Aims and Scope
LITES aims at the publication of high-quality scholarly articles, ensuring efficient submission, reviewing, and publishing procedures. All articles are published open access, i.e., accessible online without any costs. The rights are retained by the author(s).

LITES publishes original articles on all aspects of embedded computer systems, in particular: the design, the implementation, the verification, and the testing of embedded hardware and software systems; the theoretical foundations; single-core, multi-processor, and networked architectures and their energy consumption and predictability properties; reliability and fault tolerance; security properties; and on applications in the avionics, the automotive, the telecommunication, the medical, and the production domains.

Contents

Regular Papers

Special Issue on Software and Compilers for Embedded Systems

Editors: Editor Heiko Falk and Sander Stuijk

Randomized Caches Can Be Pretty Useful to Hard Real-Time Systems

Enrico Mezzetti[1], Marco Ziccardi[1], Tullio Vardanega[1], Jaume Abella[2], Eduardo Quiñones[2], and Francisco J. Cazorla[2,3]

1   University of Padova
    {emezzett,mziccard,tullio.vardanega}@math.unipd.it
2   Barcelona Supercomputing Center
    {jaume.abella,eduardo.quinones,francisco.cazorla}@bsc.es
3   Spanish National Research Council (IIIA-CSIC)

Abstract

Cache randomization per se, and its viability for probabilistic timing analysis (PTA) of critical real-time systems, are receiving increasingly close attention from the scientific community and the industrial practitioners. In fact, the very notion of introducing randomness and probabilities in time-critical systems has caused strenuous debates owing to the apparent clash that this idea has with the strictly deterministic view traditionally held for those systems. A paper recently appeared in LITES [17] provides a critical analysis of the weaknesses and risks entailed in using randomized caches in hard real-time systems. In order to provide the interested reader with a fuller, balanced appreciation of the subject matter, a critical analysis of the benefits brought about by that innovation should be provided also. This short paper addresses that need by revisiting the array of issues addressed in the cited work, in the light of the latest advances to the relevant state of the art. Accordingly, we show that the potential benefits of randomized caches do offset their limitations, causing them to be – when used in conjunction with PTA – a serious competitor to conventional designs.

1   Introduction

Timing analysis techniques aim at deriving upper bounds to the worst-case execution time (WCET) of a software program or parts of it. Timing analysis techniques are generally classified as either static or measurement-based [19]. Static WCET analysis techniques try to compute a provably trustworthy WCET bound for a given program from an abstract model of a processor and the source or executable of the program. Measurement-based analysis aims to contain the intrinsic pessimism of static techniques, by measuring the execution of small program fragments (typically basic blocks) run on the target hardware, and then combining them into WCET values using structural information on the program under analysis.

The recent advent of approaches based on probabilistic arguments [6, 5] suggests that the family of timing analysis techniques can also be usefully classified in relation to their using either deterministic (DTA) or probabilistic reasoning (PTA). Four distinct approaches can then be singled out in that respect: static and measurement-based deterministic timing analysis (SDTA and MBDTA), on the one end of the spectrum; static and measurement-based probabilistic timing analysis (SPTA and MBPTA), on the other [1].

Leibniz Transactions on Embedded Systems, Vol. 2, Issue 1, Article No. 1, pp. 01:1–01:10

Leibniz Transactions on Embedded Systems
LITES   Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

As a precondition to computing sound and tight bounds to worst-case execution time, DTA and PTA alike postulate predictability in the timing behavior of the system of interest, from both the processor and the application perspective (whether top-down or bottom-up). Yet they do so with different flavors. Deterministic approaches require or assume execution platforms whose hardware and software components can be accurately abstracted into a fully deterministic model. Conversely, probabilistic methods allow combining deterministic and randomized timing behavior in processor resources [15], to produce WCET estimates that are guaranteed to be exceeded only with a given (arbitrarily low) probability of occurrence.

Cache memories indeed are one of the processor resources whose timing is randomized in the context of use of PTA [12, 13]. This notwithstanding, PTA should *not* be understood as a timing analysis method *for* randomized caches [1]. Conversely, time randomization (applied to cache memories, but not only) should be regarded as an enabling factor to PTA.

In a recent work, Reineke [17] has critically reviewed the impact of randomized caches on timing analysis [12, 13], raising an array of serious concerns and concluding that randomized caches are harmful for hard real-time systems. While the author's concerns are valid and need to be addressed, the conclusions arrived at are negatively slanted by failing to equitably consider the benefits that randomized caches do bring about.

In this short paper we review the observations and conclusions made in [17], extending them in a way that: (1) the advantages of using randomized caches in hard real-time systems are captured and understood; (2) recent results, appeared subsequently to Reineke's cited work, are recalled to mitigate some of the author's concerns; and (3) finer-grained reasoning compounded by the fresher perspective provided by the point above, is presented for the original conclusions that we deemed too general.

Overall, our work provides substantive arguments to contend that randomized caches used together with PTA should be regarded as competitive alternatives to conventional caches used together with DTA. This is especially true in scenarios where set-associative caches are used in the target processor (which is very common indeed) and timing analysis is performed with measurement-based approaches: there, randomized caches together with MBPTA increase the user's ability to compute trustworthy and tight WCET estimates with less effort.

2 Recap of Main Arguments Against the Adoption of Randomized Caches

The arguments made against the adoption of randomized caches in hard real-time systems in [17] follow a rather schematic flow. The random replacement and placement policies are in fact evaluated for efficiency and precision against their deterministic counterparts – Least Recently Used (LRU) replacement and modulo placement – with respect to both static (SDTA and SPTA) and measurement-based (MBDTA and MBPTA) WCET analysis techniques. The reasoning uses both fully associative and set-associative caches.

The relevant observations and conclusions are recalled below. In the interest of brevity, in the following narrative we do not use verbatim quotes from [17], but rather our own interpretations of the author's observations and conclusions.

2.1 Random Replacement in Fully Associative Caches and SPTA

The first part of the argument considers fully associative caches and develops a comparison between a predictable deterministic replacement policy (LRU) and an instance of random replacement

(Evict on Miss, EoM).[1] After commenting on the correctness of the formula bounding the hit probability (originally proposed in [20] and recently improved in [9]), the superiority of LRU is claimed, building on the following two main observations:

O1 – LRU Dominates EoM on Reuse Distance.   LRU always guarantees better hit probabilities than EoM when the analysis exploits reuse distances.[2]

O2 – Current SPTA Cannot Exploit Stack Distance.   When LRU replacement is used in place of Random replacement, current cache analyses – used with SDTA – can benefit from bounding stack distances[3] that can be arbitrarily lower, hence more accurate and less pessimistic, than reuse distances.

The conclusion drawn in this regard is that:

C1 – LRU Replacement is Always Preferable to EoM.   With simple, state-of-the-art analysis methods, LRU replacement is preferable to Random replacement in S*TA.

2.2   Random Replacement in Fully Associative Caches and MBPTA

The mirror argument against EoM in MBPTA builds on the restrictive assumptions made in the early (still immature) formulation of it presented in [8]. Namely, a variant of the method limited to single-path programs, with known worst-case initial state and no input dependence.

The conclusion here is that:

C2 – Deterministic Replacement Enables More Efficient MBPTA.   (Under restrictive assumptions) a single execution with LRU is sufficient to exhibit the worst-case behavior against hundreds in MBPTA with Random replacement, which makes deterministic replacement far more efficient for MBPTA than Random replacement.

2.3   Random Placement in Set-associative Caches and SPTA

In contrast to fully associative caches, set-associative caches rely on mapping memory blocks to cache sets. In discussing placement policies, [17] compares the classic deterministic modulo placement policy (MOD) with random placement (RAND). The author argues the superiority of MOD over RAND on the observation that the formula for the computation of hit probabilities suffers from weaknesses similar to those noted in the replacement case, in that RAND analysis fails to account for existing dependencies.

A counterexample is presented to sustain that:

---

[1] Note that, from the PTA perspective, LRU exhibits a *degenerate* hit probability, which is either 0 or 1, but nothing in between.

[2] The notion of reuse distance captures the number of *cache accesses* that occur in-between two consecutive accesses to the same address. For instance, in the sequence of accesses a, b, c, b, a, the reuse distance for the second occurrence of access a is 3.

[3] The notion of stack distance captutes the number of *unique addresses accessed* in-between two consecutive accesses to the same address. For instance, in the sequence of accesses a, b, c, b, a, the stack distance for the second occurrence of access a is 2.

O3 – The Independence Assumption Fails.   With RAND, hit probabilities depend on prior accesses to the same set, which breaks the independence hypothesis[4] and compels the user to always assume zero probability of hit.

And therefore:

C3 – Random Placement Nullifies SPTA.   Random replacement cannot be adopted in so far as SPTA cannot handle conditional probabilities.

2.4   Random Placement in Set-associative Caches and MBPTA

Random placement is claimed inappropriate for MB*TA approaches as caches may exhibit extremely poor performance for unfortunate, yet very rare, mapping (conflict) conditions. With MB*TA, therefore, one should either capture those cases in the measurement observations, thus incurring possibly massive overestimation, or ignore (and miss) them, thus becoming unsound. Moreover, those extreme cases, being so far apart, may even cause the statistical test on identical distribution to fail [10]. In any such case, MBPTA will fail.

The firm conclusion here is that:

C4 – Random Placement is not Adequate for MBPTA.   Random placement causes MBPTA to fail.

3   Arguments in Favor of Randomized Caches

In the following we first systematically analyze each claim made in [17] and recalled in Section 2, and then provide specific arguments in support of randomized caches and their use in probabilistic timing analysis approaches.

Revisiting C1 (O1+O2)

It is indeed correct to say that, if we limit ourselves to the reuse distance metric, current state-of-the-art SPTA approaches cannot do better than LRU. Observation O2 is therefore accurate, in the sense that current SPTA does not exploit stack distances. Yet, no intrinsic trait of SPTA allows making final claims on this matter, as further improvements, such as e.g. those outlined in [4, 3], may well arrive at mitigating that limitation.

As noted in [17], current SPTA approaches cannot do better with information on reuse distances considered in isolation. However, having information on individual reuse distances implies also having global knowledge on all reuse distances. Interestingly, the hit probability of an access may benefit from the information on the reuse distances of all the accesses that have occurred within its reuse distance. As observed in [4], full knowledge on the reuse distances of all accesses in a sequence (as assumed by SDTA) allows information to be derived on the cache contention. The cache contention $con(e_l, T)$ of an access e_l in a sequence T is conceptually defined in [4] as the number of accesses within the reuse distance of e_l that have been assigned a non-zero hit probability. All accesses in $con(e_l, T)$ are assumed to use their own separate location in memory (see [4] for a formal definition of con). Using this information, a more precise lower bound to

---

[4] Independence in this context refers to whether the outcome of one cache access depends or not on the output of previous accesses. For instance, the fact that a previous access is a miss leads to an eviction, which may decrease the hit probabilities of subsequent accesses.

■ **Table 1** Comparison of random and LRU hit probabilities for all accesses in the sequence $a, b, c, d, f, a, b, c, d, f$ when reuse and stack distances of all accesses are known (same information) in a cache with associativity 4.

	a	b	c	d	f	a	b	c	d	f
stack distance	∞	∞	∞	∞	∞	4	4	4	4	4
$P(hit_{LRU})$	0	0	0	0	0	0	0	0	0	0
$P(hit_{random})$	0	0	0	0	0	$\left(\frac{3}{4}\right)^4$	$\left(\frac{3}{4}\right)^4$	$\left(\frac{3}{4}\right)^4$	0	$\left(\frac{3}{4}\right)^4$

the hit probability of an access is provided in [4], captured by Equation 1, where k is the reuse distance and N is the associativity of the cache. It is important to observe that $con(e_l, T) \leq k$.

$$P(hit_{random}(e_l)) = \begin{cases} 0 & con(e_l, T) \geq N \\ \left(\frac{N-1}{N}\right)^k & \text{otherwise} \end{cases} \tag{1}$$

Table 1 reports an example taken from [4] where, building on contention information, random replacement is shown to outperform LRU, even when the latter relies on stack distance.

