MicroTools: Automating Program Generation and Performance Measurement

Jean Christophe Beyler
Intel Corporation
jean.christophe.beyler@intel.com

Farid Chabane Thibault Fighiera
Jean-Philippe Halimi William Jalby
Vincent Palomares Nicolas Triquenaux
Exascale Computing Research
(first.last)@exascale-computing.eu

Abstract

Tuning an application to a given architecture has become a complex procedure. Sophisticated hardware obfuscates the path to easily writing peak-performance applications. During the optimization process, before utilizing the hardware correctly, the user must understand out-of-order execution and micro-operations.

Understanding the software's performance on a given target architecture is the goal of the *MicroCreator* and *MicroLauncher* tools. MicroCreator automatically generates a set of benchmark programs from a XML file, whereas MicroLauncher executes them in a stable and closed environment.

With these tools, the user has a better understanding of the underlying architecture. The two programs, through the execution of hundreds of micro-programs and from a single-core execution to the parallel world using OpenMP, give insight on performance issues. Looking into unrolling, strided memory accesses, vectorized programs, and parallel programs allow quick and efficient calculations of the latencies and bottlenecks of the architecture.

1. Introduction

It is becoming increasingly complex to understand performance of current applications on modern hardware: whether the architecture shares the caches or not, what algorithms are included in the prefetch mechanisms, how instructions are decoded, how many queues exist in the processor, and the out-of-order execution paradigm, are only a handful of elements making performance predictions difficult.

Performance analysis is either static, dynamic, or a combination of both. Static analysis allows the comprehension of the performance by analyzing the program's assembly or even binary code [7]. Profilers such as gprof [12] or Vtune [19] allow the comprehension of general program execution. Static solutions cannot handle run-time variables and context; dynamic techniques use techniques such as sampling to reduce overhead but automatically introduce a loss of precision. Furthermore, the tools focus on application optimization and understanding the lack of performance; whereas, architecture knowledge is as important, if not as necessary, to the tuning process.

MicroCreator and MicroLauncher, the two presented tools, provide a methodology to understand the behavior of code variations on a given architecture in a simple and empirical manner. The initial step to understanding complex applications, especially during parallel executions, is understanding micro-kernel programs and their interaction with the underlying architecture, the operating system and any environmental noise. Running lengthy studies is impossible with the decreasing lifespan of new architectures. These MicroTools are the solution. MicroCreator creates a plethora of microbenchmark programs and MicroLauncher executes them on a given machine in a quick and efficient manner. The generated benchmarks, whether for sequential execution or multi-core executions, give insight into the effects of unrolling and strides as well as streaming loads and stores.

The MicroTools contributions are:

- Provide an automatic solution for microbenchmarks generation in an easy and reusable way
- Illustrate the usefulness of MicroLauncher with key microbenchmarks to characterize hierarchical memory latencies
- Variations give insights on architecture specifications with an automatic process

The remainder of the paper is organized as follows. First, the paper presents a motivation example by considering the

```
void multiplySingle (int iter, double *A,
                     double *B, double *C)
{
 for (i = 0; i < iter; i++)
 {
  double *first = A + i * iter;
  double *second = B + i * iter;
  for (j = 0; j < iter; j++)
  {
   double *res = first + j;
   *res = 0;
   for (k = 0; k < iter; k++)
   {
    double *third = C + k * iter;
    *res += second[k] * third[j];
   }
  }
 }
}
```

Figure 1. Naive matrix multiply

classic matrix multiplication algorithm. By considering a simple implementation, the paper shows how the MicroTools are able to test variations and help define a few optimizations for a given architecture. Then, MicroCreator is presented as a code generator which creates, from a single description file, as many variations as requested. Next, the paper presents MicroLauncher, the microbenchmark program launcher, restraining the environmental noise to a minimum; which is essential to studying the effects of small code variations. Finally, experimental results and related works are provided.

2. Motivation: Matrix Multiply

A common computer science problem is optimizing matrix multiplications on a given architecure. Mathematic libraries such as MKL and ATLAS work hard to provide the right implementation to achieve optimal results. There are many different aspects of a matrix multiply algorithm: the matrix alignment, the matrix sizes for future tiling optimizations, and the assembly instructions used for the kernel itself.

