

Visualization of Procedural Abstraction

Stefan Schaeckeler¹, Weijia Shang², Ruth Davis³

*Department of Computer Engineering
Santa Clara University
Santa Clara, CA 95053, USA*

Abstract

Visualizing impacts of an optimization pass helps to reason about, and to gain insight into, the inner workings of the optimization pass. In this paper, we visualize the impacts of two procedural abstraction passes. For this, we modified two procedural abstraction post pass optimizers to visualize for each the difference in machine code before and after optimization by drawing abstracted fragments in the original program. We then explain how the generated visualizations aid in better understanding the optimization passes.

Keywords: Visualization of computational processes, Program visualization, Program understanding, Compiler understanding, Code compaction, Procedural abstraction, Post pass optimization

1 Introduction

Visualizations are often used in mechanical engineering, chemistry, physics, and medicine [3], but are occasionally used in computer science as well to aid program understanding (see for example the ACM Symposia on Software Visualization (SOFTVIS), the IEEE Workshops on Visualizing Software for Understanding and Analysis (VISSOFT), or the Program Visualization Workshops (PVW)). For program understanding, program executions often generate very large traces. It is a challenging task to represent these masses of data in a digestible form and a lot of research is conducted for appropriate visualization techniques.

Visualizations for understanding optimization passes are not always so complex. We found natural visual representations, that are powerful yet simple enough to *completely* understand in their entirety. We believe visualizations can be a great aid for compiler writers in understanding their optimization passes in greater depth, and we hope the insight thus gained might help them to improve the optimization passes.

¹ Email: sschaeck@engr.scu.edu

² Email: wshang@scu.edu

³ Email: rdavis@scu.edu

In this paper, we visualize procedural abstraction. Typically, all we see from running a size optimization pass such as procedural abstraction is one number — reflecting the reduction in program size. To make its inner workings visible, we generate from the internal data structures of our optimization passes several visualizations. After a brief review of procedural abstraction in the next section, we introduce these visualizations in sections 3 and 4 and explain how they help in better understanding procedural abstraction. Section 5 discusses future work, and section 6 concludes the paper.

2 Background on Procedural Abstraction

Optimizing compilers traditionally target execution speed, but may also target code size as this becomes increasingly important for embedded systems. A common technique for compacting code is *procedural abstraction*, or *abstraction* for short. In its standard form, equivalent code fragments are identified, abstracted in a new procedure, and eventually replaced by procedure calls. This saves all but one occurrence of the fragments and adds a small overhead of one procedure call per fragment and one return instruction per abstracted procedure. Abstracted procedures are minimalistic functions without function prologues or epilogues.

Example 2.1 (Procedural Abstraction) Fig. 1 provides an example of procedural abstraction. In the original code of Fig. 1(a), either two code fragments of four instructions (Fig. 1(b)) or three code fragments of three instructions (Fig. 1(c)) can be abstracted. Whatever abstraction is more beneficial in terms of code size can then be chosen.

| | | | |
|-----|--|--|---|
| (a) | <pre>load r1, \$5200 add r1, r2 rot r1, \$2 mul r1, r1</pre> | <pre>load r1, \$5200 add r1, r2 rot r1, \$2 mul r1, r1</pre> | <pre>load r1, \$5300 add r1, r2 rot r1, \$2 mul r1, r1</pre> |
| (b) | <pre>call f</pre> | <pre>call f</pre> | <pre>load r1, \$5300 add r1, r2 rot r1, \$2 mul r1, r1 f: load r1, \$5200 add r1, r2 rot r1, \$2 mul r1, r1 ret</pre> |
| (c) | <pre>load r1, \$5200 call f</pre> | <pre>load r1, \$5200 call f</pre> | <pre>load r1, \$5300 call f f: add r1, r2 rot r1, \$2 mul r1, r1 ret</pre> |

Fig. 1. Example of Procedural Abstraction

The challenge of procedural abstraction is to efficiently find fragments for abstraction. Fraser et al., Cooper and McIntosh and Schaeckeler and Shang use suffix trees to identify fragments in $O(n \cdot \log(n))$ time [1,4,10]. The details do not concern us in this paper; we assume fragments for abstraction have already been identified.

3 Visualization of Procedural Abstraction

We implemented in [10] a procedural abstraction post-pass optimizer for Intel’s 32-bit architecture IA32. The optimization pass applied to seven programs from the MediaBench suite [7] resulted in an average code size reduction of 2.502%. In this paper we use the mpeg encoder `mpeg2enc` as a running example because, with 13,599 instructions and 49,927 bytes, its visualizations fit on single pages. We found in this program 333 abstracted fragments which could be abstracted into 66 procedures, resulting in a reduction of 1.160%.

