Quantifying Performance Bottleneck Cost Through Differential Analysis

Souad Koliaï
Exascale Computing Research
Versailles, France,
souad.koliai@exascale-computing.eu

Zakaria Bendifallah
Exascale Computing Research
Versailles, France,
zakaria.bendifallah@exascale-computing.eu

Mathieu Tribalat
Exascale Computing Research
Versailles, France,
mathieu.tribalat@exascale-computing.eu

Cédric Valensi
Exascale Computing Research
Versailles, France,
cedric.valensi@exascale-computing.eu

Jean-Thomas Acquaviva
Exascale Computing Research
Versailles, France,
jean-thomas.acquaviva@exascale-computing.eu

William Jalby
University of Versailles
Versailles, France,
william.jalby@uvsq.fr

ABSTRACT

Accurate performance analysis is critical for understanding application efficiency and then driving software or hardware optimizations. Although most of static and dynamic performance analysis tools provide useful information, they are not completely satisfactory. Static performance analysis does not provide an accurate view due to the lack of runtime information (eg: cache behavior). On the other hand, profilers, generally mixed with hardware counters, provide a wide range of performance metrics but lack the ability to correlate performance informations with the appropriate code fragment, data structure or instruction. Finally, cycle accurate simulators are too complex and too costly to be used routinely for optimization of real life applications. This paper presents the Differential Analysis method, an approach designed for simple and automatic detection of performance bottlenecks. This approach relies on DECAN, a tool which generates different binary variants obtained by patching individual or groups of instructions. The different variants are then measured and compared, allowing to evaluate the cost of an instruction group and therefore its optimization potential benefit. Differential analysis is illustrated by the use of DECAN on a range of HPC applications to detect performance bottlenecks.

Categories and Subject Descriptors

D.2.8 [**Software Engeneering**]: Metrics—performance measures

Keywords

Bottleneck detection, differential analysis, binary rewriting, performance evaluation

ICS'13, June 10–14, 2013, Eugene, Oregon, USA.

1. INTRODUCTION

A critical problem in High Performance Computing is to bridge the gap between peak and achieved performance. Despite impressive raw performance, the achievable fraction on production computing systems remains a frustration for the application developer. The performance gap has been acknowledged, and numerous tools (cf. Section 5) have been developed to bridge it. In fact, one of the key issues to resolve is not only to find the performance bottlenecks but also to correlate back to the source code, which is the natural place where the application developer has to insert the optimizations. For example, knowing through performance counters that a loop generates a high number of cache misses is not enough. First, the loop may contain ten arrays accessed in different ways and only a few of these accesses are responsible for most of the cache misses. Second, knowing the number of cache misses is not enough, quantifying its impact is also critical. It might happen that most of the cache misses are in fact near hits (apparent latency is less than 10 cycles) while a few of them have a very high latency. Therefore these few cache misses have the largest impact on overall performance. In fact, this last category of misses is the one to be optimized in the first place. This leads to the more general notion of potential performance gain for optimization. Let us assume that routine A consumes 60% of the total execution while routine B consumes 30% and, at the same time, the maximum potential performance gain for A is estimated at 10% (resulting in an overall performance gain of 6%) while performance gain for routine B is estimated at 50% (resulting in an overall performance gain of 15%). In such a case it is clear that more effort should be spent on the optimization of routine B. However, the difficult part is how to evaluate the potential performance gain.

To answer most of the problems listed above, simulation tools might seem the best approach. They are accurate and enable not only bottleneck detection but the correlation of performance issues with the source code as well. They also give a clear quantification of performance impact. Unfortunately, such tools do not take into account all of the system aspects of the target machine such as the memory configuration and the OS. Furthermore they are slow and in general not user friendly for an application developer.

To remedy this situation, we developed the DECAN tool, which relies on differential analysis to identify and quantify performance bottlenecks. The DECAN principle is very similar to standard debugging techniques where programmers remove statements one after another to identify the bug. The basic concept is fairly simple: the original binary is first measured and then several variants, each of them addressing a potential performance bottleneck, are generated and measured. By comparing performance measurements of these variants with the reference timing, we can detect bottlenecks and assert their performance impact. All of the measurements are done on the real target machine, taking into account all of the system aspects and offering decent analysis time when compared with simulation. The DECAN tool performs binary level transformations. It does not consider the arithmetic compliance with the original binary, and it makes sure not to change the control flow. Furthermore, the transformation scope of DECAN is at the instruction level.

1.1 Motivating Example

To illustrate DECAN, let us consider the following piece of code (extracted from a dense Singular Value Decomposition library):

Algorithm 1 Code example

```
real * 8 A(N,16), scal, s(16) {Column oriented storage}
DO i = 1,16 (Parallel loop)
  DO k= 1, N
    A(k,i) = A(k,i)/scal
    s(i) = s(i) + A(k,i) * A(k,i)
  ENDDO
ENDDO
```

We consider different values of N ranging from 200 up to 1000K. The target machine is a 4-core Sandy-Bridge architecture (see section 4.1). The parallelization of the outermost loop is ideal and results in a perfect load balance between the cores. The innermost loop contains a stride 1 access to an array. It results in a very good vectorization. The only two potential issues are: 1) the reduction operation and 2) the division operation which is very costly on Sandy-Bridge architectures. The division operation can not be factored out of the innermost loop because the use of a reciprocal operation followed by a multiply affects numerical properties.

First, DECAN is used to generate two binary variants to detect for which data range the loop is CPU bound or data access bound:

- *LSI_STREAM* is a binary variant in which all of the floating-point (FP) arithmetic operations have been suppressed: only the data access instructions and the address/loop instructions have been kept.
- *FPI_STREAM* is a binary variant in which all of the load / store operations have been suppressed: only the FP instructions and the address/loop instructions have been kept.

Measuring LSI_STREAM (resp. FPI_STREAM) enables the contribution of data access (resp arithmetic) instructions to the overall execution time to be evaluated. Figure 1 shows that for values of N less than 400K the bottleneck is the arithmetic operations while for N values greater than 400K the bottleneck is the data access.

To investigate the performance impact of reduction and division operations, two more variants were generated:

- *ORIG_NODIV* is a binary variant in which only the FP division is suppressed.

