

DF00100 Advanced Compiler Construction

DF21500 Multicore Computing

Autotuning

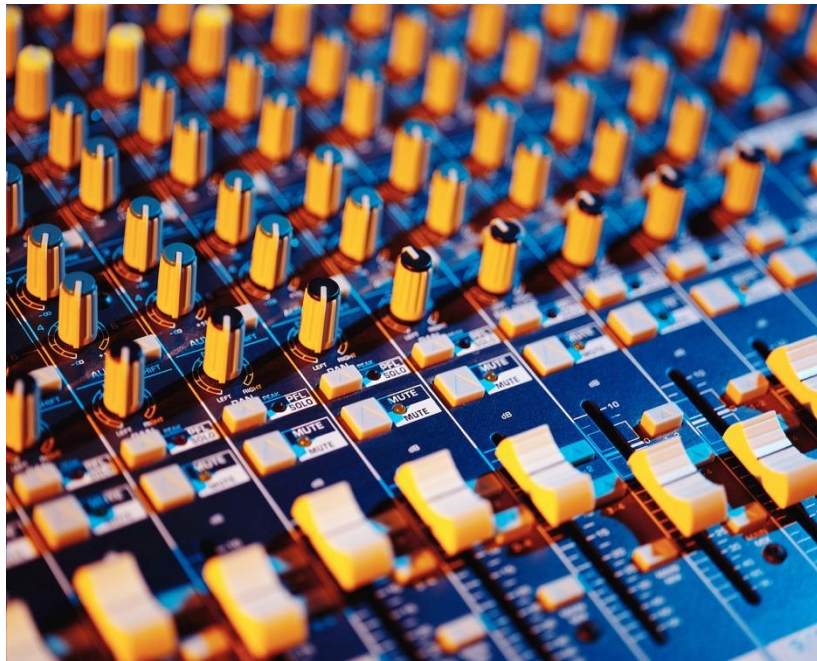
A short introduction

Motivation

- Modern (high-end) computer architectures are (too) complex
 - Some final machine parameters may not be statically (well-)known
 - Caches (multiple levels, capacity, associativity, replacement policy)
 - Memory latency
 - ILP and pipelining:
Dynamic dispatch, out-of-order execution, speculation, branching
 - Parallelism and contention for shared resources
 - OS scheduler
 - Paging
 - Performance not well predictable,
e.g. for manual or compiler optimization
- Some program parameters (problem sizes, data locality etc.) may not be statically known
- Different algorithms / implementation variants may exist for a computation
- Hardcoded manual optimizations lead to non-performance-portable code
- Compiler optimizations are limited and may have unexpected side effects / interferences

Motivation (cont.)

- Thousands of knobs that we could turn to tune performance!
- Which ones and how?
 - Avoid hardcoding of performance tuning



Performance Portability for User-level code?

Avoid hard-coded adaptations / optimizations such as:

```
if (avail_num_threads() > 1)
    in_parallel {
        sort( a, n/2); // on first half of resources
        sort( &a[n/2], n-n/2); // on the other half
    }
else ... (do it in serial)
```

NO!

```
if (available(GPU))
    gpusort(a,n);
else
    qsort(a,n);
```

NO!

```
if (n < CACHESIZE/4)
    mergesort(a,n);
else
    quicksort(a,n);
```

NO!

Idea: Autotuning – Automatic optimization for unknown target system using Machine Learning

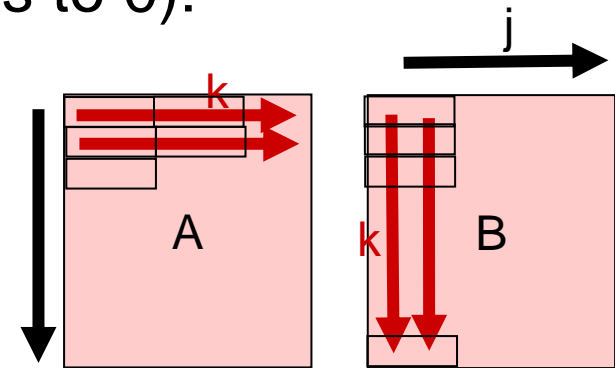
- Given: Training data and initial program version
 - Observed performance on target
 - Machine learning algorithm
 - Optimization strategy (choice of some parameter(s))
 - Automatic code generation / adaptation for target platform and possibly repeat this process
- for libraries: **autotuning library generators**,
for compilers: **iterative compilation**
for dynamic composition: **context-aware composition**
- Typical examples:
 - Find the best blocking factor(s) for loops or loop nests to automatically adapt to target cache behavior
 - Find the right sequence and settings of compiler optimizations
 - Select among different algorithms for same operation
 - How many cores/threads / which processors/accelerators to use?

