Towards Dynamic Cache and Bandwidth Invasion

Carsten Tradowsky[1], Martin Schreiber[2], Malte Vesper[1], Ivan Domladovec[1], Maximilian Braun[1], Hans-Joachim Bungartz[2], and Jürgen Becker[1]

[1] Institute for Information Processing Technology, Karlsruhe Institute of Technology
{tradowsky,becker}@kit.edu,
{malte.vesper,ivan.domladovec,maximilian.braun}@student.kit.edu
[2] Scientific Computing in Computer Science, Technische Universität München (TUM)
{martin.schreiber,bungartz}@in.tum.de

Abstract. State-of-the-art optimizations for high performance are frequently related to particular hardware parameters and features. This typically leads to optimized software for execution on particular hardware configurations. However, so far, the applications lack the ability to modify hardware parameters either statically before execution of a program or dynamically during run-time.

In this paper, we first propose to utilize the flexibility of underlying invasive hardware to adapt to the needs of the software. This enables us to ask for more than just processing power by, e. g., requesting particular cache parameters that correspond to certain application properties. The adaptive hardware architecture therefore is able to dynamically reconfigure itself dependent on the availability of the resources in order to achieve an optimized working point for each application scenario. Secondly, we present requirements for dynamical scheduling of computing resources to resource-competing applications. This becomes mandatory to account for memory-access characteristics of concurrently executed applications. We propose consideration of such characteristics with bandwidth-aware invasion.

With this novel approach, we are able to show that dynamic hardware and software co-design leads to improved utilization of the underlying hardware resulting in higher throughput in means of efficiency such as application-throughput per time-unit.

Keywords: invasive computing, adaptive, application-specific microarchitecture, reconfigurable cache, compute-bound, memory-bound, HPC.

1 Introduction

Invasive computing is regarded as the paradigm of building a platform that has a multitude of heterogeneous resources. It allows for dynamic allocation and utilization depending on the resources' availability to solve various computing problems. Due to additional and changing demands of both hardware and software requirements, respective changes have to be considered for each of those

D. Goehringer et al. (Eds.): ARC 2014, LNCS 8405, pp. 97–107, 2014.

building blocks of the invasive hardware and software architecture. Both upcoming sections discuss the different views on dynamic adaptive hardware and software.

1.1 Dynamic Adaptive Software

Following the trend towards many-core systems and extrapolating the number of cores exceeding one thousand during the next decades, dynamically changing resources get mandatory for energy-efficiency, throughput optimizations and further upcoming requirements like reliability or security. This leads to several new demands to the application developer to be able to express this changing resource demands.

Applications should provide information on their requirements, targeting at improved application- and memory-throughput, real-time requirements, energy efficiency, etc. Different application scenarios such as multi-resolution image processing and dynamic adaptive hyperbolic simulations [1,6] set up the requirements from static resource requirements towards dynamic resource scheduling. With such dynamically changing requirements of applications, further referred as different *phases*, it would be beneficial for applications making particular hardware sets available to other applications or deactivating them for energy efficiency reasons. With memory expected to be one of the main bottlenecks in a few years, dynamic reconfigurable memory related components such as caches and memory bandwidth is our main focus for this work.

Running only a single program on a particular number of cores, numerical applications such as a matrix-matrix multiplication (MMul) are typically (a) optimized to a particular cache-line size *during compile time*, (b) are *not able to consider changing cache-line sizes* and (c) are *not able to adapt the hardware to their requirements*. To our knowledge, those dynamic optimizations so far target only non-adaptive hardware resources in current HPC studies[1].

1.2 Dynamic Adaptive Hardware

Adaptive application-specific processors lead to higher efficiency by dedicated support of applications [9]. E. g., *i*-Core provides enhanced flexibility within the microarchitecture itself by enabling the application developer's interaction with the microarchitecture [19].