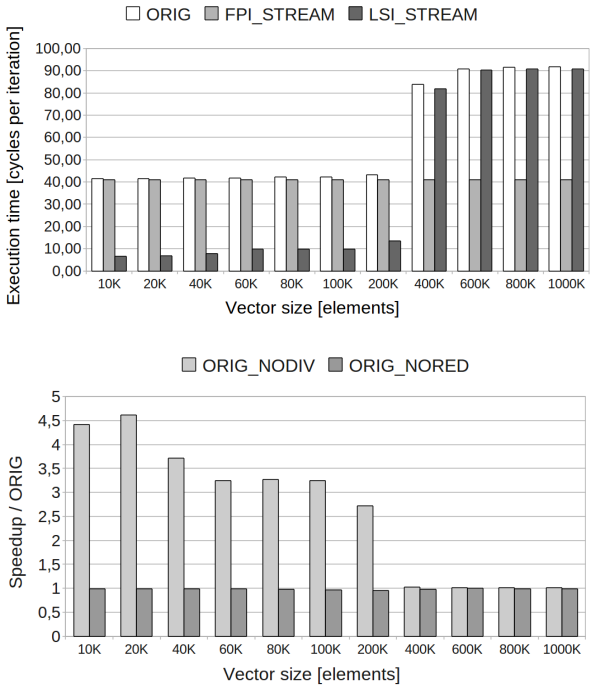

Figure 1: The upper figure shows the streams analysis with the variants FPI and LSI. The lower figure shows the elementary transformations NODIV and NORED. The experiments are performed on 4 cores

- *ORIG_NORED* is a binary variant in which the dependencies between iterations relative to the addition (second statement of the original loop) are suppressed. This is achieved by inserting an XOR (Exclusive OR) instruction which sets to zero the content of the register where the sum is normally accumulated. Inserting such an instruction automatically breaks the dependencies between consecutive executions of the addition instruction.

Figure 1 presents the relative performance gains of these variants with respect to the original. First, it clearly shows that the reduction operation induces no performance penalties across the whole data range (ORIG_NORED performance is identical to the original). Second, the division operation is very costly (i.e. it is the main performance bottleneck) for values of N less than 400K while the division operation cost is hidden by the data access for larger N values. Therefore, the technique of keeping the division operation within the loop to increase numerical accuracy had a performance impact only for smaller N values. A third variant, ORIG_NORED_NODIV, not shown here, was generated to detect potential interaction between the division and the reduction operations. This third variant had exactly the same performance as the ORIG_NODIV variant.

Similar observations for the impact of the division operation could have been achieved at the source code level by suppressing it but the compiler could have then suppressed the original statement from the generated code. Another approach is to replace the division with a multiply operation, yet the compiler may still generate an altered code. Operating with DECAN at the binary level allows "surgical" operations (with minimal intrusion) to be performed, keeping the DECAN variant code very close to the original target code. Now suppressing reductions at the binary level is much more

challenging and in most of the cases would result in a code very different from the original one.

1.2 Paper Contributions and Organization

The main contribution of this paper is the DECAN tool which allows at the loop level:

- to quickly and accurately identify key performance limitations
- to quickly assess quantitatively the potential performance gain associated with removing a given performance bottleneck
- to efficiently correlate performance problems with source code.

These three key features allow the efficiency of overall application performance tuning process to be improved considerably.
The paper is organized as follows: Section 2 introduces the general infrastructure of the DECAN tool as well as the techniques used to preserve the original program flow. Section 3 discusses the DECAN framework and describes the instruction transformation process itself; it also deals with DECAN limitations. Section 4 shows the experimental results on different codes. Section 5 briefly reviews other tools used in application performance analysis. Finally the conclusion summarizes and gives an outline of the future works.

2. DECAN INFRASTRUCTURE

First, we describe the global base environment on top of which the tool is built. Then, we discuss the program (and loop) control flow management, which is essential. Indeed, altering the binary has a major impact on the overall program behavior and special care has to be taken so that this remains under control. The section also addresses handling of OpenMP and MPI codes.

2.1 Disassembling and identifying target binary loops

DECAN relies on a software stack composed of two major components:
MADRAS (Multi Architecture Disassembler Rewriter and Assembler) [16], is a tool to disassemble a file in ELF [12] format and to return a sequence of structures containing information about the assembly instructions. MADRAS is also able to patch a file by inserting function calls or assembly instructions, deleting instructions or modifying them by changing their opcode or operands.
MAQAO (Modular Assembly Quality Analyzer and Optimizer) [5] is a framework which performs control and data flow analyses on the list of instructions provided by MADRAS. The resulting high level structure is a control flow graph augmented with an accurately constructed loop hierarchy. Other useful information is provided via dataflow analysis such as the list of registers used and *Use-Def* chains.

The loop hierarchy provided by MAQAO is made available to the user through a loop topology file. Its form is an intermediate representation of the chosen part of the program to study. The representation can be obtained by different kinds of requests. For example, a request could be the loop hierarchy contained between two source lines. The file is updated with the desired transformations and fed into DECAN.

2.2 Control Flow Management

Since DECAN directly modifies loop binaries, the overall program behavior might be affected and the program might even crash. We will distinguish two levels of impact and correspondingly two levels of control flow: outside of the modified loop (*outer control flow*), and within the modified loop (*inner control flow*). In order to keep control of the program behavior, we will use two approaches: first, limit the scope of the loops transformed by DECAN and second, restore correct behavior of the application.

Preserving outer control flow: A more general approach (used in [10]) to tackle this problem is the injection of a recovery loop, right after the loop modified by DECAN, to restore the correct states of registers and memory. Basically, the original loop is not only replaced by the transformed one, but also duplicated; when the DECANNed loop execution completes, the original version is also executed just after it. The following steps summarize the mechanism:

- **Context saving**: All the registers (general purpose, vector, flags and FPU) are saved
- **Monitors activation**: Monitoring activation probes are injected
- **Modified loop execution**: The transformed version of the loop is executed
- **Monitors deactivation**: Monitoring deactivation probes are injected
- **Context restore**: All the registers are restored
- **Original loop execution**: The original version of the loop is executed to recover the correct state of registers and memory

To be entirely correct, this technique also requires that all the stores are deleted in the transformation process because no corrupted data should be written.
In addition to this general mechanism, we also use a more specific technique which is ideal when targeting only one loop at a time. The principle is to insert an exit sequence right after the execution of the modified binary loop to make the program stop without crashing. Exiting right after the execution of the DECAN loop will prevent any propagation of the "incorrect" data generated by the DECANNed loop. This technique is implemented through an instance mode: first a given target binary loop is selected and the program is launched to monitor execution timings of different instances of that target loop. This helps to define an instance number K of interest (usually it corresponds to the largest execution time monitored). DECAN patches the binary to let the program run normally (the original binary code is executed) until the Kth instance of the target loop is detected. This triggers a branch to the timing and execution of the modified binary code of the target loop followed by an early exit.
For most of the performance studies performed so far using DECAN, the two methods have been able to function correctly.

