Discovery of Locality-Improving Refactorings by Reuse Path Analysis

Kristof Beyls and Erik H. D'Hollander

Department of Electronics and Information Systems (ELIS), Ghent University,
Sint-Pietersnieuwstraat 41, B-9000 Ghent, Belgium
{kristof.beyls, erik.dhollander}@elis.UGent.be

Abstract. Due to the huge speed gaps in the memory hierarchy of modern computer architectures, it is important that programs maintain a good data locality. Improving temporal locality implies reducing the distance of data reuses that are far apart. The best existing tools indicate locality bottlenecks by highlighting both the source locations generating the use and the subsequent cache-missing reuse. Even with this knowledge of the bottleneck locations in the source code, it often remains hard to find an effective code refactoring that improves temporal locality, due to the unclear interaction of function calls and loop iterations occurring between use and reuse.

The contributions in this paper are two-fold. First, the locality analysis is enhanced to not only pinpoint the cache bottlenecks, but to also suggest code refactorings that may resolve them. The refactorings are found by analyzing the dynamic hierarchy of function calls and loops on the code path between reuses, called *reuse paths*. Secondly, reservoir sampling of the reuse paths results in a significant reduction of the execution time and memory requirements during profiling, enabling the analysis of realistic programs.

An interactive GUI, called *SLO (Suggestions for Locality Optimizations)*, has been used to explore the most appropriate refactorings in a number of SPEC2000 programs. After refactoring, the execution time of the selected programs was halved, on the average.

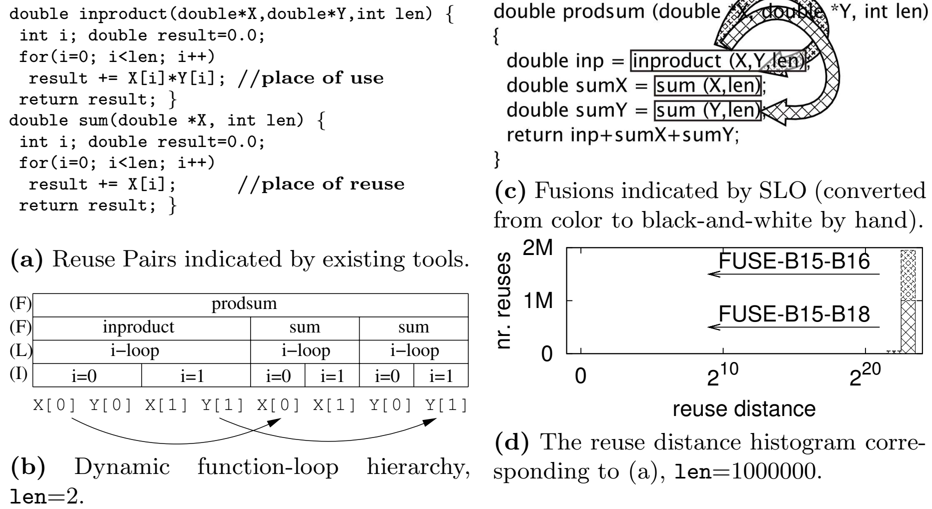
1 Introduction

The memory access time is a growing bottleneck in high performance computer systems. A suitable memory hierarchy is one of the prominent answers, and its beneficial effect is proportional to a good data locality during program execution.

Poor temporal locality occurs when between consecutive reuses of the same data, a large amount of other data is accessed. Two consecutive reuses of the same data are called a *reuse pair*. In this paper, the distance of a reuse pair is defined as the number of accesses between use and reuse

The temporal locality of a reuse pair can only be increased by reordering memory accesses, so that its distance is reduced. The best existing analysis tools [1,2] highlight both the source locations generating the use and the reuse of long-distance reuse pairs. However, using this information, it often remains

M. Gerndt and D. Kranzlmller (Eds.): HPCC 2006, LNCS 4208, pp. 220–229, 2006.
© Springer-Verlag Berlin Heidelberg 2006



(a) Reuse Pairs indicated by existing tools.

(b) Dynamic function-loop hierarchy, `len`=2.

