# Evaluating Architecture and Compiler Design through Static Loop Analysis


Yuriy Kashnikov, Pablo de Oliveira Castro, Emmanuel Oseret, and William Jalby
Exascale Computing Research – University of Versailles, France
yuriy.kashnikov@exascale-computing.eu, pablo.oliveira@prism.uvsq.fr,
emmanuel.oseret@exascale-computing.eu, william.jalby@exascale-computing.eu



***Abstract*—Using the MAQAO loop static analyzer, we characterize a corpus of binary loops extracted from common benchmark suits such as SPEC, NAS, etc. and several industrial applications. For each loop, MAQAO extracts low-level assembly features such as: integer and floating-point vectorization ratio, number of registers used and spill-fill, number of concurrent memory streams accessed, etc. The distributions of these features on a large representative code corpus can be used to evaluate compilers and architectures and tune them for the most frequently used assembly patterns. In this paper, we present the MAQAO loop analyzer and a characterization of the $4857$ binary loops. We evaluate register allocation and vectorization on two compilers and propose a method to tune loop buffer size and stream prefetcher based on static analysis of benchmarks.**



***Keywords*—Benchmarking and Assessment; Software Monitoring and Measurement; HPC Monitoring and Instrumentation; Modeling, Simulation and Evaluation Techniques**


## I. Introduction

To be efficient on a large set of applications, new architectures and compilers must be tested on as many benchmarks as possible. The cost of benchmarking grows with the number of benchmarks considered. Testing a new architecture or compiler on thousand of benchmarks may not be feasible. In this paper, we propose to use fast static analysis as a preliminary step in the tuning process.

Compilers must be tuned to better harness new architectural features. To efficiently generate the code for a given architecture, compilers use an approximate model of the architecture and profitability heuristics, which guide the code transformations. Usually, tuning the compiler heuristics is a tedious and error prone manual work done by compiler developers. Our approach simplifies this process by identifying loops with sub-optimal code (eg. badly vectorized, too big for the architecture loop buffer). The proposed static analysis gives a quick estimate of the generated code quality and therefore could be transparently integrated into an automated compiler evaluation process. We demonstrate the applicability of the proposed approach for two compiler transformations: register allocation and vectorization.

For architectures, we consider the problem of selecting good hardware parameters as the number of virtual registers, the loop buffer size or the kind and number of dispatch ports. Finding the sweet-spot for these parameters through benchmarking may require many iterations. The number of testing iterations could be reduced if an oracle provided good enough initial values. We propose a method to derive statically good initial candidates for hardware parameters.

First, we collect a body of $4857$ binary loops extracted from a set of benchmarks and real industrial applications. Each loop is dissected with the MAQAO static loop analyzer [1], [2], which extracts a large set of low level characteristics such as: number of register used, number of values read in the stack, vectorization ratio, and pressure on dispatch ports.

MAQAO statistics are aggregated for all the loops in our database. To get a realistic profile, we weight loops features proportionally to their running time and discard loops that have little impact on the application running time. This analysis allows to profile the assembly characteristics of many different code fragments. By using a very large set of benchmarks, we can identify the most frequent micro-architectural bottlenecks.

The empirical distributions of these assembly features are used to select good candidates for sizing architecture parameters. Knowing the distribution of the used registers and spill-fill per loop can be used to evaluate the pay-off (cost) of adding (removing) registers in an architecture. For example, the number of register could be selected so that 90% of the loops are free of registers spills. The port pressure distribution, may help decide how many arithmetic, load and store ports are needed to balance dispatch rate. Of course, this decision must be weighted against the power, silicium and complexity cost of adding new dispatch ports to an architecture. But knowing the empirical percentage of benchmarks that could benefit from a new feature helps focusing on big payoff features. Our method is based on the analysis of the assembly code, which is usually produced by a compiler (except for some manually optimized assembly sections in highly specific programs), therefore the compiler effect must be taken into account. Our approach estimates the impact of new architecture features for a fixed binary or the impact of different compilers for a fixed target architecture.

The main contributions of this paper are:

1) a profile of low-level characteristics of a large representative body of applications
2) an evaluation of compiler's register allocation and vectorization

3) a static assembly characterization methodology to tune the loop buffer size and the hardware prefetcher

Section II presents the MAQAO static loop analyzer architecture and explains how the assembly features are extracted from binary data. Section III discusses the analysis methodology and the characterization results on the Intel Nehalem architecture and section IV demonstrates several data analysis use cases for guiding architecture parameters selection. Section V presents related works. Finally, Section VI summarize the study and discusses future research directions.

## II. The MAQAO Loop Static Analyzer

### A. Introduction

The Modular Assembly Quality Analyzer and Optimizer (MAQAO) [1] is an application performance analysis framework. MAQAO disassembles, analyzes, and instruments arbitrary binaries. It can be extended through a convenient API for modules.

MAQAO decomposes binaries as a hierarchy of functions, loops, paths (in loops), basic blocks and instructions. This hierarchy is produced with the following steps:

1) Disassembling: from the binary code, builds the corresponding list of decoded instructions as flexible data structures.
2) Flow analysis: from the instruction list and function labels, builds the corresponding call graph (CG) and control flow graph (CFG).
3) Loop analysis: detects and extracts loops in the CFG. Loops are organized hierarchically preserving the nesting of loops in the program.
4) Path analysis: detect paths (sequences of basic blocks) inside each loop. For example, a loop with an if-then[-else] body has two paths.

MAQAO analyzes the quality of the assembly code for each path in innermost loops. To assess the quality of assembly code, MAQAO measures how well the assembly code takes advantage of the hardware resources offered by the architecture. To evaluate resources usage MAQAO relies on a machine model. Currently, MAQAO supports the following Intel micro-architectures: Core 2 (65 and 45 nm), Nehalem and Sandy Bridge. The Nehalem model used in our study is built using information (instruction latencies, throughput, port assignment, $\mu$-op decomposition, etc.) provided by Agner Fog [3], [4].

### B. Architecture

Current implementation of the analysis is decomposed into two main steps: the in-order pipeline and the out-of-order execution core. To estimate the number of cycles and derived metrics, MAQAO considers the slowest step/stage (throughput oriented). Below we describe each step in details.