Recall: Tiled Matrix-Matrix Multiplication (1)

- Matrix-Matrix multiplication $C = A \times B$
 here for square ($n \times n$) matrices C, A, B , with n large ($\sim 10^3$):
 - $C_{ij} = \sum_{k=1..n} A_{ik} B_{kj}$ for all $i, j = 1..n$
- Standard algorithm for Matrix-Matrix multiplication
 (here without the initialization of C-entries to 0):

```

for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    for (k=0; k<n; k++)
      C[i][j] += A[i][k] * B[k][j];
  
```



Good spatial locality on A, C
 Bad spatial locality on B
 (many capacity misses)

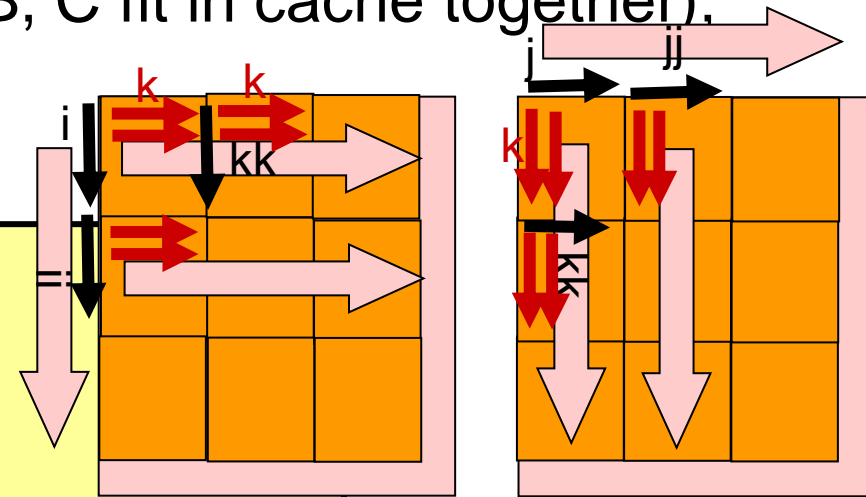
Recall: Tiled Matrix-Matrix Multiplication (2)

- Block each loop by block size S (choose S so that a block of A , B , C fit in cache together), then interchange loops

- Code after tiling:

```

for (ii=0; ii<n; ii+=S)
  for (jj=0; jj<n; jj+=S)
    for (kk=0; kk<n; kk+=S)
      for (i=ii; i < ii+S; i++)
        for (j=jj; j < jj+S; j++)
          for (k=kk; k < kk+S; k++)
            C[i][j] += A[i][k] * B[k][j]
  
```



Good spatial locality for A , B and C

What is the best choice for the blocking factor S ?

Recall: Loop Unroll-And-Jam

unroll the outer loop
and fuse the resulting inner loops:

```

for  $i$  from 1 to  $N$  do
  for  $j$  from 1 to  $N$  do
     $a[i] \leftarrow a[i] + b[j]$ 
  od
od

```

unroll&jam:

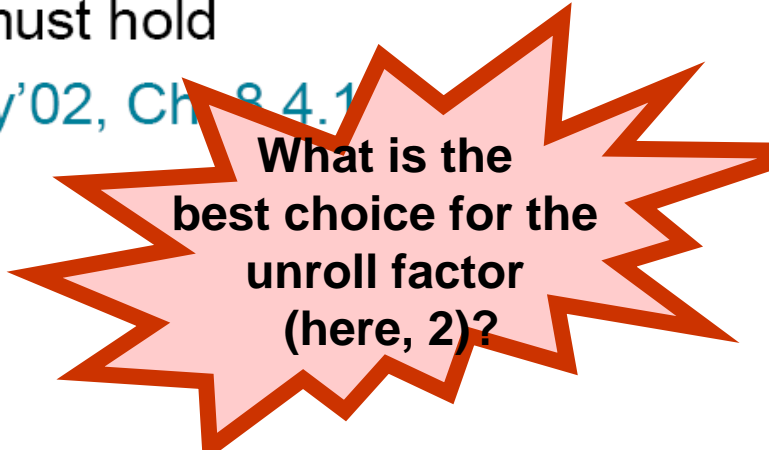
```

for  $i$  from 1 to  $N$  step 2 do
  for  $j$  from 1 to  $N$  do
     $a[i] \leftarrow a[i] + b[j]$ 
     $a[i+1] \leftarrow a[i+1] + b[j]$ 
  od
od