(c) Fusions indicated by SLO (converted from color to black-and-white by hand).

(d) The reuse distance histogram corresponding to (a), `len`=1000000.

Fig. 1. Example program containing long-distance reuses

hard to find a code refactoring that reduces the distance, due to the unclear interaction of function calls and loop iterations between the reuses.

Example 1. Consider the code in Fig. 1(a). The comments indicate the source location where the cache-missing reuse occurs and the location where the previous use of the data was. This view on the code does not reveal how to bring use and reuse closer together, since it is not clear from where the functions `inproduct` and `sum` are called.

In contrast, the presented method analyzes the *dynamic function-loop hierarchy*, which is the hierarchy of all function calls, loop executions and loop iterations at run-time. An example is given in Fig. 1(b), where (I) indicates a loop-iteration level, (L) a loop level and (F) a function-call level. Our method proceeds by analyzing the highest level in this hierarchy where "sections" are crossed between use and reuse. Significantly reducing the distance between use and reuse requires the merging of the corresponding sections. Merging these run-time sections by static source code transformations requires the following refactorings, depending on the type of sections that need to be merged:

- **(I) Iteration merging** can be done by a *tiling-like* code transformation, so that less data is accessed in a given loop iteration.
- **(L) Loop merging** can be done by fusing the corresponding loops.
- **(F) Function merging** is done by fusing the corresponding functions.

Example 2. (continued) For the reuse of `X[0]` in Fig. 1(b), the highest-level sections that are crossed are function-call sections. Therefore, bringing use and reuse closer together requires fusing `inproduct` and `sum`. This refactoring is

graphically represented by SLO as shown in Fig. 1(c). Furthermore, Fig. 1(d) shows that the other half of long-distance reuse pairs need to be optimized by fusing `inproduct` with the second call to `sum`. The need for these refactorings is not easily extracted from the bottleneck information shown by previous analyses (see Fig. 1(a)). The resulting optimized code runs 3 times faster on a 2.66Ghz Pentium4. The use of SLO as a tool to improve temporal locality has been discussed in more detail in [3]. In this paper, we present the underlying analysis.

Constructing the complete function-loop-hierarchy explicitly would result in prohibitive memory overheads, and prohibitive time overhead to find the highest-level section crossed. Therefore, we developed a data structure and accompanying algorithm to track the function-loop hierarchy for the *open reuse pairs*, i.e. those reuse pairs for which the use has occurred, but the potential reuse is still in the future, as presented in Sect. 2.

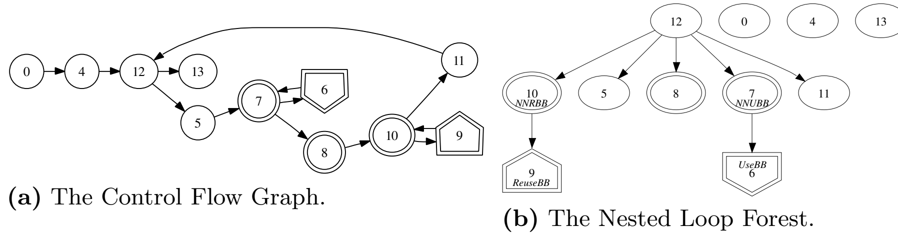
Furthermore, a sampling algorithm is introduced in Sect. 3, that speeds up the profiling of long-running programs. Previous sampling methods for cache measurement such as time and set sampling [1,4,5] only allow to control the sample rate, without guarantees about the resulting accuracy, since samples are taken systematically instead of randomly. In contrast, our algorithm takes samples randomly, which allows to derive a theoretically guaranteed accuracy.

The sampled profiling and analysis has been implemented in the GCC compiler. Using the data structures presented in Sect. 2, the analysis becomes doable for most SPEC2000 programs, albeit at a sometimes large memory overhead and a time overhead of a factor 1000 during profiling. As shown in Sect. 4, the sampling largely reduces memory overhead to an almost constant factor, and a time overhead of only a factor 5 for long-running programs compared to uninstrumented, fully-optimized execution. Using SLO, we were able to improve the locality of five already hand-optimized programs in SPEC2000, resulting in an average speedup of 2 on a number of different platforms. A comparison with related work is made in Sect. 5, indicating that the data structures and sampling techniques introduced in Sections 2 and 3 might also be profitable for a number of other program analyses. In Sect. 6, concluding remarks follow.