In its current version, DECAN does not modify loops containing subroutine calls: for example, a loop containing an MPI *Send* or *Receive* statement will not be modified. This limitation is essentially motivated by the necessity to narrow the propagation of the "side effects" in the modified binary: a loop modified by DECAN will in general compute incorrect values and therefore parameters passed to the subroutine calls will be incorrect. This "no subroutine" rule will suffer two major exceptions: 1) DECAN will still operate on loops containing calls to intrinsic functions such as exponential. 2) It also handles the calls to the runtime library such as the ones inserted by the compiler for managing the OpenMP parallel constructs. It should be noted that the "no subroutine call" limitation can be easily worked around in most of the practical cases by forcing subroutine inlining.

Preserving inner control flow: First, loops with conditional branches cannot be transformed in a naive manner because the paths' execution ratio and order must be kept unchanged. In order to fulfill this requirement, as a preliminary step (before any transformation is applied) a detailed static analysis is performed to compute *Use-Def* chains involved in the computation of the values governing the conditional branch. The resulting instruction lists are put in a blacklist and skipped during the transformation step. Using such a technique guarantees that the branch behavior will be identical in the original loop and in the DECANNed loop because all of the instructions governing the branch have not been modified.

A more subtle issue can arise with instructions such as divide or square roots. Since "incorrect" numerical values are generated in the DECANNed loop, exceptions (divide by 0, square root of a negative number) can be triggered during the execution of the modified version of the loop. A first partial solution to overcome this difficulty is simply to mask interruptions during DECANNed loop execution but this will not be entirely satisfactory because instruction timing can be affected. To preserve timing properties, we insert load instructions from specific stack locations to provide division/square root operations with operands leading to standard timing. The overall performance impact of these extra load instructions will remain limited, first because across iterations, these extra loads will always access the same stack locations resulting in L1 hits, and second because the cost of these L1 hits is negligible with respect to the cost of a divide or a square root instruction.

2.3 Information Reporting

As DECAN works on small portions of the program it is essential for it to perform an accurate performance monitoring followed by a good source level correlation.

Performance monitoring In order to accurately capture the transformation impact, DECAN places its monitoring probes at the entry and exits of the loop. For that, it creates basic blocks and redirect the control flow coming to and going out of the loop to point to those blocks. Furthermore, several probe types are available:

- Timers by the insertion of RDTSC instructions inside the blocks.
- Iteration counters by the insertion of incrementable global variables with MADRAS.
- Hardware counters by the insertion of functions calls to an API provided by a hardware counter/event monitoring tool. In the case of DECAN we use Likwid [3] whose API functions serve as triggers to start and stop the counters.

As a result the profiling overhead is greatly minimized.

Correlating back to the source code Relating assembly instructions to source code level is straightforward when debug information (DWARF format) is available. Fortunately, INTEL compilers provide such an option without altering the quality, and therefore performance, of the code generated. The option assigns to each instruction its corresponding source line and source file path. In most of the reasonable cases (multidimensional arrays) it is also possible to get information about the array allocation and therefore enable connection between instruction groups and arrays.

2.4 Dealing With Parallel Codes

OpenMP codes: Despite the fact that compilers do not fully follow the code generation structure specified in the OpenMP standard, they remain close enough to it. This enables us to recognize the important OpenMP structures (such as parallel loops) and inject the probes according to their positions. Additionally since the same loop binary can be executed by different threads, the timing probes track down the thread ID and generate a separate report for each of them.

MPI codes: DECAN handles MPI applications in the same manner as it handles sequential ones. The main difference lies in the results reporting mechanism: in the MPI case a report file is generated for each process which runs the loop. As mentioned earlier the "no subroutine call" rule will force DECAN to skip loops containing standard MPI statements such as SEND and RECEIVE.

3. TRANSFORMATION FRAMEWORK

While the previous section dealt with the control flow which might be seen as the skeleton of the program, the current section will walk through the intra-block modifications by addressing the instruction selection and transformation process.

3.1 Overview

The DECAN version presented in this paper targets the x86 instruction set with a focus on SSE/AVX instructions (floating-Point x87 instructions using the stack are not addressed).
DECAN operates on binary loops. It uses a configuration file (called template) to specify how a binary loop is modified. A template consists of two major components:

- *LIST OF INSTRUCTION SUBSETS*: a subset of instructions (extracted from the loop body) on which DECAN will apply the same specific transformation.
- *LIST OF TRANSFORMATIONS*: on a given instruction subset, specific transformations can be applied such as simple deletion, opcode or operand modification and replacement by other instructions or insertion of new instructions.

For the LSI_STREAM template, the subset of instructions to be modified consists of all of the FP SSE/AVX instructions. To this subset, the following transformation process is uniformly applied: FP instructions involving only registers are simply suppressed while FP instructions having one operand coming from memory are replaced by an equivalent LOAD operation using the same operand and the same target register.
A DECAN variant is simply a DECAN template applied to one or several binary loops.
The DECAN operating mode will specify how the DECAN variants are executed (see section 2.3).
Since one of the issues is to deal with binaries, we will first describe how to modify binaries.

3.2 Identifying instruction subsets

For each binary loop, DECAN generates a detailed Data Dependency Graph, as well as the list of addresses and SSE/AVX registers used by the binary.
From that point, DECAN will identify different instruction subsets which are potential candidates for modification (see Table 1).
The *LS SUBSET* aggregates all of the read/write instructions and therefore is not sufficient to distinguish between accesses to different arrays. To achieve that goal, *LS SUBSET* is further subdivided into groups. A group is essentially a set of read/write operations which are addressing close by memory locations. For example, loads to A(I) and A(I+1) are set within the same group. The group generation is detailed in the next subsection.

SUBSET	Definition (instructions listed below are extracted from target loop body)
LS SUBSET	all of the SSE/AVX instructions having one of its operands operating to/from memory (either as destination or source)
L SUBSET	all of the SSE/AVX instructions having one of its source operands coming from memory
S SUBSET	all of the SSE/AVX instructions having as a target operand a memory location
FP SUBSET	all of the SSE/AVX FP arithmetic instructions
DIV SUBSET	all of the SSE/AVX FP divide instructions
RED SUBSET	all of the FP or LOGICAL instructions directly involved in a reduction
LOC SUBSET	all of the instructions involved in the loop branch control operations

Table 1: Instruction subsets

For each group, we also identify the subsets of instructions modifying the address registers (base and index) defining it.

