Configurable and Efficient Memory Access Tracing via Selective Expression-based x86 Binary Instrumentation

Simone Economo, Davide Cingolani, Alessandro Pellegrini and Francesco Quaglia
DIAG - Sapienza University of Rome
{economo,cingolani,pellegrini,quaglia}@dis.uniroma1.it

***Abstract*—Memory access tracing is a program analysis technique with many different applications, ranging from architectural simulation to (on-line) data placement optimization and security enforcement. In this article we propose a memory access tracing approach based on static x86 binary instrumentation. Unlike non-selective schemes, which instrument all the memory access instructions, our proposal selectively instruments a subset of those instructions that are the most (or fully) representative of the actual memory access pattern. The selection of the memory access instructions to be instrumented is based on a new method, which clusters instructions on the basis of their compile/link-time observable address expressions and selects representatives of these clusters. This allows for reducing the runtime cost for running instrumented code, while still enabling high accuracy in the determination of memory accesses. The trade-off between overhead and precision of the tracing process is user-tunable, so that it can be set depending on the final objective of memory access tracing (say on-line vs off-line exploitation). Additionally, our approach can track memory access at different granularity (e.g., virtual-pages or cache line-sized buffers), thus having applications in a variety of different contexts. The effectiveness of our proposal is demonstrated via experiments with applications taken from the PARSEC benchmark suite.**

I. Introduction

Memory access tracing is the art of collecting streams of memory references generated by applications, for either off-line or on-line exploitation. Applications of this type of tracing include: (i) performance/energy-consumption evaluation of alternative architectural designs of memory subsystems [5], [16], [29]; (ii) the detection of security vulnerabilities or leaked/corrupted memory in programs [21]; (iii) the runtime optimization of applications in the context of highly-heterogeneous systems, especially when it comes to the determination of threads/virtual-pages affinity for optimizing memory access in parallel NUMA (Non-Uniform Memory Access) platforms [9], [13], [25].

A well-known method to perform memory access tracing transparently to the applications is *binary instrumentation*, which is based on either hardware or software facilities. The former approach makes use of Performance Monitoring Units (PMUs) embedded within modern processors, which sample machine instructions in order to associate the memory address involved in load/store operations with the instruction/thread that issued the access. The accuracy and overhead of this technique strongly depend on the frequency of sampling, given that too low frequency may lead to poor memory access tracing accuracy, while too frequent sampling may give rise to significant execution slowdown [23]. Furthermore, PMU-based sampling is essentially arbitrary, meaning that the sampled instructions might not be representative of the memory access pattern of the program, in fact they may not even interact with memory. In these scenarios, tracing gives rise to low-quality samples, while still paying the cost for taking them.

On the opposite side, *software instrumentation* resorts to either just-in-time binary translation or ahead-of-time binary rewriting. The first is a dynamic technique that patches the application code of a program on-the-fly, while the program itself is running [11], [26]. In the context of memory access tracing, dynamic patching discovers memory-referencing instructions at runtime in order to identify the target memory location depending on, e.g., the addressing mode used by the instruction and the CPU and/or program state. Ahead-of-time binary rewriting is instead a static approach that patches executables once and for all (before executing them) [18], [24]. This leads to inject instrumenting code that is typically able to identify memory locations targeted by memory-referencing instructions in a faster manner compared to dynamic patching (e.g., because of the exploitation of pre-filled data referring to the instruction category and its addressing mode). Thanks to its reduced runtime intrusiveness, the relevance of the static approach has recently grew up, leading it to become a baseline technique in all the scenarios where memory access traces are exploited for on-line (say runtime) optimizations.

Nonetheless, independently of whether one relies on the static or the dynamic approach, instrumenting all the memory-referencing instructions may still lead to excessive runtime overhead. On the other hand, arbitrary selection (e.g. random sampling) of a subset of the instructions to be instrumented, with the purpose of reducing runtime costs, may lead the memory access tracing process to provide excessively inaccurate outputs. Overall, one core problem to tackle is to identify a reduced set of memory-referencing instructions to be instrumented, so that memory access tracing still provides accurate results for generic memory access patterns. For the case of static instrumentation, this

is further complicated because compile/link-time analysis cannot exploit relevant information characterizing the actual execution flow of programs, especially in scenarios with non-determinism, where alternative (non-disjoint) execution paths might be taken along any thread.

In this article, we take the static instrumentation approach to memory tracing and propose a method that is aimed at selectively instrumenting the subsets of the memory-referencing instructions that are identified as the most (or fully) representative of the actual memory access pattern the application will materialize at runtime, along any thread. Hence our aim is to provide high accuracy of the memory access tracing process, while also reducing its runtime overhead.

In our method, the subset of memory-referencing instructions to be selected for instrumentation is based on *memory address expressions*, which are used to build clusters of instructions whose memory accesses (if materialized) can be traced by logging the access issued by a representative element of the cluster, namely the instrumented one. Further, the granularity according to which memory accesses are traced is user-tunable. This can result extremely useful in scenarios where users are not interested in tracing byte-level accesses, rather they want to trace accesses to coarser-grained memory regions—such as operating system virtual-pages or cache line-sized buffers. Thus we avoid paying runtime costs for tracing accesses with granularity values that are not fully suited for the specific (user-defined) target. Also, our instrumentation discipline is able to provide a count of the times that an access to a given target memory location has been issued throughout the execution, thus providing tracing with an additional accuracy dimension. In fact, literature solutions aimed at reducing memory access tracing costs, e.g. via the employment of techniques alternative to instrumentation such as operating system services [9], [13], do not reach this level of precision. Indeed, they are only able to determine whether a given area (mostly a virtual-page) has been accessed, but not how many times it happened along the lifetime of the application.