2 Compact Representation of the Function-Loop Hierarchy for Open Reuse Pairs

Pinpointing the refactoring for optimizing a given reuse pair is based on determining the highest level in the function-loop hierarchy where sections are crossed. Constructing the complete function-loop hierarchy would be prohibitively expensive. Therefore, a compact representation of the function-loop hierarchy for only the open reuse pairs has been developed. At each memory access, if the address has been accessed before, the highest-level sections crossed between the previous use and the current reuse is computed as follows.

First, the innermost function in the hierarchy that contains both use and reuse occur is determined. We call it the *Least Common Ancestor Function (LCAF)*



(a) The Control Flow Graph.

(b) The Nested Loop Forest.

Fig. 2. A Control Flow Graph and Nested Loop Forest. The basic blocks executed between use and reuse are indicated by double ellipses. For the considered reuse pair, the use occurs in basic block 6, while the reuse occurs in basic block 9.

since it is the least common ancestor of both use and reuse in the function-loop hierarchy. Then, the basic blocks executed between use and reuse are analyzed to find the outermost loops or loop iterations in the LCAF that are crossed between use and reuse, using the following definitions (see Fig. 2):

Definition 1. *The **Intermediately Executed Code (IEC)** of a given reuse pair is the set of basic blocks in the LCAF executed between use and reuse. The **UseBB** is the basic block in the LCAF containing the use; the **ReuseBB** contains the reuse.*

*The **Nested Loop Forest** of a function is a graph, with its basic blocks as nodes and edges going from loop headers to the basic blocks directly controlled by them. The **Outermost Executed Loop Header (OELH) of a basic block BB with respect to a given reuse pair** is the earliest ancestor of BB in the nested loop forest that has been executed between use and reuse.*

*The **Non-nested Use Basic Block (NNUBB)** is the OELH of UseBB. The **Non-nested Reuse Basic Block (NNRBB)** is the OELH of ReuseBB.*

When NNUBB=NNRBB, the highest level sections crossed are iterations of the loop for which the loop header is NNUBB. If NNUBB$\neq$NNRBB, fusion of the loops or function calls associated with the NNUBB and NNRBB are required (Each function call is located in a separate basic block).

When a data reuse is detected, the algorithm in Fig. 4 is used to calculate the LCAF, NNUBB and NNRBB. The three steps in the algorithm are discussed below; the data structure for representing basic blocks executed between use and reuse is illustrated in Fig. 3:

1. On every memory access and basic block transition, the global time is increased. For each data address, the time of its last access is stored in a hash-table. At time of reuse, the time of use is retrieved from that hash-table.
2. At run-time, a stack is maintained that reflects the call stack of the program. The LCAF is simply the latest function on the call stack that was called before the time of use.
3. For each frame in the stack, the list of executed basic blocks is maintained sorted in Most Recently Used order, e.g. Fig. 3. At the time of reuse, the

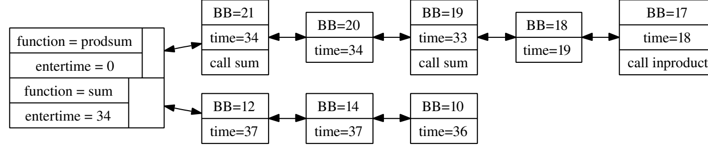


Fig. 3. Illustration of call stack, with an MRU list of basic blocks per call frame. The call stack for the code in Fig. 1 is presented. (current time=37, $t_u = 6, LCAF = prodsum, head(LCAF) = BB21, LAST(LCAF, t_u = 6) = BB17$).