From the example in Table 1 one can conclude that LRU together with SDTA does not always outperform random replacement together with SPTA.

Conclusion C1 was sustainable at that time of its writing. Yet, more recent advances [4, 3] show that it does not hold in the general case. In fact, it is overly difficult to determine whether one approach does always outperform the other, as this may largely depend on the particular system and its domain of application. Hence, no absolute claim can soundly be made against either approaches on the basis of the current state of the art.

Revisiting C2

The claim that random replacement is not efficient with MB*TA rests on the assumptions made in the initial formulation of MBPTA [8]. Considering those assumptions, conclusion C2 in [17] maintains that a single measurement would be sufficient to hit the WCET if deterministic replacement is used, instead of several hundreds with MBPTA.

LRU has been shown to provide better average performance than random replacement [12, 18]. However, systematic pathological cases may well occur with LRU (cf. [18, 16]), which cause it to behave much worse than random replacement. An example case follows. Consider a loop traversing a vector whose size is $N + M$ cache lines where N is the cache size expressed in lines. Under LRU, if $M = 0$ the vector fully fits in cache and all accesses (except cold misses) are hits. This is the best case and no other replacement policy can do better. If $M > 0$ instead, and each cache lines holds a single element of the vector, *all* accesses are misses.[5] Consider now vector $V[]$, with $N = 2$, $M = 1$, and a sequence of accesses $V[1], V[2], V[3]$ in a loop, where $V[i]$ stands for the ith position in V. At any iteration after the first one, when all accesses always miss, $V[1]$ misses and replaces $V[2]$, $V[2]$ misses and replaces $V[3]$, $V[3]$ does so with $V[1]$. In that case, there is no way to hit any line because they are *systematically* evicted before they can be reused. We refer to this phenomenon as pathological LRU behavior. Thus, if $M > 0$ any other replacement policy cannot do worse than LRU. For instance, under random replacement there is always a

---

[5] If each cache line contained D distinct data items, then there would be 1 miss and $D - 1$ hits in that case, due to spatial locality. In the rest of this discussion we ignore hits due to spatial locality since they are identical for all examples and replacement policies.

non-zero probability of hit in cache even if $M > 0$. Of course, as M increases the hit probability decreases, but with a gentle slope and not abruptly, as shown in [18].

Random replacement can indeed incur pathological cases too. Yet, it would do so with a ridiculously low probability and not systematically, which arguably is an even more important trait. Consequently, random replacement leads to increases in execution time with (rapidly) decreasing probabilities. The execution time distribution for high values thus is smooth and the slope of its tail steep, such that lower probabilities of exceedance can be reached with small increases in the WCET threshold.

For LRU instead, we have just seen that whether the scenario is pathological or not, depends on correlation between the footprint of the program data and the cache size, in other words on the application use of the cache space. Small changes in that correlation can indeed create abrupt performance variations [18].

It can thus be contended that, regardless of the 1:100 ratio in the number of runs that LRU may need in comparison with random replacement, the determination of the most convenient replacement policy is so entirely application dependent that no valid general answer can be preordained.

Furthermore, conclusion C2 rests on an assumption that needs careful consideration for a complete view of the problem space to be had.

In point of fact, C2 may only hold – if at all – for fully-associative caches. The distinctive feature of fully-associative caches is that they allow cache analysis to focus exclusively on the replacement policy, disregarding the effects of placement. If set-associative or direct-mapped caches were used instead in the very example considered in C2, a single measurement observation could only provide a WCET bound solely valid for the specific memory layout the program had in the observed run. A memory layout is specific in many respects as it affects execution timing not only because of code placement, but also as an effect of data placement and alignment in memory and on the stack (ignoring here the use of dynamic data allocation). Hence, when the layout changes (even only for small displacements), arising in consequence of re-linking or incremental software integration, the previously computed WCET becomes invalid.

MBPTA cures this problem by leveraging random placement, which allows the analysis to abstract from consideration of all possible memory placements [12, 13].

A very distinctive trait of MBPTA, as opposed to to MBDTA, is its ability to take great benefit from the combination of upper-bounding and randomization in the timing behavior of processor resources, which allows it to implicitly capture a wider scope of execution conditions than MBDTA [7]. With that base, MBPTA yields a WCET value associated with a given confidence, quantified as the probability of exceedance. MBDTA, instead, is designed to only provide a single answer to the timing analysis question, which may consequently be only either true or false. Still, the way to arrive to it in practice is often more based on "engineering judgment", for coverage of unobserved execution conditions, than on scientific evidence of sort. Conversely, by upper-bounding and randomizing the sources of execution time variation in the processor, MBPTA replaces uncertainty by scientifically quantified probabilities.

Path coverage issues should also be considered here. While it is overly difficult with MBDTA as yet to determine how sufficient path coverage can be obtained and how different path traversals may affect cache behavior, MBPTA leverages the character of randomization, which attenuates causal effects. This trait intrinsically leads to less abrupt performance variations [18, 16], which eases the problem of identifying worst-case paths, thereby reducing the hardness of the path coverage problem [14]. Arguably therefore, using random replacement can be deemed to increase the benefits of randomization without increasing the number of runs needed for MBPTA.

In fact, randomization is a veritable enabler for MBPTA, which may equally well be applied to

other processor resources than for replacement policies in caches. The application of randomization to buses, for example, has been recently presented in [11].

Revisiting C3 (O3)

Conclusion O3 is correct for current SPTA approaches. The valid point in O3 is that, when random placement is adopted, SPTA cannot assign hit probabilities greater than zero to individual accesses, in the face of residual dependence between hit probabilities. It should be noted however that existing SPTA techniques [4] have solved a similar problem for random replacement, where independence across accesses is not guaranteed either. The cited work has proven that, in the case of random replacement, dependencies can be studied locally – with low complexity – to obtain tight lower bounds for hit probabilities. Similarly, given the example presented in [17], where the sequence of 10 accesses $\{a, b, a, b, a, b, a, b, a, b\}$ is considered, one should be able to determine that the probability of having up to ten misses in that sequence – starting from an empty cache state – is exactly $\frac{1}{s}$, where s is the number of cache sets, and up to two misses otherwise.

It is therefore fair to say that there is nothing intrinsic that prevents a technique as young as SPTA from being able to solve the problem addressed by observation O3.

For the sake of comparison, it is also interesting to notice that deterministic placement has its share of weaknesses with respect to analysability. As we already noted in fact, the result of SDTA strictly and solely applies to the memory layout considered in the analysis. Any change in that, irrespective of how small, invalidates the result. Moreover, the particular memory location of objects may make accesses a and b in the previous example map to the same cache set and thus be always misses. Similarly, if the information provided to SDTA does not allow identifying whether a and b do indeed map to the same cache set, SDTA must then conservatively assume that they do, and consider all accesses as misses. It is therefore evident that SDTA may draw conclusions that are as bad as those of state-of-the-art SPTA, because of either a bad memory placement or lack of precise information.

Revisiting C4

Conclusion C4 builds on the observation that low-probability events may occur that lead to very high execution times. A low-probability event (possibly occurring, say, once in $1,000,000$ runs) is very unlikely to be observed within the few hundreds of runs typically required by MBPTA. Hence, the author of [17] maintains that MBPTA would be fooled by its very reliance on possibly very low probability of occurrence, as caused by, for instance, random placement.

At the time of Reineke's writing, conclusion C4 was a certainly valid claim. The Heart-of-Gold (HoG) technique [2], which appeared slightly later, addresses the observations that gave rise to C4. The HoG work specifically delves into the internal reasoning of MBPTA to identify why a C4-like scenario can occur and what processor resources can lead to it. In particular, the HoG authors show that the most challenging events are those whose probability of occurrence ($P_{extreme}$) is low enough for the probability of not observing any of them (P_{not_seen}) in the MBPTA runs, to be above the exceedance probability threshold set by the relevant system requirements. In the HoG work – and in much of the current PTA literature –, the assumption is made that such a threshold is related to the acceptable failure rate as sanctioned by the applicable safety-related standards for critical system components.

The work in [2] contains a detailed analysis of the processor resources that MBPTA wants randomized. The HoG method allows determining $P_{extreme}$ for the randomized resources of interest, thus identifying risky scenarios. Random placement is shown to be liable to such risks. The HoG solution to this problem is to either increase $P_{extreme}$ or the number of runs until P_{not_seen} drops

below the limit threshold. In that manner, the risky events are captured by MBPTA and the resulting WCET estimates can be regarded as trustworthy.

It can therefore be contended that conclusion C4 in [17] holds true solely for the original MBPTA method [8], but not for any MBPTA method enriched with [2].

Interestingly, a similar argument to C4 can be used to unveil a nice property of MBPTA, also highlighting a serious issue with deterministic placement policies and SDTA approaches. If an extreme pattern of memory accesses may occur that causes the cache to exhibit a pathological behavior leading to very high WCET, SDTA approaches cannot disregard that possibility, regardless of how rare it can be. With MBPTA and random placement, instead, each pathological behavior is considered only with respect to its probability of occurrence. In this light, in contrast with C4, we can conclude that random placement can in fact mitigate – as opposed to exacerbate – the effects of pathological worst-case behaviors. From an industrial perspective, the inherent robustness of MBPTA and randomized caches to pathological behavior on specific memory access patterns arguably is one of the strongest arguments in favour of probabilistic techniques.

4   Conclusions

We thoroughly enjoy the attention given to time-randomized architectures and probabilistic timing analysis in the scientific community for the assessment of what consequences they may bring to the timing analysis of hard real-time systems. We are equally interested in any critique that can be moved to those techniques as part of that scrutiny.

The work presented in [17] raised numerous issues on the adoption of randomized caches and PTA approaches. For the benefit of the interested community, this short paper places those considerations into the bigger and more complete picture where they rightly belong. To that end, we provide some additional observations and present recent research results, which collectively allow us to make the following contentions:

1. In the situations where set-associative caches and measurement-based timing analysis can be soundly used – whether by virtue of scientific authority or of social consensus among the domain stakeholders –, randomized caches together with MBPTA increase the ability to obtain trustworthy and tight WCET estimates while reducing the burden on the user.
2. In the other situations it is unclear whether deterministic analyses on top of conventional caches or probabilistic analyses on top of randomized caches are the best choice, in that the results obtained are highly dependent on the particular characteristics of the system of interest.

It can therefore be maintained that the advantages brought about by randomized caches as discussed in this paper make them a serious candidate for use with hard real-time systems in places where conventional caches exhibit limits that undermine their reputation.

References

1   Jaume Abella, Damien Hardy, Isabelle Puaut, Eduardo Quiñones, and Francisco J. Cazorla. On the comparison of deterministic and probabilistic WCET estimation techniques. In *26th Euromicro Conference on Real-Time Systems (ECRTS'14), Madrid, Spain, July 8–11, 2014*, pages 266–275. IEEE Computer Society, 2014. `doi:10.1109/ECRTS.2014.16`.