3.3 Grouping

Analyzing performance globally for all memory instructions is too coarse.
The goal is to refine the scope of the analysis to something more meaningful and isolate delinquent instructions. However, discarding a single instruction is misleading in case of *hit under miss*. For instance, if *A[0]* is a miss, then the next access to *A*, *A[1]*, will be a hit. Discarding the access to *A[0]* will simply shift the miss to *A[1]*. To pinpoint the bottleneck accurately, an aggregation scheme to regroup accesses on a cache line basis (*A[0]-A[3]*) has to be designed.
Furthermore, aggregation eases measurement interpretation. Instead of a per cache line analysis, a per data structure analysis is more relevant for an application developer. The ability to discard all the accesses to a given data structure at the same time sorts the different data structures by cost.

	Assembly code	Groups
0	LOOP:	
1	MOVSS **(%RDI, %R8, 4)**, %XMM0	1 → G1
2	ADDSS **12(%RDI, %R8, 4)**, %XMM0	2 → G1
3	ADDSS **24(%RDI, %R8, 4)**, %XMM0	3 → G1
4	MOVSS %XMM0, *12(%RDX, %R8, 4)*	4 → G2
5	INC %R8	
6	CMP %R9, %R8	
7	JB LOOP	

Table 2: advanced data-flow analysis tracks the symbolic values of registers. Based on registers value it is then possible to infer which instructions are targeting the same data structure. Such instructions are coalesced within *groups*

Definition: A group is a set of memory accesses to the same data structure. The structure is usually an array, but it can also be a memory area used for spill-fill. Two instructions are considered to belong to the same group if they target an address using the same base and index register values, the only difference being the offset (see Table 2 for an example of groups).

As shown in the example depicted in Table 2 grouping analysis requires a partial knowledge of the execution context, which is evaluated through advanced dataflow analysis. Indeed, for each used register or memory address, an internal representation is used to keep its possible formal values.

3.4 DECAN Transformations

On each instruction subset DECAN can perform different modifications (see Table 3).

Transformation	Operations performed on all of the instructions of the target subset
Transformations applicable to any subset	
DELETION	Delete instruction
N1B	Instructions are replaced by a 1 Byte NOP
NMB	Instructions are replaced by a MultiByte NOP, the size of the NOP in bytes being equal to the size of the instruction replaced
Transformations applicable to L subset or a group	
L_L1_HIT	The memory address of each Load instruction is replaced by a constant address (different for each Load) on the stack. Store instructions are deleted
Transformations applicable to a group	
GROUP_PREFETCH	Before the first instruction of a group a software prefetch instruction is inserted using the base and index registers defining the group
Transformations applicable to S subset	
S2L	This transformation replaces Store opcode by an equivalent Load opcode, the operands being swapped: the source register (resp. memory target address) of the Store becomes the source register (resp. memory source address) for the Load

Table 3: Subset Transformations

The transformations *N1B* and *NMB* are useful because the simple *DELETION* "brutally" modifies the front end pipeline behavior: some instructions being suppressed, the remaining instructions are issued at a higher rate than in the original code. The use of the N1B and NMB transformations helps to keep the issue rate of those instructions close to the issue rate of the original code.
The last three transformations of Table 3 target specific potential performance problems:

- *L_L1_HIT*: The net effect of such a transformation is that across iterations, every load instruction will repeatedly hit the same location: after an initial miss, the subsequent accesses will be L1 hits. By comparing the transformed code timing with the original unchanged code, a quick estimate of performance penalty due to data access can be obtained. Using groups, this performance penalty can be precisely attributed to a given array access.

- *GROUP_PREFETCH*: This transformation assesses the hardware prefetcher efficiency by prefetch distance, thus allowing a precise investigation of memory access behavior.

- *S2L*: This transformation aims to evaluate the cost of false sharing or more generally the cost of coherency operations. Simply suppressing the store will induce a potential reduction of the cache footprint of the code, while replacing the Store by a Load keeps the same cache footprint as the original code.

The transformations listed can have side effects with potentially large impact on performance and this defeats our purpose of tight control of modification impact. For example, deleting a single load instructions can introduce dependencies between two arithmetic instructions which were independent in the original code. To prevent this, specific instructions (zeroing a register) are inserted: the process is exactly similar to the one used for breaking dependencies within reduction constructs.

More complex transformations (not described here for lack of space) are also available: one of particular interest targets the stride associated with a group. In such a case, the stride is replaced by a smaller one to avoid writing out of array bounds. This transformation is particularly useful to assess the cost of strides accesses.

3.5 DECAN limitations

First, DECAN has only been developed and tested on X86 processors. However, most of the techniques can be easily transferred to other ISAs and very likely at a much lower cost because the X86 instruction set is one of the largest and hence one of the most complex.

Second, although DECAN overhead can be precisely controlled, it must be evaluated. When applied to a whole application a single DECAN transformation will at most double the execution time of its target loop due to the replay code inserted after each DECANNed version. If many variants have to be tested, then this factor would be multiplied by their number.

Third, DECAN can be limited by measurement accuracy. Let us consider a loop accessing 50 different arrays, all of them being accessed from the same memory hierarchy level and having similar data access behavior (same stride, same alignment). Using L_L1_HIT on one of the arrays will show at best a 2% variation in execution time, which is within standard measurement margin errors. Therefore, no conclusions can be reached on any individual data array access behavior. Fortunately, most of the loops access less than 20 arrays, making the example mentioned clearly a corner-case.

Fourth, in cases of complex control flows with many branches within the innermost loops, preserving the original flow during execution will limit the number of instructions eligible for modification by DECAN. Therefore, the potential for exploration will be reduced. It should be noted that, in practice, this problem was never encountered for HPC codes. For example, in all of the NAS benchmarks, all of the hot innermost loops have a simple control flow that is easily handled by DECAN.

3.6 DECAN Summary

Table 4 illustrates how DECAN operates on a binary. The leftmost column contains the assembly code of an example loop. Instructions being part of the *LS SUBSET* are in italic and those of the *FP SUBSET* are in bold. The middle and rightmost columns show the loops resulting from the transformation process. Beyond the deleted and replaced instructions we can see the change in the address of the branch instruction in the modified versions which is due to the code being moved by MADRAS.