In addition to the standard setting of the hardware, the software developer should provide extra input to further parametrize the *i*-Core. This enables the application developer to pass on its knowledge about the application to the *i*-Core in order to achieve an optimized processor configuration. We refer to this as resource aware programming, which can as well include configurations oo caches.

To our best knowledge, a parametrizable cache was not considered so far. This constraints the degree of freedom when defining the sizes of, e. g., the dedicated level one caches in an optimized manner.

2 Related Work

2.1 Dynamic Adaptive Memory

In the classical model of an n-way associative cache, parameters such as line length, degree of associativity and number of sets exist. These parameters are coupled by the total cache size and thus constrained by available chip resources. Until now, most evaluations on memories have only been done using special simulators and models neither considering silicon implementation possibilities nor overhead. This is necessary for invasive computing since one of the key points is resource sharing between several concurrent independent computational problems.

For this work, the approach presented in [10] and [13], to tune the cache for inner loops before the start of each run seems appealing and will be used to get an overview of the benefits of adaptation. At first glance, it seems obvious that the highest associativity (fully associative cache) would yield best performance since the forced cache misses can be minimized. However, the gain by increasing associativity diminishes vastly after four or eight while the hardware expenditure keeps rising [5,20]. Another side effect of increasing the set size is mapping larger memory area to the set. This reduces the benefit of having more possible locations for element storage. With a variable fixed cache size, decreasing this cache size by deactivating particular sets is expected to yield power benefits but should only be considered with negligible impact on application's performance.

In this work, we consider a fixed cache size as a given number of sets being defined by line length and associativity. The three performance related keys compensating reduced cache size are *prefetching*, *dense storage* and *temporal locality*. While prefetching exploits local spatiality by pre-emptive loading of data, dense storage refers to storing a particular amount of data in a small memory area. A third effect to be exploited is temporal locality, which refers to data usually being accessed multiple times in a short time frame. On the one hand, cache-oblivious algorithms (e. g., matrix multiplication [8] and dynamic adaptive simulations [3,18]) are likely to benefit of dynamic adaptive caches optimized to their particular demands. On the other hand, algorithms not being able to exploit the access locality (e. g., dot product), would improve the performance of cache-oblivious algorithms by sharing the cache resources with them.

Existing work on reconfigurable caches [14,7] uses only simulation models so far and thus stays on the hypothetical side from the hardware point of view. Besides these parameters, there are further additions to a cache that leverage the same effects introduced above. They include cache-assists (prefetch buffer, victim cache) and way management.

Prefetch Buffer: A prefetch buffer[10] is a FIFO, into which data following the location of the last miss is loaded. On the next miss the prefetch buffer will be queried before the request goes to a lower memory hierarchy level.

Victim Cache: In case of replacement, the data being replaced is stored in a small fully associative cache, which is searched in case of a miss. This is a mixture of temporal and spatial locality and works by increasing the virtual cache density. An evaluation of reconfigurable combinations of a victim cache and prefetch buffer is given in [10].

Way Management: Way management [11] introduces control bits for every line and assumes a mixed instruction and data cache. The control bits decide whether a way is writable for data or for instructions. Thus the size of the instruction and data cache can be changed and parts of the memory can be shared. Furthermore, some data can be frozen in cache by locking the line completely.

Replacement Strategies: Different replacement strategies [12] appear to be almost orthogonal to the parameters of line length and associativity. However, there are corner cases where different replacement strategies yield better results.

2.2 Application Requirements for Dynamic Memory

Without knowledge of applications and their performance, no appropriate runtime decisions can be undertaken to optimize the hardware resources towards software requirements. In order to differentiate between particular requirements of applications, we start with taxonomy of representative state-of-the art algorithms with respect to the memory-related requirements.