Our approach targets the x86 Instruction Set Architecture (ISA) and ELF objects, although it could be suited for other object formats. Further, our implementation exploits Thread Local Storage (TLS) to improve the management of memory access logs in concurrent applications based on the well-established multi-thread technology. We also offer a user-tunable parameter that can be exploited to control the trade-off between the runtime overhead and the accuracy of the tracing process, according to user-specific requirements.

Overall, the original contributions of this work can be summed up as follows:

1) we propose an innovative instrumentation scheme to select a subset of representative memory-referencing instructions, suited for characterizing the overall memory access pattern of an application at reduced runtime cost, by using only static code analysis techniques;
2) our approach is able to provide a count of how many times an access to a target buffer has been issued, while still avoiding instrumenting all the accessing instructions;
3) the instrumentation scheme can be configured in such a way to well fit the desired memory access tracing granularity;
4) the trade-off between tracing accuracy and overhead (once fixed the tracing granularity) can be further controlled through an instrumentation parameter which is user-definable, leading to well suited configurations just depending on the final objective of memory access tracing.

We have experimented our method with applications taken from the PARSEC benchmark suite. The outcoming results confirm the capability of our proposal to deliver accurate memory access tracing at reduced runtime costs.

The remainder of this article is organized as follows. In Section II we discuss related work. The method for selective identification of the memory-referencing instructions to instrument is illustrated in Section III. Experimental data are provided in Section IV.

II. Related Work

The issue of tracing the memory access pattern of applications has been long studied in literature. As for the proposals based on dynamic binary instrumentation (see, e.g., [11], [20]–[22]), they have been historically targeted at off-line analysis of memory accesses, given the non-negligible overhead they introduce. A few approaches slide along the path of making the dynamic instrumentation process more lightweight [26], e.g. by exploiting cached disassembling results generated at runtime when instrumenting previous executions of the same instructions. As for static instrumentation, which is typically less intrusive at runtime, we can categorize literature approaches depending on the objective they pursue. The work in [25] uses static source-code instrumentation to dynamically (at runtime) optimize thread/data placement on heterogeneous NUMA machines. This work is however only targeted at tracing the accesses to (large-size) arrays, in order to locate the associated virtual-pages on the same NUMA nodes where the threads mostly using them are hosted. Skeletonization [15] has been presented as a means for emulating the effects of specific inputs in terms of memory accesses performed by the application. Compared to all these approaches we have a different target, namely the one of providing a method to identify what subsets of memory-referencing instructions are those whose instrumentation would allow to capture the overall memory access pattern by the application.

A few solutions rely on application-external libraries, like MPI/OpenMP libraries (see, e.g., [7], [14], [28]), and are

aimed at determining the level of sharing in the access to library-managed data across concurrent threads/processes. The final target is to optimize the memory layout of the accessed data structures, such as the memory buffers used for data exchange. Variants of these proposals are based on either a-priori knowledge of the source/destination threads for specific data-exchange operations [12] or a knowledge base of the communication pattern built by tracing previous application executions [4], [10]. Our solution does not exploit any specific library (since we rely on x86 binary rewriting), thus being potentially usable in wider application contexts. Also, it does not require any a-priori knowledge base on memory accesses by applications.

Alternative methods to perform memory access tracing build on hardware facilities. In this category we find solutions exploiting the output from memory controllers [3], [6], software-managed TLBs [19], [27] or instruction-based sampling relying on PMUs [8]. Compared to these solutions, we do not require specialized hardware, hence our approach looks highly general. On the other hand, we share with them the idea to provide a customizable memory access tracing support. Indeed, we offer user-tunable parameters that can be exploited to determine the trade-off between tracing accuracy and tracing overhead, as well as the size of the memory area representing the target buffer for the tracing process.

As for operating system-based memory access tracing, most of the literature proposals work at the granularity of virtual-pages [9], [13], while in our approach we can select the level of granularity associated with the memory buffers whose accesses need to be intercepted and traced. Also, these literature approaches are not able to count the number of accesses to individual pages. Rather, they are mostly tailored to the determination of whether a given virtual-page is currently within the locality of the accesses by some thread, which may help to dynamically place it on specific NUMA nodes in parallel machines. Instead, our approach is able to identify the count of accesses that are materialized by the application run. Hence, we can determine both the locality of the accesses and the hotness of the different accessed locations.

III. The Selective Instrumentation Approach

As hinted, our memory access tracing approach relies on static binary instrumentation, a technique that operates on (and transforms) the binary image of a program before the program itself is run. Our tracing method is based on selecting for instrumentation those memory-referencing instructions, say instructions targeting memory as source or destination, which are believed to be more representative of the overall memory access pattern the program would exhibit at runtime. Unfortunately, the problem of choosing the representatives prior to runtime is complex mainly for two reasons:

1) Memory accesses are encoded within a binary file as *expressions*, i.e., arithmetic combinations of register variables and constants.
2) A same memory-referencing instruction may be executed multiple times targeting different memory locations, just depending on the program's control flow at runtime, along any of its threads.

These two aspects force us to see the "static" memory access pattern of a program, say the one observable at compile/link-time, as *a graph of machine instructions containing memory address expressions*. In contrast, the "dynamic" memory access pattern, say the one materialized by whichever thread at runtime, is *a sequence of machine instruction instances containing finalized memory addresses*.

Given this premise, we are left with the compile/link-time task of choosing which memory-referencing instructions to instrument. To accomplish this task, our analysis relies on a well-known program representation called *Control-Flow Graph* (CFG) [2]. It is a directed graph of all the possible ways in which a thread running a program can flow from one instruction to another at runtime. A node in the CFG is called *basic block* and represents a contiguous sequence of instructions with one single entry point at the first instruction and one single exit point at the last. An edge between two blocks in the CFG establishes that the program's control flow is allowed to move (along any thread) from the last instruction of the source, straight to the first instruction of the destination. A basic block may reach multiple destinations and be linked to multiple sources, though it does not admit sideways entrances or departures. Therefore, if control flows into a basic block, it must go through it from the first instruction all the way to the last[1]. Basic blocks represent atomic execution units for the program. For this reason, in our approach we target the selection of what memory-referencing instructions to instrument on a per-basic-block basis.

