# Performance Optimization by Dynamic Code Transformation

Josef Weidendorfer, Tilman Küstner
Department of Informatics
Technische Universität München, Germany
{weidendo,kuestner}@cs.tum.edu

Sally A. McKee
Computer Science and Engineering
Chalmers University of Technology, Sweden
mckee@chalmers.se

## ABSTRACT

Even parts of a program that are sequential or just inherently difficult to parallelize can be optimized for ILP. For instance, eliminating loop overheads and potential pipeline stalls from control flow can alleviate performance bottlenecks. Unfortunately, static compilation is limited in the extent to which it can identify opportunities to apply such optimizations. Generating code dynamically at run time, however, create much more efficient applications by usin information not available at compile time. We demonstrate our approach on a sparse-matrix PET scan code by aggressive unrolling loops and specializing code via dynamic code generation. We leverage task-level parallelism by having an auxiliary processor core concurrently generate code and feed it to the core executing the application. Our approach to fast code generation leverages patching and concatenating prepared code skeletons.

## Categories and Subject Descriptors

B.3.2 [**Hardware**]: Memory Structures—*Design Styles*; C.1 [**Computer Systems Organization**]: Processor Architectures

## General Terms

Performance, Design

## Keywords

Dynamic code generation, performance optimization

## 1. INTRODUCTION AND MOTIVATION

While parallelization is important to exploit multicore processors, there often will be program parts that are inherently difficult to parallelize. These easily can dominate overall performance. Good single-thread performance hinges on our ability to reduce potential pipeline stalls, e.g., due to data dependencies, control flow changes, or long-latency operations. Standard optimization techniques include function inlining, loop unrolling, and code specialization. However, the ability to fully apply these optimizations profitably at compile time is inherently limited. Luckily, the complex processor logic of current cores, intended to exploit potential ILP, can be leveraged dynamically to compensate for optimization opportunities missed during static compilation. Even with the current trend to use more simple cores on chip, dynamically generated, adaptively tuned code can help deliver high performance for sequential program parts. For instance, similar approaches have been demonstrated in the past: code-rewriting optimization engines such as Shade [3] and Dynamo [2] provided significant speedups. Good overviews of the JIT compilation techniques employed by these tools are provided by Franz [4] and Ayock [1]; these are essential in implementing managed languages (JavaVM, .NET, and JavaScript engines), but their applicability can and should be extended to other types of languages.

Using dynamic code generation for performance optimization can reduce indirections, both for method calls and memory accesses. One common source is the use of complex data structures, either for compressed representation of data, or when dynamically sized data are used. Examples are HPC sparse matrices/grids and dynamic re-meshing/refinement of simulation models. Sparse matrix vector multiplication (SpMV) motivates our approach: it is the heart of our MLEM image reconstruction code for medical PET imaging (see Section 3). Here, even indirect data accesses that hit in cache slow the computation. The sparse matrix is constant once initialized, which means we can dynamically optimize the application to remove the unnecessary address calculations and much of the control-flow overhead: we generate a single, unrolled, specialized code sequence embedding the full sparse matrix as immediate operands. This unrolled code quickly grows larger than the matrix, which will (again) bottleneck execution due to limited memory bandwidth. However, concurrently generating the code on a separate core of a multicore processor with sufficient speed reduces the memory pressure, since the generated code never need be written back to main memory.

## 2. CONCEPT

We propose a framework for concurrent generation and execution of code. Usually, dynamic code generation is performed when the optimized code is expected to be executed multiple times. The overhead of run-time compilation of pieces of code is expected to be compensated by multiple, faster execution runs. This holds true both for high-level


*CF'11,* May 3–5, 2011, Ischia, Italy.

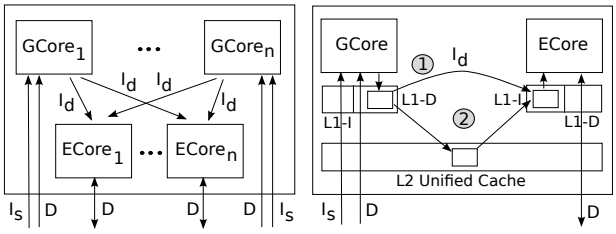


**Figure 1: Concurrent generator/executor scheme (left) and implementation for current multicore processors (right). $D$, $I_s$, and $I_d$ denote data streams and static/dynamic instruction streams.**