*1) In-order Pipeline:* The goal of the in-order pipeline is to feed execution units with ready-to-execute micro-operations. It can be viewed as a macro-pipeline with the following stages:

1) Instruction fetch
2) Instruction predecode (mainly to detect instructions boundaries)
3) Instruction queue (a FIFO to improve overall throughput)
4) Instruction decode (instructions are translated into micro-operations)
5) Micro-operations cache (on Sandy Bridge)
6) Micro-operations queue (another FIFO)
7) ROB-read (read of register operands)

For the in-order pipeline, MAQAO reports the number of cycles spent in the slowest stage.

*2) Out-of-Order Execution Core:* On supported architecture models micro-operations are dispatched in several execution ports. For example, on Nehalem it is six ports from P0 to P5, each port providing access to a subset of execution units. For instance the floating point adder is behind P1. Some execution units are duplicated like the arithmetic and logic unit (ALU), present behind P0, P1 and P5 allowing three integer instructions per cycle. For the out-of-order execution core, MAQAO displays the maximum number of cycles spent in the busiest port and in the DIV/SQRT unit (non pipelined contrary to all other units).

### C. Key Metrics

MAQAO extracts large number of dataset-independent characterization key metrics, but as any static performance analysis tool it can not estimate dynamic hazards like branch misprediction and cache misses. In order to provide time-related metrics, MAQAO assumes an infinite loop trip. Moreover, they are reported at a reduced cost, independent of the application runtime and, from them, an upper bound on performance can be estimated. The key metric definition depends on the goal, for example, code optimization or characterization. In order to demonstrate the global and diversified view we gather metrics and their combination on a large corpus of loops. Such a complete view opens many interesting opportunities for analysis and optimization. Below we present the characterization key metrics:

*1) Vectorization:* Vectorization metrics are key metrics for processors with vector units. For computation bounded loops, the vector length (number of elements per vector register) is the speedup factor that can be gained by vectorization. MAQAO provides the following vectorization metrics:

1) The proportion of vector instructions over all vectorizable instructions (already vector or scalar but admitting a vector equivalent). MAQAO provides a breakdown per instruction type (add/sub, mul, load, store and other). This metric is called “vectorization ratio” or “packed ratio”.

2) For loops featuring both integer and floating-point computations, two metric sets are computed: one considering only integer instructions and the other one, floating-point instructions.
3) An estimation of cycles if the loop could be fully vectorized, that is if all instructions in the loop body could be vector instructions (except loop control instructions and instructions with no vector equivalent).

*2) Registers and Stack Usage:* At the compiler stage and during register allocation, if there are not enough available registers (if all of them are used and the code could benefit from extra ones), the compiler has to spill/fill lacking registers from/to memory (x86: in the stack). MAQAO reports the number of registers used (at least XMM and YMM, in the release used for experiments presented in this paper) and the number of references in the stack (relative to the stack or the frame pointer register). XMM registers, 128 bits wide, are used by packed (i.e. vector) integer SSE instructions and scalar or packed FP (floating-point) SSE (or AVX, but only for scalar) instructions. YMM registers, 256 bits wide, are used by packed FP (AVX) and packed integer (AVX2) instructions. Consequently, when targeting SSE, integer and floating-point vectorized loops use XMM registers for processing data, scalar loops using XMM registers for floating-point data and general purpose (x86) registers for integer data. Spill/fill is a potential bottleneck which can often be fixed at source level. This is why it is interesting to evaluate its intensity. For not-unrolled loops that requires X logical registers, all processors having less than X of them will require spill/fill and could benefit from having more registers. If a loop that is spilling/filling registers is unrolled, it could be worth to reduce its unroll factor.

*3) Unroll Factor:* Unrolling is one of the most popular loop optimizations. Without tools, especially when having only a binary code (and not compiler optimization reports), it is difficult to know how much a loop were unrolled. MAQAO can determine, for most cases, the unroll factor of a (source) loop as soon as a peel or a tail (binary) loop were generated. Otherwise, MAQAO cannot conclude. Briefly, MAQAO considers that the unroll factor is the ratio between the number of operations (and/or moved bytes) in the main loop and the one in the peel or tail loop. A 4-way vectorized loop is implicitly unrolled by 4 and reported as unrolled by 4.

*4) Dispatch Ports Pressure:* MAQAO reports the pressure (in terms of micro-operations and cycles) on each execution port allowing to estimate which one will be bottleneck. If ports of load and store units are the most loaded, the loop is memory bound. If other ports are the most loaded, the loop is compute bound if there is no data dependencies and if data fit in the L1 cache, otherwise it could be memory bound.

## III. Static Characterization of Loops

This section presents MAQAO static analysis on a 4857 loops corpus. After describing the loop extraction, filtering and analysis methodology, we present the aggregated statistics for our loop corpus. Then, we evaluate the architecture design choices and compiler efficiency in regards to the measured assembly metrics.

### A. Methodology

MAQAO characteristics depend on the instruction set and compiler used. This study uses Intel Compiler version 12.1.0 and GCC version 4.5.1 and assumes an Intel Nehalem microarchitecture. The compilation options used for both compilers were `-g -O3 -mSSE4.2`. The `-g` flag produces debug information needed by MAQAO to map binary and source code loops and has no effect on the generated code and application performance.

The loop corpus was extracted from the set of applications and benchmarks summarized in Table I. Each application was compiled using both Intel Compiler and GCC. The binary compiled with Intel Compiler was profiled using the Intel Profiler which reports the percentage of run time spent in each function. In this study, functions whose running time is less that 1% were filtered out to concentrate on the performance hot spots. The profiling was performed using Intel Compiler option `--profile-loops=inner` on Intel Xeon L5609, Quad core, 1.86 GHz, 8GB RAM, without frequency scaling.