We shall present our method by initially introducing the baseline formalisms we rely on, and then the algorithms we use for the selection task.

A. The BID abstract model and its templates

On real processors, memory references are represented as *address expressions*, i.e., arithmetic combinations of a fixed number of quantities. As for x86 processors, which are our reference architecture, the most expressive form of addressing is called *SIB addressing* (Scale-Index-Base) and

[1]Sideways entrances and/or departures may happen due to hardware interrupts (e.g., for re-scheduling), as well as software interrupts (say, signal handling). However, hardware interrupts are well beyond the scope of user space program analysis, therefore they do not need to be captured by this representation. As for signal handlers, typically they return control to the interrupted code block so as to allow it to finalize its execution.

involves expressions that combine four quantities[2]. The *base register* b specifies the base address from which the address computation starts. The *index register* i is used as an index value whenever b points to an array-like object or a regular structure. The *scale* is used in conjunction with the index register to specify the step size and can assume one of the values in $\{1, 2, 4, 8\}$. To obtain the final memory address, a displacement d is added to the result, as in the expression $b + i \times s + d$.

An issue with x86 expressions is that compilers and programmers are not bound to adhere to the semantic of SIB addressing. On Linux-GCC, statically-allocated objects can be accessed by putting the base address of a data object into the SIB displacement, since that address is already known at compile-time. On the contrary, the base address of a dynamically-allocated object is not known until the same object effectively materializes in memory. Therefore, its base address is concealed behind a register.

Due to its dependency on compiler's and programmer's idioms, we discard SIB addressing in favor of a more abstract model called *BID addressing* where only base, index and displacement are considered[3], which we denote as a tuple $\{b, i, d\}$. Given a BID expression e, each parameter p_e in the associated tuple $\{b_e, i_e, d_e\}$ can map to either a register or an immediate value. To discriminate the two cases, we introduce the notion of *expression template* τ_e, which is a tuple where each symbol indicates whether the corresponding parameter in the expression e corresponds to a register R or to an immediate I. Due to the way BID addressing is used in x86 processors, we can model any address expression by resorting to two alternative templates only:

(i) the `RRI` template, which is suitable for address expressions used to access dynamic objects, and
(ii) the `IRR` template, which better matches address expressions associated with accesses to static data.

We note that a BID expression e can never lack its base b_e, since it gives always the most significant bits of an effective memory address. On the other hand, missing quantities (index and/or displacement), if any, are modeled in our analysis through a *nil* value, just to represent the fact that they do not contribute to the actual value of expression e (despite x86 supporting them). This leads the `IRR` template, for instance, to gracefully reduce to absolute addressing when both the registers appearing in the template are mapped to the *nil* place-mark.

[2]We omit from the discussion other forms of addressing, such as the *RIP-relative* and the *absolute* ones, since they can easily be reformulated in terms of SIB expressions.

[3]To cope with the absence of a scale parameter, we can consider a revised index quantity which is the result of multiplying the old index with the scale. This makes the shape of expressions slightly simpler, without loss of generality.

B. Stability analysis

Our approach to select the memory-referencing instructions to be instrumented (as representatives of memory accesses) within each basic block relies on rules that compare x86 BID expressions at compile/link-time. However, BID addressing is *implicit*, since it uses register values to derive the effective address that will be associated at runtime with a memory access. Hence, the complexity of dealing with implicit expressions lies in that the value the register will hold at runtime may be unpredictable. As an example, it may depend on program inputs, or may materialize as any of multiple values resulting from taking different execution paths.

To cope with this aspect, our comparison rules for BID expressions rely on *stability analysis*, a simple form of machine-code analysis used to detect when an update may have affected the value of a register. We restrict the scope of such analysis to each single basic block, as opposed to full data-flow analysis (see, e.g., [15]) which operates on the entire CFG. Our stability analysis procedure works as follows. When starting the inspection of a basic block, all the CPU registers are deemed stable. As soon as an x86 instruction that overwrites the value of some register is found while scanning the instructions forming the basic block, the register is assigned a new *incarnation number*, that is a new version number, indicating that it logically represents a new register instance. Two register parameters appearing in BID expressions are then considered as *equal* if and only if they use the same register identifier and the same register incarnation number. Clearly, this approach may lead to false negatives, since it checks for register updates on an instruction-by-instruction basis. For example, it cannot recognize when a sequence of value-changing operations yields an unchanged register value as the final result. This, in turn, may identify two BID expressions as good candidates for being instrumented, even though logging the memory access performed by a single of them via instrumentation would still allow to trace the memory accesses performed by both. However, this scenario can only lead to instrument more memory instructions than strictly needed, thus only increasing the overhead rather than reducing the accuracy.

C. Comparing BID expressions

To derive an effective algorithm for identifying the representative memory-referencing instructions within a basic block upon performing static code analysis, we need a way to compare BID address expressions.

Intuitively, two BID expressions are equal if and only if they share the same template (either `RRI` or `IRR`) and their parameters b, i, and d have the same values. If templates mismatch, then the expressions are deemed incomparable. Within the same template, parameter-wise equality is defined as expected for immediate values, while it relies on incarnation numbers for registers (according to stability analysis

as discussed in Section III-B). Formally speaking, given a basic block b that contains a set $\mathcal{E}_b$ of memory address expressions (associated with memory-referencing instructions), we introduce the *equivalence relation* $=$ between any two expressions (e, e') belonging to $\mathcal{E}_b$ as follows:

$$e = e' \Leftrightarrow (\tau_e = \tau_{e'}) \wedge (b_e = b_{e'} \wedge i_e = i_{e'} \wedge d_e = d_{e'}) \quad (1)$$

By Equation (1), all the BID expressions of the basic block b can be clustered into different classes, depending on whether they are equivalent or not. We define the set of *candidate memory expressions* $\mathcal{C}_b$ of basic block b as the one formed by picking a single representative from each equivalence class. It must be noted that the instrumentation of a single element of $\mathcal{C}_b$ leads to capture the memory accesses by all the memory-referencing instructions in the same class. Indeed, given the property of any basic block b to have all its instructions executed (if activated at runtime), then the access to some target memory area performed by an instrumented instruction in $\mathcal{C}_b$ can be counted as many times as the cardinality of the corresponding equivalence class. Therefore, this optimization has no detrimental effect on the memory access tracing accuracy for block b. In fact, by using the above notion of equivalence the possibility of non-traced accesses (in case of actual execution of the basic block b) is reduced to zero.