```

The same conditions as for loop interchange (for the two innermost loops after the unrolling step) must hold (for a formal treatment see [\[Allen/Kennedy'02, Ch. 8.4.1\]](#))

- + increases reuse in inner loop
- + less overhead

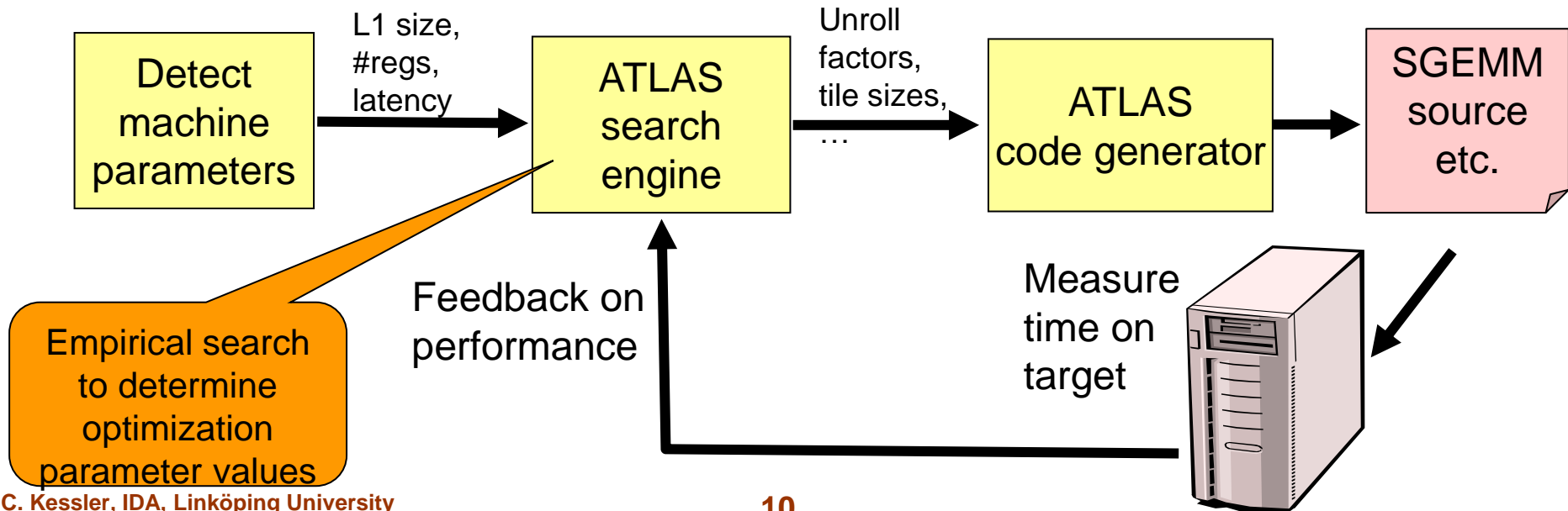


Auto-tuning linear algebra library ATLAS (1)

- **BLAS** = Basic Linear Algebra Subroutines
 - standard numerical library for Fortran, C
 - frequently used in high-performance applications
- Level-1 BLAS: Vector-vector operations e.g. dot product
- Level-2 BLAS: Matrix-vector operations
- Level-3 BLAS: Matrix-matrix operations, esp.,
generic versions of dense LU decomposition and Matrix mult.
 - **SGEMM**: $C := \alpha A * B + \beta C$
for matrices A,B,C, scalars α, β
 - ▶ is ordinary Matrix-Matrix multiplication for $\alpha=1, \beta=0$

Auto-tuning linear algebra library ATLAS (2)