For this study, we selected traditional performance benchmarks such as NAS parallel benchmarks, SPEC CPU 2006, SPEC OMP and the Test Suite for Vectorizing Compilers (TSVC) [5]. We also selected a set of industrial applications:

- Gadget2 [6] is a cosmological structure formation simulator
- QMC=Chem (QMCChem) [7] applies Quantum Monte Carlo techniques to Chemistry problems
- PARMA is a set of three computations kernels extracted from a 3D combustion simulator provided by RECOM, a Navier-Stokes equation solver provided by Dassault-Aviation, and a metal forming simulator provided by GNS. The ITEA2 project ParMA [8] put together these codes.
- Polaris, a molecular dynamics simulator provided by CEA
- SPECFEM3D [9], [10], which simulates seismic wave propagation on arbitrary unstructured hexahedral meshes.

Table I shows for each application the average percentage of time spent inside the loops analyzed by MAQAO. It shows that most of the time is spent inside loops, supporting the loop based analysis proposed in this paper. The only exception is Gadget2 for which only 27% of the time is captured. This is because two Gadget2 computation-heavy functions, hydro_evaluate and force_treeevaluate, could not be analyzed by MAQAO. The functions have a set of deeply nested `if` blocks, which generate a combinatorial explosion of possible executions paths. To avoid blowing up the available memory, MAQAO static analyzer skips loops with a number of pathes

| Benchmark | % spent in loops (icc) | loops: icc | loops: gcc |
|---|---|---|---|
| Gadget2 | 26.70 | 5 | 3 |
| NAS-SER | 85.56 | 590 | 256 |
| PARMA | 97.90 | 65 | 46 |
| Polaris | 87.70 | 68 | 64 |
| QMCChem | 65.90 | 106 | – |
| SPEC-CPU-FP | 67.62 | 1458 | 681 |
| SPEC-CPU-INT | 57.66 | 635 | 430 |
| SPECFEM3D | 93.50 | 143 | – |
| SPEC-OMP | 83.20 | 1003 | 555 |
| TSVC | 100.00 | 179 | 181 |
| **Total** | — | **4252** | **2216** |

TABLE I
BENCHMARKS AND APPLICATIONS SELECTED FOR THE BINARY LOOP CORPUS AND AVERAGE TIME PERCENTAGE SPENT IN LOOPS. LOOPS ANALYZED WITH MAQAO LOOP STATIC ANALYSIS AND PROFILED USING AUTOMATIC INSTRUMENTATION DONE BY INTEL ICC COMPILER.

greater than a given threshold. This study selects 2 as the maximum number of pathes, which is enough to analyze most of the programs in a reasonable amount of time.

MAQAO extracted 4252 loops for binaries compiled with Intel Compiler and 2216 loops for binaries compiled with GCC. Table I breaks down the number of loops extracted per benchmark and compiler.

Optimizing compilers, such as Intel Compiler, often create multiple binary loops from a single source loop to deal with corner cases of the optimizations. For example, unrolling a loop often requires a tail loop to handle the left-over iterations, because the total iteration is not necessarily a multiple of the unroll factor. Likewise, compilers may generate multiple versions of the same loop, e.g. a vectorized binary loop that is called when the data is vector-aligned and another one that is called when the data is unaligned. Intel Compiler more aggressive optimizations make heavy use of unrolling and multi-versioning, which explains the higher Intel Compiler loop count in Table I.

For this study to be relevant, we want to concentrate on "hot" loops that impact performance, therefore we filter out tail and peel loops that do a negligible number of iterations. Nevertheless, we keep multi-versioned loops, since it is not possible to determine statically which one will be called. The execution time spent inside each binary loop is approximated by aggregating profiled time for each function and dividing it equally between all the loops in a function. After removing peel and tail loops, 4857 main binary loops remain (3103 for Intel Compiler binaries and 1754 for GCC binaries).

### B. Number of Instructions per Loop

The number of instructions is one of the key characteristics of a loop. The size of the loop defines its complexity in simple terms and also influences architecture design choices, such as loop buffer size or instruction cache size. Figure 1 shows that the majority of the studied binary innermost loops are small.

The loop buffer size on the Intel Nehalem architecture is 28 $\mu$-ops or micro operations. Most of the time, but not always one instruction is equivalent to one micro operation on the Intel Nehalem architecture. To convert between instructions and $\mu$-ops, we use the correction coefficient $E = median\ \frac{Number\ of\ \mu-ops}{Number\ of\ Instructions}$. For our loop corpus the coefficient $E$ is equal to $0.96$. Therefore, the Nehalem loop buffer can store at most $28\ \mu - ops/E \simeq 29\ instructions$. Loops larger than 29 instructions may encur a performance penalty since the loop body exceeds the loop buffer size.

Figure 1 shows that most of the loops fit the Intel Nehalem loop buffer size except for some programs in the SPEC-CPU-FP, NAS and Applications groups. In Sandybridge processors the loop buffer penalty is mitigated thanks to a new $\mu$-ops cache. We note that all the studied programs fit the Sandybridge $\mu$-ops cache (1.5K $\mu$-ops).

### C. Evaluating Compilers Vectorization

Vector instructions have been successfully used for decades in vector supercomputers. Instead of operating on single values, vector instructions load a set of values contiguously placed in memory and perform arithmetic operations on vectors constructed with these values. Modern optimizing compilers provide a broad range of optimizations targeting vectorization. Figure 2 compares the vectorization of Intel Compiler and GCC across the benchmarks. A benchmark is counted as vectorized if 80% of its assembly instructions are vectorial. Intel Compiler vectorizer is vectorized more loops than GCC in all the benchmarks. Overall Intel Compiler vectorizes 39.8% of the loops and GCC vectorizes 12% of the loops.

S. Maleki et al. [11] recently evaluated vectorizing compilers on different benchmarks, including TSVC. For TSVC they report that Intel Compiler 12.0 with the same optimization flags used in this paper, vectorized 61 loops (71.77% of the total loops). For the same benchmark, our analysis in Figure 2 shows that only 56.3% were vectorized. This discrepancy is due to the fact that we considered all the TSVC source loops in our analysis (whereas S. Maleki et al. only consider 85 source loops), including 137 initialization loops that cannot be vectorized. Also, unlike S. Maleki et al. who evaluated vectorization of source loops, we count vector binary loops. By projecting our vectorization measures at the source loop level, considering that a source loop is vectorized if it produces at least one vectorial binary loop, we compute that 57 source loops were vectorized. This figure is close to the 61 loops found in [11].