The motivation of the MicroTools is to consider a kernel and try code variations and runtime environment to find the optimal point. The matrix multiply implemementation for the example uses single arrays, as shown in Figure 1. It contains no special optimizations and is the tuning process starting point. Figure 2 shows the generated inner assembly kernel using the Gcc compiler with the -O3 optimization flag. The obtained code is relatively straight forward with a load, a multiply including a load, an addition, and a store.

Optimizing the matrix multiply code is however not straightforward, due to the three matrices. The example considers square matrices and, depending on the size of the

```
.L3:
 movsd (%rdx,%rax,8), %xmm0
 addq $1, %rax
 mulsd (%r8), %xmm0
 addq %r11, %r8
 cmpl %eax, %edi
 addsd %xmm0, %xmm1
 movsd %xmm1, (%r10,%r9)
 jg .L3
```

Figure 2. Naive matrix multiply's inner assembly code

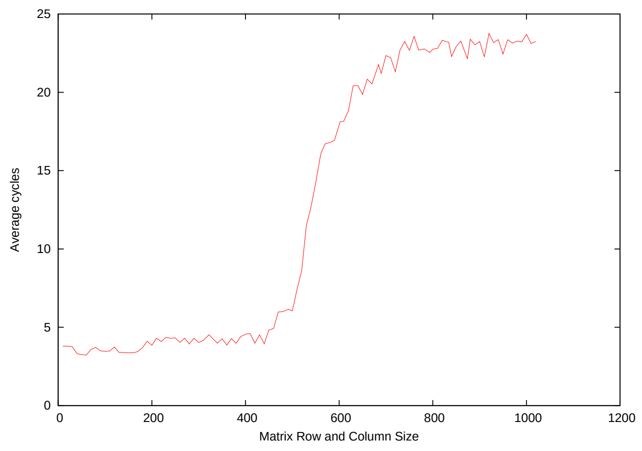

Figure 3. Cycles per iteration using different matrix sizes for the Matrix Multiplication

matrices, the data resides in different hierarchy levels. To determine the hierarchy used by a given size, consider the performance cycle per iteration when varying the size. Figure 3 provides the cycle count when varying the size. As the cycles increase, the matrix multiplication takes place higher in the memory hierarchy. The optimal size for matrix multiplications is used by optimizations such as tiling. Tiling, a very well known Matrix Multiply optimization, allows the complete multiplication to be performed in steps, each tile being calculated separately before combining the results. The right tiling size is a correct ratio between space and temporal locality. Correct size permits a full calculation step to hold in the caches. The following studies consider 200 * 200 matrices, which fit in the cache. In the figure, it is clear that 500 is one of the cutting points in performance for the matrix multiply on the considered architecture. Table 1 in Section 5 provides the architecture information.

Each matrix can have a different alignment; therefore, MicroLauncher, one of the two MicroTools programs, allows automatic alignment testing. By applying different alignments and measuring the performance per iteration, it is possible to obtain the optimal alignment for a given size. Consider Figure 4, it represents cycles per iteration obtained with different possible alignments for each matrix. The test

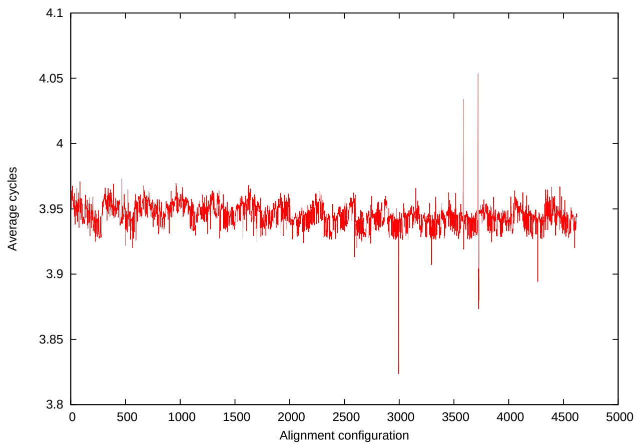

Figure 4. Cycles per iteration using different matrix alignments for the Matrix Multiplication