- ATLAS is a generator for optimized BLAS libraries
 - Tiling to address L1 cache
 - Unroll-and-jam / scalar replacement to exploit registers
 - Use multiply-accumulate and SIMD instructions where available
 - Schedule computation and memory accesses
- Outperforms vendor-specific BLAS implementations



Remark

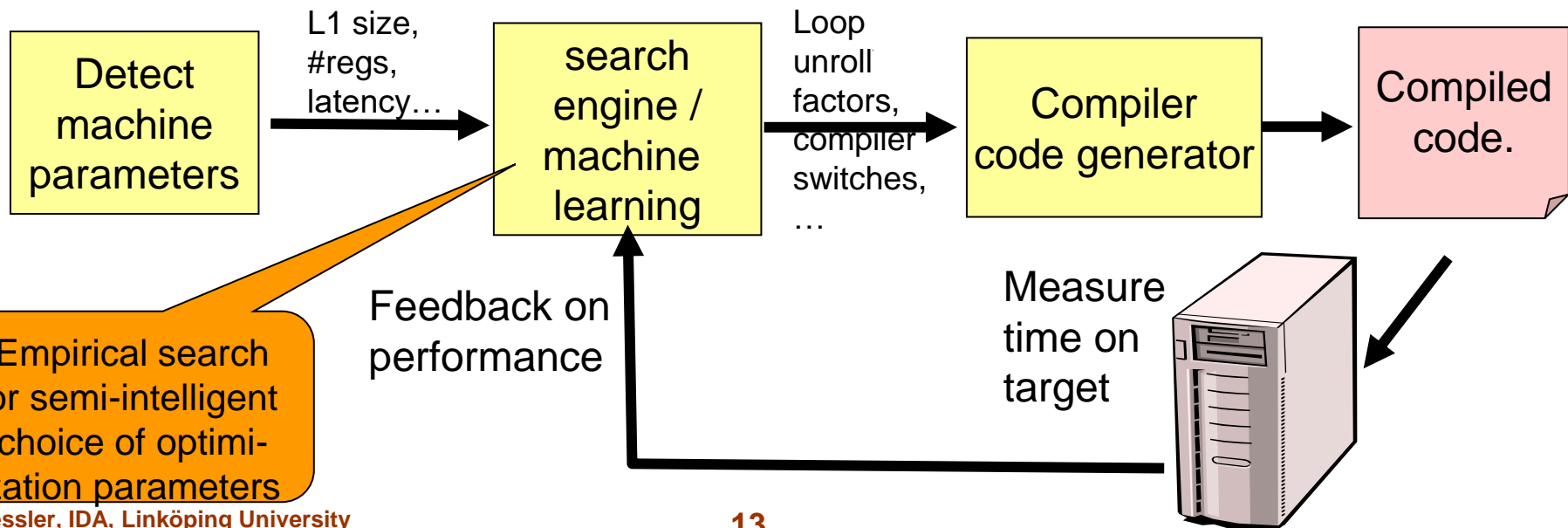
- Off-line sampling and tuning by greedy heuristic search
 - Happens once for each new system at library deployment (generation) time
 - Can be expensive
- Not practical for less static scenarios or costly sampling
 - Fast predictors needed – full execution or even simulation is not feasible
 - ▶ Usually constructed by machine learning
 - ▶ Shortens the feedback loop
 - ▶ Could be adapted dynamically (on-line sampling/tuning)

Further auto-tuning library generators

- Linear Algebra
 - ATLAS
 - PhiPAC
 - OSKI
- FFT and other signal processing
 - FFTW [Frigo'99]
 - SPIRAL [Püschel et al. 2005]
- Sorting, searching etc.
 - STAPL [Rauchwerger et al.]
 - [Li, Padua, Garzaran CGO'94]
 - [Brewer'95]
 - [Olszewski, Voss PDPTA-2004]

Generalize this in a compiler!