1: $t_u \leftarrow$ time of last use.
2: $LCAF \leftarrow FirstFunctionBeforeTimeInStack(t_u)$
3: $head \leftarrow HEAD(LCAF); last \leftarrow LAST(LCAF, t_u); iec \leftarrow \{head..last\};$
$NNRBB \leftarrow OELH(iec, head); NNUBB \leftarrow OELH(iec, last)$

Fig. 4. Algorithm to find (NNUBB,NNRBB) for a reuse pair at time of reuse

ReuseBB is the head of the list of the LCAF, so NNRBB=OELH(head). The UseBB cannot be determined from this data structure. Also, it is not possible to record the UseBB at time of use, since the LCAF is unknown at that time. However, OELH(UseBB) is the same as the OELH of the last basic block in the list with an access time before the time of use, see lemma 1 below. Therefore, NNUBB=OELH(last).

Lemma 1. *The last basic block L in the MRU list with a time more recent than the time of use, has the same OELH as the UseBB.*

Proof. (a) If UseBB is executed once between use and reuse, then UseBB=L, so OELH(UseBB)=OELH(L).

(b) If UseBB is executed multiple times between use and reuse, then the trace of basic blocks between use and reuse has the following general form: ($UseBB_1 \cdots L \cdots UseBB_N \cdots ReuseBB$), where $UseBB_1$ is the dynamic instance of UseBB where the use occurs and $UseBB_N$ is the last instance of UseBB that is executed between use and reuse. L is executed between $UseBB_1$ and $UseBB_N$, so it must lay on some cyclic path from UseBB to itself. This cyclic path is part of a loop, and goes through the loop header which is part of the IEC. Since L and UseBB have a common loop header that is executed between use and reuse, they have a common loop header which is part of the IEC. As such, the OELH for both UseBB and L is the OELH of that common loop header. □

3 Overhead Reduction by Reservoir Sampling

While the data structures introduced in Sect. 2 enable to measure the data reuses and to pinpoint the required refactorings, it still has a large profiling overhead (factor 1000 on average) and a large memory overhead (over 2GB for some

SPEC2000 benchmarks), see Sect. 4. In this section, we introduce a sampling method to reduce both the time and memory overhead. The aim is to reduce the amount of reuse pairs for which the analysis in Sect. 2 is needed, while still guaranteeing the resulting error to be within a specified confidence interval.

3.1 Determining Number of Samples Needed for a Given Accuracy

Earlier sampling methods for memory traces, such as time sampling and set sampling [5] perform systematic sampling, i.e. the samples are not taken randomly. In contrast, we take uniform random samples of the reuse pairs in the program. This allows to perform the following analysis on the number of samples needed to obtain a given confidence interval.

The following assumptions are made: (1) A refactoring R is identified by the couple (NNUBB,NNRBB). (2) A refactoring R optimizes a given fraction R_f of all reuses for a given program run. (3) For a given reuse pair P, the probability that P is optimized by a refactoring R is R_P.

When the refactorings are determined for a sample of only n out of a total of N reuse pairs in a program run, the estimation for the fraction of reuse pairs that are optimized by refactoring R is $\hat{R}_P = c/n$, where c equals the number of reuse pairs in the sample that can be optimized by R. The corresponding large sample confidence interval [6] is, without assuming any distribution on R_P, $\hat{R}_P \pm z_{\alpha/2}\sqrt{\hat{R}_P(1-\hat{R}_P)/(n-1)}$. Therefore, the expected relative error within an α% confidence interval is $e = \left(z_{\alpha/2}\sqrt{\hat{R}_P(1-\hat{R}_P)/(n-1)}\right)/\hat{R}_P$ Rearranging this formula, the number of samples n that need to be taken to estimate R_P to within a certain error e for a given confidence interval α is: $n = \left(\left(z^2_{\alpha/2}(1-\hat{R}_P)\right)/e^2\hat{R}_P\right) + 1$

In the experiments below, we wanted to find for each refactoring that optimizes at least 0.01% of all reuse pairs, the true fraction of reuses it optimizes with a maximum relative error of 10% ($e = 0.1$) with a 95% confidence interval ($z_{\alpha/2} = 1.96$). Substituting these values in the above formula, shows n must be at least 3.841.217. Therefore, a uniform random sample of at least 3841217 reuse pairs is needed to accurately pinpoint the refactorings.