ORIGINAL CODE	
MOVSD -0x8(%RDX,%R9,8),%XMM0 INC %R8 ***MULSD -0x8(%RCX,%RAX,1),%XMM0*** *MOVSD 0(%RDI,%RDX,1),%XMM1* *MOVSD 0x8(%RDI,%RDX,1),%XMM3* **SUBSD %XMM0,%XMM1** *MOVSD %XMM1,0(%RDI,%RDX,1)* *MOVSD -0x8(%RDX,%R9,8),%XMM2* ***MULSD -0x10(%RCX,%RAX,1),%XMM2*** ADD $-0x10,%RCX **SUBSD %XMM2,%XMM3** *MOVSD %XMM3,0x8(%RDI,%RDX,1)* ADD $0x10,%RDI CMP %R10,%R8 JB 402c40	
LSI_STREAM	**FPI_STREAM**
MOVSD -0x8(%RDX,%R9,8),%XMM0 INC %R8 MOVSD -0x8(%RCX,%RAX,1),%XMM0 MOVSD 0(%RDI,%RDX,1),%XMM1 MOVSD 0x8(%RDI,%RDX,1),%XMM3 MOVSD %XMM1,0(%RDI,%RDX,1) MOVSD -0x8(%RDX,%R9,8),%XMM2 MOVSD -0x10(%RCX,%RAX,1),%XMM2 ADD $-0x10,%RCX MOVSD %XMM3,0x8(%RDI,%RDX,1) ADD $0x10,%RDI CMP %R10,%R8 JB 6b897e	XORPS %XMM0,%XMM0 INC %R8 MULSD %XMM0,%XMM0 XORPS %XMM1,%XMM1 XORPS %XMM3,%XMM3 SUBSD %XMM0,%XMM1 XORPS %XMM2,%XMM2 MULSD %XMM2,%XMM2 ADD $-0x10,%RCX SUBSD %XMM2,%XMM3 ADD $0x10,%RDI CMP %R10,%R8 JB 6b897e

Table 4: Example of subset transformations. Two variants were generated one for LSI_STREAM and the other for FPI_STREAM. In the leftmost column, instructions in italic correspond to the LS SUBSET and those in bold to the FP SUBSET

4. EXPERIMENTAL RESULTS

4.1 Experimental Setup

All experiments were conducted on a XEON E31240 processor (Sandy-Bridge) with 4 cores running at 3.30GHz. Each core has a private L1(32 KB) and L2 (256 KB) whereas the L3 (8 MB) is shared among all.
All applications were compiled with the Intel Fortran Compiler (ifort 12.1.4) and the -O3 flag and the runs were made under LINUX OS.

4.2 RTM Application: Focusing optimization effort on the right loop

Reverse Time Migration (RTM) [6] is a standard algorithm used for geophysical prospection. The code used in this study is an industrial implementation of the RTM algorithm.

Our RTM code operates on a regular 3D grid. The core of the domain is processed uniformly, but a specific process is applied on the borders of the domain, the skin of the domain, to annihilate potential wave reflections. From a performance perspective, more than 90% of the execution time of the application is spent in two functions, *inner* and *damping*. These two functions are executing similar code on two different parts of the domain, *inner* is devoted to the core of the domain, while *damping* is used on the skin of the domain. Standard domain decomposition techniques are used to

spread the workload on multicore target machines. Since the grid is uniform, load balancing is easily achieved by using rectangular sub-domains.

In our study, we focused on the *inner* routine which represents a much larger execution time than *damping*.
In the *inner* function, 3 loops denoted Lapx (resp. Lapy and Lapz) are used to compute the 3 components of the finite difference operator along the 3 axis (X, Y and Z). A fourth loop computes the wave-field. These four loops are themselves surrounded by 2 outermost loops to complete the operator computation over the entire sub-domain. The sub-domain size is too large to fit in the L1 or L2 caches, therefore cache blocking has to be used to optimize cache usage.

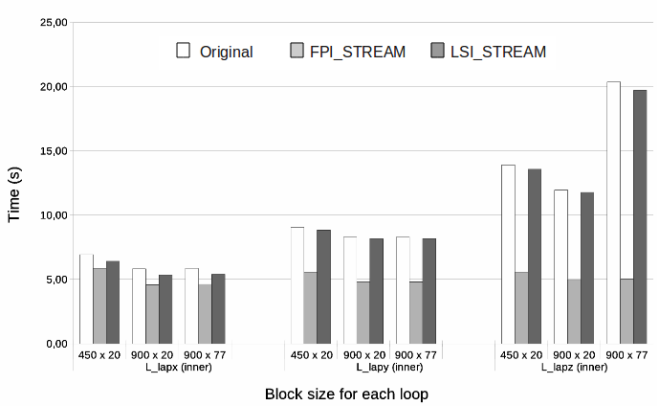

Figure 2: Stream performance comparison for the RTM application. For the three functions *lapx*, *lapy* and *lapz*, histograms represent respectively the speed of arithmetic operations and memory accesses depending on the blocking used for the data.

Figure 2 presents the results obtained by running the original code, the LSI and FPI_STREAM variants. Blocking was performed only along 2 dimensions, X and Y, and 3 different blocking strategies were studied: (Bx = 450, By = 20), (Bx = 900, By = 20), (Bx = 900, By = 77) where Bx (resp. By) denotes block size along the X (resp. Y) axis. The leftmost (resp. center and rightmost) histograms refers to Lapx execution time (resp. Lapy and Lapz). The leftmost and center histograms indicate that FPI_STREAM and LSI_STREAM are very close. This reveals that data access penalty is limited (not worth being optimized) and furthermore not very sensitive to block size. At the opposite, the large gap between FPI_STREAM and LSI_STREAM for Lapz reveals a large data access penalty: therefore the blocking strategy effort should only be focused on Lapz.

4.3 Timing The Access To Individual Memory Structure: EUFLUX application

EUFLUX is a 3D finite element CFD application from Dassault [9], ITRSOL (the iterative solver) represents over 80% of the execution and is the most time consuming routine is EUFLUXm. The EUFLUXm routine implements a sparse matrix-vector product in a quadruply nested loop. Among the four different arrays, three of them, *VECX*(2D), *OMPU*(3D) and *OMPL*(3D), are read-only and the last one *VECY*(2D) is read and written. The code of EUFLUXm is presented in Figure 3.