2   Jaume Abella, Eduardo Quiñones, Franck Wartel, Tullio Vardanega, and Francisco J. Cazorla. Heart of gold: Making the improbable happen to increase confidence in MBPTA. In *26th Euromicro Conference on Real-Time Systems (ECRTS'14), Madrid, Spain, July 8–11, 2014*, pages 255–265. IEEE Computer Society, 2014. `doi:10.1109/ECRTS.2014.33`.

3   Sebastian Altmeyer, Liliana Cucu-Grosjean, Robert I. Davis, and Benjamin Lesage. Progress on static probabilistic timing analysis for systems with random cache replacement policies. In *5th Real-Time Scheduling Open Problems Seminar (RTSOPS'14)*, 2014.

4   Sebastian Altmeyer and Robert I. Davis. On the correctness, optimality and precision of static probabilistic timing analysis. Technical report, Technical Report YCS-2013-487, University of York, 2014. URL: `http://www.cs.york.ac.uk/ftpdir/reports/2013/YCS/487/YCS-2013-487.pdf`.

5   Guillem Bernat, Antoine Colin, and Stefan M. Petters. WCET analysis of probabilistic hard real-time system. In *23rd IEEE Real-Time Systems Symposium (RTSS'02), Austin, Texas, USA, December 3–5, 2002*, pages 279–288. IEEE Computer Society, 2002. `doi:10.1109/REAL.2002.1181582`.

6   Francisco J. Cazorla, Eduardo Quiñones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, Emery D. Berger, Jaume Abella, Franck Wartel, Michael Houston, Luca Santinelli, Leonidas Kosmidis, Code Lo, and Dorin Maxim. PROARTIS: probabilistically analyzable real-time systems. *ACM Trans. Embedded Comput. Syst.*, 12(2s):94, 2013. `doi:10.1145/2465787.2465796`.

7   Francisco J. Cazorla, Tullio Vardanega, Eduardo Quiñones, and Jaume Abella. Upper-bounding program execution time with extreme value theory. In *13th International Workshop on Worst-Case Execution Time Analysis (WCET'13), July 9, 2013, Paris, France*, volume 30 of *Open Access Series in Informatics (OASICS)*, pages 64–76. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2013. `doi:10.4230/OASIcs.WCET.2013.64`.

8   Liliana Cucu-Grosjean, Luca Santinelli, Michael Houston, Code Lo, Tullio Vardanega, Leonidas Kosmidis, Jaume Abella, Enrico Mezzetti, Eduardo Quiñones, and Francisco J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *24th Euromicro Conference on Real-Time Systems (ECRTS'12), Pisa, Italy, July 11–13, 2012*, pages 91–101. IEEE Computer Society, 2012. `doi:10.1109/ECRTS.2012.31`.

9   Robert I. Davis, Luca Santinelli, Sebastian Altmeyer, Claire Maiza, and Liliana Cucu-Grosjean. Analysis of probabilistic cache related pre-emption delays. In *25th Euromicro Conference on Real-Time Systems (ECRTS'13), Paris, France, July 9–12, 2013*, pages 168–179. IEEE Computer Society, 2013. `doi:10.1109/ECRTS.2013.27`.

10   William Feller. *An Introduction to Probability Theory and Its Applications.* John Willer and Sons, 1996.

11   Javier Jalle, Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J. Cazorla. Bus designs for time-probabilistic multicore processors. In *Design, Automation & Test in Europe 2014 (DATE'14), Dresden, Germany, March 24–28, 2014*, pages 1–6. IEEE, 2014. `doi:10.7873/DATE.2014.063`.

12   Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J. Cazorla. A cache design for probabilistically analysable real-time systems. In *Design, Automation & Test in Europe 2013 (DATE'13), Grenoble, France, March 18–22, 2013*, pages 513–518. EDA Consortium San Jose, CA, USA / ACM DL, 2013. URL: `http://dl.acm.org/citation.cfm?id=2485416`.

13   Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J. Cazorla. Multi-level unified caches for probabilistically time analysable real-time systems. In *34th IEEE Real-Time Systems Symposium (RTSS'13), Vancouver, BC, Canada, December 3–6, 2013*, pages 360–371. IEEE Computer Society, 2013. `doi:10.1109/RTSS.2013.43`.

14   Leonidas Kosmidis, Jaume Abella, Franck Wartel, Eduardo Quiñones, Antoine Colin, and Francisco J. Cazorla. PUB: path upper-bounding for measurement-based probabilistic timing analysis. In *26th Euromicro Conference on Real-Time Systems (ECRTS'14), Madrid, Spain, July 8–11, 2014*, pages 276–287. IEEE Computer Society, 2014. `doi:10.1109/ECRTS.2014.34`.

15   Leonidas Kosmidis, Eduardo Quiñones, Jaume Abella, Tullio Vardanega, Ian Broster, and Francisco J. Cazorla. Measurement-based probabilistic timing analysis and its impact on processor architecture. In *17th Euromicro Conference on Digital System Design (DSD'14), Verona, Italy, August 27–29, 2014*, pages 401–410. IEEE Computer Society, 2014. `doi:10.1109/DSD.2014.50`.

16   Eduardo Quiñones, Emery D. Berger, Guillem Bernat, and Francisco J. Cazorla. Using randomized caches in probabilistic real-time systems. In *21st Euromicro Conference on Real-Time Systems (ECRTS'09), Dublin, Ireland, July 1–3, 2009*, pages 129–138. IEEE Computer Society, 2009. `doi:10.1109/ECRTS.2009.30`.

17   Jan Reineke. Randomized caches considered harmful in hard real-time systems. *Leibniz Transactions on Embedded Systems (LITES)*, 1(1):03:1–03:13, 2014. `doi:10.4230/LITES-v001-i001-a003`.

18   Mladen Slijepcevic, Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J. Cazorla. DTM: degraded test mode for fault-aware probabilistic timing analysis. In *25th Euromicro Conference on Real-Time Systems (ECRTS'13), Paris, France, July 9–12, 2013*, pages 237–248.

IEEE Computer Society, 2013. `doi:10.1109/ECRTS.2013.33`.

19 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), 2008. `doi:10.1145/1347375.1347389`.

20 Shuchang Zhou. An efficient simulation algorithm for cache of random replacement policy. In *7th IFIP International Conference on Network and Parallel Computing (NPC'10), Zhengzhou, China, September 13–15, 2010*, volume 6289 of *Lecture Notes in Computer Science*, pages 144–154. Springer, 2010. `doi:10.1007/978-3-642-15672-4_13`.

More General Optimal Offset Assignment

Sven Mallach

Universität zu Köln
Albertus-Magnus-Platz, 50923 Köln, Germany
`mallach@informatik.uni-koeln.de`

─── **Abstract** ───

This manuscript presents exact approaches to the general offset assignment problem arising in the address code generation phase of compilers for application-specific processors. First, integer programming models for architecture-dependent and theoretically motivated special cases of the problem are established. Then, these models are extended to provide the first widely applicable formulations for the most general problem setting, supporting processors with several address registers and complex addressing capabilities. Existing heuristics are similarly extended and practical applicability of the proposed methods is demonstrated by experimental evaluation using an established and large benchmark set. The experiments allow us to study the impact of exploiting more complex memory addressing capabilities on the address computation costs of real-world programs. We also show how to integrate operand reordering techniques for commutative instructions into existing solution approaches.

1 Introduction

We study the offset assignment problem arising in the address code generation phase of compilers for digital signal and other application-specific processors. To save silicon area, such processor designs often have narrow instruction widths limiting the number of bits available for memory addressing. Typically, there is no support for indirect addressing modes that combine a base address held in a register with an immediate offset to build the effective address of a memory operand (sometimes called *base plus offset* addressing). However, specialized Harvard architectures often provide an address generation unit (AGU) supporting pointer arithmetic to be done in parallel to the main data path. The additional hardware can help to at least partially compensate the drawbacks if exploited properly. It supports instructions that permit to manipulate an address register (AR) in the same clock cycle as another instruction referencing it. Either, the respective modifications are encoded implicitly (effectively moving the encoding of the offset into the instruction opcode) or the instructions permit to add (subtract) values within a small architecture-dependent *auto-modify range* $[-r, r]$ to (from) the address held in the AR [14]. AR modifications by absolute values larger than r however cannot be handled like this and need additional explicit address arithmetic instructions.

With such specialized instructions at hand, one can consider the complexity of indirect addressing to have been moved from hardware to software, relying on compilers in order to exploit the processor's capabilities for fast memory addressing of statically allocated variables. However,
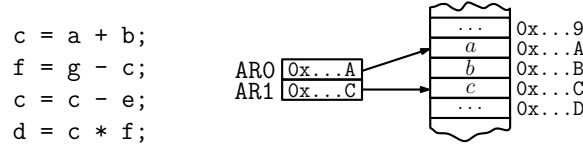
```
c = a + b;
f = g - c;
c = c - e;
d = c * f;
```

Figure 1 A sample code fragment and an illustration of ARs referencing memory locations of variables.

optimal exploitation asks for the optimal solution of two interdependent problems, namely to determine a stack memory layout of the variables accessed and to select an address register responsible for each of the accesses. Conversely, address computation overhead may result from two main issues. An inappropriate storage layout may necessitate additional explicit address arithmetic instructions for 'jumps' to addresses that have a distance larger than r. Further, if the processor provides multiple ARs, a poor choice of the ARs responsible for particular accesses may result in superfluous immediate AR loads and unnecessary 'jumps' as well. Since address calculations make up a significant part of machine instructions, optimizing these decisions may considerably reduce the code size and speed up the program at the same time. Indeed, experimental studies, e.g. [9, 11, 14], show that optimized configurations lead to significant savings in practice.

During compilation, the instruction scheduling phase determines the *access sequence* to local stack variables. It can be extracted by simply concatenating the referenced variables of each three-address-code instruction $c = a\ op\ b$ in the order $a\ b\ c$. For example, the program fragment in the left of Fig. 1 refers to the variables $\mathcal{V} = \{a, b, c, d, e, f, g\}$ that are accessed in the order $S = a\ b\ c\ g\ c\ f\ c\ e\ c\ c\ f\ d$. Tab. 1 shows pseudo machine code for this program fragment and three potential stack layouts A, B, and C of $\mathcal{V}$. Layout A complies with the order of first use of the variables in S. On a processor with only a single AR and with autoin-/decrement capability ($r = 1$) only, this layout would require six explicit address arithmetic instructions (`ADAR` and `SBAR`). An optimized layout (B) already reduces the necessary number of such instructions to three by increasing the use of autoin-/decrement instructions (with `*(ARx)+`/`*(ARx)-`). If the memory layout is optimized for a use of two ARs (C) and an optimal AR assignment is computed as well, it becomes possible to cover the access sequence even without any explicit address arithmetic at all. Assuming the cost of an immediate AR load and the cost of an address arithmetic instruction to be both one, the optimal total cost with one AR is four, with two ARs it is two. Notably, layout A and B have no register assignment that leads to a total cost smaller than three with two or more ARs.

2   Related Work and Contribution

The *general offset assignment* (GOA) problem is defined for processors with k address registers and an auto-modify range of r. Given an access sequence $S = \{s_1, s_2, \ldots, s_m\}$ of program variables $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$, it asks for a stack memory layout of the variables, i.e., a permutation $\pi : \mathcal{V} \rightarrow \{1, \ldots, n\}$, and an assignment $\gamma : S \rightarrow \{1, \ldots, k\}$ of accesses to address registers exploiting the auto-modify range r such that the number of accesses requiring extra address arithmetic instructions is minimum.