3.2 Profiling Based on Reservoir Sampling

The goal of the sampling is to collect the (NNUBB, NNRBB) for n reuse pairs that have been uniform randomly selected. At the start of the program, it is unknown how many reuse pairs there will be in the program run, so it's impossible to select a fixed sampling rate (apart from the fact that a fixed sampling rate does not result in a completely random set of reuse pairs). Therefore, the implementation is based on reservoir sampling [7]. At the start of execution, each reuse pair detected is inserted in the sample, until there are n reuse pairs. From then on, the formula presented in [7, Alg. L] determines the number of reuse pairs to be skipped, before the next reuse pair is chosen to replace one of the pairs

already present in the sample. As the program runs longer, the distance between reuse pairs that are sampled also grows larger. At the end of the execution, each reuse pair has an equal probability of being present in the sample [7].

In our implementation in the GCC compiler, every memory access, basic block and function call is instrumented. For each basic block and function call, a call to a run-time library is inserted that maintains the call stack data structure illustrated in Fig. 3. For each memory access, the following code is inserted:

```
++time;
if (HASH_ARRAY[addr & mask] != 0)  __full_check (addr, ref);
if (--accesses_to_next_traced == 0)  __track_address (addr, ref);
```

`time` is the global time. `HASH_ARRAY` is a large array that is indexed using the accessed address modulo the size of `HASH_ARRAY`. `HASH_ARRAY` contains non-zero entries for all addresses that are currently part of an open reuse pair that is sampled. `accesses_to_next_traced` is the number of accesses that need to be skipped until the next access that will be entered in the sample, according to [7, Alg. L]. When `HASH_ARRAY` is substantially larger than the number of samples n to be taken, and when the number of accesses already executed in the program run is substantially larger than n, both if tests are likely to be false. As a result, for most accesses in long running programs, the overhead of memory access instrumentation is just incrementing the `time` and `accesses_to_next_traced` variables and performing the two conditions in the if-test.

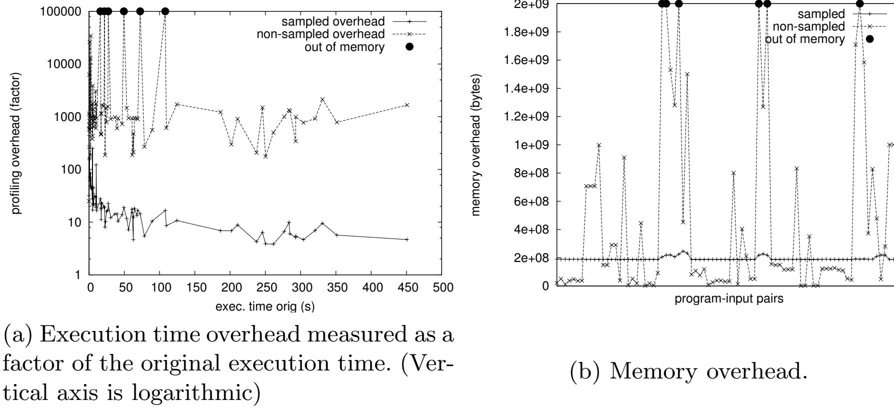
4 Experimental Results

We implemented the presented method in the GCC 4.1 compiler. The patch is available at `http://www.elis.ugent.be/~kbeyls/slo`. We used the SPEC2000 benchmarks, with all their available inputs, to evaluate the reduction in time and memory overhead due to the sampling. All profiling overhead tests were run on an AMD Opteron 1.6Ghz processor running Linux.

Fig. 5(a) shows the time overhead of profiling versus the original execution time of the non-profiled fully-optimized programs. It shows that profiling without sampling has a more or less constant overhead of about a factor 1000. Using sampling, the time overhead reduces to about 5 for long-running applications.

Fig. 5(b) shows the memory overhead. Using non-sampled profiling, a few runs lead to more than 2GB memory overhead, resulting in a program crash due to out of memory on the 32-bit platform. These results agree with those found by Fang et al. [8, Sect. 4.1] who instruments programs to measure the branch history between use and reuse: there are too many different paths executed between reuses to be able to store them in a reasonable amount of memory. Therefore, it seems that sampling techniques are required to practically measure the paths executed between reuses for arbitrary programs.