- Iterative compilation / autotuning compilers
 - Optimization of compiler transformation sequences
 - GCC MILEPOST project 2007-2008
 - CAPStuner, www.caps-entreprise.com
 - ActiveHarmony search engine + CHiLL source-to-source loop transformation framework
 - And many more recent works

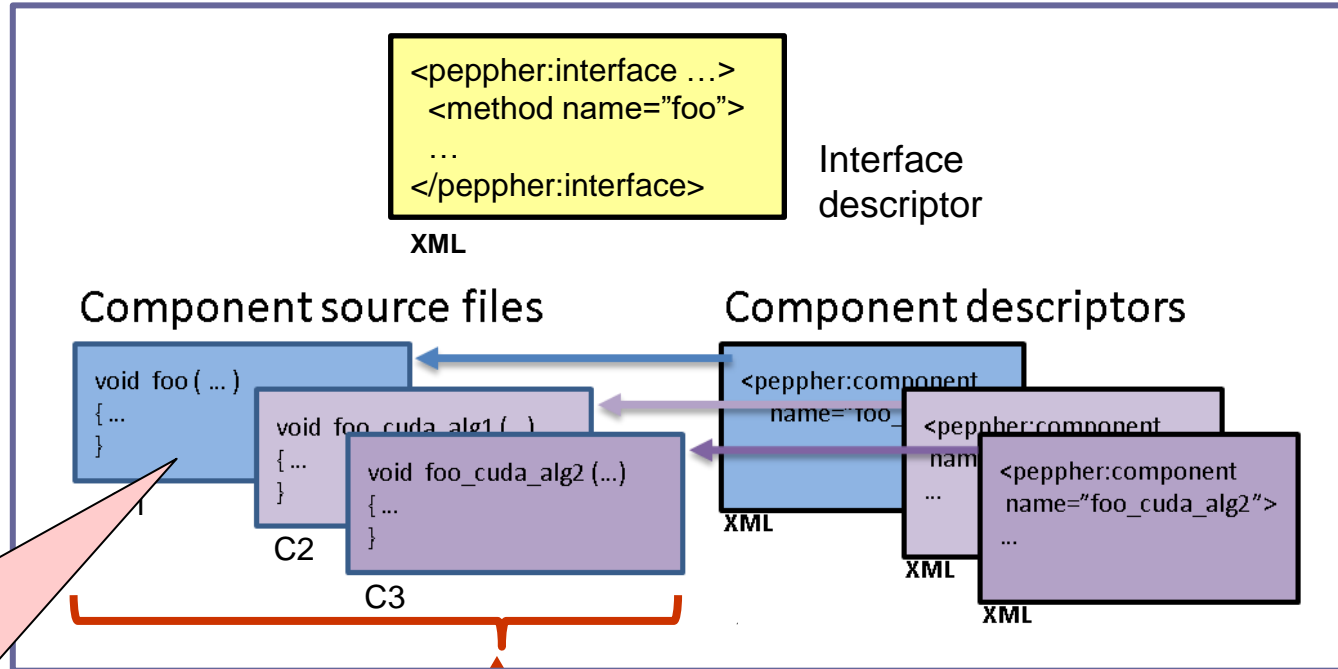


One step further: Auto-tunable software components and run-time composition

- Component programmer exposes the knobs for optimization in a **performance tuning interface**
 - Tunable function parameters e.g. problem sizes
 - Equivalent implementation variants (different algorithms, ...) at calls
 - Possible loop transformations, code specializations
 - Resource allocation and scheduling for independent tasks
- **At run time**, automatically select
 - expected best implementation variant for each call,
 - expected best resource allocation and schedule for indep. subtasks, given run-time information on actual parameters and available resources. Look up model / dispatch tables prepared off-line (by machine learning)
- **Examples**
 - Performance-aware parallel software components [[K./Löwe 2007/2012](#)]
 - Autotuning SkePU ([Dastgeer, Enmyren, K. 2011](#); [Dastgeer, K. 2013](#))
 - EU FP7 project PEPPER ([Benkner et al. IEEE Micro Sep/Oct. 2011](#))
 - Related work: Merge, Elastic Functions, PetaBricks

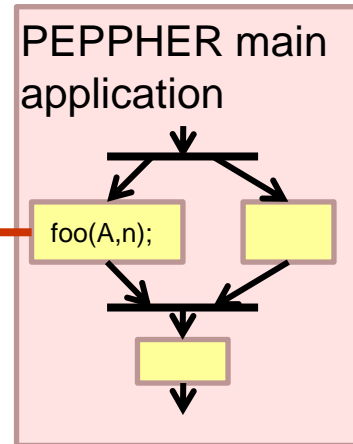
Performance-aware components: Interfaces, implementations, descriptors