### D. Number of Independent Memory Streams

We define a stream as a group of load/store instructions whose target addresses differ only by a constant. In other words, instructions in the same stream have identical values for address and index registers but different offsets. A stream captures load/store instructions accessing the same data structure. Figure 3 presents an array copy example with two streams. Each stream in this example consist of two memory

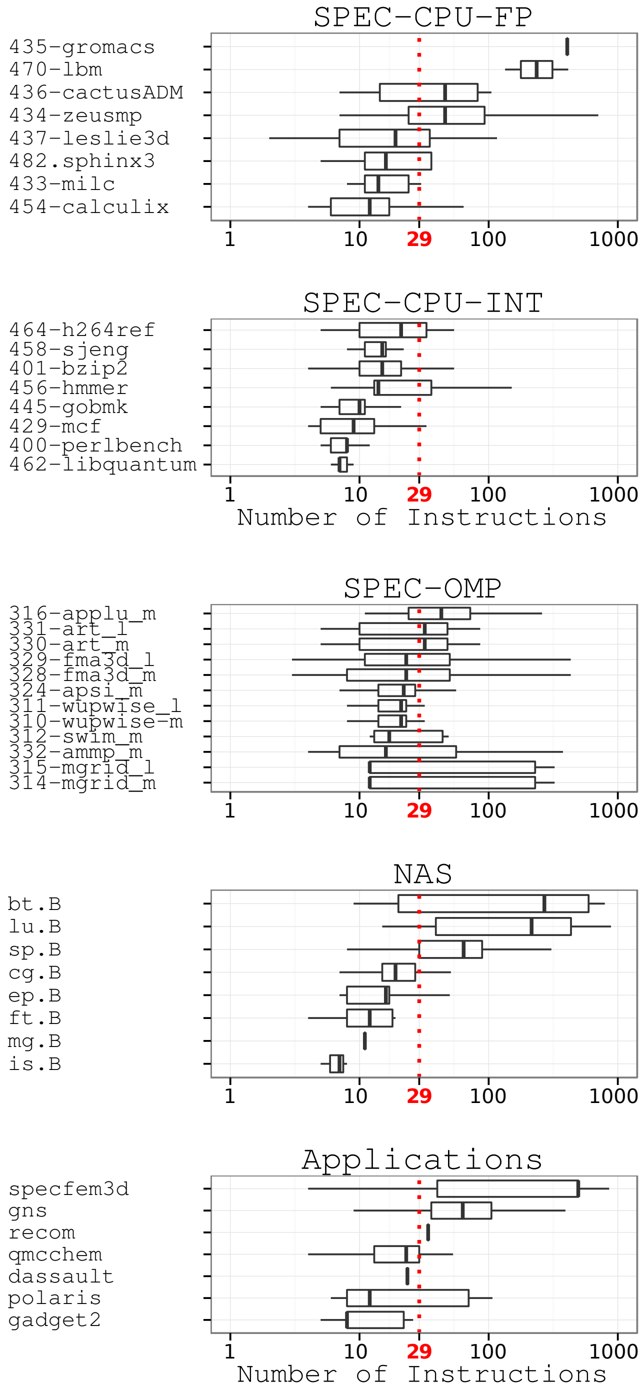


Figure 1. Number of instructions per loop. The distribution is weighted by execution time and represented as a box plot. For a given program, 50% of the execution time is spent in loops with a number of instructions lower than the bold bands. Most of the loops fit into the Nehalem loop buffer (dotted red line), all of them fit into the Sandy Bridge $\mu$-ops cache.

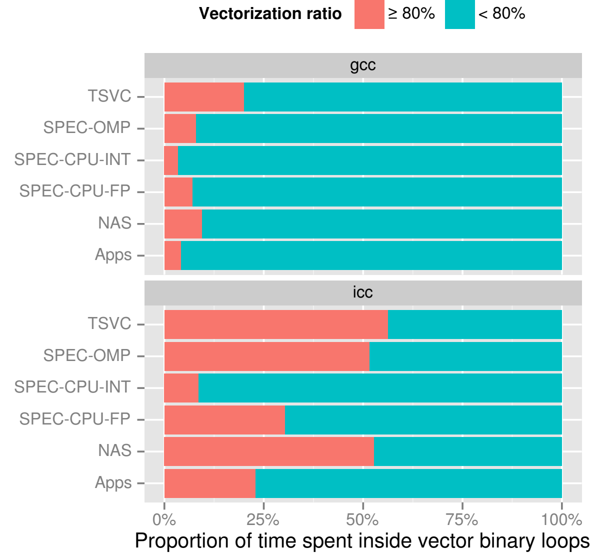


Figure 2. Vectorization across benchmarks. Distribution is weighted by execution time. Intel Compiler vectorizer is considerably better than GCC. The vectorization is higher in NAS, SPEC-OMP and TSVC compared to SPEC and Applications.

```
   loop
   lea     (%rcx,%rcx,1),%edi
   inc     %ecx
   movslq  %edi,%rdi
   cmp     %eax,%ecx
1  mov     (%rsi,%rdi,8),%r8
2  mov     %r8,0x0(%r13,%rdi,8)
1  mov     0x8(%rsi,%rdi,8),%r9
2  mov     %r9,0x8(%r13,%rdi,8)
   jb      .loop
```

Figure 3. Simple array copy kernel, unrolled twice. MAQAO detects two streams: stream 1 reading from the source array, and stream 2 writing to the target array.

instructions. The first stream is a pair of loads and second is a pair of stores. In most of the cases, MAQAO is able to detect the different streams in a binary loop by combining static analysis and abstract interpretation:

- By analyzing the memory access instructions, MAQAO determines the addressing mode (fixed address, offset from a base address, induction variable, etc.) and the registers involved in the address calculation.
- Then MAQAO simulates the loop execution and tries to determine the possible address values for each access.
- Access hitting the same memory segment are counted as a single stream.
- The number of streams is invariant to the unroll factor of the loop.