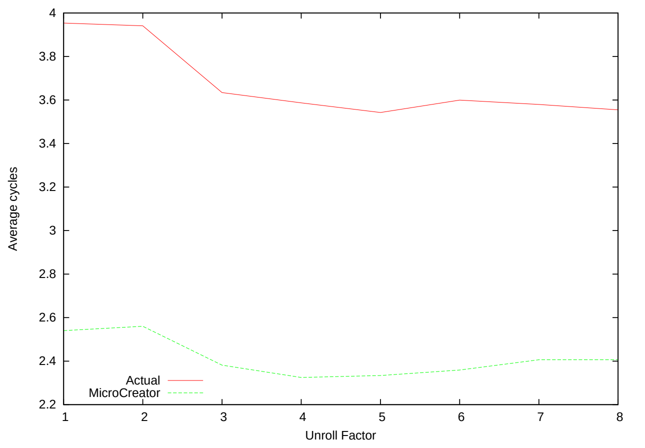

Figure 5. Comparing cycles per iteration using different unrolling factors for the Matrix Multiplication and between the actual code and the micro-benchmarks

was performed on 200 * 200 elements for each matrix. On the considered hardware, with a 200 * 200 size, the chosen alignment does not impact the 200 * 200 matrix multiply. The variation is less than 3% for any alignment configuration. However, in Section 5.2.2, two examples are given where alignment does have a 30% impact.

Finally, consider the matrix multiply assembly code. An easy optimization to apply is unrolling but selecting the correct factor is often a difficult problem. By abstracting the assembly operations in the MicroCreator format and testing various unrolling factors, the MicroTools study the kernel's performance variation. Consider Figure 5, it shows the cycle count per unrolling factor for the 200 * 200 factor between the original code and the microbenchmark equivalent. As it is seen, unrolling provides a 9% difference between not unrolling the code and unrolling it eight times. In the MicroTools version, the expected improvement was 8.2%, which is similar. Therefore, in the 200 * 200 matrix multiplication case, the programmer can either use compiler assisted hints to correctly unroll the code or rewrite the code in assembly format, in order to optimize the code.

The previous example shows the real motivation behind the MicroTools: tuning a given kernel is not an easy task and many factors affect the final results. Each architecture and each kernel are different and, by considering a given problem and testing slight variations in the code or runtime environment, the MicroTools help automate the tuning process and provide insight to optimal points in the domain search. The following sections detail the two tools and how both interact.

3. MicroCreator

MicroCreator automatically creates micro-programs for evaluating effects of minor changes in a program on an architecture. The tool allows a user to easily create thousands of variations of a given XML format template, generating the user desired programs. In one example, presented in the evaluation section, MicroCreator automatically generates more than two thousand benchmark programs from a single input file. These benchmark programs would not have been tested otherwise because generating them by hand or even with semi-automatic methods would be too time consuming. The generated programs are either in assembly format or in C source code.

3.1 Unrolling Loads and Stores

The following section shows the input format for MicroCreator. The input file example generates kernels using the regular expression (Load—Store)+.

Figure 6 shows an example kernel. A kernel is a series of instructions, which can form a loop for example. An instruction node in the XML defines a load *movaps* instruction or store instruction. A memory operand followed by a register operand represents a load instruction. A store instruction is the opposite. The memory operand offset allows the definition of the memory offset when required. When using XMM registers, provide their name with a minimum and maximum field so as to generate a different XMM register per unrolling iteration. Doing so reduces register dependency. Consequently it is important to explicitly define the use of the term *unrolling*.

The unrolling node signifies the desire to unroll the kernel from one to eight. An example includes two induction variables. The *movaps* instruction uses *r1*, a logical register name. The hardware detection system associates *r1* to a physical register such as *%rsi* or *%rdi*. The second induction variable, *r0*, is the counter used to continue or stop the loop. It is considered linked to the first because of the striding or unrolling of the first register *r1*. The same behavior is desired for *r0*. The last node in Figure 6 provides the branch information via a label and the jump instruction. MicroCreator also allows the user to provide move semantics, such as the number of bytes to be moved, without specifying exactly which instruction to use. Doing so enables MicroCreator to try different variations such as aligned versus non-aligned instructions or using vectorized or scalar instructions. The variability allows the tool to create as many examples as required to fully test the hardware.