Bandwidth Limited Applications: Typical memory access patterns for bandwidth limited applications are streams, stencils and in general a relatively small computation / memory access (CM) ratio for current architectures. Considering a *dot product* [16] under the assumption that the sum is kept in a register, two vector components have to be loaded followed by two computations of multiplication of both values and adding the result to the value in the register with CM $= 2 : 2 = 1$. *Stencil operations* are frequently used in image processing for border detection and scientific computing for iterative solvers [4]. Considering a simplified sparse 2D stencil operation computing second order derivatives with a stencil size 3x3, five values have to be loaded, each followed by a multiplication with the stencil value assumed to be available in a register and an add operation. Using blocking techniques for cache-reutilization and assuming a single boundary-data for blocks of size $\sqrt{S} \times \sqrt{S}$ still stored in cache, this leads to CM $= 6 : 5 \approx 1.2$

Compute Bound Algorithms: We consider numerical quadrature of a computational intensive function [2]. With frequent evaluation of such functions with $n >> 1$ instructions with higher order quadrature formula, those computations are clearly compute bound due to avoidance of data access assuming that all instructions to evaluate the function fit into L1 instruction-cache.

Latency Bound Algorithms: Unpredictable access to memory occurs especially with interactively driven computations such as steering, image editing as well as spatial-residual aware iterative solvers [17]. Since the access occurs randomly, those algorithms are unlikely to fully exploit cache features. Those classes

of unpredictable algorithms depend on the dynamically changing memory access patterns itself. Therefore no clear statement on memory dependency can be given and those algorithms are not further evaluated in this work.

3 Dynamic Scheduling and Adaptive Hardware

We propose a novel approach to reconfigure particular parts in hardware, which so far was only statically exploitable to the software developers. Currently, the concept involves a model of cache tiles. Depending on control signals, these tiles either form larger memory sections for deeper ways or are used in parallel as different ways. Each tile incorporates the control logic and can store tags. The implementation of replacement strategies and the cache assistant is considered as orthogonal.We consider it reasonable to store line associated management information in the tiles, since this memory grows automatically with the addition of tiles and its connections are managed more naturally if the tiles are dynamically assigned to different cores in a later step.

Our partitioning of the cache puts the actual memory of data and management information in one module and control logic like fetch on miss and replacement in another. This hides the choice on reconfiguring the number of ways or size from the control module and sets up new tasks: deciding where to put the reconfiguration management, the logic ensuring that the data is in the right places after changing the size or associativity. On the one hand, it should be fairly general. On the other hand, detailed knowledge about the memory layout could help to speed up the implementation. Cache coherency will have to be covered once we cross the single-core boundary. This will open up new degrees of freedom, such as partly shared caches or dynamical redistribution of cache, which is one of the main reasons why we consider re-sizing.

Runtime reconfiguration leads to the issue of changing data layout in the cache, introducing the need to reorder data before continuing. Reconfiguration is shown to be a feasible process and provides a solution that flushes half the cache and realigns the remaining entries in cache for a change in associativity [15]. It is claimed that one would need a buffer of half the cache size to fully reorder entries in the cache on an associativity increase [14]. We expect to circumvent this with our tile approach.

We present an extension of the invasive programming constructs with support of invadable memory hierarchy. This leads to modification of the cache within the processor depending on application-specific requirements. With dynamically changing number of resources for invasion, interfaces have to be provided by the application developer for distributed memory and by an invasion-safe programming style on shared memory systems to assure stability. For our invadable memory hierarchy, changing resources do not change the reliability as long as the cache-coherency among processors is guaranteed. The worst-case scenario is a severe slowdown, but no stability issues. Due to optimizations on requested hardware layouts, no multiplexing of the claimed resources is allowed.

4 Case Study on Potentials of Cache and Bandwidth-Aware Invasions

We first present a case study of varying cache parameters on representative algorithmic kernels. Secondly, we present required extensions of resource managers for concurrently executed memory- and compute-bound applications.