Programs are usually visualized either graph or pixel based. For procedural abstraction, we worked out several pixel based visualizations, which are not only a natural choice, but also avoid known shortcomings of graph based visualizations such as scalability, layout and mapping problems [9].

We visualize instructions in the original program as what we call a *program map*. For program maps, there can be two levels of abstraction in which pixels represent either whole instructions or individual bytes of instructions, in ascending order from left to right, starting in the upper left corner and wrapping around at the end of each line. As the main purpose of program maps is to identify fragments, it is convenient to introduce a new term and call all pixels representing an individual fragment a *string*.

Pixels have length and area. Color may be used to emphasize pixels and strings. In the byte representation, the lengths and areas of pixels and strings are proportional to the sizes of instructions and fragments, and their quantities can be easily estimated from the visualization. If this is not required, then the more compact instruction representation may be sufficient and can be used instead.

Procedural abstraction has a flat view on the code. We abstract fragments, i.e. sequences of instructions, which are in turn bytes. Hence, two levels of abstraction are enough to capture the essence of procedural abstraction.

We generated program maps for the first time in [10]. We used the same color for each abstracted instruction, and it was impossible to distinguish adjacent abstractions from single abstractions. In Fig. A.1, we refined this program map by using light gray for the last pixel of an otherwise gray string. This revealed adjacent fragments which we had not been able to see before.

In Fig. A.1, we see many fragments consisting of only one instruction. Table 1 gives a detailed breakdown. More than 50% of all fragments are individual instructions (this is possible if the instruction length is larger than the length of the replacing function call instruction), while the remaining fragments consist of two to seven instructions. Fig. A.2 gives the program map over bytes, and light gray — here of the last five pixels — is again used to mark the ends of strings. It can be seen that there are a lot of short fragments. Table 2 gives a detailed breakdown. More than 50% of all fragments are rather short, with six or seven bytes. Most of the remaining fragments extend gradually up to 20 bytes, and there are two additional fragments of 26 bytes each.

One would expect short fragments to result in small net gains. We further investigate the net gains of fragments by visualizing the overhead. A light gray pixel in Fig. A.1 can be interpreted to represent the function call instruction responsible

Table 1
Number of Fragments and Procedures over their Lengths in Instructions

| fragment length [instr.] | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ≥ 8 |
|--------------------------|---|-----|----|----|----|----|----|---|----------|
| fragments [#] | 0 | 182 | 43 | 23 | 34 | 39 | 10 | 2 | 0 |
| procedures [#] | 0 | 18 | 12 | 7 | 12 | 11 | 5 | 1 | 0 |

Table 2
Number of Fragments and Procedures over their Lengths in Bytes

| fragment length [bytes] | ≤ 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|-------------------------|----------|----|-----|---|----|----|----|----|----|
| fragments [#] | 0 | 47 | 144 | 8 | 10 | 7 | 39 | 22 | 9 |
| procedures [#] | 0 | 4 | 15 | 2 | 1 | 2 | 7 | 9 | 4 |

| 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | ≥ 27 |
|----|----|----|----|----|----|----|----|----|----|----|----|----|-----------|
| 17 | 10 | 8 | 4 | 2 | 2 | 2 | 0 | 0 | 0 | 0 | 0 | 2 | 0 |
| 8 | 4 | 4 | 2 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

for the overhead per fragment, but this is a distorted view as instruction lengths vary on IA32 from one to 17 bytes. The return instruction `ret` is, for instance, one byte and the function call instruction `call <32-bit address>` is five bytes in length, one byte for the opcode and four bytes for the address field. A byte representation is necessary for capturing the function call overhead. Because all light gray pixels of strings in Fig. A.2 have exactly the size of a function call overhead, this figure can be used to analyze the overhead. Net gains of abstracted fragments are then represented by the remaining gray pixels. For each abstracted procedure, there is also an overhead of one byte for the return instruction. The accumulated area for all 66 return instructions occupies 27.5% of the figure width and is given in Fig. A.2 as a black line.

The areas of all 333 abstracted fragments, e.g. of all gray and light gray pixels, comprise 4.627% of the whole program map, i.e. 4.627% or 2,310 bytes of code is abstractable. As the overhead is five bytes per abstracted fragment and one byte per abstracted procedure, including the abstracted procedures this total comes to 1,731 bytes or 3.461% of the program size, and what remains is a net gain of merely 579 bytes or 1.160%.