A quick inspection of the code reveals that most of the array

```
do icb=1,ncbt
  igp   = isg
  isg   = icolb(icb+1)
  igt   = isg - igp
  do ig=1,igt
     e     = ig + igp
     i     = nnbar(e,1)
     j     = nnbar(e,2)
     do k=1,ndof
        do l=1,ndof
           vecy(i,k) = vecy(i,k) + ompu(e,k,l)* vecx(j,l)
           vecy(j,k) = vecy(j,k) + ompl(e,k,l) * vecx(i,l)
        enddo
     enddo
  enddo
enddo
```

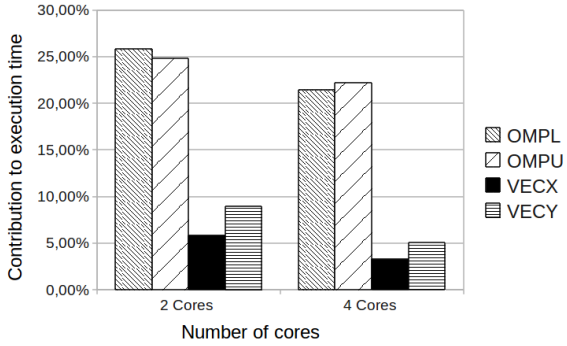

Figure 3: The upper figure presents the source code of the matrix-vector product in EUFLUXm. The lower figure shows the individual contribution in the overall execution time of memory instructions targeting each array of the EUFLUXm routine. Results are presented for 2 and 4 cores.

accesses suffer from bad stride access. The two innermost loops could be interchanged or alternatively arrays could be restructured (transposed). Restructuring an array is a complex and expensive code modification because it has to be propagated throughout the whole application. Therefore the real issue is to determine which arrays should be prioritized for restructuring.

The groups analysis (see Section 3.3) applied on the loop shows that among the 4 arrays, the accesses to *OMPU* and *OMPL* in Figure 3 are the most time consuming. The two arrays represent more than 40% of the total execution time, followed by *VECY* and *VECX* with an individual time share lower than 10%. Any optimization effort has to consider that performance issues in the loop are essentially tied to the *OMPU* and *OMPL* arrays.

4.4 Cache Coherence Protocol Analysis

Cache organization is a critical design decision in multicore processors; we propose to apply differential analysis in order to investigate the cost of cache coherence protocols. This study is conducted on the OpenMP version of the RTM application considered in the first case study. From a coherency perspective RTM is an interesting case: the code consists of iterations over a data parallel loop. Between two consecutive iterations the same array is written then read in the following iteration. First the write operations can generate false sharing: a cache line being simultaneously accessed in write mode by two different cores. Furthermore, the computation on a sub-domain requires access to data from other neighboring sub-domains: typically, all of the data on the border of the

sub-domain are first written by a core and then in the next iteration read by a different core. Such writes followed by reads on the same memory location induce coherency traffic.

The methodology of measurements involves a simple transformation (S2L): all the store instructions of one loop are replaced by load instructions without changing either the memory location or the accessed register. Such a transformation will annul all of the coherency actions which would have been normally triggered by the store. Figure 4 denotes the execution time of all the innermost loops executed. The bars corresponding to the S subset with the S2L transformation are all at the same level (one) indicating a zero performance impact. For reference, the variant based on simple store deletion has been run. The results show an almost equivalent performance with the original versions except for two loops which present a non negligible store cost. From these results it can be stated that the cache coherence overhead remains negligible for the RTM application.

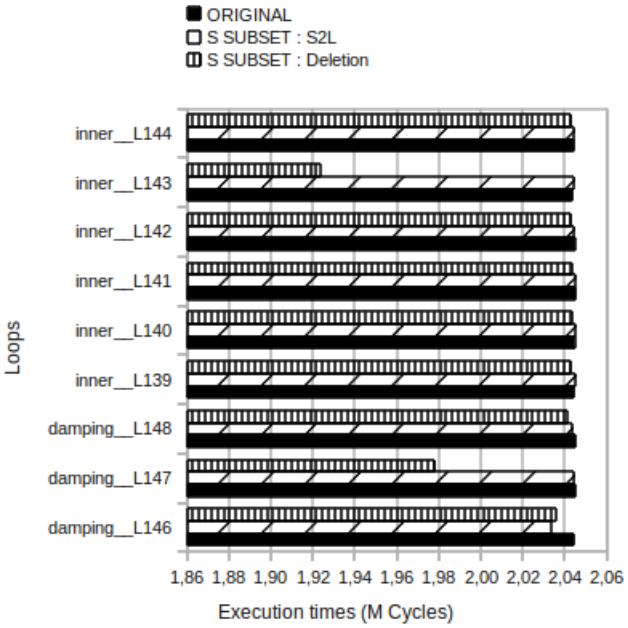

Figure 4: Evaluation of the cost of cache coherence protocol. The S SUBSET with S2L transformation variants show similar performances as their corresponding original versions. The S SUBSET with DELETION transformation variants show similar performances also, except for two loops which present a relatively non negligible store cost.

4.5 NAS Benchmarks Characterization

In order to test the scalability of the DECAN tool regarding complex codes, we chose the NAS Parallel Benchmarks *NPB-OMP3.0* as a test suite and performed a basic characterization of the behavior of its parallel loops. As known, most of the NAS Benchmarks implement iterative algorithms, therefore the alteration of the loops semantic leads to an earlier end of the program. This issue allowed us to use the sampling technique introduced in Section 2.2.
Figure 5 presents the results obtained on five of the benchmarks: *BT*, *CG*, *LU*, *SP* and *EP*. For each of them the hottest functions are selected and only their loops were considered. Two DECAN templates were applied: LSI_STREAM and the FPI_STREAM.

The results provide a clear insight on the trend of each benchmark. In the *BT* benchmark for example, the first two loops are driven by memory operations. As the execution time of the loop decreases we see more balance between the two versions. The loops of *CG* are highly memory bound. The hint for this is given by the big gap between the ratios of the two templates, especially in the upper loops. The only binary loop captured in *EP* performs a series of stores which explains the extremely low ratio for the FPI_STREAM version.