Most of the literature considers special cases of the problem, where either $r = 1$, $k = 1$ or both. For $k = 1$, the problem is called the *simple offset assignment (SOA)* problem. It reduces to the task to find a stack memory layout that allows as many accesses as possible to be performed by auto-modify instructions on the single available address register. It was first considered by

■ **Table 1** Pseudo machine codes for the code fragment from Fig. 1 assuming different memory layouts A, B and C and either one (A and B) or two (C) available address registers.

```
Instruction    AR0        Instruction    AR0        Instruction    AR0  AR1
LDAR  AR0, &a   &a        LDAR  AR0, &a   &a        LDAR  AR0, &a   &a
LOAD  *(AR0)+   &b        LOAD  *(AR0)+   &b        LOAD  *(AR0)+   &b
ADD   *(AR0)+   &c        ADD   *(AR0)              ADD   *(AR0)+   &c
STOR  *(AR0)+   &g        ADAR  AR0,2     &c        STOR  *(AR0)
LOAD  *(AR0)-   &c        STOR  *(AR0)-   &g        LDAR  AR1, &g        &g
SUB   *(AR0)              LOAD  *(AR0)+   &c        LOAD  *(AR1)-        &e
ADAR  AR0,2     &f        SUB   *(AR0)+   &f        SUB   *(AR0)+   &f
STOR  *(AR0)              STOR  *(AR0)-   &c        STOR  *(AR0)-   &c
SBAR  AR0,2     &c        LOAD  *(AR0)              LOAD  *(AR0)
LOAD  *(AR0)              ADAR  AR0,3     &e        SUB   *(AR1)
ADAR  AR0,3     &e        SUB   *(AR0)              STOR  *(AR0)+   &f
SUB   *(AR0)              SBAR  AR0,3     &c        MUL   *(AR0)+   &d
SBAR  AR0,3     &c        STOR  *(AR0)+   &f        STOR  *(AR0)
STOR  *(AR0)              MUL   *(AR0)+   &d
ADAR  AR0,2     &f        STOR  *(AR0)
MUL   *(AR0)
ADAR  AR0,2     &d
STOR  *(AR0)
```

$A = a\ b\ c\ g\ f\ e\ d$ $B = a\ b\ g\ c\ f\ d\ e$ $C = a\ b\ c\ f\ d\ e\ g$
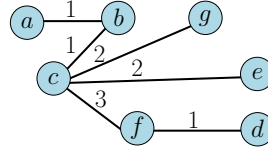

■ **Figure 2** Access graph for the code fragment from Fig. 1.

Bartley [2] in 1992. Assuming also $r = 1$, he proposed to model the variable relationships contained in an access sequence by an *access graph* $G = (V, E)$. The set of vertices V corresponds to the variables and there is an edge $e = \{u, v\} \in E$ with weight $w(e)$ if the variables u and v appear subsequently in the access sequence for $w(e)$ times. Fig. 2 shows the access graph that corresponds to the sample code fragment from Fig. 1.

Bartley recognized a close relationship of SOA (with $r = 1$) to the maximum-weight Hamiltonian path problem and developed a first greedy heuristic to solve it. In subsequent research, Liao [17] showed that this problem is equivalent to a maximum-weight path cover problem instead and gave a formal proof of its strong $\mathcal{NP}$-hardness. Based on these results, he proposed a simpler and faster heuristic producing solutions with the same quality as Bartley's and also a first exact Branch-and-Bound procedure. In the sequel, a number of heuristics and as well an exact approach capable to solve larger instances have been developed and also been subject to experimental studies [9, 11, 14].

Since 2014, the situation is similar for the general variant with $k \geq 1$ and $r = 1$. Using the exact approach presented in [18], several GOA heuristics [24, 16, 15] could be evaluated in terms of quality relative to the optimum for the first time. In practice, GOA is often solved heuristically by first partitioning the set of program variables w.r.t. the available ARs and then solving a SOA problem for each of the ARs. This strategy allows to reuse available SOA algorithms but inherently constrains all accesses to a particular variable to be carried out by the same AR. This may preclude optimal results as is comprehensively discussed by Huynh et al. [9]. Even more, the results in [18] suggest not to partition the variables a priori, but to first compute a memory layout for them and an address register assignment afterwards.

In this paper, we highlight the key ideas and techniques that let the exact GOA approach for
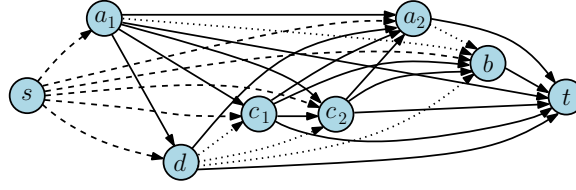
Figure 3 Minimum cost flow network for $S = a\ d\ c\ c\ a\ b$ assuming $\mathcal{L} = d$ - a - c - b.

$k \geq 1$ and $r = 1$ presented in [18] be successful in solving a wide range of instances to optimality in reasonable time. The mentioned article also contains a correction of the only previous exact approach by Ozturk et al. [20] that was originally designed for arbitrary ranges r. However, as the experiments in [18] show, the method is not capable to solve larger instances, even for $r = 1$. It suffers from a quickly growing number of variables and constraints and does neither exploit the combinatorial structure nor symmetries inherent to the problem. Here, we develop an alternative formulation that takes advantages of the techniques presented in [18], yet generalizing to arbitrary auto-modify ranges and still remaining relatively moderate in size. Using this approach and by extending existing heuristics, we provide the first experiments for ranges $r > 1$ on a larger set of instances and study the effect of exploiting larger auto-modify capabilities on the total address computation costs. Further, we address a commonly observed criticism associated with GOA, namely that it is not clear how it relates to operand reordering techniques such as, e.g, [23, 1, 4], which may also result in reduced address computation overhead. We present a method to integrate commutativity-based operand reordering into the address register assignment part of the optimization process. Combined with our optimal approaches for GOA, this allows to create globally optimal address code.

3 Optimal Address Register Assignment

3.1 A Minimum-Cost Flow Model

Suppose for now that a memory layout $\mathcal{L}$ of the program variables $\mathcal{V}$ has already been fixed and we are now asked to compute an optimal address register assignment (ARA) for k address registers w.r.t. $\mathcal{L}$ and the input access sequence S. For $r = 1$, an exact solution to this problem has been proposed by Gebotys [7, 8]. It is based on a minimum-cost circulation network that contains a vertex for each access in S and a directed arc for each pair of accesses u, v such that v succeeds u in S. In [18], it is shown how the circulation problem can be transformed into an equivalent minimum cost flow problem. We will now describe the corresponding flow network $N = (V_N, A)$. We denote with V_S an ordered set of vertices that is associated with the accesses in sequence S, i.e., if $u < v$ for $u, v \in V_S$, then u is accessed prior to v in S. The vertex set of V_N of the network N is given by V_S and artificial source (s) and sink (t) vertices, i.e., $V_N = V_S \cup \{s, t\}$. The flow arc set A is composed as the union of the arc sets $\{(s, v) \mid v \in V_S\}$, $\{(v, w) \mid v, w \in V_S, v < w\}$, and $\{(v, t) \mid v \in V_S\}$. So in essence, each of the access vertices has an entering arc from the source s and a leaving arc to the sink t. Further, there is an arc $(u, v) \in A$ between two access vertices $u, v \in V_S$ if and only if u precedes v in the access sequence.

The total flow leaving the source is restricted to be at least one unit and at most k units because at least one register must be used and at most k ones can be used in order to cover the accesses. All arcs have capacity one and each sequence vertex is constrained to receive and supply exactly one unit of flow. Hence, each unit of flow leaving s essentially delivers a path of accesses before it proceeds to t. The cost of an arc between two accesses is zero if and only if the two
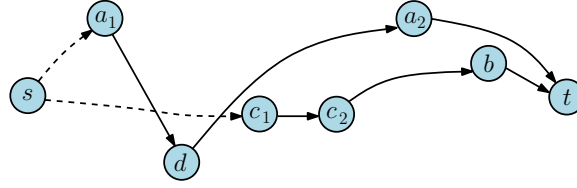
Figure 4 An optimal solution to the example from Fig. 3.

associated variables are either equal, or adjacent in $\mathcal{L}$. Otherwise the arc has cost coefficient c_A that shall reflect the costs of of an address arithmetic instruction. Finally, the cost associated to an immediate AR load is installed on every s-leaving arc and the costs of all arcs entering t are zero. So if the selection of paths through the network is based on a min-cost criterion, then each of the resulting paths can be interpreted as an optimal series of accesses performed using a distinct AR.

As a small example, let $\mathcal{V} = \{a, b, c, d\}$, $S = a\ d\ c\ c\ a\ b$ and assume $\mathcal{L} = d$ - a - c - b (which is optimal for $k \geq 2$ ARs). The associated minimum cost flow network looks as depicted in Fig. 3. Zero-cost arcs are drawn solid, access transitions with cost c_A are shown as dotted arcs and the s-leaving arcs with cost c_L are illustrated using dashes. An optimal solution (assuming $c_L = c_A$, and $k \geq 2$) to this example is depicted in Fig. 4.

Let $y_{u,v}$ be a flow variable for each arc $(u, v) \in A$ and $c_{u,v}$ its associated cost. The LP formulation corresponding to the described min-cost flow problem is then:

$$\min \sum_{(u,v) \in A} c_{u,v} y_{u,v}$$

$$\text{s.t.} \sum_{(v,w) \in A} y_{v,w} = 1 \qquad \text{for all } v \in V_S \qquad (1)$$

$$\sum_{(u,v) \in A} y_{u,v} = 1 \qquad \text{for all } v \in V_S \qquad (2)$$

$$\sum_{v \in V_S} y_{s,v} \leq k$$

$$y_{u,v} \geq 0 \qquad \text{for all } (u, v) \in A$$

$$y_{u,v} \leq 1 \qquad \text{for all } (u, v) \in A$$

The restriction of the in- and out-degrees of all sequence-vertices to one by constraints (1) and (2) highlights the aforementioned 'path selection' property of this model. Actually, they make the usual flow conservation constraints of network flow problems obsolete; any unit of flow sent from s to satisfy the equations must finally arrive at t. Further, lower bounds on the flow on out- (in-) arcs of the source (sink) are not necessary since the former (latter) must be satisfied due to the in- (out-) degree equation of the first (last) access vertex. Stated as an LP, the model permits all variables to take values from the interval $[0, 1]$. However, the constraint matrix that arises from the described LP formulation is unimodular [7, 13] and, hence, optimal LP solutions will always be integral. Besides that, combinatorial minimum-cost flow algorithms can be used in order to obtain integral solutions in polynomial time.

3.2 Taking Commutativity Into Account

Sometimes address computation overhead can also be avoided by simply reordering the accesses to operands where this is possible due to the commutativity of the respective operations. For example, assume that variable b was last accessed and `c = a + b` is the next instruction. Then it is typically beneficial to access b first if a and b are not within range r in the stack layout.
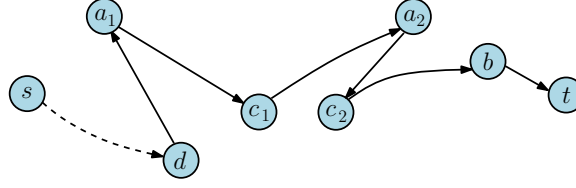
■ **Figure 5** An optimal solution to the example from Fig. 3 exploiting commutativity.

