A Generic Tool Supporting Cache Design and Optimisation on Shared Memory Systems

Martin Schindewolf[1], Jie Tao[2*], Wolfgang Karl[3] and Marcelo Cintra[4]

[1]Universität Karlsruhe (TH), Zirkel 2, 76131 Karlsruhe, Germany
schindew@ira.uka.de
[2]Universität Karlsruhe (TH), Zirkel 2, 76131 Karlsruhe, Germany
jie.tao@iwr.fzk.de
[3]Universität Karlsruhe (TH), Zirkel 2, 76131 Karlsruhe, Germany
karl@ira.uka.de
[4]University of Edinburgh, Mayfield Road, EH9 3JZ Edinburgh, United Kingdom
mc@inf.ed.ac.uk

Abstract: For multi-core architectures, improving the cache performance is crucial for the overall system performance. In contrast to the common approach to design caches with the best trade-off between performance and costs, this work favours an application specific cache design. Therefore, an analysis tool capable of exhibiting the reason of cache misses has been developed. The results of the analysis can be used by system developers to improve cache architectures or can help programmers to improve the data locality behaviour of their programs. The SPLASH-2 benchmark suite is used to demonstrate the abilities of the analysis model.

1 Motivation

As Moore's Law — the number of transistors per die doubles every 18 months — still holds, higher clock rates for the cores are feasible due to shorter signal distances. Higher processor speed demands faster access to the requested data. A computer system can not exploit its computing capacity if the processor spends time waiting for the data to arrive. Subsequently, the performance increasingly relies on the efficient use of the caches. Therefore, a high cache hit rate is indispensable. The common approach is to design caches whose performance is acceptable for a wide range of applications, as this concept yields the best trade-off between performance and costs. Anyways, if only a few applications have to be considered, an application specific cache design allows for better performance and improved energy efficiency. The idea is to perform a cache miss analysis and use the results to guide the user through the optimisation process. This paper presents a tool, that helps system developers to discover application specific cache parameters (such as cache

*Dr. Jie Tao is now at the Institute for Scientific Computing, Forschungszentrum Karlsruhe, Hermann-von-Helmholtz-Platz 1, 76344 Eggenstein-Leopoldshafen, Germany

size, line size). Further, programmers are supplied with the cause of the cache miss for performing source code optimisation (see section 4.2). A concise analysis model is designed and implemented. The analysis is based on the cache event trace acquired from the SIMICS simulation environment. Cache misses are classified according to their cause. Analysis results help the designers to deduce the best cache configuration for individual applications. Furthermore, an interface to an existing visualisation tool is implemented, enabling a graphical representation of the analysis results. Information in this form allows the human user to easily detect the optimisation target and identify bottlenecks.
The remainder of this paper is organised as follows. Section 2 first gives a brief introduction to related work. This is followed by a detailed description of the proposed analysis model and its implementation in section 3. Evaluation results are presented in section 4. The paper concludes in section 5 with a brief summary.

2 Related Work

Any cache optimisation, either targeting the cache architecture or the source code, relies on the knowledge about the cache miss reasons. During the last decades, cache miss analysis is in the focus of computer architecture research. Dubois et al. [DSR+93] present an approach towards cache miss estimation on multi-processor systems. Their work defines cache miss metrics assuming infinite caches. Consequently, the caches need no replacement strategy and reveal no conflict or capacity misses. Under these circumstances, their *Pure True Sharing Misses* are the same as the *true sharing misses* and the *true sharing invalidation misses* (definitions are given in section 3.1.1). Beyls and D'Hollander [BD01] introduce the *reuse distance* as a concept for cache miss estimation. This work implements the reuse distance concept and uses it to distinguish between cold, conflict and capacity misses. Jeremiassen and Eggers [JE95] demonstrate how compiler techniques avoid false sharing misses. The techniques identify shared data and use padding to extend it to the full cache line size.
Our approach combines the methodologies of the first and the second work. Similar to the first work, we consider multi-processor systems and, hence, also calculate coherence misses. Furthermore, we incorporate the cache metrics of the second work to target realistic cache architectures. An accurate algorithm was designed for this computation.

3 The Cache Model

The base of the analysis model is a cache event trace that records every cache event. The *g-cache* module of SIMICS generates this event trace. The *parse simics conf* tool captures the cache configuration and delivers it to the analysis tool. The analysis tool processes every cache event and classifies the misses.