However, if we simply instrument all the memory-referencing instructions associated with BID expressions in $\mathcal{C}_b$, we might still perform redundant memory access tracing work. In particular, the granularity according to which memory access tracing needs to be carried out can play a role in determining what elements of $\mathcal{C}_b$ to be actually instrumented. Indeed, two or more elements, although ideally targeting different memory locations, might target a same coarse-grained buffer. Hence, if the user is interested in tracing the accesses at such coarse-grain level, then just one memory referencing instruction among the ones targeting the same coarse-grained buffer could be selected for instrumentation (as the one representative of all the accesses).

To take the requested memory access tracing granularity into account for driving the selection of the elements in $\mathcal{C}_b$ to be instrumented, we introduce the notion of *expression distance* between two candidates e and e'. Informally, two expressions e and e' belonging to $\mathcal{C}_b$, which are associated with different templates, cannot be related through a distance. On the contrary, when e and e' have the same template, we can measure their distance by comparing their characterizing quantities $\{b_e, i_e, d_e\}$ and $\{b_{e'}, i_{e'}, d_{e'}\}$ on a parameter-by-parameter basis.

Figure 1 shows the algorithms we devised to implement the distance functions $\delta_{RRI}(e, e')$ and $\delta_{IRR}(e, e')$ for expressions associated with the two alternative templates. Formally speaking, we call *score triplet* a tuple of the form $\langle s_b, s_i, s_d \rangle$, where each value is a score for the respective parameter $p \in \{b, i, d\}$. We define with aliases s_3, s_2 and s_1 the scores assigned to each parameter in decreasing order. To obtain a valid scoring system, a triplet must satisfy the constraints $s_1 < s_2 < s_3$ and $s_1 + s_2 < s_3$. Then, for each parameter p characterizing e and e', the algorithms in Figure 1 increment the current distance between e and e' by a value s_p whenever $p_e \neq p_{e'}$ (for register parameters) or $|p_e - p_{e'}| \geq C$ (for immediate parameters). As a result, similar expressions end up being 'closer' than other pairs of expressions which have less similar parameter values. The value C in the pseudo-code is the *chunk size* and allows to specify the user-desired memory access tracing granularity. By the structure of the algorithms that compute the distance function for the two alternative templates, it comes out that two expressions e and e' belonging to $\mathcal{C}_b$, which target two different memory addresses with relative offset less than C, will have distance zero.

```
procedure δ_RRI(e, e')
    δ ← 0
    if b_e = b_e' then
        if i_e = i_e' then
            if |d_e − d_e'| ≥ C then
                δ ← δ + s_3
        else
            δ ← δ + s_1
    else
        if i_e = i_e' then
            δ ← δ + s_2
        else
            δ ← δ + s_1 + s_2
    return δ
```

```
procedure δ_IRR(e, e')
    δ ← 0
    if |b_e − b_e'| ≥ C then
        δ ← δ + s_3
    else
        if i_e = i_e' then
            if d_e ≠ d_e' then
                δ ← δ + s_1
        else
            if d_e = d_e' then
                δ ← δ + s_2
            else
                δ ← δ + s_1 + s_2
    return δ
```

Figure 1. Algorithms implementing the distance functions for the two templates `RRI` and `IRR`. Score triplets are respectively $\langle s_2, s_1, s_3 \rangle$ and $\langle s_3, s_2, s_1 \rangle$.

Coming back to the selection of the memory-referencing instructions to be instrumented, if the tracing process is aimed at determining accesses to buffers of size at least C, then we could discard instrumenting one of the two zero-distance expressions. However, an issue that comes out is when we need to trace accesses to buffers of size C with a particular memory alignment. In such a case we might have two expressions e and e' at distance zero (i.e., at relative offset less than C) which lead to hit two different memory buffers of size C. An example is shown in Figure 2, where e and e' hit two different memory locations with relative offset less than $C = 4KB$ (meaning that we are tracing memory accesses at page-level granularity) which fall in two adjacent virtual-pages within the address space.

Anyhow, two expressions at distance zero have less value, if both instrumented, since instrumenting a single of them would likely allow capturing the access to a given target buffer of size C. This concept is further exploited in our approach when coming to the specification of the user affordable per-basic-block tracing overhead, and its trade-off vs accuracy. Overall, once we have computed distances between same-template expressions belonging to $\mathcal{C}_b$, we

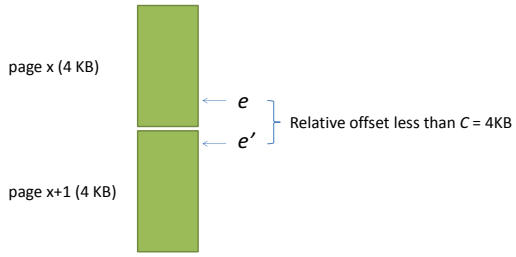

Figure 2. Expressions at distance zero (say with relative offset less than $C = 4KB$), which do not actually access the same 4KB sized buffer.

compute a so called "value of instrumentation" v_e to be associated with expression e in $\mathcal{C}_b$. Such value is structured as a tuple $\langle |[e]|, \bar{\delta}_e \rangle$, where $|[e]|$ is the size of the respective equivalence class to which e belongs, while $\bar{\delta}_e$ is the average distance of e towards all the other same-template candidates based on an arithmetic mean. Given two values v_e and $v_{e'}$, we have that:

$$\begin{aligned} v_e < v_{e'} \text{ iff } \quad & |[e]| < |[e']| \text{ or } |[e]| = |[e']| \wedge \bar{\delta}_e < \bar{\delta}_{e'} \\ v_e = v_{e'} \text{ iff } \quad & |[e]| = |[e']| \wedge \bar{\delta}_e = \bar{\delta}_{e'} \end{aligned} \quad (2)$$

just to represent the fact that, the access count being equal, expressions in $\mathcal{C}_b$ associated with greater average distance from the others will exhibit a higher value of instrumentation.