This analysis works for most of the simple cases, but has some limitations in complex kernels that use exotic addressing patterns.

The number of streams provides a worst case evaluation when tuning the cache associativity. First, we consider that associativity problems can only happen between different streams. Indeed, access in the same stream will not be mapped to the same associativity set in a reasonable window of iterations, because the stride between accesses is small enough (instructions are hitting the same data structure). Therefore, if the number of streams in a loop is below the cache associativity, there will be no evictions due to associativity.

The number of streams can also be used to fine-tune the hardware stream prefetcher. Modern hardware prefetchers can track different memory streams concurrently. The number of actual streams reported by MAQAO can be used to decide how many concurrent streams the prefetcher should track.

The Nehalem architecture provides a 8-way associative L1 data cache, therefore, as shown in Figure 4, Nehalem handles without problem most of the loops. The Nehalem stream prefetcher can keep track of 12 forward-stride streams and 4 backward-stride streams, which is enough for 92.7% of the loop corpus assuming all streams have a forward-stride. Figure 4 shows that very few execution time is spent in loops hitting more than 16 simultaneous memory streams.

*E. Dispatch Port Pressure*

MAQAO estimates the port pressure for each of the micro-architecture dispatch ports. Each loop can be attributed to one of the five categories listed below:

- Memory-Saturated (contains three sub categories):
  - Memory-Balanced: the loop is memory saturated but P2 pressure equals the pressure on P3 and P4.
  - Load-Saturated: P2 pressure is largest in number of cycles.
  - Store-Saturated: P3 or P4 pressure is largest in number of cycles.
- Arithmetic-Saturated: P0 or P1 or P5 pressure dominates in number of cycles.
- Balanced: Pressure on arithmetic ports (P0,P1,P5) matches pressure on either memory ports (P2, P3, P4).

Figure 5 shows how much time is spent in each category for the different benchmarks. We note that the pressure on the load dispatch port is high accross benchmarks, in particular for applications.

*F. Evaluating Register Allocation and Spill Fill*

A compiler may generate spill code in different situations:

- It has exhausted the available virtual registers of the architecture
- It has exhausted the available registers to pass function input parameters

When a compiler spills a register, it temporarily writes its content to the stack. Each spill usually translates into two

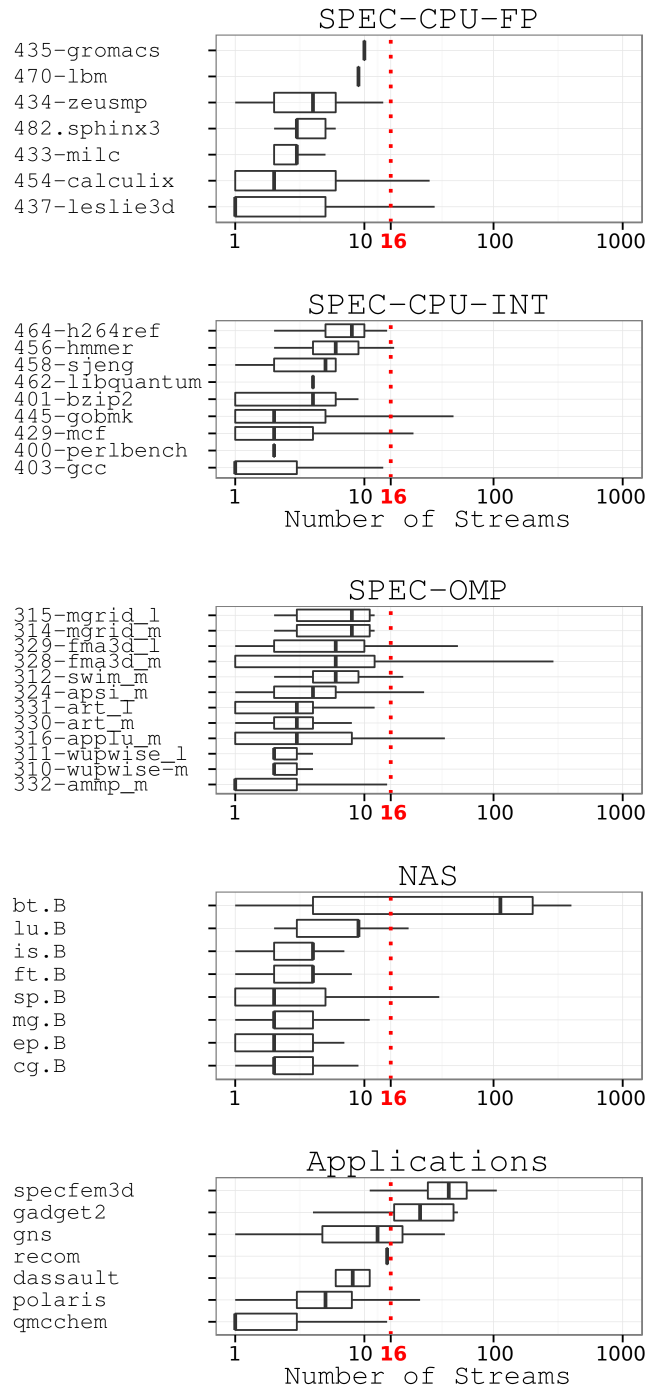


Figure 4. Number of streams per loop. The distribution is weighted by execution time and represented as a box plot. For a given program, 50% of the execution time is spent in loops with a number of streams lower than the bold bands. The Nehalem prefetcher is able to track up to 16 independent streams. For 87% of the loops the number of simultaneous streams used is less than 8.

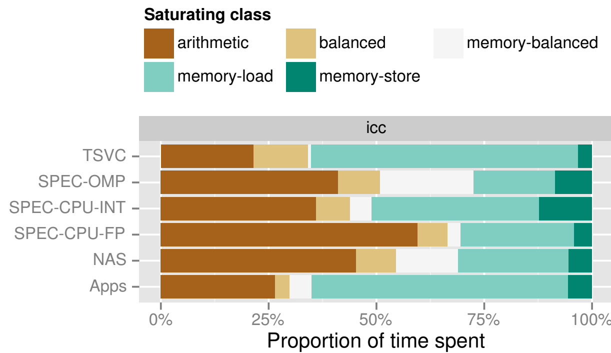


Figure 5. Bottleneck on dispatch ports. Weighted distribution of loops accross five saturating classes. The pressure on the load dispatch port is high accross benchmarks, in particular for applications.