For accuracy and flexibility we used SIMICS to provide the cache event trace. SIMICS is an efficient and instrumented system level instruction set simulator [Rev06]. SIMICS

simulates the hardware running the operating system and the application. Caches are configurable modules called *g-cache*.

3.1 Tool Chain

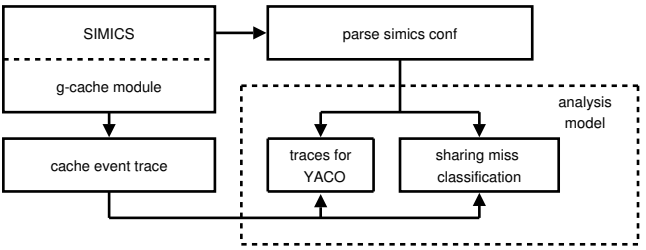

Figure 1: Tool chain interfacing with SIMICS.

Figure 1 depicts the interface between SIMICS and the developed tool, where the left side represents the simulation environment and the right side shows our own work. The *g-cache* module of SIMICS was slightly modified in order to capture the cache events, which are written to a trace file. Each cache event is processed by the analysis model. For this task the analysis model also needs the cache configuration parameters which have been used by SIMICS to generate the trace. This information is delivered by the *parse simics conf*-tool. The main work of the analysis model is to find out the reason of each cache miss. A statistical output is delivered to the user for comparing the cache behaviour of different configurations. Analysis results are also recorded in traces required by the existing visualisation tool YACO. The visualisation presents the analysis results in a user-understandable way. This helps the programmer to detect access bottlenecks and optimisation strategies.

3.1.1 Cache Miss Categories

Traditionally, cache misses are classified as cold, conflict and capacity misses [HS89]. Cold misses are caused by the first reference. Capacity misses occur when the cache is smaller than the working set size, while conflict misses occur due to mapping conflicts. For multi-processor machines the coherence problem has to be solved - this results in coherence misses. When many processors execute an application, data are shared. A write invalidate protocol invalidates the modified, shared data, thus, causing cache misses. To see whether the invalidation is necessary, the sharings are differentiated in *true sharing* (at least two processors access the same data) and *false sharing*.

A sharing miss is defined straightforwardly as a miss occurring on an address having a block address which has been shared. *True sharing misses* are an inevitable effect of parallel execution. However, *false sharing misses* shall be eliminated. Sharing misses are typically identified by examining whether the miss is caused by an earlier invalidation.

Actually, this calculation is not exact, because the replacement strategy might replace the line before the miss. Then the miss must be attributed to the replacement strategy and not classified as a coherence miss. This leads to our refined definition of coherence miss:

true sharing invalidation miss (tsim): *true sharing miss* that would not have been replaced by the local replacement strategy before the miss.

false sharing invalidation miss (fsim): analogue to the *true sharing invalidation miss*.

3.1.2 Miss Classification Implementation

In order to classify each cache miss, the cache event trace is processed. The following subsections give a short description on the implementation of the algorithms.

Recognization of Cold Miss
Each processor records every access to a memory address in a linked list. Subsequently, the first access to a block address is not found in this list. Therefore, this block address has not been accessed before by the corresponding processor. Hence, a *cold miss* is detected.

Detecting Conflict and Capacity Miss
Introduced in [BD01], the *reuse distance* is the concept to distinguish *conflict* and *capacity misses*. It is defined as the number of unique block addresses between two references to the same block. We implemented the *reuse distance* as follows. Every block address is associated with a reuse distance counter and a time stamp with the date of the last reference. Every time a miss occurs, the linked list is traversed and the time stamps of the last reference of the entries and the time stamp of the last reference of the miss are compared. If the time stamp of the entry is greater than the time stamp of the miss, the entry's reuse distance counter is increased by one. This is done because the last reference to the miss occurred before the last reference to the entry. Therefore, the block address of the miss is distinct from the other block addresses accessed since the last reference to the block address of the list entry.

The classification of *conflict* and *capacity miss* compares the reuse distance counter of the miss. If the reuse distance counter is smaller than the number of cache lines, the miss is a *conflict miss*. Otherwise, the miss is a *capacity miss*.