D. Selecting BID expressions

After v_e is computed for any element in $\mathcal{C}_b$, the final selection of what memory-referencing instructions to instrument is mapped to a (0,1)-knapsack problem, constructed as follows. We consider the cost introduced by the instrumentation of a single memory-referencing instruction associated with an element in $\mathcal{C}_b$ as our *instrumentation cost unit*. The total weight of our knapsack problem is defined as $W = \lceil \omega \cdot m \rceil$, where ω is a user-defined cost-controlling parameter in $[0, 1]$, and m is the cardinality of the set $\mathcal{C}_b$. We solve the knapsack problem iteratively according to the following steps:

Step A: Starting from $\mathcal{C}_b$, we generate the set $\widehat{\mathcal{C}_b}$ where for any set of expressions originally present in $\mathcal{C}_b$ having distance 0, we include in $\widehat{\mathcal{C}_b}$ a single element e, say the one with maximum v_e;

Step B: We select elements in $\widehat{\mathcal{C}_b}$, corresponding to memory-referencing instructions to be instrumented, by solving the linear programming problem in Equation (3), whose tractability is guaranteed by having equally-weighted items:

$$\begin{aligned} \max \sum_{e \in \widehat{\mathcal{C}_b}} v_e x_e \qquad & \text{such that } \sum_{e \in \widehat{\mathcal{C}_b}} x_e \leq W \\ & x_e \in \{0, 1\} \quad v_e \geq 0 \end{aligned} \quad (3)$$

Step C: We eliminate from $\mathcal{C}_b$ any element e that has been already selected in Step 2, and if room is still available in the knapsack, then we repeat selecting elements by restarting from Step A, with the only constraint that the elements already in the knapsack are still kept there when running again Step B. So we fill the knapsack incrementally across subsequent iterations of these algorithmic steps.

E. Accuracy measures

By the above structure of our knapsack solving algorithm, we initially discard instrumenting memory-referencing instructions associated with expressions at distance zero from each other, since as discussed these would add lower contribution to the accuracy of the tracing process. However, if the user-specified cost-controlling parameter ω still allows for instrumenting more instructions, then we start recovering the instrumentation of the originally discarded instructions by iterating again over the selection steps.

On the other hand, if the user is not interested in memory alignment while tracing accesses, then the discarded memory-referencing instructions could be somehow reconsidered by counting the memory accesses they will perform via some instrumented instruction at distance zero. More in detail, if we have two zero-distance expressions e and e' in the same basic block, associated with two equivalence classes, such that e is instrumented and e' is not instrumented, we can associate the sum of the two counters $|[e]|$ and $|[e']|$ to e, so that if e materializes at runtime, then it will contribute to the memory access trace to its target chunk via $|[e]| + |[e']|$ accesses.

By the above reasoning, we can discriminate between two different accuracy expressions for memory access tracing. The *minimal accuracy* $MA_b \in [0, 1]$ is defined as the ratio of instrumented accesses weighted by the cardinality of the corresponding equivalence class, over all accesses in the basic block b. Overall, MA_b can be expressed as:

$$MA_b = \frac{\sum_{e \in \mathcal{K}_b} |[e]|}{n} \quad (4)$$

where $\mathcal{K}_b$ is the knapsack of exactly W expressions chosen by our selection algorithm among all the possible candidates in $\mathcal{C}_b$, and n is the total number $|\mathcal{E}_b|$ of memory-referencing instructions in the basic block. Note that, by virtue of our selection algorithm, MA_b is indirectly affected by the chunk size. Indeed, by varying the size of the chunk, we affect the number of zero-distance pairs seen by the selection algorithm, which in turn can alter the selection of candidates in $\mathcal{C}_b$. As a result, it is possible for a discarded zero-distance expression e' to have an access count $|[e']|$ greater than the one of another expression included in the knapsack.

The second expression for the accuracy, which we refer to as *Alignment-Independent Accuracy* $AIA_b \in [0, 1]$ takes into account zero-distance expressions within a basic block, and

the possibility to count the accesses by non-instrumented instructions (those discarded in **Step A** of our algorithm, and not eventually recovered) via instrumented ones. Given an expression $e \in \mathcal{K}_b$, we associate with it expressions e' at distance zero, which do not belong to the knapsack $\mathcal{K}_b$ ([4]). Denoting with $\widehat{|[e]|}$ the sum of the cardinalities of all the equivalence classes targeted by this association process having e as fulcrum, AIA_b can be expressed as:

$$AIA_b = \frac{\sum_{e \in \mathcal{K}_b} \widehat{|[e]|}}{n} \tag{5}$$

By construction, Equation (5) expresses an upper bound on the accuracy that our approach can guarantee for a generic basic block. In fact, the actual accuracy may be lower in scenarios where the alignment of the accessed memory areas must be considered. More in detail, associating zero-distance expressions e and e' (and summing the cardinality of the corresponding equivalence classes) may lead to both false positives and false negatives in the access trace. The former arise when e and e' touch subsequent memory aligned chunks, but the accesses by e' are falsely considered as targeting the memory-aligned chunk targeted by e. The latter may arise because the access count of e' is erroneously associated with the same chunk touched by e, and is not associated with the adjacent chunk actually targeted by e'. Hence the accesses to some memory-aligned chunk by the expressions in the equivalence class associated with e' are lost. However, if memory alignment is a stringent constraint, the determination of the accuracy can still (conservatively) resort to MA_b as expressed in Equation (4).