4.1 Variation of Cache Parameters

For this case study, we use an inner-loop blocked matrix-matrix multiplication (MMul). During the execution of the benchmark, the cache is invalidated after each run of the outer loop. For evaluation, we use a Xilinx XUPV5-lx110t prototyping board. We use the Gaisler Leon3, which has two sets of $8\,kB$ ($16\,kB$) of instruction cache and four sets of $4\,kB$ ($16\,kB$) of data cache. This enables the use of different cache parameters that can be defined at design time. Our *baseline* for comparisons with parametrizable caches is given by this basic configuration.

Parametrizable Instruction-Cache: On the one hand, the program size of numerical cores such as MMul and stencil operations is typically small while on the other hand, the code binary of functions demanding many different computations integrated typically by numerical quadrature is by far larger. Consequently, this underlines the potential of changing the cache sizes. As a case study, we disable one set and halved the instruction cache and benchmarked the blocked MMul. This had only minor impact to the program's execution time with a variance of less than 1 % relative to the baseline. This provides us with additional memory resources that can be *assigned to the data cache* or deactivated for *energy efficiency*.

Parametrizable Data-Cache: Applications optimized for spatial-local access such as the blocked MMul and stencil operations target at exploiting data caches in an optimal way. Efficiency for such algorithms is gained by cache-oblivious access of the matrix-matrix multiplication data for particular cache-parameters. Additionally to the complete execution time, we compare the performance of every outer loop's iteration to the baseline of the default configuration.

At first, the cache size is kept constant at $16\,kB$. This just leaves two options for variation: the number of sets with respective adjustments of the set size. Fig. 1 (left) shows halved number of sets (2) and doubled set size ($8\,kB$). We see a slight overall efficiency gain (approx. 3 %), however, more importantly, we can see that different input data to the MMul is handled very well. In Fig. 1 (right), a single set with a $16\,kB$ set results in increased efficiency (approx. 7 %) relative to the baseline. Especially the 40 block-size and 384 matrix-size input benefit from the change in cache parameters.

Secondly, we half the size of the data cache. One option is to half the set size to $2\,kB$ per set (see Fig. 2, left) or the other option is to half the number of sets down to two (see Fig. 2, right). As we compare the two results, almost the same relative performance for the application is achieved in both settings. In contrast to our first results, modifications of cache parameters did not affect the

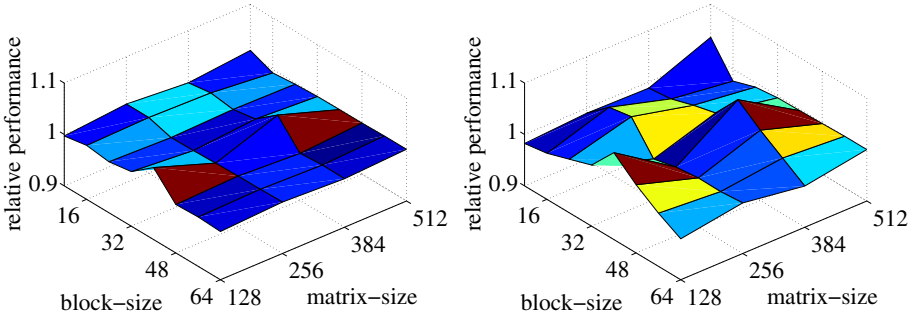

Fig. 1. Relative performance using a 16 kB data cache consisting of two sets with 8 kB per set (left) and one set with 16 kB (right)

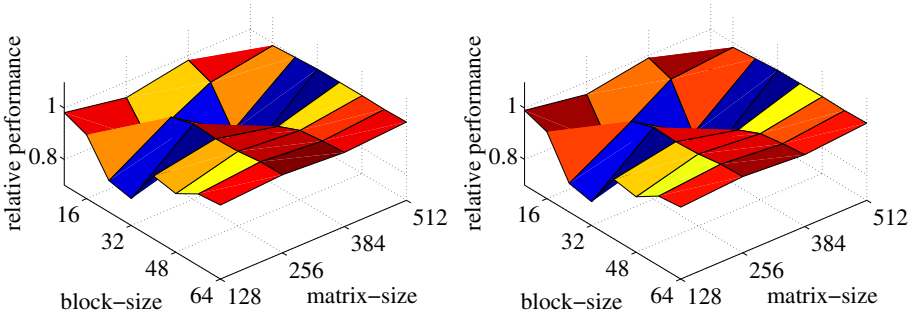

Fig. 2. Relative performance using a 8 kB data cache consisting of four sets with 2 kB per set (left) and two sets with 4 kB (right)