That the overhead is almost three times the net gain is quite disappointing. This observation motivated us to investigate alternative computer architectures with different function call/return overheads. If the function call/return overhead were less, then not only would there be less overhead for abstraction, but additional fragments would also emerge for abstraction, because more fragments would then have a non-negative benefit, i.e. would be larger than the size of the function call instruction. Table 3 gives statistics for function call and return instructions of varying sizes. The upper limit is for no function call/return overhead and would result in a reduction of 20.548%. Because there is no overhead, even (small) single instructions can be

Table 3
 Statistics over the Lengths of Call and Return Instructions

| | | | | | | |
|----------------------|--------|--------|-------|-------|-------|-------|
| size of call [bytes] | 0 | 1 | 2 | 3 | 4 | 5 |
| size of ret [bytes] | 0 | 0 | 0 | 0 | 0 | 0 |
| procedures [#] | 654 | 600 | 370 | 226 | 138 | 79 |
| fragments [#] | 4,026 | 3,470 | 1,610 | 969 | 651 | 376 |
| overhead [bytes] | 0 | 3,470 | 3,220 | 2,907 | 2,604 | 1,880 |
| net gain [bytes] | 10,259 | 6,230 | 3,339 | 2,025 | 1,186 | 658 |
| reduction [%] | 20.548 | 12.478 | 6.688 | 4.056 | 2.375 | 1.318 |
| size of call [bytes] | 0 | 1 | 2 | 3 | 4 | 5 |
| size of ret [bytes] | 1 | 1 | 1 | 1 | 1 | 1 |
| procedures [#] | 653 | 491 | 309 | 180 | 121 | 66 |
| fragments [#] | 4,024 | 3,231 | 1,458 | 852 | 609 | 333 |
| overhead [bytes] | 643 | 3,722 | 3,225 | 2,736 | 2,557 | 1,731 |
| net gain [bytes] | 9,605 | 5,630 | 2,969 | 1,799 | 1,048 | 579 |
| reduction [%] | 19.238 | 11.276 | 5.947 | 3.603 | 2.099 | 1.160 |

abstracted. The corresponding program maps of Figs. A.3 and A.4 show the high redundancies of instructions in a program: for each instruction represented by a gray pixel there exists at least one further identical instruction in the code.

Interesting scenarios that can be implemented in hardware, are:

call instr. size = 5 bytes; return instr. size = 0 bytes: Encoding the length of the abstracted procedure in a function call instruction can reduce the program size by 1.318%. This also has other interesting consequences: there can be now different entry/exit points and abstracted procedures can overlap [8] or do not need to be abstracted in new procedures at all [6]. This might lead to further reductions.

call instr. size = 3 bytes; return instr. size = 1 byte: A new function call instruction with a relative address mode can reach all procedures in programs $\leq 65,536$ bytes with an address field size of two bytes. This can reduce the program size by 3.603%. If this addressing mode is also used for regular functions, then a further reduction can be expected.

call instr. size = 3 bytes; return instr. size = 0 bytes: This scenario combines the previous two and results in a reduction of 4.056%.

The previous scenarios show that changing the function call/return mechanism on IA32 would enable procedural abstraction to produce significant reductions in code size.

Not all fragments can be used for abstraction. Fragments must be single entry–single exit regions and can extend in our implementation up to single basic blocks (see [4] for details). Furthermore, fragments should not include stack accesses (this includes also function calls), as calls to abstracted procedures modify the stack by pushing the return address on the stack so that wrong stack slots might be accessed within the abstracted procedures (see [4,10] for details).

Keeping fragments within basic blocks and abstracting stack accesses would result in a higher net gain. Figs. A.5 and A.6 give such program maps. Non-white colors indicate abstracted instructions. Gray is used for regularly abstractable instructions, e.g. for the instructions of Figs. A.1 and A.2, black for instructions accessing the stack and light gray for the remaining instructions. Apparently, not abstracting stack accesses results in a three times lower net gain. Figs. A.5 and A.6 suggest that this huge loss is due to both black pixels, i.e. directly due to stack access instructions, and light gray pixels, i.e. indirectly due to stack access instructions, which, when part of a fragment, can reduce the abstractable part below the size of the call instruction or influence combinations of fragments for abstraction and leave some fragments unabstracted.