Clearly, the memory tracing accuracy of the entire program execution can be computed as the weighted sum of the accuracy of each basic block (either MA or AIA), with weights corresponding to the number of instances of the execution of each basic block, divided by the total number of basic block executions.

F. Complexity of the instrumentation algorithm

Our selective algorithm instruments a single basic block in $\Theta(n^2)$ time in the worst case, with n being $|\mathcal{E}_b|$ and assuming that the cost of applying either δ_{RRI} or δ_{IRR} is constant. It takes at most n^2 steps to produce the set $\mathcal{C}_b$, since equivalence classes can be computed in no more than a single quadratic pass over instructions in the basic block. To compute distances between representatives of each class we need m_{RRI}^2 applications of the function δ_{RRI} and m_{IRR}^2 applications of the function δ_{IRR}, where m_{RRI} and m_{IRR} are the number of candidates belonging to the two templates. A linear scan of m steps is needed to compute the average distances for all elements in $\mathcal{C}_b$. The selection algorithm can be implemented by sorting all candidates in $\mathcal{C}_b$ according to their values in decreasing order. Candidates are then read from the priority queue one at a time. If the next available candidate must be discarded given the current granularity C, we keep it in the queue and move to the next element. Otherwise, we detach it from the queue in constant time and flag as 'discarded' all others zero-distance elements in the queue. Overall, scanning the queue takes between W and m steps. The cost of discarding zero-distance elements is linear in their number, provided that they are all logically connected. If this connection can be established while computing distances, overall discarding elements will take between 0 and $m-1$ steps. Finally, the queue can be scanned at most W times, when each visit detaches just a single element.

[4]If e' is at distance zero from more than one element in the knapsack, then we associate e' with one element only.

IV. Experimental Assessment

A. Implementation details

In our implementation we resorted to the Hijacker open source framework [24], augmenting its capabilities in order to fully support the proposed instrumentation method and the algorithms it relies on[5]. Also, we offer to the user different operation modes. One is based on decorating an instrumented memory access instruction with a patch that only logs any access it performs. A second one where, upon logging the access, the instrumenting code also reports the size of the equivalence class the instruction belongs to. This mode is useful for generating memory access traces that comply with MA. Finally, a third operation mode associates with each logged access the size of all the equivalence classes of expressions at distance zero which have been aggregated, if any. This mode is useful for generating memory traces adhering to AIA.

To avoid synchronization costs by the instrumenting code when dealing with multi-threading, per-thread buffers need to be used to temporarily log the access trace. Since our implementation targets x86 and ELF objects, we resorted to inserting a new symbol in the `.tbss` section used to implement TLS. The symbol refers to the per-thread memory area where each thread can log the accesses in isolation with respect to any other concurrent thread. Finally, the implementation also offers the possibility to transparently inject in the binary a call to a stub-function that can be used to consume the produced log in a custom manner depending on the final target of memory tracing[6].

B. Results

We report a set of representative data collected with five benchmark applications taken from the PARSEC suite [17], namely `blackscholes`, `fluidanimate`, `canneal`, `freqmine`, and `swaptions`. All the experiments have

[5]Source code available at `https://github.com/HPDCS/hijacker`

[6]For example, the function could simply flush the logged accesses onto files for post-analysis.