One PEPPER component of the application



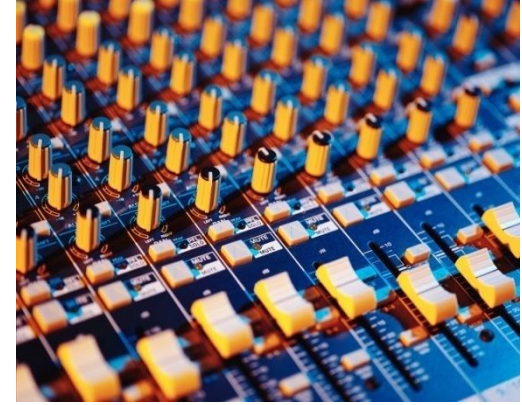
Implementation variants, e.g. different algorithms, exec. units, ...

Composition:
Variant selection
(static or dynamic or both)



C. Kessler, W. Löwe: Optimized composition of performance-aware parallel components. *Concurrency & Comput. Practice and Experience*, April 2012.
 U. Dastgeer, L. Li, C. Kessler: The PEPPER Composition Tool: Performance-aware composition for GPU-based systems. *Computing*, 96(12):1195-1211 (2014), DOI 10.1007/s00607-013-0371-8. Springer

Summary: Auto-tuning



- **Code optimization is difficult and very platform specific. Avoid hardcoding.** Instead, expose what is tunable and let the system learn suitable configurations from training data.
- **Auto-tuning library generators**
 - Fixed domain, implicit or explicit human guidance of search space
- **Auto-tuning compilers**
 - General-purpose programs (HPC)
 - Program structure (loop nests) defines optimization search space
 - Limited influence by programmer (e.g., some #pragmas)
- **Auto-tuning application-level software synthesis (software component composition)**
 - Programmer-exposed performance tuning interfaces, install-time learning, run-time composition
 - Can incorporate library and compiler based autotuning

References

- **On ATLAS:**
J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, K. Yelick: Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE* **93**(2):293-312, Feb. 2005
- **On FFTW:**
M. Frigo: A fast Fourier transform compiler. Proc. PLDI-1999, p.169-180, ACM.
- **On SPIRAL:**
M. Püschel et al.: SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE* **93**(2):232-275, Feb. 2005
- **On iterative compilation:**
 - **On ActiveHarmony + CHILL:**
A. Tiwari, C. Chen, J. Chame, M. Hall, J. Hollingsworth: A scalable auto-tuning framework for compiler optimization. Proc. IPDPS-2009, pp. 1-12, IEEE.
- **On general software autotuning:**
 - C. Kessler, W. Löwe: A framework for performance-aware composition of explicitly parallel components. Proc. ParCo-2007 conference, IOS press, pp. 227-234.
 - C. Kessler, W. Löwe: Optimized composition of performance-aware parallel components. *Concurrency & Computation Practice and Experience*, April 2012.
 - J. Ansel, C. Chan, Y. Wong, M. Olszewski, Q. Zhao, A. Edelman, S. Amarasinghe: PetaBricks: A language and compiler for algorithmic choice. Proc. PLDI-2009. ACM.
 - J. Wernsing, G. Stitt: Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing. Proc. LCTES, 2010.
 - U. Dastgeer, L. Li, C. Kessler: The PEPHER Composition Tool: Performance-aware composition for GPU-based systems. *Computing*, vol. 96 no. 12 (2014), pages 1195-1211 (DOI 10.1007/s00607-013-0371-8). Springer.

References (2)

Generic Autotuning Tools

- AutoTune
 - R. Miceli et al.: AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications, 2013
- OpenTuner
 - J. Ansel et al.: OpenTuner: An extensible framework for program autotuning. Proc. PACT 2014.
- (for CUDA)
 - M. Khan et al.: A Script-Based Autotuning Compiler System to Generate High-Performance CUDA Code. ACM TACO, 2013
- KTT Kernel Tuning Tool (for OpenCL)
 - J. Filipovic et al.: Autotuning of OpenCL Kernels with Global Optimizations, 2017
- KernelTuner (for GPU)
 - Ben van Werkhoven: Kernel Tuner: A search-optimizing GPU code autotuner. FGCS, 2018
- ATF Autotuning framework
 - A. Rasch et al.: ATF: A Generic Auto-Tuning Framework. HPCC 2017