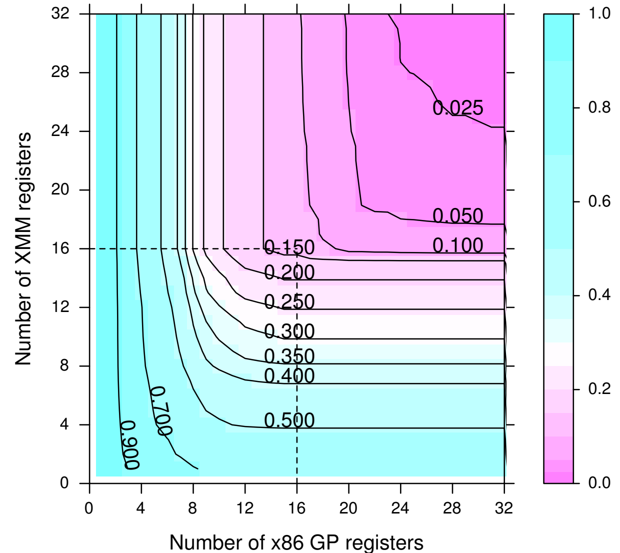


Figure 6. Projection of the proportion of spilling loops assuming a different number of XMM or x86 registers for Intel Compiler compiled loops. The current Nehalem configuration is signaled by the intersecting dashed segments.

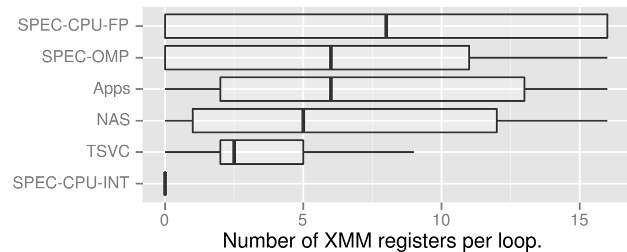


Figure 7. XMM compiler-visible register occupation per loop. Distribution is weighted by execution time. Most of the time is spent in loops with less than 8 XMM registers. The loops from SPEC CPU 2006 (INT) applications utilize almost no XMM registers, because it is poorly vectorized.

memory access: one store to write it to the stack and one load to read it back later. Spill access are quite fast, since they usually hit L1, but still more costly than a register access. There are two obvious ways to reduce spill overhead: improve the compiler spill allocation heuristics or add more virtual registers to the architecture. This section is divided in two parts: first, we will evaluate the compiler register allocation; second, we will study the impact of adding new virtual registers to existing architectures.

Among all the main binary loops, only 13.8% with Intel Compiler and 22% with GCC of the loops have spill access: both compiler are doing a good job in avoiding spills. Sometimes the compiler is willing to increase the register pressure to unroll a loop, even if this generates some spill access. Indeed 7.7% with Intel Compiler and 5.99% with GCC of the spilling loops are unrolled.

Now we evaluate the impact of adding or removing new virtual registers to an architecture. MAQAO evaluates the number of different locations in the stack that are accessed using XMM or x86 instructions. By adding as many XMM or x86 registers as spill stack locations, we can free a loop from spill access. Likewise, by reducing the number of register below the current register usage of a loop, the loop must spill. Figure 6 shows the proportion of spilling loops under different register configurations. The slope of the levelplot is steeper in the horizontal direction, this means that the impact of adding or removing x86 GP registers is higher. We attribute this to the fact that x86 registers are used both for data and address manipulations while XMM are only used for data. Figure 7 shows that many loops do not use all the alloted virtual registers. Therefore, for many loops it might be profitable to increase the unrolling factor potentially improving their performance. Compiler developers could benefit from the results of such an analysis and improve the default unroll factor selection heuristics.

## G. Limitations

Like any other, our approach also has limitations and has to be applied wisely. Programming languages like Java and Python which use just-in-time compilation, or autotuners which generate the code at runtime are out of the scope of our purely static analysis method. To analyze the Java bytecode one can use, for example, the Soot framework [12]. Similarly, server applications or applications that strongly depend on the dataset, and dramatically change their behavior at runtime, will be poorly characterized by the static approach. Finally, our approach examines the application on a new architecture using a binary generated for another architecture; however the code generated for this new architecture could be different, because the compiler heuristics would use another model to transform the code. Nevertheless, our approach is applicable to many existing applications, which are compiled once and shipped everywhere, like applications for Microsoft Windows, legacy

applications, or prebuilt binary packages for GNU/Linux.

## IV. Summarizing Resource Utilization

This section discusses how the static analysis presented in section III can be used to select optimal architecture parameters. The study considers three key architectural features: the loop buffer size, the number of simultaneous streams handled by the hardware prefetcher, and the number of vector registers. To estimate application stress on the three hardware parameters, we will consider respectively three MAQAO metrics: the number of instructions per loop, the number of memory streams, and the number of used XMM registers.

Table II summarizes the static analysis of section III for Intel Compiler compiled programs. Table II presents the median of the three MAQAO metrics for each program. Programs are grouped by benchmark suite. The gray lines compute the median metrics for all the programs belonging to the same benchmark suite. The metrics for the applications set are within the benchmark's metrics range showing that the benchmarks are representative of the group of applications that were selected.

To avoid loop buffer penalties, the loop buffer size should accommodate most of the program's loops. First column in Table II presents the median loop's size in instructions. One option to estimate the loop buffer size is taking the maximum loop size across all benchmarks, guaranteeing that at least half of the loops of every application fit. This option is impractical because of outliers: some programs, such as 470-lbm, have large loops that would require large and costly loop buffers. Instead, we propose to select a value satisfying 90% of the programs, that is to say we take the $0.9$ quantile of the values in the first column.

The number of concurrent prefetcher streams can be estimated with the number of independent memory streams in Table II second column. Likewise, to estimate the number of vector registers, we look at the number of registers used across programs in Table II third column. To select good initial candidates we consider, as before, the $0.9$ quantile.