Commutativity-based reordering opportunities can be incorporated into the original approach by Gebotys and as well into the just described flow model. If u and v are the two respective operands belonging to a commutative instruction, one can replace the corresponding flow arc $(u, v) \in A$ by a flow *edge* $\{u, v\}$ that is undirected and can hence be used in both directions. Since we may reasonably assume to have three-address-code instructions only, the model guarantees that each vertex of the network has at most one neighbor that is adjacent by an edge rather than by an arc. At these particular vertex pairs, the flow in the network is then permitted to also move 'backward' while the constraints that each vertex must have exactly one incoming and outgoing unit of flow will preserve the overall feasibility of the problem and correctness of the solutions.

Suppose the access sequence $S = a\ d\ c\ c\ a\ b$ from the previous subsection is stemming from the computations `c = a * d; b = c * a`. Then both operations are commutative and the arcs (a_1, d) and (c_2, a_2) in the flow network become edges in the proposed methodology. Indeed, sending flow 'backward' along these edges allows for a better solution with cost only one as is depicted in Fig. 5.

4 Optimal General Offset Assignment

To solve GOA to global optimality, we need to solve the two interdependent problems in an integrated fashion, i.e., we need to find a memory layout that will allow us to create the best possible address register assignment.

A key observation is that the objective function is the only point where the memory layout influences the concrete ARA network problem to be solved. The cost of an access transition (u, v) in the network described in Sect. 3 is zero if and only if the variables associated with u and v are equal or neighbors in the memory layout. Otherwise, a positive cost c_A reflecting the overhead of an additional address arithmetic instruction is assigned. Moreover, there is no reason to not redefine this rule for $r > 1$, i.e., to assign arcs $(u, v) \in A$ the cost zero if u and v are no more than r positions apart from each other and cost c_A otherwise.

However, in terms of modeling the feasible solutions of GOA problems, it makes a considerable difference whether r is equal to one or may be larger. We will now first discuss the approach for $r = 1$ from [18] and then proceed to the more general case.

4.1 GOA with $r = 1$

If $r = 1$, then an access transition can only be realized without extra address arithmetic if the two successively referenced variables are either identical or neighbors in the memory layout.

Clearly, a memory layout is basically a permutation of the variables. As already indicated in Sect. 2, possible permutations of the variables $\mathcal{V}$ can, e.g., be modeled by the Hamiltonian paths of a complete undirected graph G with vertex set $\mathcal{V}$. From an integer programming point of view, it is however more suitable to model and compute Hamiltonian cycles instead of paths [11, 18]. So we construct a complete graph $G' = (V, E)$ with $V = \mathcal{V} \cup \{z\}$ where z is an artificial vertex. It
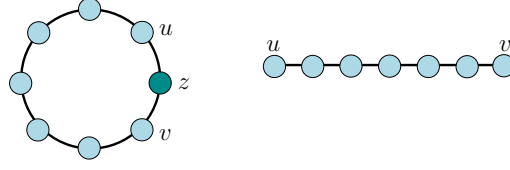
Figure 6 Extracting a Hamiltonian path from a Hamiltonian cycle.

serves as a cutting point in order to obtain a (up to reversion) unique Hamiltonian path from each Hamiltonian cycle in G' as is truly intended (cf. Fig. 6).

To model GOA completely, we will now always consider two graphs. Firstly, a complete graph $G = (V, E)$ where $V = \mathcal{V} \cup \{z\}$ as just described. Secondly, we have a network $N = (V_N, A)$ with $V_N = V_S \cup \{s, t\}$ where V_S is a vertex set related to the accesses contained in the input sequence S, just like in Sect. 3. Let $A_S = \{(v, w) \mid v, w \in V_S, \ v < w\}$ and $A = A_S \cup \{(s, v) \mid v \in V_S\} \cup \{(v, t) \mid v \in V_S\}$. Since all access vertices in V_S reflect instances of the program variables represented by the vertices $\mathcal{V}$, we may define a corresponding unique mapping $\sigma : V_S \to \mathcal{V}$. For ease of reference, we further split the set A_S into $A_S^{\neq} = \{(u, v) \in A_S, \ \sigma(u) \neq \sigma(v)\}$, i.e., the set of arcs between accesses that do not refer to the same associated program variable and, analogously, the set $A_S^{=}$.

In addition to the flow arc variables $y_{u,v}$ for each arc $(u, v) \in A$, we associate edge decision variables $x_{u,v} \in \{0, 1\}$ with the edges $\{u, v\} \in E$ that have no associated costs. The variable $x_{u,v}$ is equal to one if the edge $\{u, v\}$ is part of the computed Hamiltonian cycle (u and v are neighbors in the memory layout), and zero otherwise. Since G is undirected, the variables $x_{u,v}$ are only defined for $u < v$. Slightly disregarding mathematical precision, we write $x_{\sigma(u),\sigma(v)}$ when referring to the associated edge decision variable of $y_{u,v}, (u, v) \in A_S^{\neq}$, irrespective of whether $\sigma(u) < \sigma(v)$ or $\sigma(u) > \sigma(v)$. Exploiting these variable relationships, we can express the cost of an access transition $y_{u,v}$ with $(u, v) \in A_S^{\neq}$ by $(1 - x_{\sigma(u),\sigma(v)})c_A$ while the cost of each variable $y_{u,v}$ for $(u, v) \in A_S^{=}$ is zero. This leads to a first quadratic integer programming formulation.

4.1.1 Quadratic Formulation

$$
\min \sum_{(u,v) \in A_S^{\neq}} (1 - x_{\sigma(u),\sigma(v)})c_A y_{u,v} + \sum_{v \in V_S} c_L y_{s,v}
$$

$$
\begin{array}{llll}
\text{s.t.} & \displaystyle\sum_{\{u,v\} \in E} x_{u,v} & = 2 & \text{for all } v \in V & (3) \\[2ex]
& x(E(W)) & \leq |W| - 1 & \text{for all } W \subsetneq V, \ |W| \geq 2 & (4) \\[2ex]
& \displaystyle\sum_{(u,v) \in A} y_{u,v} & = 1 & \text{for all } u \in V_S & \\[2ex]
& \displaystyle\sum_{(u,v) \in A} y_{u,v} & = 1 & \text{for all } v \in V_S & \\[2ex]
& \displaystyle\sum_{v \in V_S} y_{s,v} & \leq k & & \\[2ex]
& x_{u,v} & \in \{0, 1\} & \text{for all } (u, v) \in E & \\[1ex]
& y_{u,v} & \in \{0, 1\} & \text{for all } (u, v) \in A &
\end{array}
$$

This integer program is essentially the min-cost flow formulation from Sect. 3 appended by inequalities enforcing the x-variables to correspond to a Hamiltonian cycle of G and with a new objective function linking the two subproblems. The objective function simply sums up the terms $(1 - x_{\sigma(u),\sigma(v)})c_A$ for all arcs $(u, v) \in A_S^{\neq}$ and all costs $\sum_{v \in V_S} c_L y_{s,v}$ for initial address register loads.

The set of feasible Hamiltonian cycles of G can be expressed using the standard formulation of the traveling salesman problem (TSP): The equations (3) force any vertex to be adjacent to exactly two other vertices. Inequalities (4) are called *subtour elimination constraints* [5]. Here, $E(W) = \{\{u, v\} \in E \mid u, v \in W\}$ for any $W \subseteq V$ such that the inequalities exclude solutions containing any cycle w.r.t. the vertex sets W and $V \setminus W$ from the feasible set.

The above integer program is quadratic in its objective function only. We may linearize it using a standard linearization approach [19]. However, first we simplify. The term

$$\min \sum_{(u,v) \in A_S^{\neq}} (1 - x_{\sigma(u),\sigma(v)}) c_A y_{u,v}$$

can also be written as

$$\min \Big(\sum_{(u,v) \in A_S^{\neq}} y_{u,v} - \sum_{(u,v) \in A_S^{\neq}} x_{\sigma(u),\sigma(v)} y_{u,v} \Big) c_A.$$

We then need $|A_S^{\neq}|$ new variables $z_{u,v}$ to model the products $x_{\sigma(u),\sigma(v)} y_{u,v}$ and three linearization constraints for each of the new variables:

$$
\begin{aligned}
z_{u,v} &\leq x_{\sigma(u),\sigma(v)} \\
z_{u,v} &\leq y_{u,v} \\
z_{u,v} &\geq x_{\sigma(u),\sigma(v)} + y_{u,v} - 1
\end{aligned}
$$

After this transformation, the objective function becomes:

$$\min \sum_{(u,v) \in A_S^{\neq}} c_A y_{u,v} - \sum_{(u,v) \in A_S^{\neq}} c_A z_{u,v} + \sum_{v \in V_S} c_L y_{s,v}$$

Clearly, the number of product variables to be introduced, $|A_S^{\neq}|$, must be strictly smaller than the total number of flow arc variables which is $(|S| + 2) \cdot (|S| + 1)/2$. Hence, in total the linearized version of the above formulation has strictly less than $|\mathcal{V}| \cdot (|\mathcal{V}| - 1)/2 + ((|S| + 2) \cdot (|S| + 1)/2)$, i.e., $\mathcal{O}(|\mathcal{V}|^2 + |S|^2)$ variables. The number of subtour elimination constraints is exponential in $|\mathcal{V}|$ such that it is preferable to not consider all of them in the solution process from the beginning, but to separate them instead as we will describe more detailed in Sect. 5.1. The number of remaining nontrivial constraints is strictly less than $1 + |\mathcal{V}| + 2(|S| + 2) + 3((|S| + 2) \cdot (|S| + 1)/2)$, i.e., $\mathcal{O}(|\mathcal{V}| + |S|^2)$. Remarkably, all these numbers are independent from the number k of ARs available. The more access pairs in S refer to the same variable, the less product variables and associated constraints are needed.

4.1.2   Linear Formulation

By further inspection and by exploiting the fact that there are only two cases for each arc $(u, v) \in A_S^{\neq}$, namely that it either has the assigned cost c_A or assigned cost zero, an inherently linear formulation of the problem can be established. The main idea is to replace every variable (arc) between two accesses $y_{u,v}, (u, v) \in A_S^{\neq}$, by two new variables (arcs) $y_{u,v}^0$ (for the zero-cost case) and $y_{u,v}^c$ (for the arc with cost c_A) as is depicted in Fig. 7. The set $A_S^{\neq}$ is therefore further split into the corresponding new arc sets A_S^0 and A_S^c. For every arc $(u, v) \in A_S^{=}$, we keep the former variable $y_{u,v}$ with zero cost as before. We also skip the superscript when referring to flow variables disregarding their costs or if only one instance exists.