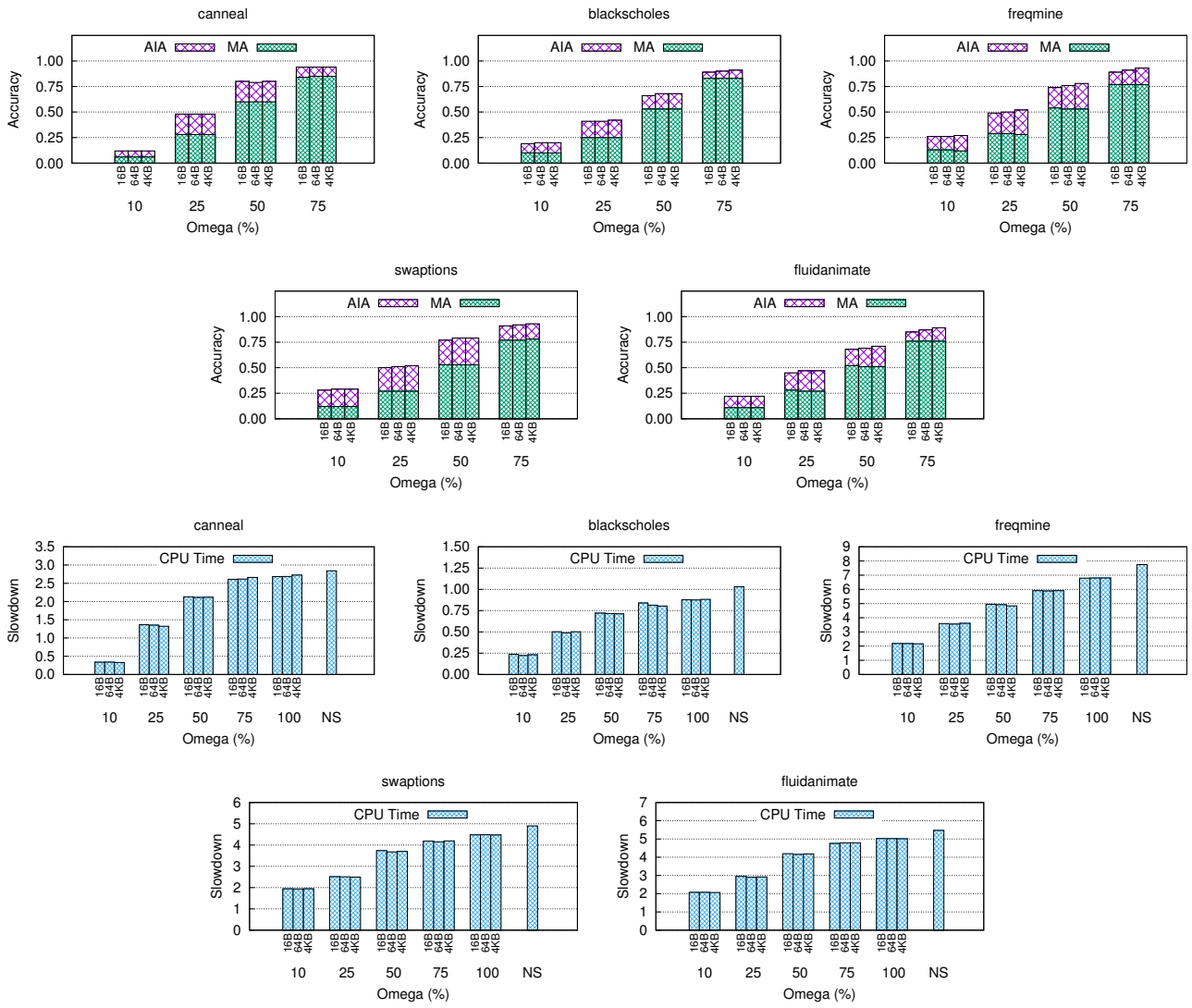

Figure 3. Results with five representative benchmark applications from PARSEC.

been executed by running the applications on top of a 32-core HP ProLiant server equipped with 64 GB of RAM, running Linux (kernel 2.6). The achieved results are shown in Figure 3, where we report both memory access tracing accuracy results and results on the slowdown caused by instrumentation, while varying the cost-controlling factor ω between 0.1 and 1 (where 1 means instrumenting all the memory-referencing instructions falling in different equivalence classes). We also report results for the case of non-selective instrumentation (not considering equivalence classes and expression distances), where any memory-referencing instruction is instrumented, which is used as a baseline especially for slowdown assessment of our proposal. This configuration is marked as 'NS' in the plots. In these experiments, slowdown is computed by including on the critical path of the application execution only the latency for producing the memory access trace within TLS, not for logging (e.g. on stable storage) and manipulating it. The actual need for generating logs and processing the logged trace is in fact dependent on the specific target of memory tracing, which can be considered as somehow orthogonal to our technique. Also, the slowdown value reported in the plots represents the percentage increase in the CPU time used by the benchmark applications with respect to the one observed with no instrumentation at all. As for the accuracy, we report both MA and AIA, so as to provide indications on how our method behaves with different user requirements, in terms of memory alignment of the memory buffers towards which the accesses need to be traced. In this set of experiments we also varied the chunk size C determining the granularity

of memory access tracing between 16 bytes and 4KB. All the runs have been carried out by relying on 32 threads, and each reported sample (either for accuracy or slowdown) represents the average over three executions.

By the data we see how our approach provides accuracy values that scale with a factor grater than one with respect to the scaling of the cost-controlling parameter ω. This allows achieving values of AIA that are close (or slightly above) 0.5 when configuring ω in such a way to only instrument one fourth of the memory-referencing instructions. Similarly, except for `canneal`, we achieve values of AIA of the order of 0.25 when instrumenting one tenth of the memory-referencing instructions. As expected, the reduction of the slowdown achieved by our approach compared to NS scales (almost) linearly when reducing the value of ω. Also, except for settings where ω has the value 10%, MA appears to be (at least) the 60%/70% of AIA, which indicates how a conservative instrumentation decision—based on the will to determine memory accesses by also considering strict alignment of the accessed chunks—does not lead to excessive degradation of the accuracy of the tracing process (once fixed the value of the cost-controlling factor ω). By the data we also observe an impact (although limited) of the chunk size C on both accuracy and slowdown. Indeed, for all the benchmarks, except `canneal`, we observe an increase of the accuracy (especially AIA) when increasing the size of the chunk. Contextually, we generally note a slight reduction of the slowdown with greater values of C. As for the limited impact of the chunk size on accuracy, this is also motivated by the fact that PARSEC applications are generally characterized by basic blocks that are made up by a reduced amount of memory-referencing instructions (on average no more than 7). Therefore, the likelihood of finding zero-distance expressions given any value of C is low. Lastly, we do not exploit the possibility of correlations between expressions in different basic blocks. Studying this type of correlation will be the object of future research work. Overall, our approach appears to provide memory access tracing accuracy that can well scale vs the runtime cost spent by the instrumented code, especially when the percentage of memory-referencing instructions that are instrumented is low.

V. Conclusions

We have presented a static instrumentation method for x86/ELF objects allowing memory access tracing at different granularity levels. The method is based on a policy that selects what memory-referencing instructions to instrument depending on the expressions that characterize the overall set of memory accesses within each basic block of the program. Our method also offers a user-configurable parameter that enables the determination of suited trade-offs between the runtime costs caused by the instrumented memory accesses and the accuracy of the memory access tracing process. Also, with our solution we do not only identify the memory buffers that are accessed by a program, rather we can also count the number of accesses to them (although not necessarily instrumenting all the memory referencing instructions), thus also being able to identify the relative hotness of the different accessed memory areas. We have also provided an open source implementation of our instrumenting tool, and we tested it with benchmark applications taken from the PARSEC suite. A step ahead along this research path would the one of not only performing selective instrumentation of the memory-referencing instructions within each single basic block of code forming the program. Rather, we plan to also include algorithms for the identification of the relative importance of the different blocks of the program, in terms of impact on memory access, so as to further improve the trade-off between the cost one would admit—in terms of amount of instrumented memory accesses—and the actual benefits—in terms of accuracy of the tracing process, just depending on the final objective of memory tracing.