3.2 MicroCreator's Passes

The MicroCreator compiler currently contains nineteen passes. Figure 7 summarizes various passes in the source-to-source compiler. The first instruction selection pass handles instruction repetition and random instruction selection. Instruction selection is a generic instruction scheduling pass which generates as many microbenchmark programs the user requires.

The creator then selects the strides for each induction variable and the values of the immediate variables. For each element, if there are multiple choices, a separate version of the kernel is created. The user can limit the number of benchmark programs if it is superfluous. After, the system executes the operand swap phase, the unrolling, the operand swap after the unrolling, register allocation, and inserting the induction variables into the instruction stream. Finally, the creator generates the obtained code.

There are two operand swap passes augmenting the tool's versatility. Consider a twice unrolled load instruction. When the tool swaps the operands before the unrolling, it generates either two loads or two stores. If the tool swaps after the unrolling, it creates the same two benchmark programs but one program with a load instruction followed by a store instruction also. In addition, a final program is created with a store instruction followed by a load instruction.

The passes included in MicroCreator give the user a powerful tool allowing a myriad of possibilities when generating microbenchmark programs.

3.3 MicroCreator's Plugins

To further augment the possibilities for various users, MicroCreator provides a plugin system resembling the GCC technique [9]. Users provide their dynamic libraries containing a few predefined functions. A user may replace or rewrite any of the internal passes with the fully exposed API. The user must provide an initialization function named *pluginInit*, giving the user the ability to modify the tool's default behavior. The user can easily add, remove, or modify a pass without recompiling the system.

MicroCreator also permits a redefinition of any pass gate, the function returning a boolean deciding whether or not to execute the pass. Most internal passes are performed because their gates always return true. A user may modify it so as not to always execute the pass, or to execute it later. As opposed to general compiler passes, the passes in MicroCreator are entirely independent.

```
<instruction>
    <operation>movaps</operation>
    <memory>
        <register> <name>r1</name> </register>
        <offset>0</offset>
    </memory>

    <register>
        <phyName>%xmm</phyName>
        <min>0</min>
        <max>8</max>
    </register>

    <swap_after_unroll/>
</instruction>

<unrolling>
    <min>1</min>
    <max>8</max>
</unrolling>

<induction>
    <register>
        <name>r1</name>
    </register>
    <increment>16</increment>
    <offset>16</offset>
</induction>

<induction>
    <register>
        <name>r0</name>
    </register>
    <increment>-1</increment>

    <linked>
        <register>
            <name>r1</name>
        </register>
    </linked>

    <last_induction/>
</induction>

<branch_information>
    <label>L6</label>
    <test>jge</test>
</branch_information>
```

Figure 6. Internal part of the kernel description for a (Load—Store)+ definition

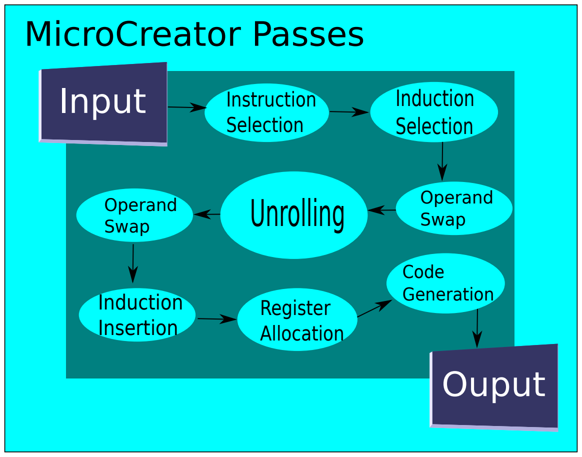

Figure 7. Passes in the source to source compilers

```
.L6:
    #Unrolling iterations
    movaps %xmm0, 0(%rsi)
    movaps 16(%rsi), %xmm1
    movaps %xmm2, 32(%rsi)
    #Induction variables
    add $48, %rsi
    sub $12, %rdi
    jge .L6
```

