates a limitation of the code interval model, occurring when control-flow bypasses a large section. As an example, Figure 7 shows a situation in which fine-grained intervals could not be generated because a bypass edge blocks the traversal of successors in Algorithm 2.

Figure 6 Influence of the amount of fuel on the code interval cover length (node-centric regions). `nsichneu` omitted for readability.

Some of our benchmarks (such as for example `expint` or `select`) resulted in a memory access profile containing a single dominant block (surrounded by negligibly short blocks). This is generally caused by `if` statements or loops enclosing most of the code as illustrated in Figure 7.

Figure 7 The same CFG as in Figure 1 is no longer dividable in a code interval cover longer than 2 if we add a bypass edge from the leftmost to the rightmost block. In red is the longest possible code interval cover.

4 Related work

The authors of [2] present TIPs, a technique to extract memory access profiles from code using trace enumeration. Similarly, Oehlert et al. present in [12] a technique to extract event arrival functions using IPET, with memory accesses as a particular case of intervals. These two papers generate memory access profiles per *interval*, with an interval defined as a time interval in task execution. StAMP, in contrast, first generates a profile in the code domain and then converts its results to the time domain. This allows introducing specific code (i.e. synchronization) for tighter identification of interference cost [15, 16].

The time-domain output of the TIPs method differs significantly from StAMP and the method presented by Oehlert et al. While the latter provides an upper bound of the number of memory accesses left to do after a certain point in time, TIPs provides more precise data

that describe the worst-case number of memory accesses during the ongoing time interval. However, this is achieved via exponential time algorithms over the length of the program, while StAMP is in polynomial time over the length of the program.

The PREM (PRedictable Execution Model, [13]) allows to separate phases that use shared resources from those that do not and allows a scheduling technique that avoids the co-scheduling of phases that interfere with each other [14, 18]. Similarly to PREM, StAMP identifies code intervals with different memory access patterns to shared resources but offers more flexibility in the identified access patterns.

The MRSS task model introduced in [3] characterizes how much stress each task places on resources and how sensitive it is to such resource stress and presents schedulability tests using this model. Memory access profiles such as those produced by StAMP produce information at a finer granularity than in [3] (interval level instead of task level) and could be used to improve the schedulability tests of [3].

Code interval covers are similar to super blocks as defined in [1] in the sense that when an interval in the cover is executed, all other intervals will be executed exactly the same number of times, and as such enforce full coverage of the CFG. In contrast to the concept of super block as defined in [1] intervals in our sequences are made of sub-graphs of the CFG and not basic blocks.

5 Conclusion

We have presented StAMP, a technique that generates worst-case memory access profiles from compiled code. StAMP links bidirectionally time-domain and code-domain information. The technique generates a segmentation of a compiled program as a code interval cover in polynomial time and generates a time-domain representation of the worst-case number of memory accesses in this cover.

Future work should first experimentally evaluate the differences between the memory access profiles generated by StAMP and the ones obtained by time-based technique of [2], and most importantly their respective impact on scheduling strategies. Other directions for future work are to improve the expressiveness of the method by improving the code interval model (i.e., move from sequences of intervals to directed graphs), and to enhance memory access profiles to detail when memory accesses occur within code intervals, similarly to [12]. We also believe improvements to the worst-case complexity of the generation of non-overlapping node-centric SESE regions are possible.

───── **References** ─────

1   Hiralal Agrawal. Dominators, super blocks, and program coverage. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 25–34, New York, NY, USA, 1994. Association for Computing Machinery. `doi: 10.1145/174675.175935`.

2   Thomas Carle and Hugues Cassé. Static extraction of memory access profiles for multi-core interference analysis of real-time tasks. In Christian Hochberger, Lars Bauer, and Thilo Pionteck, editors, *Architecture of Computing Systems - 34th International Conference, ARCS 2021, Virtual Event, June 7-8, 2021, Proceedings*, volume 12800 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2021. `doi:10.1007/978-3-030-81682-7_2`.

3   Robert I. Davis, David Griffin, and Iain Bate. Schedulability Analysis for Multi-Core Systems Accounting for Resource Stress and Sensitivity. In Björn B. Brandenburg, editor, *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, volume 196 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:26, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ECRTS.2021.7`.

4   Théo Degioanni and Isabelle Puaut. Stamp: Static analysis of memory access profiles for real-time tasks: supplementary material, 2022. URL: `https://files.inria.fr/pacap/puaut/papers/WCET_2022_appendix.pdf`.

5   Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen WCET benchmarks: Past, present and future. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, volume 15 of *OASIcs*, pages 136–146. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010. `doi:10.4230/OASIcs.WCET.2010.136`.

6   Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. The Heptane static worst-case execution time estimation tool. In *International Workshop on WCET Analysis*, pages 8:1–8:12, 2017.

7   Mohamed Hassan and Rodolfo Pellizzoni. Analysis of Memory-Contention in Heterogeneous COTS MPSoCs. In Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:24, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ECRTS.2020.23`.

8   Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 171–185, New York, NY, USA, 1994. Association for Computing Machinery. `doi:10.1145/178243.178258`.

9   Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154, 2014. `doi:10.1109/RTAS.2014.6925998`.

10   Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC: 32nd ACM/IEEE conference on Design automation*, pages 456–461, 1995.

11   Claire Maiza, Hamza Rihani, Juan Maria Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Comput. Surv.*, 52(3):56:1–56:38, 2019. `doi:10.1145/3323212`.

12   Dominic Oehlert, Selma Saidi, and Heiko Falk. Compiler-based extraction of event arrival functions for real-time systems analysis. In Sebastian Altmeyer, editor, *30th Euromicro Conference on Real-Time Systems, ECRTS 2018, July 3-6, 2018, Barcelona, Spain*, volume 106 of *LIPIcs*, pages 4:1–4:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. `doi:10.4230/LIPIcs.ECRTS.2018.4`.

13   Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for cots-based embedded systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279, 2011. `doi:10.1109/RTAS.2011.33`.

14   Benjamin Rouxel, Steven Derrien, and Isabelle Puaut. Tightening contention delays while scheduling parallel applications on multi-core architectures. *ACM Trans. Embed. Comput. Syst.*, 16(5s):164:1–164:20, 2017. `doi:10.1145/3126496`.

15   Stefanos Skalistis and Angeliki Kritikakou. Timely fine-grained interference-sensitive run-time adaptation of time-triggered schedules. In *IEEE Real-Time Systems Symposium, RTSS 2019, Hong Kong, SAR, China, December 3-6, 2019*, pages 233–245. IEEE, 2019. `doi:10.1109/RTSS46320.2019.00030`.

16   Stefanos Skalistis and Angeliki Kritikakou. Dynamic Interference-Sensitive Run-time Adaptation of Time-Triggered Schedules. In Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:22, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ECRTS.2020.4`.

17 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3):36:1–36:53, 2008.

18 Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Emiliano Betti, and Marco Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real Time Syst.*, 48(6):681–715, 2012. `doi:10.1007/s11241-012-9158-9`.