We separate our programs into two different sets: the first contains the industrial applications in our corpus, the other contains the benchmark suites (SPEC, SPEC OMP, NAS, and TSVC). Table III presents the estimated values for each set. The selected loop buffer size is comparable for applications (43.50) and benchmarks (41.40). The selected buffer is a bit larger than for the Nehalem micro-architecture, suggesting that doubling the loop buffer size could improve the performance of future architectures. The suggested number of prefetcher stream is 6.9 for benchmarks and 17 for applications. The Nehalem prefetcher handles up to 16 simultaneous memory streams, and is therefore adequate for our set of programs. The number of vector registers suggested is around 8 for both applications and benchmarks. This value confirms the analysis of section III-F that suggest that there are plenty of potential candidates for higher unroll factor.

| | # Instructions | # Streams | # XMM Registers |
|---|---|---|---|
| Apps | 11.00 | 8.50 | 2.00 |
| dassault | 18.50 | 8.50 | 4.00 |
| gadget2 | 8.00 | 17.00 | 0.00 |
| gns | 81.00 | 13.00 | 11.50 |
| polaris | 15.50 | 4.00 | 3.00 |
| qmcchem | 6.00 | 2.00 | 1.00 |
| recom | 11.00 | 1.00 | 2.00 |
| specfem3d | 7.00 | 17.00 | 2.00 |
| NAS | 13.50 | 2.50 | 2.00 |
| bt.B | 24.00 | 58.00 | 8.00 |
| cg.B | 8.50 | 2.00 | 2.00 |
| ep.B | 16.00 | 2.00 | 1.00 |
| ft.B | 9.00 | 3.00 | 2.00 |
| is.B | 7.00 | 3.50 | 0.00 |
| lu.B | 61.00 | 9.00 | 11.00 |
| mg.B | 11.00 | 2.00 | 2.00 |
| sp.B | 29.00 | 2.00 | 11.00 |
| SPEC-CPU-FP | 24.00 | 3.50 | 2.25 |
| 433-milc | 29.00 | 2.00 | 2.00 |
| 434-zeusmp | 35.00 | 3.00 | 9.00 |
| 435-gromacs | 19.00 | 4.00 | 2.50 |
| 436-cactusADM | 50.00 | 6.00 | 7.00 |
| 437-leslie3d | 7.00 | 1.00 | 1.00 |
| 454-calculix | 12.00 | 3.00 | 0.00 |
| 470-lbm | 273.50 | 9.00 | 16.00 |
| 482.sphinx3 | 12.00 | 5.00 | 0.00 |
| SPEC-CPU-INT | 9.50 | 3.50 | 0.00 |
| 400-perlbench | 10.00 | 5.00 | 0.00 |
| 401-bzip2 | 15.00 | 6.00 | 0.00 |
| 429-mcf | 8.00 | 1.00 | 0.00 |
| 445-gobmk | 9.00 | 1.00 | 0.00 |
| 456-hmmer | 14.00 | 6.00 | 0.00 |
| 458-sjeng | 11.00 | 1.00 | 0.00 |
| 462-libquantum | 8.00 | 4.00 | 0.00 |
| 464-h264ref | 6.00 | 3.00 | 0.00 |
| SPEC-OMP | 16.50 | 2.50 | 5.25 |
| 310-wupwise-m | 18.00 | 2.00 | 6.00 |
| 312-swim_m | 17.00 | 6.50 | 3.00 |
| 314-mgrid_m | 33.00 | 7.00 | 8.00 |
| 316-applu_m | 43.00 | 3.00 | 7.00 |
| 324-apsi_m | 16.00 | 3.00 | 5.00 |
| 328-fma3d_m | 11.00 | 1.00 | 1.00 |
| 330-art_m | 12.00 | 2.00 | 2.00 |
| 332-ammp_m | 16.00 | 1.50 | 5.50 |
| TSVC | 11.00 | 5.00 | 2.00 |

TABLE II
MEDIAN VALUES OF THE STATIC ANALYSIS METRICS FOR EVERY STUDIED PROGRAMS

| | Loop buffer size | #Prefetcher Streams | #Vector Registers |
|---|---|---|---|
| Apps | 43.50 | 17.00 | 7.00 |
| Benchs | 41.40 | 6.90 | 8.80 |

TABLE III
SELECTED ARCHITECTURE PARAMETERS

Our method allows to measure the stress of a set of benchmarks for some key architecture parameters based on assembly metrics. One disadvantage of the method, is that the metrics may be too tied to the compiler. For example, the number of XMM registers over GCC compiled programs is lower than for Intel Compiler compiled programs because GCC vectorizes less loops. To factor out the compiler effect, we would need compiler-independent metrics, measured at the source level. The advantage of our approach is that it measures

directly what is executed on the processor, unlike source level metrics.

## V. Related Work

ROSE is a well known framework for performing data flow and static analysis. Shalf, E., et al. [13] used ROSE to analyse an iterative co-design process focusing on exascale applications.

MAO is an extensible micro-architectural assembly to assembly optimizer [14]. Similar to MAQAO it operates at a low micro-architectural level, but unlike MAQAO it is not able to decompile the binary and requires the assembly code to be manually extracted. MAO focuses mainly on optimization: optimization passes are designed in the spirit of compiler optimization passes and operate mostly on the programs control-flow graph. MAO executes the program to evaluate the impact of each optimization pass, which is costly compared to the MAQAO pure static approach.

Intel Architecture Code Analyzer (IACA) is a performance static analysis tool which operates on a binary file and estimates port pressure and cycles taking into account data dependencies. Unlike MAQAO, which automatically analyzes all innermost loops belonging to given functions, IACA needs to recompile the application to delimit the section of code to analyze. IACA assumes an optimal front-end throughput (four uops per cycle) contrary to MAQAO which simulates it, allowing the latter to detect bottlenecks lying there. IACA displays only low-level metrics (cycles, dispatch) on instructions sequences while MAQAO can provide high-level metrics on source level loops, breakdown of a source loop into binary loops and unroll factor, and can help an application developer to optimize his code by providing him a threefold report: what are the detected performance issues, how much speedup can be expected by fixing them and how to fix them.