BT

LOOP	% of Exec time	LSI_ STREAM	FPI_ STREAM
z_solve__L162	**6,87%**	0,96	0,30
y_solve__L155	**5,52%**	0,96	0,45
x_solve__L146	**5,27%**	0,92	0,59
z_solve__L161	**4,20%**	1,00	0,79
y_solve__L154	**4,15%**	1,00	0,79
x_solve__L147	**3,42%**	1,00	0,79
compute_rhs__L134	**0,17%**	0,94	0,87
y_solve__L151	**0,11%**	0,93	0,83
z_solve__L158	**0,10%**	0,94	0,85
x_solve__L143	**0,10%**	0,95	0,90

SP

LOOP	% of Exec time	LSI_ STREAM	FPI_ STREAM
z_solve_j__L264	**1,92%**	1,00	0,32
z_solve_j__L270	**1,58%**	0,99	0,14
y_solve_k__L243	**1,44%**	1,00	0,77
y_solve_k__L245	**1,34%**	1,00	0,44
x_solve_k__L219	**1,33%**	0,94	0,64
z_solve_j__L266	**1,31%**	0,90	0,79
y_solve_k__L241	**1,27%**	0,92	0,28
x_solve_k__L224	**1,13%**	0,99	0,51
x_solve_k__L222	**1,07%**	0,99	0,76
y_solve_k__L247	**1,04%**	0,92	0,20

EP

LOOP	% of Exec time	LSI_ STREAM	FPI_ STREAM
MAIN___L6	**100,00%**	1,00	0,01

CG

LOOP	% of Exec time	LSI_ STREAM	FPI_ STREAM
conj_grad__L25	**26,38%**	1,00	0,02
conj_grad__L13	**24,89%**	1,00	0,10
conj_grad__L3	**24,44%**	1,00	0,05
conj_grad__L16	**12,27%**	0,98	0,09
conj_grad__L9	**7,14%**	1,00	0,13
conj_grad__L23	**2,39%**	0,98	0,50
conj_grad__L19	**0,37%**	0,99	0,30
conj_grad__L6	**0,35%**	1,00	0,39
conj_grad__L12	**0,19%**	0,98	0,78
conj_grad__L14	**0,18%**	0,94	0,85

LU

LOOP	% of Exec time	LSI_ STREAM	FPI_ STREAM
rhs__L89	**4,33%**	0,98	0,16
rhs__L97	**3,45%**	1,00	0,41
rhs__L107	**3,39%**	0,90	0,39
rhs__L105	**2,94%**	1,00	0,24
rhs__L116	**2,13%**	0,90	0,46
rhs__L106	**1,86%**	0,94	0,89
rhs__L118	**1,39%**	0,94	0,62
rhs__L100	**1,73%**	0,89	0,59
rhs__L96	**1,68%**	0,93	0,83
rhs__L103	**1,36%**	0,95	0,78

Figure 5: Results of the application of the two DECAN templates LSI_STREAM and FPI_STREAM on 5 of the NAS Parallel Benchmarks 3.0. In each table the 10 hottest parallel loops are considered. For the two templates applied on each loop, the ratio of the execution time on the original one is reported.

5. RELATED WORKS

The comparison with previous works needs to be done at two levels: first, the basic mechanisms used for understanding low-level interaction between hardware and software and second how these mechanisms are integrated in a performance evaluation tool.

In terms of basic mechanisms, the closest work to DECAN is the technique proposed by Fursin et al. [10] to determine a lower bound on execution time. In their work, Fursin *et al.* use assembly level instrumentation to determine the lower bound on the execution time. From that bound they derive the potential gain of optimizations. Basically the idea is to transform all memory accesses in order to target a unique memory location. The goal is to lead all memory instructions into triggering cache hits. The timing of such a modified code determines a best case performance where all data accesses are L1 hits. They also introduce the idea of recovery code to overcome the semantic loss. Our DECAN greatly extends this idea by applying instruction suppression/modification to a much larger class of instructions (not only load/store) but also by using much more complex modification schemes (cf. section 3).

Besides modifying the binary as proposed in DECAN, there are two standard approaches for analyzing low-level interactions between hardware and software: 1) hardware performance counters/ events and 2) simulation tools.

5.1 Hardware Performance Counters

For two decades hardware counters/events have attracted a large deal of attention [13]. Counters have the ability to monitor various events occurring within the processor at very low overhead without altering the numerical output of the system. Hardware counters are consequently used in a large number of tools; they are now natively supported in the Linux kernel since the version 2.6 through the *perf* subsystem [8].

Hardware events are excellent at capturing how a given piece of hardware is used but very often they fail in evaluating the exact performance impact: for example, hardware events can count cache misses but what matters is not the number of cache misses but the total impact of cache misses on performance i.e. the product of the number of cache misses and the average cost of cache misses. In general, hardware counters fail at evaluating accurately such average costs. In contrast, DECAN can directly provide a detailed estimation on how much a given array access costs.

Another difficulty with hardware counters is the correlation between source code and hardware event counts. Hardware counters can be triggered on and off at the beginning and the end of a loop structure or a function but the results are reported globally for the whole loop/function and not for individual source code statements.

To refine further the link between performance issues and source code, architects have extended the counter monitoring capabilities with Precise Event Based Sampling (PEBS) [11]. PEBS is introduced to link exactly a performance event with the delinquent instructions. First due to the statistical nature of the loop, there must be a sufficient number of samples to get realistic information. In general, this prevents using these mechanisms on too short loops (less than a few ten thousands of cycles). Second, due to the reordering mechanism of most x86 processors, it remains challenging to pinpoint the offending instruction [13].

In contrast to performance counters, DECAN can operate on loops with a very short duration (1000 cycles), the main limitation being the timer accuracy. Furthermore, the transformations are directly performed on a few selected subsets of instructions allowing a direct correlation between performance impact and the binary/source code to be made.

One final difficulty with hardware performance events is their complexity. First, their number is fairly high (in general over a thousand) making them hard to use. Secondly, many of them refer to low-level microarchitectural details which are not publicly available making counter information hard to decipher. For example, knowing that the reservation station is full and generates partial stalls in the front end pipeline does not give a precise clue of what to do to optimize the code. Interestingly enough, when used in conjunction with hardware events, DECAN helps in the identification of the instructions which are causing reservation station overflow.

5.2 Cache Simulators

In order to bridge the gap between the low-level information brought back by hardware counters and the real needs of the optimization, cache simulators are used. The main drawback of these simulators is the large overhead induced by the need to track every memory access. Among the performance analysis tools built on the top of cache simulators two well-known tools are detailed below.