As mentioned before, fragments lie within single basic blocks. The program map in Fig. A.7⁴ shows how fragments fill out basic blocks. Abstracted fragments are represented as gray pixels and basic block boundaries as black pixels. For this, we replace each jump and branch instruction with a black pixel and insert at each jump or branch target a black pixel. This distorts the program map somewhat, but the sizes of individual basic blocks remain the same. From Fig. A.7 we can see that over a third (precisely, 36.949%) of the abstracted fragments are whole basic blocks while almost two-thirds (precisely, 63.051%) are not.

To reduce the cost of finding fragments, Debray et al. limit the search for fragments in their compactor to whole basic blocks, only [2]. We learned from our visualization that this would drastically reduce the efficiency of at least our compactor.

4 Visualization of Procedural Abstraction Variants

A variant of procedural abstraction, hereafter called *tail merging procedural abstraction*, allows not only identical fragments for abstraction, but also fragments identical to (but differing in length from) tails of the longest fragment. The longest fragment is then abstracted into a procedure, and all fragments are replaced by procedure calls to their corresponding start instruction *somewhere* in the abstracted procedure.

Example 4.1 (Tail Merging Procedural Abstraction) Fig. 2 gives an example for tail merging procedural abstraction. The first fragment is identical to the second fragment and so both are abstracted in a new procedure. These two fragments are replaced by procedure calls to the first instruction of the procedure. The last three instructions of the third fragment are identical to the last three instructions of the

⁴ The program map over instructions is enough as a program map over bytes does not give us in this case any additional information.

abstracted procedure (e.g. its tail) and can then be replaced by a procedure call into the procedure to its second instruction.

```
(a) load r1, $5200   load r1, $5200   load r1, $5300
    add r1, r2       add r1, r2       add r1, r2
    rot r1, $2       rot r1, $2       rot r1, $2
    mul r1, r1       mul r1, r1       mul r1, r1

(b) call f1         call f1         load r1, $5300  f1: load r1, $5200
                                                call f2         f2: add r1, r2
                                                        rot r1, $2
                                                        mul r1, r1
                                                        ret
```

Fig. 2. Example of Tail Merging Procedural Abstraction

Earlier work on tail merging procedural abstraction in [5] and [8] did not provide any comparison with traditional procedural abstraction, and it remained unclear whether there is an actual improvement for real programs. This lack of comparison data motivated us to write not only a traditional, but also a tail merging procedural abstraction post pass optimizer. The reduction for `mpeg2enc` under traditional procedural abstraction is 1.160% and for tail merging procedural abstraction is 1.242%. The reduction over all seven MediaBench programs is on average 2.502% and 2.716%, respectively.

To understand from where the improvements are coming, we generated in Fig. A.8(a) a program map for traditional procedural abstraction and in Fig. A.8(b) a program map for tail merging procedural abstraction. As they are quite similar, it is instructive to generate the *difference map* of Fig. A.8(c) as well. Gray pixels represent instructions that can be abstracted by both abstraction methods. Black pixels represent instructions that can only be abstracted with tail merging abstraction, and light gray pixels represent instructions that can only be abstracted with traditional procedural abstraction. The black pixels in Fig. A.8(c) indicate the higher code size reduction for tail merging procedural abstraction.

As before, the program maps of Figs. A.8(a) and A.8(b) have been directly generated from our optimization passes using their internal data structures. These program maps are then input to a script generating the difference map of Fig. A.8(c).

The difference map shows that the improvements are coming both from extended fragments, e.g. black pixels left adjacent to gray pixels, and from newly emerging fragments, e.g. black strings in isolation. Roughly a third of the improvements are coming from extended fragments while roughly two-thirds are coming from newly emerging fragments.

5 Future Work

We intend to write an interactive program map, e.g. a java applet which lets the user interactively explore abstractions in a program. It will be able to not only display the program maps discussed here, but it will also allow one to see references

between abstractions of a procedure, e.g. clicking on a fragment will highlight multiple occurrences of the same fragment. Interactively removing and re-adding fragments will show the current reduction and, if sufficient profiling information is available, show the estimated run-time of the program.

We hope that such an interactive map will give us playful insight into interactions between abstractions, run-time, and code reduction.

6 Conclusion

We hope this paper may inspire other compiler writers to visualize optimization passes to help them to reason about, and to understand, the inner workings of various optimization techniques.

Visualizations can be also used in compiler classes to make optimizations less abstract and to give students a better understanding of where and how often optimizations are applied in the code.