Sharing Invalidation Miss
For recognising the *sharing invalidation miss*, we apply another definition: *set reuse distance*. Based on the *reuse distance*, the *set reuse distance* is also the number of unique block addresses between two accesses to the same block, but only blocks mapped to the same set with the observed address are counted. This value is used to exclude misses that are caused by the replacement strategy of the cache.

If a miss is perceived on a sharing, the *set reuse distance* of that block address is compared to the number of lines in the set (associativity). If the associativity is equal or less than the set reuse distance, then this block address would already have been replaced by the LRU

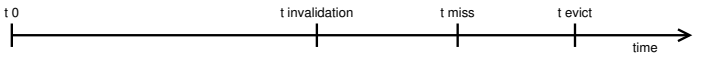

Figure 2: Timeline illustrating the sharing invalidation miss.

strategy, resulting in a *conflict* or *capacity miss*. Otherwise, a *sharing miss* is detected because the replacement strategy would not have evicted this block address. Figure 2 illustrates the *sharing invalidation miss*. The following terms are applied:

- t_0 represents the last reference to this block address which resets the *reuse distance* and the *set reuse distance* instances of this block address. Afterwards, for every different block address referenced, the *reuse distance* is increased by one and if the block belongs to the same set the *set reuse distance* is accumulated as well.
- $t_{invalidation}$ represents the time of the invalidation. The sharing of the block address is classified and saved.
- t_{miss} represents the time at which a miss occurs on that block address.
- t_{evict} is the point in time, where the LRU strategy would have replaced this block address.

If $t_{miss} >= t_{evict}$, a replacement strategy miss is detected, because the associativity is equal or less than the *set reuse distance*. On the contrary, if $t_{miss} < t_{evict}$, which correlates with the *set reuse distance* being less than the associativity, a *sharing invalidation miss* is detected (Figure 2).

4 Evaluation

Parameter	Value
L1 Line Size	32 Bytes
L1 Associativity	2-way
L1 Replacement Policy	Least Recently Used
L1 Write Policy	Write Through
L1 Allocate Policy	Write Allocate
L2 Line Size	32 Bytes
L2 Associativity	4-way
L2 Replacement Policy	Least Recently Used
L2 Write Policy	Write Back
L2 Allocate Policy	Write Allocate

Parameter	private L2	shared L2
Number of Processors	8	8
L1 Number of Caches	8	8
L1 Size (each)	4 KBytes	4 KBytes
L1 Number of Lines	128	128
Coherency Protocol	MESI	MESI
L2 Number of Caches	8	1
L2 Size (each)	128 KBytes	1024 KBytes
L2 Number of Lines	4096	32768
Coherency Protocol	MESI	None

Table 1: Common cache parameters (left) and case specific parameters for 8 processors (right).

In order to verify the functionality, the cache analysis model has been evaluated using the SPLASH-2 benchmark suite. This section first briefly describes the benchmarks and the results obtained with the multi-processor configuration. Then we show a sample view of the visualisation.

4.1 Results with SMP-architectures

For evaluating the analysis model and achieving valuable conclusions for cache optimisation, several experiments have been conducted. Throughout these experiments, the caches are configured as shown in Table 1. This setup allows for examining the SPLASH-2 Benchmark Suite [spl] towards scaling performance on SMP-architectures. SPLASH-2 aims at evaluating cache-coherent shared memory architectures. In order to adapt the SPLASH-2 programs to the x86 architecture and SIMICS the hints given by [Bar, Hei] are performed. The m4 macros [Sto], developed by Bastian Stougie [Sto03], are used to parallelise the benchmark.

Corresponding to the existing and emerging multi-processor cache designs, this experiment examines different structures of the level 2 cache(s), **private** (each processor has an exclusive level 2 cache) and **shared** (one level 2 cache for all processors). Table 1 depicts the applied configuration, with the left side for the common parameters and the right side for the L2 specific one (a sample configuration with 8 processors). This work, as well as many other studies concerning cache miss estimation, uses normalised miss rates: $miss_{rate} = \frac{miss_{sum}}{access_{sum} * number\ of\ processors}$.