The new network N has now the arc set $A = A_S^0 \cup A_S^c \cup A_S^{=} \cup \{(s, v) \mid v \in V_S\} \cup \{(v, t) \mid v \in V_S\}$ and the new objective is to minimize the selected arcs with positive costs assigned. With the new variables, this is a linear expression. However, we now have to restrict the use of zero-cost
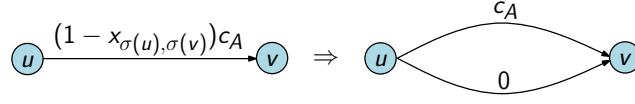
Figure 7 Replacing arcs with dynamic costs by two arcs with static costs.

arcs. As before in the quadratic model, it should only be possible to use them if the respective corresponding variables are neighbors in the access sequence. With the newly introduced variables this can easily be enforced using the following constraints:

$$y^0_{u,v} \leq x_{\sigma(u)\sigma(v)} \quad \text{for all } (u,v) \in A^0_S$$

Further, the following constraint is valid for the model:

$$y^c_{u,v} \leq 1 - x_{\sigma(u)\sigma(v)} \quad \text{for all } (u,v) \in A^c_S$$

However, since zero-cost arcs are preferred by the objective function, there will never be a variable $y^c_{u,v} = 1$ in an optimum solution where also $x_{\sigma(u)\sigma(v)} = 1$, even if these constraints are not present. Therefore, they will also have only marginal impact on the solution process and can be omitted. A complete linear IP formulation for GOA with $r = 1$ is then:

$$
\begin{aligned}
\min \quad & \sum_{(u,v)\in A^c_S} c_A y^c_{u,v} + \sum_{v\in V_S} c_L y_{s,v} \\
\text{s.t.} \quad & \sum_{\{u,v\}\in E} x_{u,v} = 2 && \text{for all } v \in V \\
& x(E(W)) \leq |W| - 1 && \text{for all } W \subsetneq V,\ |W| \geq 2 \\
& \sum_{(u,v)\in A} y_{u,v} = 1 && \text{for all } u \in V_S \\
& \sum_{(u,v)\in A} y_{u,v} = 1 && \text{for all } v \in V_S \\
& \sum_{v\in V_S} y_{s,v} \leq k && \\
& y^0_{u,v} \leq x_{\sigma(u)\sigma(v)} && \text{for all } (u,v) \in A^0_S \\
& x_{u,v} \in \{0,1\} && \text{for all } (u,v) \in E \\
& y_{u,v} \in \{0,1\} && \text{for all } (u,v) \in A
\end{aligned}
$$

The number of variables is the same as in the quadratic formulation since essentially, for each $(u,v) \in A^{\neq}_S$, the product variable $z_{u,v}$ is replaced by a second flow arc variable $y^c_{u,v}$. However, $2|A^{\neq}_S|$ less constraints are needed.

4.2 GOA with $r \geq 1$

If the processor supports offset ranges r strictly larger than one, the mathematical modeling of the associated optimization problem becomes much more complicated. Simple vertex adjacencies are not anymore sufficient in order to decide on the costs of access transitions. Rather, the auto-modify instructions can be applied as soon as there is a *path* from u to v of length smaller or equal to r in a memory layout. This cannot be modeled by linear expressions in the x-variables of the preceding subsection. Also, to assign weights to paths instead of edges and to then select paths based on these weights is not a promising strategy, especially since the possible number of paths to consider may be exponential in the number of variables.

The only straightforward way to model the problem correctly appears to be via using variables that encode the position of the program variables explicitly. This can be done based on the famous *assignment problem* [3], in a similar way as it was also proposed by Wess and Gotschlich [25] for SOA, i.e., $k = 1$. The GOA model by Ozturk et al. [20] has also an assignment-based character. However, as discussed in [18], their formulation was flawed and is not applicable to a wider range of instances. We will now develop a different formulation that has considerably less variables and constraints while preserving the advantages of our model for $r = 1$.

In the assignment problem, we have variables $x_{i,p}$ that take value one if item i is placed at position p and zero otherwise. To model the stack memory layout for $n = |\mathcal{V}|$ variables, we need exactly n^2 variables, since any variable may be placed at any position $p \in P, P = \{1, \ldots, n\}$. Clearly, every variable $v \in \mathcal{V}$ must be assigned exactly one position $p \in P$ and each position $p \in P$ must be assigned exactly one variable $v \in \mathcal{V}$. Hence, the corresponding constraints of the assignment problem are:

$$\sum_{p \in P} x_{v,p} = 1 \quad \text{for all } v \in \mathcal{V}$$
$$\sum_{v \in \mathcal{V}} x_{v,p} = 1 \quad \text{for all } p \in P$$

Let $x_{u,a} = 1$ and $x_{v,b} = 1$ with $u \neq v$ and $a \neq b$. Then, an access transition $u \to v$ has cost zero if and only if $|b - a| \leq r$. For the following discussion, we introduce auxiliary variables $r_{u,v}$ for each pair of different variables $u, v \in \mathcal{V}$, expressing whether u and v are placed within range r ($r_{u,v} = 1$) or not ($r_{u,v} = 0$). We will however not need these variables for the subsequently developed integer program. With the auxiliary variables at hand, we may express the following constraints:

$$r_{u,v} \geq x_{u,b} + x_{v,a} - 1 \quad \text{for all } u < v \in \mathcal{V} \text{ and } a < b \text{ s.t. } b - a \leq r$$
$$r_{u,v} \geq x_{u,a} + x_{v,b} - 1 \quad \text{for all } u < v \in \mathcal{V} \text{ and } a < b \text{ s.t. } b - a \leq r$$

The constraints force $r_{u,v}$ to become one as soon as the assignment of positions to u and v is such that their distance is at most r. Further, they never enforce $r_{u,v}$ to be greater than one. We may also directly combine the constraints as follows:

$$r_{u,v} \geq \underbrace{x_{u,b} + x_{u,a}}_{\leq 1} + \underbrace{x_{v,a} + x_{v,b}}_{\leq 1} - 1 \quad \text{for all } u < v \in \mathcal{V} \text{ and } a < b \text{ s.t. } b - a \leq r \quad (5)$$

On the other hand, we must make sure that $r_{u,v}$ is never assigned value one if the two variables are not placed within range r using the following constraints:

$$r_{u,v} \leq 2 - x_{u,b} - x_{u,a} - x_{v,a} - x_{v,b} \quad \text{for all } u < v \in \mathcal{V} \text{ and } a < b \text{ s.t. } b - a > r \quad (6)$$

In total, this would already amount to $\mathcal{O}(n^4)$ constraints, having not yet formulated constraints that restrict the use of the zero-cost flow arc variables. However, we can still improve on that. First of all, for the same reason as in Sect. 4.1.2, we do not need to care about the positive cases that permit to use a flow arc variable $y_{u,v}^0$ but only forbid those cases where the use of $y_{u,v}^0$ is prohibited. Hence, we can completely omit the constraints (5). Now, we take a closer look on constraints (6) again, keeping in mind that each variable $v \in \mathcal{V}$ can be assigned at most one position from any strict subset Q of P, i.e., $\sum_{p \in Q} x_{v,p} \leq 1$ for all $v \in \mathcal{V}$ and $Q \subset P$. Say variable u is fixed at position a, then all the positions of v that make the transition $u \to v$ have non-zero cost are the positions $p \in [1, a - r - 1]$ and $p \in [a + r + 1, n]$. Hence, by fixing one position, we

can reformulate (6) by:

$$r_{u,v} \leq 2 - x_{u,a} - \underbrace{\sum_{p=1}^{a-r-1} x_{v,p} - \sum_{p=a+r+1}^{n} x_{v,p}}_{\leq 1} \qquad \text{for all } u < v \in \mathcal{V} \text{ and } a \in P$$

We now exactly characterized under which conditions two variables u and v are not within range r, so we can again exploit this to apply the correct restriction on the use of each zero-cost flow arc variable $y_{u,v}^0$.

$$y_{u,v}^0 \leq 2 - x_{u,a} - \sum_{p=1}^{a-r-1} x_{v,p} - \sum_{p=a+r+1}^{n} x_{v,p} \qquad \text{for all } (u,v) \in A_S^0 \text{ and } a \in P \quad (7)$$

Since, for any of the at most $\binom{|S|}{2}$ variables $y_{u,v}^0$, there are at most $|\mathcal{V}|$ positions where u can be fixed at, we obtain only $\mathcal{O}(|S|^2 \cdot |\mathcal{V}|)$ constraints (7) in total. The full IP formulation is then:

$$
\begin{aligned}
\min \quad & \sum_{(u,v) \in A_S^c} c_A y_{u,v}^c + \sum_{v \in V_S} c_L y_{s,v} && \\
\text{s.t.} \quad & \sum_{p \in P} x_{v,p} = 1 && \text{for all } v \in \mathcal{V} \\
& \sum_{v \in \mathcal{V}} x_{v,p} = 1 && \text{for all } p \in P \\
& \sum_{(u,v) \in A} y_{u,v} = 1 && \text{for all } u \in V_S \\
& \sum_{(u,v) \in A} y_{u,v} = 1 && \text{for all } v \in V_S \\
& \sum_{v \in V_S} y_{s,v} \leq k && \\
& y_{u,v}^0 \leq 2 - x_{u,a} - \sum_{p=1}^{a-r-1} x_{v,p} - \sum_{p=a+r+1}^{n} x_{v,p} && \text{for all } (u,v) \in A_S^0 \text{ and } a \in P \\
& x_{v,p} \in \{0,1\} && \text{for all } v \in \mathcal{V} \text{ and } p \in P \\
& y_{u,v} \in \{0,1\} && \text{for all } (u,v) \in A
\end{aligned}
$$

Being more precise, the model has $|\mathcal{V}|^2 + ((|S|+2) \cdot (|S|+1)/2)$, i.e., again $\mathcal{O}(|\mathcal{V}|^2 + |S|^2)$ variables. The number of nontrivial constraints is bounded from above by $1 + 2|\mathcal{V}| + 2(|S|+2) + ((|S|+2) \cdot (|S|+1)/2) \cdot |\mathcal{V}|$, i.e., $\mathcal{O}(|\mathcal{V}| \cdot |S|^2)$. In contrast to that, the formulation by Ozturk et al. has $\mathcal{O}(k \cdot |\mathcal{V}| \cdot |S| + |\mathcal{V}|^2)$ variables and $\mathcal{O}(|\mathcal{V}|^3 + k \cdot |S| \cdot |\mathcal{V}|^2)$ constraints (with larger constants).

5 Algorithms

5.1 Exact Branch-and-Cut Algorithms

Branch-and-Cut algorithms are similar to conventional LP-based Branch-and-Bound methods with the main difference that, at each Branch-and-Bound node, several LPs may be solved before a branching step is carried out. After solving an LP, it is tested whether the LP solution violates previously neglected or additional valid inequalities. If this is the case, these inequalities are added to the LP as so-called 'cutting planes' and it is solved again. They 'separate' or 'cut off' the respective LP solution - it will not be a feasible solution of the LP solved next. As a consequence,

the addition of inequalities may also improve the lower bounds on the objective function value provided by the LP solutions which is essential, e.g., for proving optimality of a known solution. Either if no further violated inequality can be found or after some predefined number of iterations, a branching step takes place if the LP solution is still non-integral.

Cutting plane approaches are suitable especially if the number of constraints of an integer program is too large to consider all of them from the beginning but a test for violation of these constraints can be done efficiently. This is the case for the problem formulations presented. The model for $r = 1$ from Sect. 4.1 contains the subtour elimination constraints (SECs) whose number is exponential in the number of program variables $\mathcal{V}$. However, the associated separation problem can be solved in polynomial time using minimum cut algorithms [21]. This is exploited in our implementation which we will refer to as `GOA-IP`. It has an additional exact and polynomial-time separation procedure for two-matching-inequalities [22]. Violation of these is tested whenever a fractional LP solution did not violate any SEC. We also decided to separate inequalities (7) from Sect. 4.2 in the assignment-based solver implementation, subsequently named `GOA-AIP`. Although their number is polynomial in the size of the input data, they quickly become a limitation for larger instances due to increased LP solution times. Moreover, typically only a fraction of them is in fact required (i.e., ever violated) during the solution process. We implemented `GOA-IP` and `GOA-AIP` using CPLEX 12.6 [10]. Internal presolving techniques that are not compatible with the application of cutting planes were disabled but we adopted all other default parameters. The optimization starts by relaxing integrality and the respective classes of inequalities that shall be separated as just described. In addition to the omitted inequalities, CPLEX may decide to separate further general cutting planes for integer programs and also the selection of the variables to branch on is left to CPLEX.