References

- [1] Cooper, K. D. and N. McIntosh, *Enhanced code compression for embedded RISC processors*, in: *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation* (1999), pp. 139–149.
- [2] Debray, S. K., W. Evans, R. Muth and B. De Sutter, *Compiler techniques for code compaction*, *ACM Trans. Program. Lang. Syst.* **22** (2000), pp. 378–415.
- [3] Diehl, S., “Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software,” Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.
- [4] Fraser, C. W., E. W. Myers and A. L. Wendt, *Analyzing and compressing assembly code*, in: *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction* (1984), pp. 117–121.
- [5] Gyimóthy, T., R. Ferenc, G. Lehotai, A. Kiss and A. Bicsak, *US patent nr. 7,293,264: Method and a device for abstracting instruction sequences with tail merging* (2005).
- [6] Lau, J., S. Schoenmackers, T. Sherwood and B. Calder, *Reducing code size with echo instructions*, in: *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems* (2003), pp. 84–94.
- [7] Lee, C., M. Potkonjak and W. H. Mangione-Smith, *Mediabench: a tool for evaluating and synthesizing multimedia and communications systems*, in: *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture* (1997), pp. 330–335.
- [8] Liao, S., S. Devadas and K. Keutzer, *A text-compression-based method for code size minimization in embedded systems*, *ACM Trans. Des. Autom. Electron. Syst.* **4** (1999), pp. 12–38.
- [9] Marcus, A., L. Feng and J. I. Maletic, *3D representations for software visualization*, in: *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization* (2003), pp. 27–ff.
- [10] Schaeckeler, S. and W. Shang, *Procedural abstraction with reverse prefix trees*, submitted, under review.

APPENDIX

A Visualizations

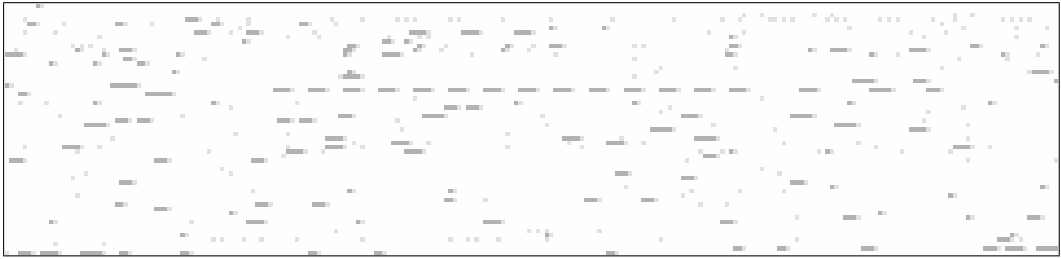


Fig. A.1. Visualization of Abstracted Fragments [instructions]: Fragments include a light gray End Marker.