Figure 8. The output with an unroll of 3 for the input given by Figure 6

3.4 MicroCreator's Output

The output of MicroCreator is an assembly file executed by the MicroLauncher tool. Figure 6 is the internal kernel the tool uses to generate a simple unroll version of a load. As a possible output among other generated kernels Figure 8 contains a kernel three times unrolled, consisting in two stores and one load. Since the input contains an operand swap request, there are infinite possible variations of loads or stores in the kernels. If the swap occurs before the unrolling, there are load or store kernels, unrolled with various factors. The usage of various XMM registers removes the dependencies between loads and stores.

The final node in the Figure 6 example contains the branch information and jump instruction defining the label name and jump instruction.

3.5 Current Uses

Though not in the scope of this paper, users are modeling unrolled codes and stencil codes with the MicroCreator tool. Others also use it to detect the effect of strides on various microbenchmark program templates. Additional uses involve detecting the effect of alignment on different kernels, how many arithmetic instructions are hidden by the latencies of a memory-based kernel, etc. MicroCreator's current work focuses on automatically generating programs on new architectures and launching them with MicroLauncher.

4. MicroLauncher

MicroCreator automatically creates microbenchmark program variations from a given kernel. Evaluating the cost or performance of each program set element and distinguishing the best variation is the next step. A second tool for executing and comparing these is MicroLauncher.

MicroLauncher executes a benchmark program in a contained and controlled environment. Distinguishing important performance differences between the execution of slightly differing programs. Alignment of the various arrays modifies program performance. Therefore, the launcher tests the effect of the alignment on the kernel execution. For certain kernels, alignment issues greatly affect performance.

In order to generate stable results, the system first runs the benchmark program to load the caches with data from the kernel. Second, it executes an outer loop simulating a high number of experiments. An experiment comprises of an inner loop repeatedly executing the targeted microkernel. Jalby et al. [14] used a similar experimental setup and showed how it contributed for stability. The two loops have different intents: the outer loop allows the user to verify the stability of the experiments; the inner loop augments the evaluation time of the kernel, further stabilizing the results.

For sequential execution, the program is pinned on a given default core or chosen by the user. For parallel execution, the system handles thread core pinning.

4.1 MicroLauncher's Input

As input, the launcher accepts any assembly, source code (C or Fortran), object file, or even a dynamic library. The kernel code must contain a function representing the entry point. A command-line parameter provides the function name to the launcher, allowing users to write their own functions or wrappers and modify the launcher's execution behavior. At execution time, the launcher compiles the kernel code, if necessary, into a dynamic library loaded at run-time.

A second input type is a stand-alone program. In the case of an application, MicroLauncher forks its execution to run the program as a stand-alone application and times it. The advantage of using MicroLauncher is the multi-core aspect. MicroLauncher internally pins the processes on various cores and synchronizes before executing the application.

4.2 MicroLauncher's Options

MicroLauncher's default behavior is perfect for new users. However, more experienced users generally want to modify the default behavior, therefore, there are currently more than thirty options in the MicroLauncher tool for behavior tweaking. These options include modifying the input file, kernel's function name, number of arrays the kernel requires, size

```
<induction>
    <register>
        <phyName>%eax</phyName>
    </register>
    <increment>1</increment>
    <not_affected_unroll/>
</induction>
```

Figure 9. Counting the number of iterations

of the arrays, their alignment ranges, number of repetitions, CPU pinning, or number of cores on which to run the program. The user may switch the evaluation library to a custom library if the default *rdtsc* register [5] is not required.

4.3 MicroLauncher's Output

The output of the launcher is a generic CSV file providing the execution time of the benchmark program which is by default the number of cycles per iteration. As an option, the tool may output the full kernel function's execution.