The LP solutions obtained during the optimization process will typically not be integral. Here, *primal heuristics* play an important role. Their purpose is to construct good feasible solutions by exploiting the current LP solution. The implemented primal heuristics follow the general idea that variables with an LP value close to one are likely to be part of a good or even optimal solution. In particular, we greedily construct a memory layout as is indicated by the subsequent pseudocode. In case of `GOA-IP`, this means to greedily construct a Hamiltonian cycle. We do this by selecting feasible edges $\{u, v\} \in E$ in non-increasing order of the LP values of their corresponding variables $x_{u,v}$. These edges are collected in the array `select`. A not yet selected edge $\{u, v\}$ is feasible if the degrees of u and v are both less than two and either the number of selected edges is $|V| - 1$ ($\{u, v\}$ will close the Hamiltonian cycle) or there is yet no path from u to v using edges from `select` ($\{u, v\}$ will not create a cycle of length smaller than $|V|$). The latter property is maintained using a common disjoint set data structure where initially each vertex is the representative of its own set, the operation `Union` joins the sets of the (representatives of) two yet disjoint sets and the operation `Find` returns the representative of each set. In `GOA-AIP`, we assign each program variable $v \in \mathcal{V}$ the position that is mostly preferred by its assignment variables $x_{v,p}$, $p \in \{1, \ldots, n\}$. After constructing a memory layout like this, the arc costs of the network flow problem from Sect. 3.1 are set accordingly and the problem is solved to find an optimal ARA.

```
 1: function PRIMALHEURISTIC(x, N = (V_N, A))
 2:     OA ← COMPUTEOFFSETASSIGNMENT(x)
 3:     SETCOSTS(N, OA)                                    # Set arc costs based on distances in OA.
 4:     ARA ← MINCOSTFLOW(N)
```

```
 1: function COMPUTEOFFSETASSIGNMENT(x)                               # GOA-IP version.
 2:     G = (V, E) ← Complete graph from the GOA-IP model
 3:     SORT(E, x)                                      # Sort edges non-increasingly w.r.t. their LP values.
 4:     INITIALIZEUNIONFIND(V)                          # Set up a disjoint-set data structure for V.
 5:     n ← |V|, m ← |E|
 6:     select ← ∅, count ← 0
 7:     for i = 1 → n do
 8:         deg(i) ← 0                                  # Initialize degrees to zero.
 9:     for i = 1 → m do
10:         e = {u, v} ← E[i]
11:         if deg(u) < 2, deg(v) < 2, and (FIND(u) ≠ FIND(v) or count = n − 1) then
12:             select ← select ∪ {e}, count ← count + 1
13:             deg(u) ← deg(u) + 1, deg(v) ← deg(v) + 1
14:             UNION(u, v)                             # Join the sets that contain u and v.
15:     OA ← EXTRACTOA(select)        # Remove artificial vertex z, map path positions to offsets.
16:     return OA
```

```
 1: function COMPUTEOFFSETASSIGNMENT(x)                               # GOA-AIP version.
 2:     n ← |V|
 3:     for p = 1 → n do
 4:         pos[p] ← free                # Initialize all positions p ∈ P = {1, . . . , n} to be unassigned.
 5:     X_max ← Array of LP-values max{x_{v,p} | p ∈ {1, . . . , n}} for each v ∈ V
 6:     SORT(V, X_max)                                  # Sort variables V non-increasingly w.r.t. X_max.
 7:     for i = 1 → n do
 8:         v ← V[i]
 9:         X_v ← Array of LP-values x_{v,p} for each p ∈ {1, . . . , n}.
10:         SORT(P, X_v)                                # Sort positions P non-increasingly w.r.t X_v.
11:         for j = 1 → n do
12:             p ← P[j]
13:             if pos[p] is free then
14:                 pos[p] = v
15:                 OA[v] = p
16:     return OA
```

5.2 Heuristics

For $r = 1$, the predominant heuristic strategy to solve GOA is to first partition the set of variables
w.r.t. the available ARs, and to call a SOA algorithm to compute memory sublayouts for each
of the partitions afterwards. Computational experiments [18] reveal that it is typically more
suggestive to first set up a full memory layout and to compute then an optimal ARA based on
that layout. For $r > 1$, this appears to be even more advisable since most of the existing SOA
algorithms used as subroutines are designed for $r = 1$ (they iteratively select edges [14, 11]) and it
is not trivial to generalize them. In contrast to that, Gebotys' network model is easy to adapt for
arbitrary auto-modify ranges as discussed in Sect. 4 and also exploited by our primal heuristics.

Like in [18], we combine the optimal ARA approach (the min-cost flow (MCF) problem from
Sect. 3.1) with two simple strategies to compute memory layouts. The first one constructs a naive
memory layout that corresponds to the order of first use (OFU) of the program variables. The
algorithm will be referred to as GOA-OFU-MCF. The second algorithm uses the most successful SOA
heuristic from [14, 11] to create a memory layout. In these articles, it is called SOA-INC-TB since it is

an incremental algorithm that iteratively selects edges $\{u, v\} \in E$ from an access graph $G = (V, E)$ and it is combined with a tie-break function for equally weighted edges. The combination of this algorithm with the min-cost-flow based ARA computation is called `GOA-ITB-MCF`.

5.3   Min-Cost Flow Implementation

For all minimum cost flow computations (by the primal heuristics used in the exact solvers as well as by `GOA-OFU-MCF` and `GOA-ITB-MCF`), we called the network simplex algorithm provided by the LEMON C++ library [6] in version 1.3. The asymptotic running time of the network simplex algorithm strongly depends on the used pivoting rule. We relied on the default block search rule of the library [12] that has a worst-case time bound of $\mathcal{O}(nm^2C)$ with $n = |V_N|$, $m = |A|$, and C being the largest assigned cost coefficient (if $c_A = c_L$, setting $C = 1$ is reasonable), but typically performs much better in practice.

6   Experimental Evaluation

6.1   Setup and Test System

For our experimental evaluation, we use the OffsetStone benchmark set that has been extracted from 31 real-world application codes written in ANSI C. Among them are computationally intensive programs (e.g., audio, video and image compression, Fourier transformation) as well as control-dominated applications (e.g., gzip). It has been frequently used in publications dealing with offset assignment and therefore allows for meaningful comparisons. For details on how the instances were extracted, we refer to the original paper [14].

Being able to compute optimal solutions for various combinations of available address registers k and auto-modify ranges r on a larger set of instances for the first time, we evaluate the effect of varying k, r or both on the total offset assignment costs as well as on the quality of heuristic solutions. We assume the costs for address arithmetic instructions c_A and immediate AR loads c_R to be both equal to one and perform the experiments once for one, two, four and eight ARs and auto-modify ranges one, three and seven. We considered all instances that consist of at least three program variables, these are $2,785$ in total.

Our experiments were run single-threaded with an Intel Core $i7$-$3770T$ processor (2.5 GHz) on a Debian Linux system with 8 GB RAM, `g++` 4.7.2, and optimization level `-O2`. We measure the address computation overhead and average solution CPU times of five runs.

6.2   Results

Since extensive evaluation w.r.t. the quality of heuristics for $r = 1$ has been presented already in [18], we only briefly discuss the results for this case. The performance of `GOA-IP` only marginally differs from the mentioned reference since we used a more recent version of CPLEX. The number (and percentage) of instances that timed out after ten seconds is shown in the left of Tab. 2. The timeouts for $k = 1$ are only shown for the sake of completeness. Since this is the SOA case, the min-cost flow part is not needed and the problem can be solved instead by assigning static cost coefficients to the variables of the Hamiltonian cycle part of the problem. Even though only a few instances are not solved within the time limit, it would therefore be more suggestive to use the exact algorithm from [11] that is even more effective. `GOA-IP` however also routinely solves most of the instances when increasing k.

In Tab. 2, we also see that `GOA-AIP` is not at all competitive to `GOA-IP`, especially for $r = 1$. This was expectable; the primal heuristics work well, but for many instances the basic constraints of the assignment problem part do not suffice in order to obtain lower bounds that prove optimality

■ **Table 2** Number of instances timed out by the exact solvers after ten seconds.

GOA-IP	$r = 1$
$k = 1$	11 (0.40%)
$k = 2$	33 (1.19%)
$k = 4$	56 (2.01%)
$k = 8$	54 (1.94%)

GOA-AIP	$r = 1$	$r = 3$	$r = 7$
$k = 1$	$1,253$ (44.99%)	924 (33.18%)	472 (16.95%)
$k = 2$	$1,245$ (44.70%)	880 (31.60%)	471 (16.91%)
$k = 4$	$1,246$ (44.74%)	888 (31.89%)	477 (17.13%)
$k = 8$	$1,252$ (44.96%)	881 (31.63%)	477 (17.13%)

of known solutions. However, with increasing auto-modify ranges, this effect more and more diminishes which can be similarly explained. With increasing r, the concrete memory layout becomes less important for an optimal address register assignment since more arcs in the min-cost flow network can be used without cost in any case. This effect is even stronger if the access sequence lengths are rather small which is often the case in OffsetStone as we will further discuss below. Hence, the lower bounds obtained from the LP and the upper bounds obtained by solutions found by the primal heuristics are much closer to each other and optimality of the latter can be proven more often.

Another expected result is that the property of `GOA-IP` to be relatively insensitive to the number k of ARs is inherited by `GOA-AIP`, especially since their numbers of variables and constraints are independent from k. Even for $r = 1$, `GOA-AIP` solves already $10 - 20\%$ more instances than the implementation of the fixed approach by Ozturk et al. in [18] and there is only a single instance where it failed to complete due to memory limitations which was frequently the case for the latter. This particular instance has a single very large access sequence leading to a flow network with more than $\binom{3,442}{2}$ arc variables.

We identified $1,480$ instances with up to 200 program variables that `GOA-AIP` could solve to optimality for all tested choices of k and r within the time limit of ten seconds. If an exact solver for arbitrary r is desired for practical application, `GOA-AIP` could be improved by several means, e.g., by adding additional cutting planes that are valid for (quadratic) assignment problems. In general, the quadratic nature of the problem and its interdependent structure of two subproblems suggests a reformulation as a semidefinite program or the application of Bender's decomposition approach. The latter is even more suggestive due to the fact that the min-cost-flow subproblem is polynomial-time solvable. In this sense, `GOA-AIP` is to be seen as a 'proof of concept' to produce first results for $r > 1$ that allow to evaluate heuristics and the effect of exploitation of larger auto-modify ranges on the quality of address code generation.

Investing more computation time, we derived optimal solutions for $1,918$ of the $2,785$ instances for all mentioned combinations of r and k. The OffsetStone instances are such that there are usually multiple access sequences associated with one program variable set. Hence, multiple min-cost flow problems need to be solved per instance, all referring to the same memory layout. Further, sequences may refer to disjoint subsets of the program variables such that the instances can be decomposed. The left image of Fig. 8 shows cumulated access lengths for the $1,918$ instances. The right one gives a fine-grain distribution of single access sequence lengths after decomposition. Unfortunately, the number of longer access sequences is rather small which is one of the already addressed reasons why the instances become 'easier' or faster to solve for `GOA-AIP` with increasing r.