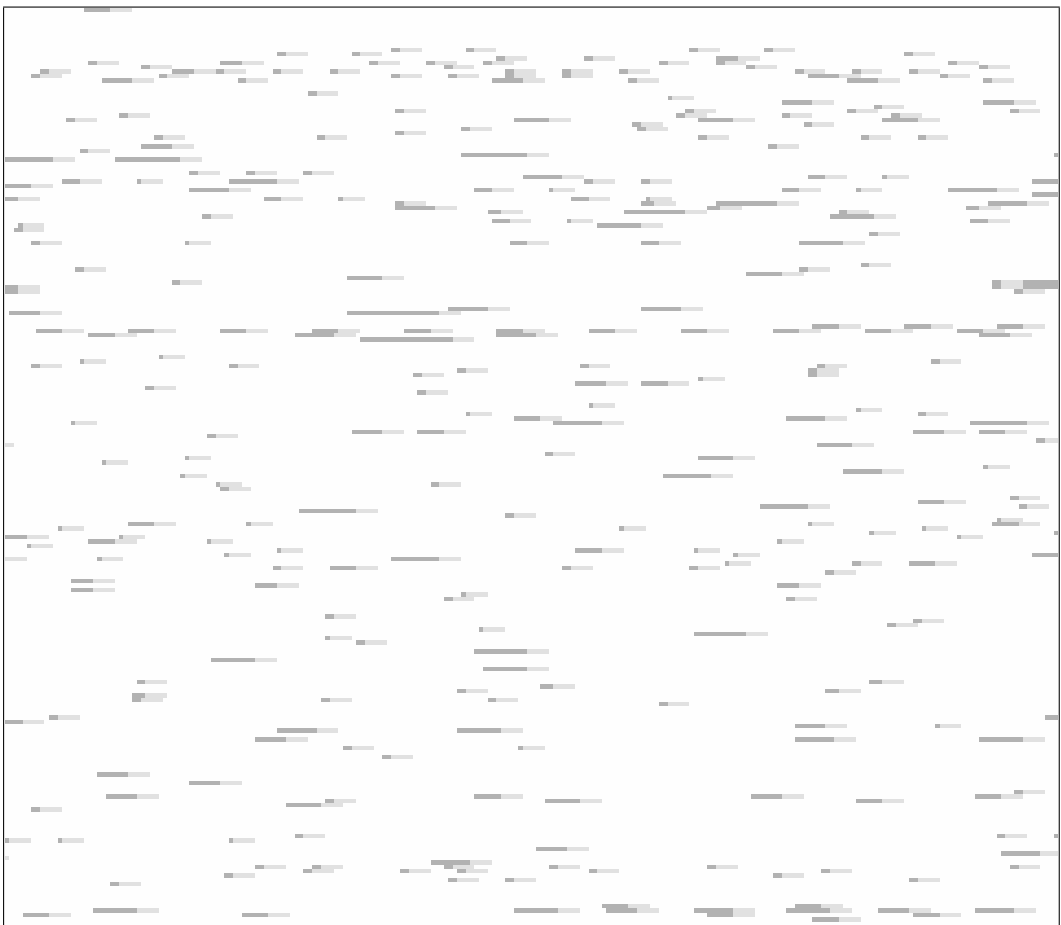


Fig. A.2. Visualization of Abstracted Fragments [bytes]: Fragments include a five Pixel long light gray End Marker of the Size of the Function Call Overhead.

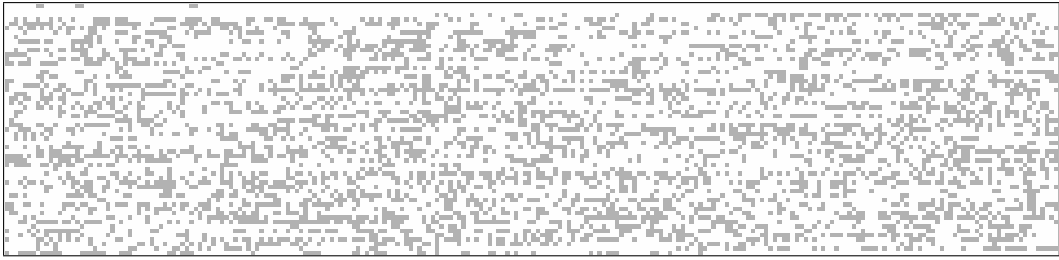


Fig. A.3. Visualization of Abstracted Fragments [instructions]: Fragments for no Function Call/Return Overhead.