4.1.1 Overall Performance

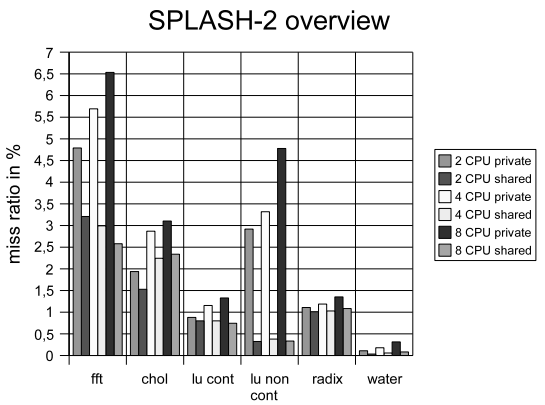

Figure 3: Miss rates grouped by SPLASH-2 programs.

Figure 3 shows the overall miss rate of the benchmarks simulated with 2, 4 and 8 processors. Two adjacent bars belong to the same number of CPUs, the left bar gives the miss ratio for privately owned caches, whereas the right bar refers to the shared case. Surprisingly, for all applications with all processor numbers the shared level 2 cache yields better performance. The improvements of a shared level 2 cache over private level 2 caches, calculated by, $\frac{miss_{private} - miss_{shared}}{miss_{private}}$, range from 9% (LU with continuous blocks measured using 2 CPUs) up to 89% (LU with non continous blocks using 2 CPUs).

For a better understanding of the reasons for the observed results, the misses are further classified (according to section 3.1.1).

4.1.2 Miss Characteristics

The analysis model computes accurately the number of misses in each miss category. This allows us to observe the cause of every cache miss. Figure 4-6 show sample results with the *cholesky*, the *water-n*^{2}, and the *lu with non continuous blocks* programs.

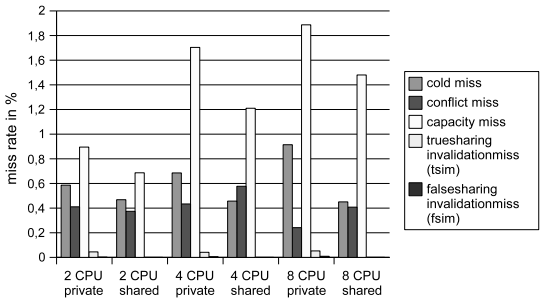

Figure 4: *cholesky* program for shared and private level 2 caches.

Cholesky
The *cholesky* program performs a matrix decomposition using the numerical cholesky method [WOT$^+$95]. As shown in Figure 4, the reduced cold and capacity miss rates mainly contribute to the better performance with shared level 2 caches. The conflict misses, on the other hand, decrease the performance of the shared level 2 caches on 4 and 8 processors.

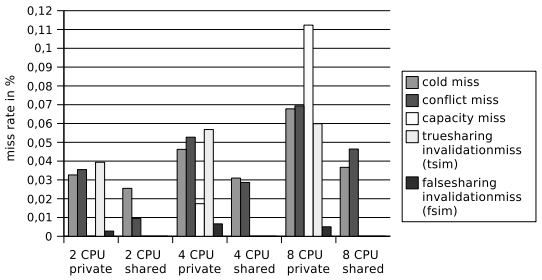

Figure 5: Water n^2 benchmark for shared and private level 2 caches.

Water-n^2
The *water-n*^{2} benchmark simulates a multi dimensional body [WOT$^+$95]. As shown in

Figure 5, the private caches show disadvantageous behaviour concerning the number of coherence misses and the increasing capacity miss ratio, which rises from ∼0% to 0,11%. According to [WOT+95], the second working[1] set exceeds the capacity of the private caches. As the private caches become smaller the more processors are used, while the data set size stays the same, the second working set does not fit in the caches, resulting in capacity misses. In the shared cases the second working set does not exceed the cache capacity, as capacity misses do not occur. Additionally, cold and conflict misses are also decreased with the shared cache.

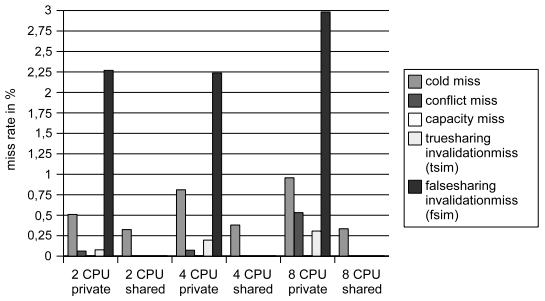

Figure 6: LU with non continuous blocks benchmark for shared and private level 2 caches.