Fig. 9 shows the results of our experiments for the $1,918$ optimally solved instances. The left image shows the impact of the various configurations for r and k on the total offset assignment cost, accumulated over all of the instances. The central observation is that the amount of address code can be considerably reduced when exploiting larger auto-modify ranges. At least for the evaluated instances, it appears that the reduction potential by increasing r is much higher than by increasing
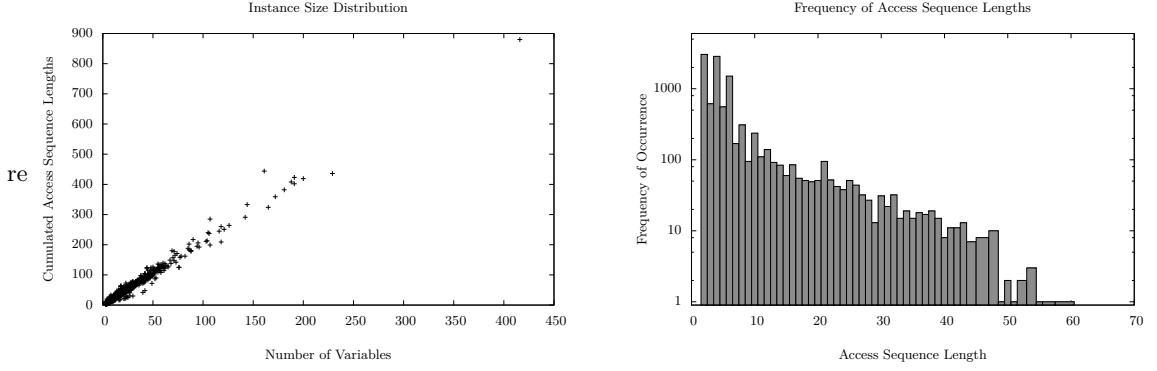
Figure 8 Distribution of the number of variables and access lengths across the $1,918$ instances.
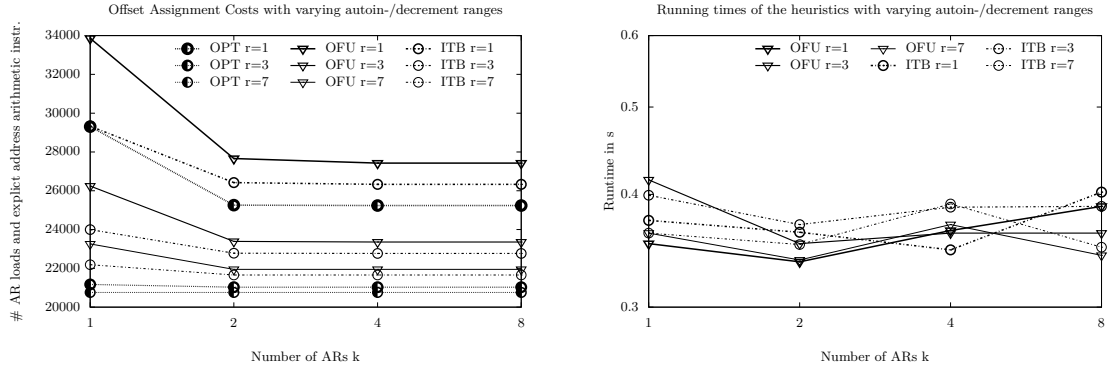
Figure 9 Offset assignment costs for all tested configurations and running times of the heuristics.

k since the offset assignment costs do not further decrease significantly for $k > 2$. However, this is partially also due to the just discussed character of the instances. The results approve the already in [18] observed impression, that the performance loss when using the proposed heuristics is rather small - as well for increasing auto-modify ranges. The `GOA-ITB-MCF`-layout is clearly a better basis for the ARA part than an order-of-first-use layout. However, since it just considers adjacencies of program variables, there is even further potential in achieving near-optimal solutions by taking the larger auto-modify ranges into account already when generating the memory layouts. Using an optimum address register assignment is worthwhile and computationally not too intensive so that it can be performed in production compilers. As can be seen in the right image, the heuristics are fast and sum up to less than half of a second in total although many small min-cost flow problems need to be solved.

7 Conclusion

We have described integer programming formulations for the general offset assignment problem. In particular, an already computationally successful model for the case where a processor supports only autoin-/decrement instructions could be extended to deal with more general auto-modify capabilities. Our experimental results show that Branch-and-Cut algorithms that use the proposed models as a basis are able to solve a wide range of real-world instances from the commonly used OffsetStone benchmark set. While, for the autoin-/decrement case, nearly all instances can be

solved in less than ten seconds of CPU time, the more general approach is, depending on the range of allowed auto-modify instructions, less applicable especially for larger instances. However, it can solve more instances than previously proposed models and has the useful property that its size (in terms of variables and constraints) is independent from the number of address registers. This allowed for new experiments that reveal two new important insights concerning the problem setting in general. At first, the exploitation of larger auto-modify ranges can help to significantly reduce the address computation overhead in real-world programs. Further, computing an optimal address register assignment based on a good (or optimal) memory layout is a good strategy and computationally feasible in practice, also for auto-modify ranges beyond one. Since we could also show how to incorporate operand reordering techniques into an existing optimal address register assignment algorithm, this approach is even more attractive.

References

1 Sunil Atri, Jagannathan Ramanujam, and Mahmut T. Kandemir. Improving offset assignment on embedded processors using transformations. In Mateo Valero, Viktor K. Prasanna, and Sriram Vajapeyam, editors, *Proc. of the 7th International Conference on High Performance Computing (HiPC'00), Bangalore, India, December 17–20, 2000*, volume 1970 of *Lecture Notes in Computer Science*, pages 367–374. Springer, 2000. `doi: 10.1007/3-540-44467-X_33`.

2 David H. Bartley. Optimizing stack frame accesses for processors with restricted addressing modes. *Softw., Pract. Exper.*, 22(2):101–110, 1992. `doi: 10.1002/spe.4380220202`.

3 Rainer E. Burkard, Mauro Dell'Amico, and Silvano Martello. *Assignment Problems*. SIAM, 2009. URL: `http://www.ec-securehost.com/SIAM/OT106.html`.

4 Yoonseo Choi and Taewhan Kim. Address assignment combined with scheduling in DSP code generation. In *Proc. of the 39th Design Automation Conference (DAC'02), New Orleans, LA, USA, June 10–14, 2002*, pages 225–230. ACM, 2002. `doi:10.1145/513918.513975`.

5 George B. Dantzig, David R. Fulkerson, and Selmer M. Johnson. Solution of a large-scale traveling-salesman problem. *Operations Research*, 3:393–410, 1954.

6 Balázs Dezso, Alpár Jüttner, and Péter Kovács. LEMON - an open source C++ graph template library. *Electr. Notes Theor. Comput. Sci.*, 264(5):23–45, 2011. `doi:10.1016/j.entcs.2011.06.003`.

7 Catherine H. Gebotys. DSP address optimization using a minimum cost circulation technique. In Ralph H. J. M. Otten and Hiroto Yasuura, editors, *Proc. of the 1997 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'97), San Jose, CA, USA, November 9–13, 1997*, pages 100–103. IEEE Computer Society / ACM, 1997. `doi:10.1109/ICCAD.1997.643380`.

8 Catherine H. Gebotys. A minimum-cost circulation approach to DSP address-code generation. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 18(6):726–741, 1999. `doi:10.1109/43.766724`.

9 Johnny Huynh, José Nelson Amaral, Paul Berube, and Sid Ahmed Ali Touati. Evaluating address register assignment and offset assignment algorithms. *ACM Trans. Embedded Comput. Syst.*, 10(3):37, 2011. `doi:10.1145/1952522.1952530`.

10 IBM. *IBM ILOG CPLEX Optimization Studio Version 12.6*, 2013.

11 Michael Jünger and Sven Mallach. Solving the simple offset assignment problem as a traveling salesman. In Henk Corporaal and Sander Stuijk, editors, *Proc. of the International Workshop on Software and Compilers for Embedded Systems (M-SCOPES'13), Sankt Goar, Germany, June 19–21, 2013*, pages 31–39. ACM, 2013. `doi:10.1145/2463596.2463601`.

12 Zoltán Király and Péter Kovács. Efficient implementations of minimum-cost flow algorithms. *Acta Universitatis Sapientiae, Informatica*, 4(1):67–118, 2012. URL: `http://www.acta.sapientia.ro/acta-info/C4-1/info41-5.pdf`.

13 Eugene Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, 1976.

14 Rainer Leupers. Offset assignment showdown: Evaluation of DSP address code optimization algorithms. In Görel Hedin, editor, *Proc. of the 12th International Conference on Compiler Construction (CC'03), held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS'03), Warsaw, Poland, April 7–11, 2003*, volume 2622 of *Lecture Notes in Computer Science*, pages 290–302. Springer, 2003. `doi: 10.1007/3-540-36579-6_21`.

15 Rainer Leupers and Fabian David. A uniform optimization technique for offset assignment problems. In Francky Catthoor, editor, *Proc. of the 11th International Symposium on System Synthesis (ISSS'98), Hsinchu, Taiwan, December 2–4, 1998*, pages 3–8. ACM / IEEE Computer Society, 1998. `doi:10.1109/ISSS.1998.730589`.

16 Rainer Leupers and Peter Marwedel. Algorithms for address assignment in DSP code generation. In *Proc. of the 1996 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'96)*, pages 109–112, 1996. `doi:10.1109/ICCAD.1996.569409`.

17 Stan Yi-Huang Liao. *Code generation and optimization for embedded digital signal processors.* PhD thesis, Massachusetts Institute of Technology, 1996. URL: `http://hdl.handle.net/1721.1/11048`.

18 Sven Mallach and Roberto Castañeda Lozano. Optimal general offset assignment. In Henk Corporaal and Sander Stuijk, editors, *Proc. of the 17th International Workshop on Software and Compilers for Embedded Systems (SCOPES'14), Sankt Goar, Germany, June 10–11, 2014*, pages 50–59. ACM, 2014. `doi:10.1145/2609248.2609251`.

19 Garth P. McCormick. Computability of global solutions to factorable nonconvex programs: Part I – convex underestimating problems. *Mathematical Programming*, 10(1):147–175, 1976. `doi:10.1007/BF01580665`.

20 Ozcan Ozturk, Mahmut T. Kandemir, and Suleyman Tosun. An ILP based approach to address code generation for digital signal processors. In Gang Qu, Yehea I. Ismail, Narayanan Vijaykrishnan, and Hai Zhou, editors, *Proc. of the 16th ACM Great Lakes Symposium on VLSI (GLSVLSI'06), Philadelphia, PA, USA, April 30 – May 1, 2006*, pages 37–42. ACM, 2006. `doi:10.1145/1127908.1127919`.

21 Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Rev.*, 33(1):60–100, 2 1991. `doi:10.1137/1033004`.

22 Manfred W. Padberg and Mendu R. Rao. Odd minimum cut-sets and b-matchings. *Mathematics of Operations Research*, 7(1):67–80, 1982. URL: `http://www.jstor.org/stable/3689360`.

23 Amit Rao and Santosh Pande. Storage assignment optimizations to generate compact and efficient code on embedded dsps. In Barbara G. Ryder and Benjamin G. Zorn, editors, *Proc. of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'99), Atlanta, Georgia, USA, May 1–4, 1999*, pages 128–138. ACM, 1999. `doi:10.1145/301618.301653`.

24 Nobuhiko Sugino, Satoshi Iimuro, Akinori Nishihara, and Nobuo Fujii. DSP code optimization utilizing memory addressing operation. *IEICE Transactions on Fundam. of Electr., Commu. and Comp. Sc.*, 79(8):1217–1224, 1996.

25 Bernhard Wess and Martin Gotschlich. Optimal DSP memory layout generation as a quadratic assignment problem. In *Proc. of the 1997 IEEE International Symposium on Circuits and Systems (ISCAS'97)*, volume 3, pages 1712–1715, Jun 1997. `doi:10.1109/ISCAS.1997.621465`.