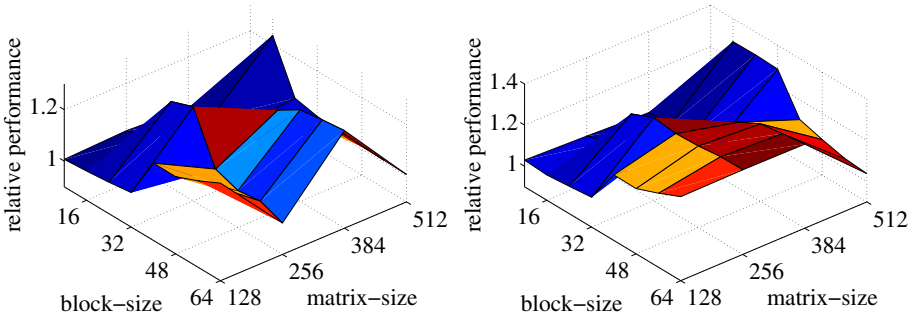

Fig. 3. Relative performance using a 32 kB data cache consisting of four sets with 8 kB per set (left) and a 64 kB data cache consisting of four sets with 16 kB per set (right)

efficiency of particular problem sizes, by the reduction of cache size. However, the performance drops by less than 20% compared to a standard execution. We expect that the information on performance change play a crucial role for the dynamical resource management. It is very promising to be able to change the cache set parameter for MatMul with larger problem sizes.

At last, we make potential additional cache resources available to the MatMul application. Fig. 3 shows the benefits of this approach. Interestingly, the application still benefits differently from the change in cache configuration. While the relative performance is higher for the 24 & 32 block size by 256 matrix-size configuration (left), MatMuls on larger matrices only benefit from a further increase of the set size (right).

4.2 Intermixing Bandwidth- and Compute-Bound Applications

Concurrently executed applications with different characteristics with respect to CM ratios have to be considered in an orthogonal way by the resource manager on a software level. Therefore, our next test case is on the concurrent execution of bandwidth- and compute-bound applications. Since these applications have different demands on memory parameters, those parameters are expected to lead to further efficiency gain once the dynamic adaptive hardware is available, e.g. by deactivating cache or statically reassigning cache sets to applications.

The experiments are conducted on a four-socket Intel(R) Xeon(R) CPU E7-4850 running at $2\,GHz$. For our test cases, we only use the physical cores on the CPU on the first socket. Our benchmark is based on a representative application for memory-bound problems, a streaming benchmark[1], and a representative application for compute-bound problems, a mandelbrot[2] computation.

The scalability graphs of both applications are presented in Fig. 4. The scalability of the representative memory-bound application almost reaches its peak with six cores due to the overloaded memory bus. For our representative compute-bound application, the scalability is almost linear for all ten cores.

Next, we consider the concurrent execution of one memory- and one compute-bound application. We pin each application to an exclusive set of cores. The description and results for the concurrent execution of our considered applications are given in Fig. 4. They indicate, that the memory-bound application is independent to the concurrently executed compute-bound application and vice versa.

Considering the application's optimal throughput being directly related to the scalability, we find the optimal throughput by searching the extrema of the sum of both scalability graphs, as shown in Fig. 5. For the optimal throughput in our benchmark, two cores should be assigned to the memory-bound application and the remaining eight cores to the mandelbrot for maximizing the throughput.