In contrast, the sampled profiling has an almost constant memory overhead of about 200MB, allowing to profile all SPEC2000 programs. Of those 200MB, about 150MB is consumed by the sample buffer containing 4 million reuse pairs, and the associated data. The slight fluctuation in memory overhead between



(a) Execution time overhead measured as a factor of the original execution time. (Vertical axis is logarithmic)

(b) Memory overhead.

Fig. 5. Time and memory overhead of sampled and non-sampled profiling for the SPEC2000 benchmarks

Table 1. Speedups on different platforms for five SPEC2000 programs after applying temporal locality optimizations based on suggestions made by SLO. The cache sizes of the largest cache level are indicated between parentheses for each platform.

	Speedup				**Analysis Time**	
	Pentium4 (512KB)	**Itanium (2MB)**	**Alpha (8MB)**	**Average**	**Non-Sampling**	**Sampling**
Art	4.11	1.54	1.16	2.39	12h13m	0h16m
Equake	1.10	2.93	3.09	2.30	59h33m	0h22m
VPR.route	1.51	1.40	1.41	1.44	18h32m	0h16m
Galgel	1.92	2.37	2.39	2.22	65h15m	0h24m
Applu	1.63	2.46	1.69	1.92	100h59m	0h28m

different program runs is due to the different number of basic block that are executed between use and reuse of those 4 million sampled reuse pairs.

We visually explored the analysis results using the SLO tool (available at `http://www.elis.ugent.be/~kbeyls/slo`) [3]. We optimized five SPEC2000 programs with a high cache miss rate by following the suggestions made by SLO. While some of the suggested refactorings could not be legally applied due to true data dependences, we still found a number of useful refactorings, resulting in an average speedup of 2 on a number of different platforms, see Tab. 1. The table also shows that thanks to sampling, the profiling time of these programs with reference input is reduced to less than 30 minutes for all programs.

Furthermore, SLO was also used to optimize the data locality of a video decoder that is being implemented in hardware, leading to a two-fold reduction of off-chip memory accesses. Since the video decoder is bandwidth-limited, this results in a doubled decoding frame rate [9].

5 Related Work

Performance Debugging Tools for optimizing Cache Behavior Too many profiling tools have been proposed to pinpoint cache bottlenecks in the source code to explicitly mention them all here. However, most of these tools focus on pinpointing the source lines, or data structures on which cache misses occurs. As illustrated in the introduction, our SLO tool aims at going further by pinpointing the refactorings that are needed to optimize the temporal locality.

An objective comparison of the usefulness of the different tools in finding good optimizations is hard, due to the human factor involved. Nonetheless, as far as we know, two other works have attempted to improve the data locality of Equake based on profiling results. In both [1] and [10], spatial locality is optimized by rearranging data layout, resulting in reported speedups of 1.4 and 1.24. In contrast, our method results in increased temporal locality, with an average speedup of 2.3. Both methods could be combined to further increase the overall locality and speedup. We are not aware of any earlier successful attempts of optimizing the temporal locality of the Art, VPR, Galgel and Applu programs.

Sampling Reuse Distance and Cache Simulation. The two most used methods to sample memory access traces for cache simulation are time sampling and set sampling [5]. In time sampling, a number of windows of consecutive memory accesses are traced, with large inter-window gaps of non-traced accesses. This method has its limitations for measuring long-distance reuses, since both the use and the reuse must be in the same window. Since there are billions of memory accesses between long-distance reuses in some applications, the windows should be billions of memory accesses large. In set sampling, only a subset of data addresses are sampled. For both time and set sampling, the accuracy cannot be determined theoretically, since the samples are not taken in a uniform random way. Typically, in order to reach less than 2% error, 10 to 20% of all accesses must be sampled [5]. While the calculation of error rates is not directly comparable, our sampling method allows to pinpoint all refactorings that optimize at least 0.01% of all reuses with at most 10% error by only sampling 0.09% of all accesses for the SPEC2000 programs.