4.4 Linking MicroCreator and MicroLauncher

There are two important issues to correctly link the generated programs from MicroCreator to MicroLauncher. First, the user must create the prototype of the kernel function. The function signature is:

```
int myFunction (int n [, void * [...]]);
```

The first parameter is the trip count requested by the MicroLauncher. Depending on the function, the other parameters are dynamically allocated arrays. The user provides the number of arrays with the option *–nbvectors*.

The second concern is to link the tools together and define the calculation of the number of cycles per iteration. The kernel program must return the executed number of iterations. On a x86 architecture, the ABI determines the return value is stored in register *%eax*. To provide the per-iteration calculation, the user adds the node in Figure 9. By defining the induction variable as unaffected by the unrolling process, at the end of the kernel execution, *%eax* contains the final number of iterations executed by the loop.

MicroLauncher retrieves the iteration count and, with the benchmark program's elapsed time, calculates the number of cycles per iteration.

4.5 Execution Overhead

The pseudo code in Figure 10 presents how MicroLauncher handles a kernel execution. The overhead calculation removes the function call cost and any other noise from the final calculation. Before executing the evaluation code, the instruction and data caches are filled with the kernel's data by calling the benchmark function once. After the first execution, MicroLauncher performs a number of repetitions of the benchmark program and evaluates the time for the runs. Finally, the tool provides the actual cycles per iteration, if

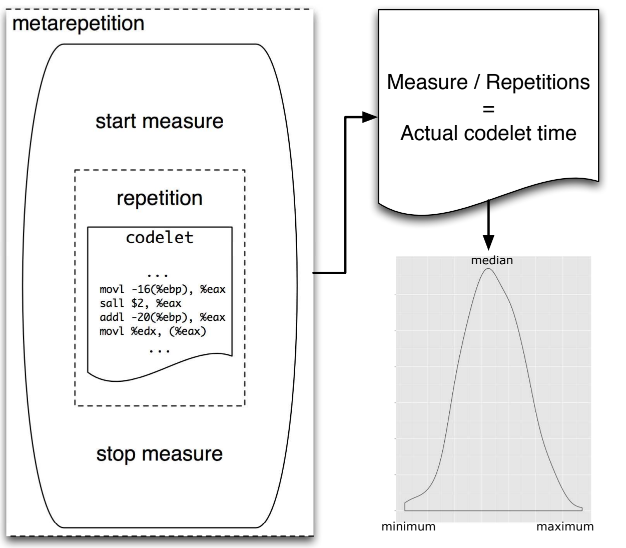

Figure 10. MicroLauncher's timing pseudo-algorithm

requested, by dividing the measured time by the number of repetitions and iterations.

4.6 Multiple Nodes

To determine the effect of the same kernel running on separate cores, MicroLauncher also allows running the same benchmark on various cores. It forks its execution into multiple launchers, pins each to a separate core; after synchronization, it records the time taken to execute the benchmark.

4.7 Experimental Runs

Stable results are MicroLauncher's priority. Executing the tool multiple times on the same architecture with the same kernel must give the same result. To achieve stability, the launcher: modifies the alignment of data arrays, disables interruptions, and pins the experiments onto particular cores. Doing so helps reduce system's global environmental issues and produces similar results between runs.

Inner core stability issues are handled by heating the instruction and data cache. First executing the kernel before evaluation and running the microbenchmark program for a number of repetitions. Finally, executing those repetitions a number of times.

All these elements contribute to obtaining stable results to comparing the number of cycles per iteration correctly.

5. Experimental Results

The experiments were executed on multiple platforms and GCC version 4.4.3, using the optimization level O3, compiled each program. Though on generated assembly programs, it is not relevant. Table 1 associates the subsequent figures of the paper and the architecture used. The MicroTools were deployed on each architecture without any additional work required to obtain results. Using the independent

Architecture	**Associated Figures**
Sandy Bridge Intel Xeon E31240 - 3.30 GHz (1 x 4GB) + (2 x 2GB)	17, 18
Dual-Socket Nehalem Intel Xeon X5650 - 2.67 GHz 8 GB	2, 3, 4, 5, 11, 12, 13, 14,
Quad-Socket Nehalem Intel Xeon X7550 128 GB	15, 16