However, this optimal application throughput is *only valid if at most one bandwidth-bound application has to be considered*. In case of two memory-bound

[1] `http://www.cs.virginia.edu/stream/`, C-ver., *Add* BW, N=50 mio, NITER=2.

[2] `http://www.cs.trinity.edu/~bmassing/CS3366/SamplePrograms/OpenMP/mandelbrot-omp-by-rows.c`, exec. with $maxiter = 2048$, $(x, y) = (0, 0)$, $size = 1$.

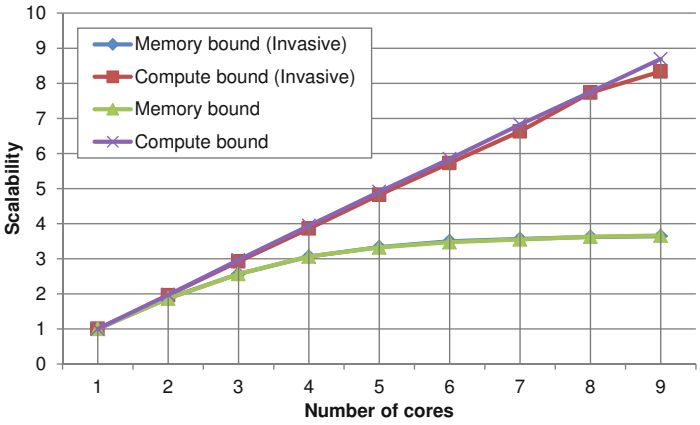

Fig. 4. Scalability of memory-bound application (streaming benchmark) and compute-bound application (mandelbrot). The *invasive* versions are executed concurrently: If n cores are assigned to the memory-bound application, then $10 - n$ cores are assigned to the compute-bound application. Both applications almost do not influence each other.

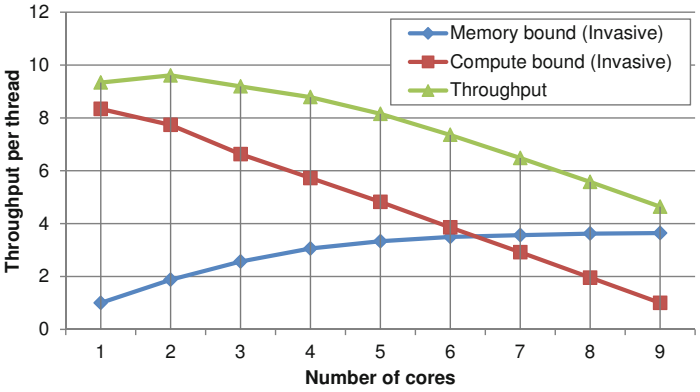

Fig. 5. Searching optimal distribution of cores to applications. The theoretical optimal throughput is given for assigning two cores to the bandwidth- and eight cores to the compute-bound application.

applications, a concurrent execution of both applications would influence the throughput and thus invalidate their scalability graphs. This yields requirements for information on memory characteristics which can then be utilized on software and hardware level for optimizations and is part of our ongoing research.

5 Conclusion and Future Work

We present an approach that exploits parameterization of cache parameters and computing resource parameters on hardware and software level. In contrast to

the state-of-the-art HPC hardware, we are able to exploit this parameterization to the application developer and offer an optimized application-specific hardware and software realization. This moves away from today's way of application programming, as the developer needs to be aware of the underlying hardware configuration and resources. Consequently, we are able to show a relative performance increase on an adaptive prototyping system, on which we dynamically change cache parameters. Furthermore, concurrently executed applications with different bandwidth characteristics extend invasive scheduling parameters for more efficient execution.

For future work, we plan to simulate and experimentally evaluate the presented concept. Also, further details of the hardware realization will be specified and evaluated. Another complex task will be the demonstration and complete integration into a single System-on-Chip.

Acknowledgement. This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Invasive Computing" (SFB/TR 89).