Berg [1] uses the MMU in the processor to speed up the detection of reuses, resulting in a time overhead of only 40%. It is not clear how their method could be extended to also measure the code that is executed between reuses.

Other Program Optimization Strategies Based on Analysis of Data Reuses. In recent years, a number of techniques have been proposed that are based on profiling long-distance data reuse, e.g. inserting prefetch instructions selectively [8], inserting cache hints to improve replacement decisions [11], improving spatial locality [12], speculative memory disambiguation of memory instructions [13], optimizing the bandwidth usage in hardware implementations [9], predicting the execution time of programs [14], estimating energy consumption [15], detecting phases in program executions [16], etc. . The sampling proposed in Sect. 3 might be an interesting extension to these methods to reduce their profiling overhead.

6 Conclusion

Reuse path analysis is a new way to look at the locality behavior of a program. A method has been presented to suggest the required refactoring that improves temporal data locality for any given reuse pair. The algorithm and associated data structure make the implementation of the method practical for realistic applications. In addition, reservoir sampling of the reuse paths drastically reduces time and memory overhead. The techniques have been implemented in the GCC compiler and the visualizer SLO was developed to analyze the results. Using these tools, we optimized the temporal locality of five SPEC2000 programs, resulting in an average two-fold speedup on a number of different platforms.

References

1. Berg, E., Hagersten, E.: Fast data-locality profiling of native execution. In: SIGMETRICS. (2005) 169–180
2. Beyls, K., D'Hollander, E.H., Vandeputte, F.: RDVIS: A tool that visualizes the causes of low locality and hints program optimizations. In: ICCS. Volume 3515 of LNCS. (2005) 166–173
3. Beyls, K., D'Hollander, E.H.: Intermediately executed code is the key to find refactorings that improve temporal data locality. In: Computing Frontiers. (2006) 373–382
4. Martonosi, M., Gupta, A., Anderson, T.: Effectiveness of trace sampling for performance debugging tools. In: ACM SIGMETRICS. (1993)
5. Uhlig, R.A., Mudge, T.N.: Trace-driven memory simulation: a survey. ACM Comput. Surv. **29**(2) (1997) 128–170
6. Walpole, R., Myers, R.: Probability and Statistics for Engineers and Scientists. Prentice Hall (1993)
7. Li, K.H.: Reservoir-sampling algorithms of time complexity o(n(1 + log(n/n))). ACM Trans. Math. Softw. **20**(4) (1994) 481–493
8. Fang, C., Carr, S., Önder, S., Wang, Z.: Path-based reuse distance analysis. In: Compiler Construction. Volume 3923 of LNCS. (2006) 32–46
9. Devos, H., Beyls, K., Christiaens, M., Campenhout, J.V., D'Hollander, E.H., Stroobandt, D.: Finding and applying loop transformations for generating optimized FPGA implementations. (Transactions on HiPEAC) submitted.
10. Buck, B.R., Hollingsworth, J.K.: Data centric cache measurement on the intel itanium 2 processor. In: Proceedings of SuperComputing. (2004)
11. Beyls, K., D'Hollander, E.H.: Generating cache hints for improved program efficiency. J. of Systems Architecture **51**(4) (2005) 223–250
12. Zhang, C., Ding, C., Ogihara, M., Zhong, Y., Wu, Y.: A hierarchical model of data locality. In: POPL. (2006)
13. Fang, C., Carr, S., Onder, S., Wang, Z.: Instruction based memory distance analysis and its application to optimization. In: PACT. (2005)
14. Marin, G., Mellor-Crummey, J.: Cross-architecture performance predictions for scientific applications using parameterized models. In: SIGMETRICS. (2004)
15. VanderAa, T., Jayapala, M., Barat, F., Corporaal, H., Catthoor, F., Deconinck, G.: Instruction and data memory energy trade-off using a high-level model. In: ODES. (2004)
16. Shen, X., Zhong, Y., Ding, C.: Locality phase prediction. In: ASPLOS-XI. (2004) 165–176