language VMs (JavaVM, .NET, and JITted scripting languages), as well as auto-tuning approaches in HPC [5, 7, 8] that typically tune the code before production runs. In contrast, we perform light-weight code generation for single-execution only. E.g., we fully unroll and specialize the sparse matrix vector operation. Code is generated immediately before it is executed, written into a buffer and subsequently overwritten with new code. Generated code resides in the cache and is never written back to main memory. Fig. 1 shows on the left the generic idea, where cores are partitioned into generators and executors, which can be arbitrarily arranged for load balancing. Our implementation on current multicore architectures (e.g., Intel Nehalem) is shown on the right, where we push code buffers down and up through cache levels. In our initial feasibility experiments, a Xeon X5670 at 2.93 GHz allows an isolated 8-byte NOP instruction generator to write about 20 GB/s of code into L3. Coupled with another core executing the generated sequence, we realize an effective bandwidth of around 13 GB/s for the code generation and 12 GB/s for its execution (i.e., the generator is slowed down by the executor).

Full unrolling of deeply nested loops eliminates loop overhead. The unrolled code can be specialized by inserting values of complex data structures as immediate operands into the instruction stream. Additionally, software prefetch instructions can be inserted into the generated code with dynamically chosen prefetch distances. This execute-once code generation technique can also be used to exploit reduced-complexity processors; for example, straight-line code needs no branch prediction (potentially allowing simpler predictor implementation and better power management by turning the predictor off when not needed). However, viability of our approach hinges on extremely fast code generation. To this end, we use prepared code skeletons and transform them by patching and concatenating them. Ideally, such a fast code transformer itself would be compiler-generated. First, the assumed compiler needs to produce skeletons of machine code, e.g., from loop bodies of user code. These skeletons become the input of the generator function, also to be produced by the compiler. At run time, this generator function builds specialized instruction streams by traversing user data structures. Here we write the code generators manually, since our purpose is to explore the potential of our approach for one specific application. Our skeletons require no time-consuming register allocation pass. For concurrent generation and execution, a lightweight run-time system manages a ring of code buffers.

# 3. RESULTS FOR REAL-WORLD SPMV

We have a working prototype for concurrent code generation and execution, but here we only present initial (encouraging) results for code generation into main memory and subsequent execution. The MLEM algorithm reconstructs images obtained from PET (positron emission tomography) scanners, and iteratively performs SpMV using a large sparse matrix (in our case 645 million non-zero elements, 0.12% population, and total size 2.45 GB) with integer elements, where 67% of non-zero elements have value one. The matrix is stored in a special format [6]. This application is a good scenario for our approach, as column numbers and matrix values are encoded into the instruction stream, and most multiplications can be avoided.

An optimization technique commonly applied to SpMV is cache blocking of the source vector (ours is around 3 MB). The results show that the statically compiled, standard SpMV implementation cannot benefit from high blocking factors on the X5670 because of loop overhead (best performance is 610 MElements/s with 16 blocks). However, dynamic code generation yields twice the performance: 1040 MElements/s with 64 blocks. On the downside, the code generator is slowed when performing indirect data accesses. This can be alleviated by running several generators in parallel.

# 4. CONCLUSION

We present a new approach to exploiting the benefits of dynamic code generation for multicore processors. First results for our concurrent code generation/execution scheme show that dynamic code generation can boost effects of other optimization strategies. We expect that our scheme will be especially beneficial on future architectures with on-chip local storage buffers for generated code.

*Acknowledgments.* Colleagues at the Department of Nuclear Medicine, Klinikum rechts der Isar, Munich, providied the sparse matrix in our MLEM application.

# 5. REFERENCES

[1] J. Aycock. A Brief History of Just-in-Time. *ACM Computing Surveys (CSUR)*, 35(3), 2003.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a Transparent Dynamic Optimization System. *Proc. ACM SIGPLAN PLDI*, 2000.

[3] B. Cmelik and D. Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. *Proc. ACM SIGMETRICS*, 1994.

[4] M. Franz. *Code Generation On the Fly: A Key to Portable Software*. PhD Thesis, ETH-Zurich, 1994.

[5] M. Frigo. A Fast Fourier Transform Compiler. *Proc. ACM PLDI*, 1999.

[6] T. Kuestner, J. Weidendorfer, J. Schirmer, T. Klug, C. Trinitis, and S. Ziegler. Parallel MLEM on Multicore Architectures. In *Proc. ICCS*, 2009.

[7] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A Library of Automatically Tuned Sparse Matrix Kernels. *J. of Physics: Conference Series*, 16:521–530, 2005.

[8] R. C. Whaley, A. Petitet, and J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 27(1-2):3–35, 2001.