Acknowledgements

Davide Cingolani and Alessandro Pellegrini are also working with Value Up S.r.l., an InResLab partner. This work is partially supported by the HeterOpt - “Transparent Optimization of Software in the Era of Multi-Core Heterogeneous Systems” project funded by Nikesoft with the support of MISE research funds.

References

[1] Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986)

[2] Allen, F.E.: Control flow analysis. In: Proceedings of a Symposium on Compiler Optimization, 1970, pp. 1-19.

[3] Awasthi, M., Nellans, D.W., Sudan, K., Balasubramonian, R., Davis, A.: Handling the problems and opportunities posed by multiple on-chip memory controllers. In: Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques, 2010, pp. 319–330.

[4] Barrow-Williams, N., Fensch, C., Moore, S.W.: A communication characterisation of splash-2 and parsec. In: Proceedings of the International Symposium on Workload Characterization, 2009, pp. 86–97.

[5] Carrington, L., Snavely, A., Gao, X., Wolter, N.: A performance prediction framework for scientific applications. In: Proceedings of the Workshop on Performance Modeling and Analysis, 2003, pp. 926–935.

[6] Casanova, H., Desprez, F., Suter, F.: On cluster resource allocation for multiple parallel task graphs. J. Parallel Distrib. Comput. 70(12), 1193–1203 (2010).

[7] Chen, H., Chen, W., Huang, J., Robert, B., Kuhn, H.: MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters. In: Proceedings of the 20th International Conference on Supercomputing, 2006, pp. 353–360.

[8] Dashti, M., Fedorova, A., Funston, J.R., Gaud, F., Lachaize, R., Lepers, B., Quéma, V., Roth, M.: Traffic management: a holistic approach to memory placement on NUMA systems. In: Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems, 2013, pp. 381–394.

[9] Diener, M., da Cruz, E.H.M., Navaux, P.O.A., Busse, A., Heiß, H.: kmaf: automatic kernel-level management of thread and data affinity. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2014, pp. 277–288.

[10] Diener, M., Madruga, F.L., Rodrigues, E.R., Alves, M.A.Z., Schneider, J., Navaux, P.O.A., Heiss, H.: Evaluating thread placement based on memory access patterns for multi-core processors. In: Proceedings of the 12th International Conference on High Performance Computing and Communications, 2010, pp. 491–496.

[11] DynamoRIO: http://www.dynamorio.org/

[12] Fujimoto, R.M., Panesar, K.S., Panesar, K.S.: Buffer management in shared-memory time warp systems. In: Proceedings of the Workshop on Parallel and Distributed Simulation, 1995, pp. 149–156.

[13] Di Gennaro, I., Pellegrini, A., Quaglia, F.: OS-based NUMA optimization: Tackling the case of truly multi-thread applications with non-partitioned virtual page accesses. In: Proceedings of the 16th Intrnational Symposium on Cluster, Cloud and Grid Computing, 2016, pp. 292–300.

[14] Karlsson, C., Davies, T., Chen, Z.: Optimizing process-to-core mappings for application level multi-dimensional MPI communications. In: Proceedings of the International Conference on Cluster Computing, 2012, pp. 486–494.

[15] Ketterlin, A., Clauss, P.: Efficient memory tracing by program skeletonization. In: Proceedings of the International Symposium on Performance Analysis of Systems and Software, 2011, pp. 97–106.

[16] Laurenzano, M., Simon, B., Snavely, A., Gunn, M.: Low cost trace-driven memory simulation using simpoint. Computer Architecture News 33(5), 81–86 (2005).

[17] C. Bienia, Benchmarking Modern Multiprocessors, Princeton University, 2011, January

[18] Laurenzano, M., Tikir, M.M., Carrington, L., Snavely, A.: PEBIL: efficient static binary instrumentation for linux. In: Proceedings of the International Symposium on Performance Analysis of Systems and Software, 2010, pp. 175–183.

[19] Marathe, J., Mueller, F.: Hardware profile-guided automatic page placement for ccnuma systems. In: Proceedings of the Symposium on Principles and Practice of Parallel Programming, 2006, pp. 90–99.

[20] Marathe, J., Mueller, F., Mohan, T., Mckee, S.A., De Supinski, B.R., Yoo, A.: Metric: Memory tracing via dynamic binary rewriting to identify cache inefficiencies. ACM Transactional on Programming Languages and Systems 29(2), 12(2007).

[21] Nethercote, N., Seward, J.: How to shadow every byte of memory used by a program. In: Proceedings of the 3rd International Conference on Virtual Execution Environments, 2007, pp. 65–74.

[22] Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the Conference on Programming Language Design and Implementation, 2007, pp. 89–100.

[23] Nowak, A., Bitzes, G.: The overhead of profiling using PMU hardware counters (July 2014), http://dx.doi.org/10.5281/zenodo.10800

[24] Pellegrini, A.: Hijacker: Efficient static software instrumentation with applications in high performance computing. In: Proceedings of the 2013 International Conference on High Performance Computing & Simulation, 2013, pp. 650–655.

[25] Piccoli, G., Santos, H.N., Rodrigues, R.E., Pousa, C., Borin, E., Quintão Pereira, F.M.: Compiler support for selective page migration in NUMA architectures. In: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques, 2014, pp. 369–380.

[26] Pin: http://www.pintool.org/

[27] Tikir, M.M., Hollingsworth, J.K.: Hardware monitors for dynamic page migration. Journal of Parallel and Distributed Computing 68(9), 1186–1200 (2008).

[28] Trahay, F., Rué, F., Faverge, M., Ishikawa, Y., Namyst, R., Dongarra, J.: Eztrace: A generic framework for performance analysis. In: Proceedings of the 11th International Symposium on Cluster, Cloud and Grid Computing, 2011, pp. 618–619 (2011).

[29] Uhlig, R.A., Mudge, T.N.: Trace-driven memory simulation: A survey. ACM Computing Surveys 29(2), 128–170 (1997).