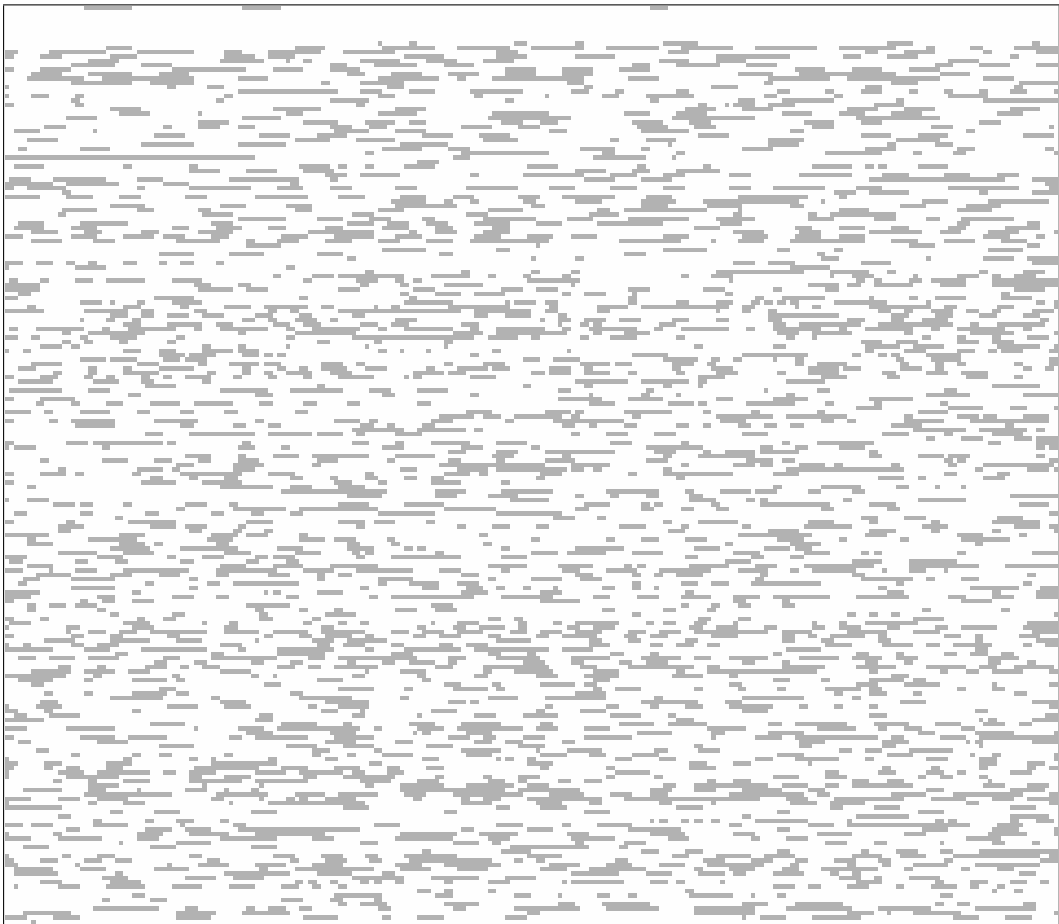


Fig. A.4. Visualization of Abstracted Fragments [bytes]: Fragments for no Function Call/Function Overhead.

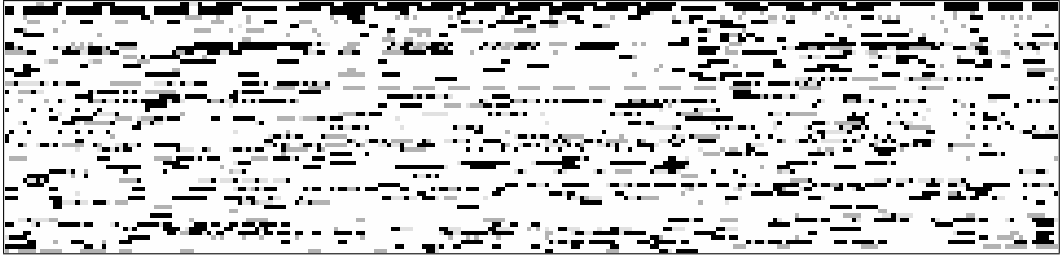


Fig. A.5. Visualization of Abstracted Fragments [instructions]: Fragments include Function Calls and Stack Accesses in black.