Cachegrind: Cachegrind [4] is the cache profiler included in the Valgrind instrumentation framework. When using Valgrind, the original instructions never run on the host processor. Instead, Valgrind converts instructions on-the-fly to an intermediate representation. Valgrind companion tools can easily and directly manipulate the intermediate representation which is then recompiled for the target architecture. Cachegrind is based on the simulation of configurable L1I, L1D, and L2 caches. It identifies the number of cache misses for each line of the source code, with per-function, per-module and whole-program summaries.

ThreadSpotter: Acumem AG [1] offers the commercial product ThreadSpotter specially targeted at analyzing data access issues. It relies on a statistical analysis of address traces to estimate various potential problems with data access (stride, false sharing).

Both Cachegrind and ThreadSpotter are excellent at capturing general code properties such as temporal and spatial localities. However, since both of them are using very simplified cache models they fail to correctly assess the performance impact of the problems detected. Finally, since they focus only on data access, they are missing all of the potential performance problems relative to the arithmetic units.

5.3 Performance Evaluation Tools

Scalasca: Scalasca mainly focuses on MPI programs and is very efficient for quickly identifying communication problems such as late sender-early receiver. For OpenMP programs, Scalasca can identify load balancing issues in data parallel loops and synchronization issues. Scalsca uses source level instrumentation which is well suited for the communication problems listed above (minimal interference with the compiler). However for more general performance bottlenecks Scalasca does not provide any specific exploration technique besides hardware performance counters which by themselves are not always adequate.

TAU: Tuning and Analysis Utilities (TAU) Performance System [14, 15] is a performance profiling and tracing framework. As

such, it offers much more flexibility in the performance investigation techniques than Scalasca. The TAU framework addresses performance problems on three levels: instrumentation, measurement, and analysis. It provides instrumentation at different levels and performs tracing on parallel programs.
Although TAU offers many possibilities of using (inserting / triggering) various performance counters, it basically inherits all of the key limitations of hardware performance counters and in many cases, it will not be very helpful for performance bottleneck investigation.

PerfExpert [7] goes a step further by trying to analyze performance bottlenecks and provide optimization guidelines. Again, it mostly rely on hardware performance counters to evaluate performance problems and suffers from the same problems as the others. However, the approach of synthesising hardware counter information to derive performance optimization is very powerful.

XE Amplifier: XE Amplifier [2] is an Intel tool for performance analysis. It has different features, including stack sampling, thread analysis and hardware event sampling. Some traditional features, such as identifying the hottest modules and functions in a whole application or tracking call sequences, are also supported. XE Amplifier leverages hardware counters for in-depth analysis of the memory system and architectural tuning and associates performance issues with the source code. If no symbol sources are found in the binary, XE Amplifier navigates through the disassembled code at a basic bloc granularity.

With a reduced set of tests (variants), DECAN can offer the ability to determine the source of issues that are worth investigating (high potential Return On Investment). Complementary analyses such as the ones provided by PerfExpert or XE Amplifier can be used to derive what are the most profitable optimizations.

6. CONCLUSIONS AND FUTURE WORK

This article presents a novel approach based on partitioning the instruction flow according to either the hardware resource accessed or the memory targeted. Generating different variants from the original binary allows the cost of each fraction of the flow to be quantified. We call the general method the "differential" analysis and propose the DECAN tool as an implementation. DECAN by its concept is intrusive and can break the numerical stability of the application; numerical accuracy is not an important matter in terms of performance analysis except in cases where the control flow of the application is altered. However, the tool supports multiple schemes to detect or circumvent control flow hazards introduced by the differential analysis.
Through several case studies, we have demonstrated how DECAN can identify and quantify performance bottlenecks. We have also shown the current main DECAN limitations: sensitivity to timer accuracy, potential difficulties in interpreting the results and finally limitations in dealing with complex code structure.

Future work will first focus on addressing the limitations listed above. Furthermore, we plan to develop a systematic methodology for generating variants and combining the obtained results in a more meaningful manner.

Acknowledgments

This work has been carried out by the Exascale Computing Research laboratory, thanks to the support of CEA, GENCI, Intel, UVSQ, and by the PRiSM laboratory, thanks to the support of the French Ministry for Economy, Industry, and Employment throught the ITEA2 project H4H. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the CEA, GENCI, Intel, or UVSQ.

7. REFERENCES

[1] Acumem. `http://www.roguewave.com/`.
[2] Amplifier xe. `http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe-documentation/`.
[3] Likwid: Lightweight performance tools. `http://code.google.com/p/likwid/`.
[4] Valgrind: instrumentation framework for building dynamic analysis tools. `http://www.valgrind.org`.
[5] D. Barthou, A. Charif Rubial, W. Jalby, S. Koliai, and C. Valensi. *Performance Tuning of x86 OpenMP Codes with MAQAO*, page 95. 2010.
[6] E. Baysal, D. Kosloff, and J. Sherwood. Rerverse time migration:geophysics, 1983.
[7] M. Burtscher, B.-D. Kim, J. Diamond, J. McCalpin, L. Koesterke, and J. Browne. Perfexpert: An easy-to-use performance diagnosis tool for hpc applications. In *2010 ACM/IEEE*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.
[8] A. C. de Melo. Performance counters on linux: the new tools. 2009.
[9] Q. V. Dinh, A. Naim, and G. Petit. rapport final de synthèse sur l'optimisation des logiciels de simulation numérique de l'aéronautique. *Technical report, Dassault Aviation*, pages xii, 51, 53, 54, 70, 71, 83, 2007.
[10] G. Fursin, M. F. P. O'Boyle, O. Temam, and G. Watts. A fast and accurate method for determining a lower bound on execution time: Research articles. *Concurr. Comput. : Pract. Exper.*, 16(2-3):271–292, Jan. 2004.
[11] D. Levinthal. Performance analysis guide for intel® coretm i7 processor and intel® xeontm 5500 processors.
[12] H. Lu. Elf: From the programmer's perspective. *NYNEX Science and Technology Inc*, page 95, 1995.
[13] T. Moseley, N. Vachharajani, and W. Jalby. Hardware performance monitoring for the rest of us: a position and survey. In *Proceedings of the 8th IFIP international conference on Network and parallel computing*, NPC'11, pages 293–312, Berlin, Heidelberg, 2011. Springer-Verlag.
[14] S. Shende, A. Malony, S. Moore, P. Mucci, and J. Dongarra. Integrated tool capabilities for performance instrumentation and measurement. 2007.
[15] S. S. Shende and A. D. Malony. The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20:287–331, 2006.
[16] C. Valensi and D. Barthou. Madras: Multi-architecture binary rewriting tool. Technical report.