LU with non continuous blocks

LU with non continuous blocks performs a matrix decomposition using the LU factorisation. As shown in Figure 6 the only perceived misses using a shared second level cache are cold misses with a miss rate of around 0,35%. The private case is dominated by coherence misses, precisely false sharing invalidation misses, that yield rates between 2,27% and 2,98%. The true sharing invalidation miss rate is between 0,08% and 0,32%. Thus, a wrong cache line size is indicated. Compiler techniques, padding data to full cache line sizes, are indicated to prevent false sharing invalidation misses. Capacity misses are not detected, which is due to the small benchmark working set size.

Improved Cache Configuration

The analysis of the *cholesky* and the *water-n²* benchmarks reveals conflict misses. In order to reduce the conflict misses and improve the performance, we increased the level 2 cache associativity from 4-way to 8-way set-associative and repeated the simulation. The results of the *water-n²* program are shown on the left hand side of Figure 7 and the *cholesky* program on the right hand side. The cache miss rate of the *water-n²* program improves by at least 5.3% (4 CPUs with private caches) whereas the *cholesky* program improves by at least 1.5% (8 CPUs with private caches) compared to the 4-way set-associative level 2 caches. The increased associativity has a greater effect on the shared level 2 caches as the

[1] working set 2 corresponds to the second knee of the function in cache size and miss rate

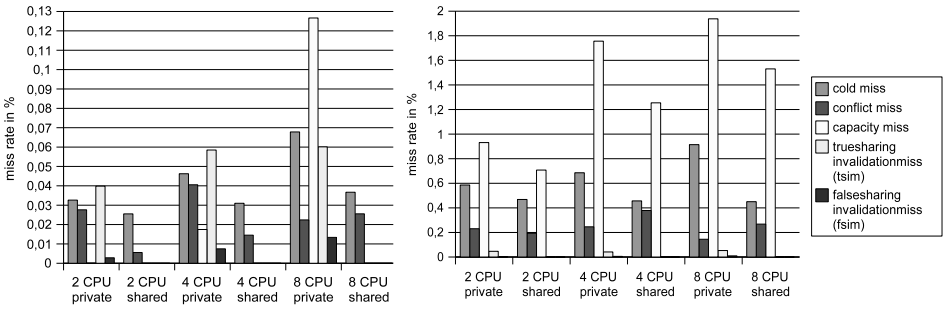

Figure 7: Water n^2 (left) and cholesky (right) with 8-way set-associative level 2 caches.

conflict misses make up a larger fraction of the overall miss rate. Therefore, the miss rate benefits more from the decreased number of conflict misses.

Summary
Overall, the shared architecture generally benefits from less cold and capacity misses. The former can be explained by the fact that shared data only causes one cold miss with the processor first accessing it. For the latter a larger cache is available for the working set. In addition, shared caches have no coherence misses. Further, the analysis tool is shown to be useful. As expected, increasing the cache associativity causes the number of conflict misses to decrease.

4.2 Visualisation

The analysis results can also be applied to understand the cache and program access pattern and further achieve an optimised application. For this, YACO is used for representing the results. YACO [QTK05] is a cache visualisation tool specifically designed for cache optimisation. It uses a set of several graphical views to guide the user to detect the problem, the reason, and the solution.

Figure 8 is a sample view used to highlight cache critical variables, i. e. the access bottlenecks. The simplicity of the graphical representation helps the programmer to clearly identify the bottlenecks throughout program execution. The relation between the name and the miss rate points the programmer to the variables worth optimising. The *fft* program shows that except **umain** all other main data structures have to be optimised. In the next step, programmers can use YACO's data access and cache views to analyse the access pattern and further to detect the optimisation strategies. Optimisation examples are found in [QTK05].