References

1. Bader, M., Bungartz, H.-J., Gerndt, M., Hollmann, A., Weidendorfer, J.: Invasive programming as a concept for hpc. In: Proc. of the 10h IASTED Int. Conf. on Par. and Dist. Comp. and Netw. (2011)
2. Brechmann, E.C., Schepsmeier, U.: Modeling dependence with c-and d-vine copulas: The r-package cdvine. Journal of Statistical Software 52 (2012)
3. Bungartz, H.-J., Mehl, M., Weinzierl, T.: A Parallel Adaptive Cartesian PDE Solver Using Space–Filling Curves. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 1064–1074. Springer, Heidelberg (2006)
4. Bungartz, H.-J., Riesinger, C., Schreiber, M., Snelting, G., Zwinkau, A.: Invasive Computing in HPC with X10. In: X10 Workshop, X10 2013 (2013)
5. Damien, G.: Study of different cache line replacement algorithms in embedded systems. PhD thesis, KTH (2007)
6. Gerndt, M., Hollmann, A., Meyer, M., Schreiber, M., Weidendorfer, J.: Invasive computing with iomp. In: Specification and Design Languages, FDL (2012)
7. Gordon-Ross, A., Vahid, F.: A self-tuning configurable cache. In: Proceedings of the 44th Annual Conference on Design Automation, DAC (2007)
8. Heinecke, A., Trinitis, C.: Cache-oblivious matrix algorithms in the age of multi- and many-cores. In: Concurrency and Computation: Practice and Experience (2012)
9. Henkel, J., Bauer, L., Hübner, M., Grudnitsky, A.: *i*-Core: A run-time adaptive processor for embedded multi-core systems. In: International Conference on Engineering of Reconfigurable Systems and Algorithms (2011)
10. Ji, X., Nicolaescu, D., Veidenbaum, A., Nicolau, A., Gupta, R.: Compiler Directed Cache Assist Adaptivity. In: High Performance Computing (2000)
11. Malik, A., Moyer, B., Cermak, D.: A low power unified cache architecture providing power and performance flexibility (poster session). In: Proceedings of the 2000 International Symposium on Low Power Electronics and Design (2000)

12. Marty, M.R.: Cache Coherence Techniques for Multicore Proc. PhD thesis (2008)
13. Nicolaescu, D., Ji, X., Veidenbaum, A.V., Nicolau, A., Gupta, R.: Compiler-directed cache line size adaptivity. In: Chong, F.T., Kozyrakis, C., Oskin, M. (eds.) IMS 2000. LNCS, vol. 2107, p. 183. Springer, Heidelberg (2001)
14. Nowak, F., Buchty, R., Karl, W.: A Run-time Reconfigurable Cache Architecture. In: International Conference on Parallel Computing: Architectures, Algorithms and Applications (2007)
15. Nowak, F., Buchty, R., Karl, W.: Adaptive Cache Infrastructure: Supporting dynamic Program Changes following dynamic Program Behavior. In: Proceedings of the 9th Workshop on Parallel Systems and Algorithms, PASA (2008)
16. Ogita, T., Rump, S.M., Oishi, S.: Accurate sum and dot product. SIAM Journal on Scientific Computing 26 (2005)
17. Rüde, U.: Mathematical and computational techniques for multilevel adaptive methods. Society for Industrial and Applied Mathematics (1993)
18. Schreiber, M., Bungartz, H.-J., Bader, M.: Shared memory parallelization of fully-adaptive simulations using a dynamic tree-split and -join approach. In: IEEE Int. Conf. on High Performance Comp, HiPC (2012)
19. Tradowsky, C., Thoma, F., Hubner, M., Becker, J.: Lisparc: Using an architecture description language approach for modelling an adaptive processor microarchitecture (best work-in-progress (wip) paper award). In: 7th IEEE International Symposium on Industrial Embedded Systems, SIES (2012)
20. Zhang, C., Vahid, F., Najjar, W.: A Highly Configurable Cache Architecture for Embedded Systems. In: 30th Annual Int. Symp. on Computer Architecture (2003)