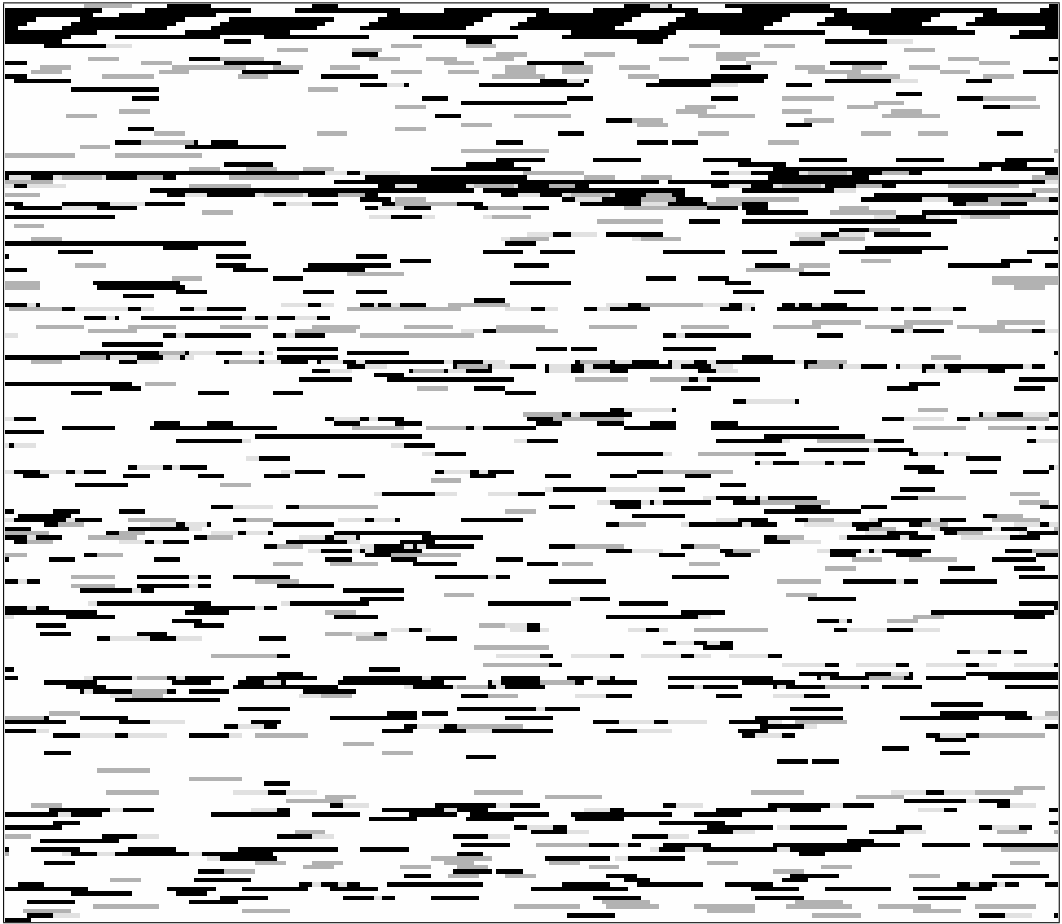
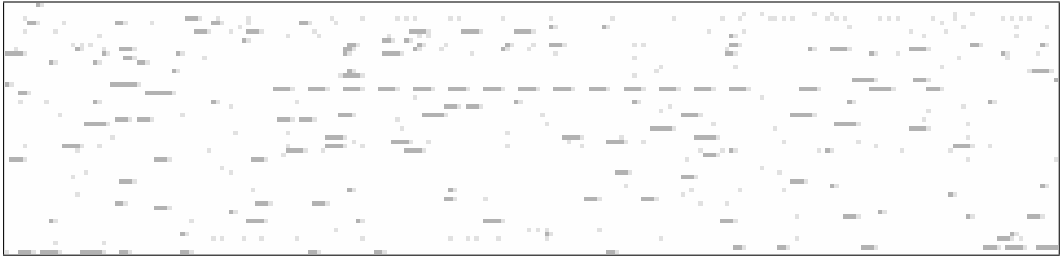


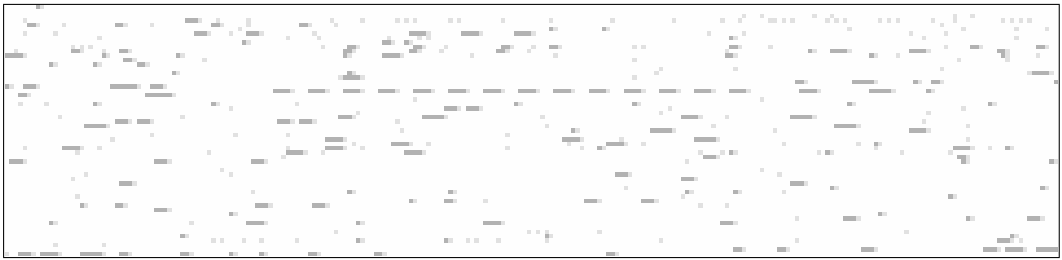
Fig. A.6. Visualization of Abstracted Fragments [bytes]: Fragments include Function Calls and Stack Accesses in black.



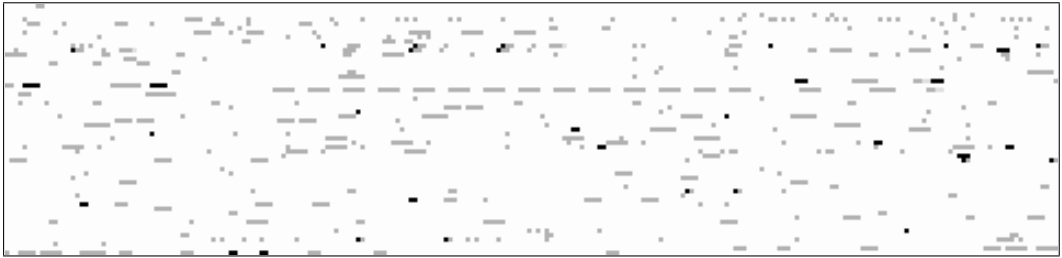
Fig. A.7. Visualization of Abstracted Fragments [instructions]: Fragments within Basic Blocks.



(a) Traditionally Abstracted Fragments



(b) Tail Merged Abstracted Fragments



(c) Difference of both Abstractions

Fig. A.8. Visualization of Abstracted Fragments.