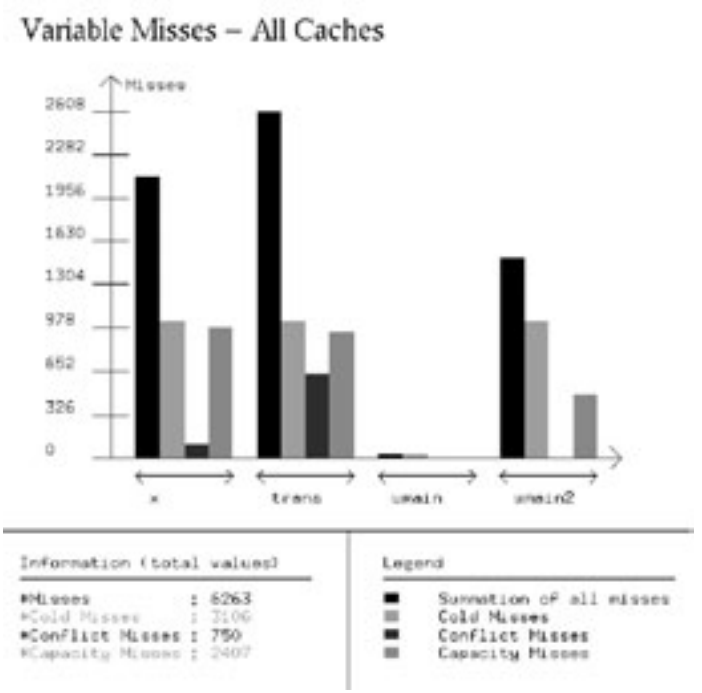

Figure 8: YACO's misses per variable.

5 Conclusion

This work uses an analysis approach to investigate the feature of cache misses on multiprocessor machines. The g-cache module of SIMICS is used to create a cache event trace. A miss classification model is applied to the cache event trace in order to distinguish cold, conflict, capacity, and sharing invalidation misses. A component for generating traces of performance facts, which can be delivered to an existing visualisation tool for graphical presentation of cache bottlenecks, is implemented as well. The following results are achieved. For all considered benchmarks the shared level 2 cache is the better choice as it improves the cache miss rate. The overall advantage of the shared level 2 cache is the lack of coherence misses.

The best example for an improved cache miss rate by eliminating coherence misses is the *lu with non continuous blocks* benchmark. The coherence misses reveal the false sharing of cache lines.

Other programs as the *water-n*^{2} yield better miss rates in the shared case, because the working set size exceeds the private caches resulting in a higher capacity miss rate.

The results obtained from the *cholesky* program indicate that the most benefit is drawn from an increased cache size, as the coherence miss rates are negligible. In the shared

cases conflict misses are visible. As shown in section 4.1.2 a higher cache associativity reduces the conflict misses and increases the performance.

Section 4.2 and section 4.1.2 show how the developed tool can guide the user to an improved adjustment of application and cache.

References

[Bar] Ken Barr. http://kbarr.net/splash2.html. Online; accessed January 2, 2008.

[BD01] K. Beyls and E. D'Hollander. Reuse distance as a metric for cache behavior. In *PDCS '01: Proceedings of the Conference on Parallel and Distributed Computing and Systems*, pages 617–662, August 2001.

[DSR+93] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The Detection and Elimination of Useless Misses in Multiprocessors. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, 1993.

[Hei] Wim Heirman. http://trappist.elis.ugent.be/~wheirman/simics/splash2/. Online; accessed January 2, 2008.

[HS89] M. D. Hill and A. J. Smith. Evaluating Associativity in CPU Caches. *IEEE Trans. Comput.*, 38(12):1612–1630, 1989.

[JE95] Tor E. Jeremiassen and Susan J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 179–188, New York, NY, USA, 1995. ACM Press.

[QTK05] B. Quaing, J. Tao, and W. Karl. YACO: A User Conducted Visualization Tool for Supporting Cache Optimization. In *HPCC '05: High Performance Computing and Communications: First International Conference*, volume 3726 of *Lecture Notes in Computer Science*, pages 694–703. Springer, September 2005.

[Rev06] Virtutech AB, Nortullsgatan 15, SE-113 27 STOCKHOLM, Sweden. *Simics User Guide for Unix*, February 2006. Simics Version 3.0.

[spl] SPLASH-2: Stanford Parallel Applications for Shared Memory. http://www-flash.stanford.edu/apps/SPLASH/. Online; accessed January 2, 2008.

[Sto] Bastiaan Stougie. http://kbarr.net/files/splash2/pthread.m4.stougie. Online; accessed January 2, 2008.

[Sto03] Bastiaan Stougie. Optimization of a Data Race Detector. Master's thesis, Delft University of Technology, October 2003.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, New York, NY, USA, 1995. ACM Press.