Traditionally, hardware architects use simulators to model and explore the design space. Unfortunately, simulating the whole application is costly and must be redone for every architecture generation. Focusing only on performance critical parts reduces the required simulation time. Several approaches [15], [16] extract synthetic benchmarks from existing benchmarks or applications that preserve a large set of performance features of the original code. These synthetic benchmarks reduce simulation time and enable profiling against proprietary workloads without sharing the code.

Phansalkar et al. [17] study the redundancy inside SPEC CPU2006 by using Principal Components Analysis on dynamic features of the benchmarks, they show that a reduced set of programs can capture most of the information of the full suite.

Marin and Mellor-Crummey [18] use static and dynamic analysis to build architecture-independent models for scientific kernels. Using these models they predict application performance and L1, L2, and TLB cache miss counts. Unlike our study Marin and Mellor-Crummey focus on memory issues of the memory-bound NAS benchmarks.

## VI. Conclusion

In this paper we applied MAQAO loop static analysis on a large set of applications, concentrating on the most relevant MAQAO low-level metrics. Such a quantitative analysis provides insights to hardware architects and compiler designers. The analysis quantifies how a large corpus of assembly loops stresses various parts of modern hardware and how well are the compiler heuristics tuned for a target architecture. We weighted the metrics with time profiling information so that our analysis accurately describes a realistic execution of the programs.

The low cost of the static analysis allowed us to examine more than $4857$ loops. We showed that the analysis of the number of instructions per loop and of the number of independent memory streams is important to tune hardware parameters such as the loop buffer or the hardware prefetcher. We proposed a simple strategy to automatically select reasonable values for these parameters given a set of benchmarks. For example, our number of instruction analysis shows that doubling the Nehalem loop buffer size could improve many benchmarks, whereas the Sandybridge $\mu$-ops cache satisfies all the benchmarks.

Finally, we evaluated Intel Compiler and GCC register allocators and vectorizers. Intel compiler and GCC register allocation heuristics are comparable in terms of number of spilling loops. On the other hand, Intel compiler vectorizes $39.8\%$ of the loops, whereas GCC vectorizes $12\%$ only.

## Acknowledgments

The authors would like to thank Mathieu Tribalat who developed the MAQAO memory streams module for his valuable help. The authors would also like to thank Michaël Quisquater for his valuable feedback on statistical methods. This paper is a result of work performed in the Application Characterization team from the Exascale Computing Research Lab with support provided by CEA, GENCI, Intel, and UVSQ. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the CEA, GENCI, Intel or UVSQ.

## References

[1] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J. Acquaviva, W. Jalby *et al.*, "Maqao: Modular assembler quality analyzer and optimizer for itanium 2," in *The 4th Workshop on EPIC architectures and compiler technology, San Jose*, 2005.

[2] S. Koliai, S. Zuckerman, E. Oseret, M. Ivascot, T. Moseley, D. Quang, and W. Jalby, "A balanced approach to application performance tuning," in *Proceedings of the 22nd international conference on Languages and Compilers for Parallel Computing*, ser. LCPC'09. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 111–125. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-13374-9_8

[3] A. Fog, *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*, 2012.

[4] ——, *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*, 2012.

[5] D. Callahan, J. Dongarra, and D. Levine, "Vectorizing compilers: A test suite and results," in *Proceedings of the 1988 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 1988, pp. 98–105.

[6] V. Springel, "The cosmological simulation code gadget-2," *Monthly Notices of the Royal Astronomical Society*, vol. 364, no. 4, pp. 1105–1134, 2005.

[7] A. Scemama, M. Caffarel, E. Oseret, and W. Jalby, "QMC=Chem: a quantum monte carlo program for large-scale simulations in chemistry at the petascale level and beyond," in *Proceedings of the VECPAR 2012 10th International Meeting High Performance Computing for Computational Science*, 2012.

[8] "Parma: Parallel Programming for Multi-core Architectures - ITEA2 Project (06015)," http://www.parma-itea2.org/.

[9] D. Komatitsch, D. Michéa, and G. Erlebacher, "Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA," vol. 69, no. 5, pp. 451–460, 2009.

[10] D. Komatitsch, "Fluid-solid coupling on a cluster of GPU graphics cards for seismic wave propagation," *C. R. Acad. Sci., Ser. IIb Mec.*, vol. 339, pp. 125–135, 2011.

[11] S. Maleki, Y. Gao, T. Wong, D. Padua *et al.*, "An evaluation of vectorizing compilers," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 2011, pp. 372–382.

[12] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing Java bytecode using the Soot framework: Is it feasible?" in *Compiler Construction, 9th International Conference (CC 2000)*, 2000, pp. 18–34. [Online]. Available: www.sable.mcgill.ca/publications

[13] J. Shalf, D. Quinlan, and C. Janssen, "Rethinking hardware-software codesign for exascale systems," *Computer*, vol. 44, no. 11, pp. 22–30, Nov. 2011.

[14] R. Hundt, E. Raman, M. Thuresson, and N. Vachharajani, "Mao - an extensible micro-architectural optimizer," 2011.

[15] A. Joshi, L. Eeckhout, R. H. Bell, Jr., and L. K. John, "Distilling the essence of proprietary workloads into miniature benchmarks," *ACM Trans. Archit. Code Optim.*, vol. 5, no. 2, pp. 10:1–10:33, Sep. 2008. [Online]. Available: http://doi.acm.org/10.1145/1400112.1400115

[16] K. Ganesan, J. Jo, and L. John, "Synthesizing memory-level parallelism aware miniature clones for SPEC CPU2006 and ImplantBench workloads," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 33–44.

[17] A. Phansalkar, A. Joshi, and L. John, "Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite," in *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2. ACM, 2007, pp. 412–423.

[18] G. Marin and J. Mellor-Crummey, "Cross-architecture performance predictions for scientific applications using parameterized models," *SIGMETRICS Perform. Eval. Rev.*, vol. 32, no. 1, pp. 2–13, Jun. 2004.