Table 1. Association between the figures and the target architectures

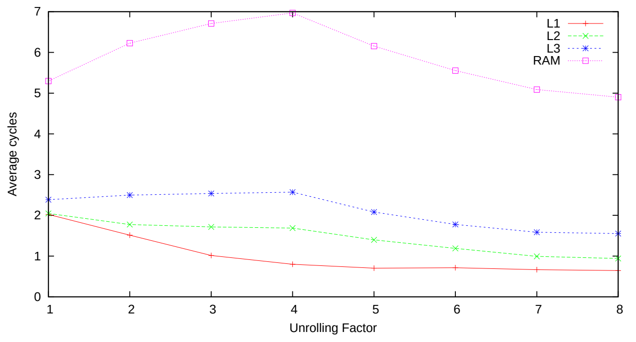

Figure 11. Average cycles per load and store using movaps by modifying the unroll factor and the level of memory hierarchy

XML language, the tools also generated the assembly and executed on the architectures also with no additional cost.

5.1 Sequential Execution

Figures 11 and 12 show the number of cycles per instruction and per iteration when varying the unrolling factor on various benchmarks. Using a single input file, MicroCreator generated 510 benchmark program variations with different unroll factors and whether the instruction was a load or a store. The figures evaluate the difference between the use of *movss* or *movaps* instructions. Since the scalar instruction *movss* moves four bytes of memory, whereas the vectorized *movaps* moves sixteen bytes, the latency difference is clearly visible.

Four groups of these 510 benchmark programs created the figures to determine the different impacts from the instructions *movss*, *movsd*, *movaps*, and *movapd*. Presented are the *movss* and *movaps* figures. The *movapd* figures are the same as their *movaps* counterparts. The *movsd* versions are similar to the *movss* ones with slightly higher latencies because of the higher data movement rate. For each unroll

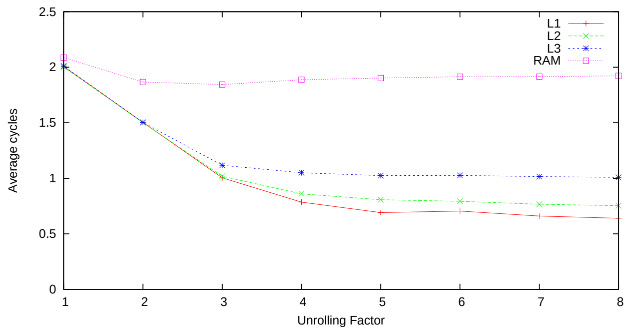

Figure 12. Average cycles per load and store using movss by modifying the unroll factor and the level of memory hierarchy

group, the minimum value was taken though the variance was minimal. The X-axis represents the number of load or store instructions in the loop; the Y-axis is the performance measured in cycles per iteration. The plot lines are different considered memory hierarchy.

The figures use the terms L1, L2, L3, and RAM for the memory hierarchy containing the array, achieved by using twice the underlying memory hierarchy size. The mention *L1* actually represents where the array is half the size of the architectures' first cache level. The *L2* plot line uses an array twice the size of the hardware's first cache level, etc.

Accessing data from RAM with vectorized instructions has a greater latency impact because vectorized instructions access four times more data than regular *movss* instructions. Not surprisingly, on the architecture the two vectorized instructions *movaps* and *movapd* have the same impact on performance. Illustrating, for the general case, unrolling is advantageous and useful on the architecture. However, the 8-unrolled case, the *movss* cycle number per iteration is one cycle per load in L3. Four *movss* instructions are the same workload as the *movaps* version. Therefore, the vectorized version is better since it executes at less than two cycles per load per iteration.

Figure 13 exposes the performance variation when modifying frequencies for a kernel with eight loads unrolled, using the *rdtsc* counter [5] which is independent on the frequency. The timing varies with the frequency for L1 and L2 accesses; however, L3 and RAM remain constant, proving on-core frequency modifications do not affect the off-core frequency.

5.2 Parallel Execution

In the parallel domain, studying benchmark programs is more complex. Two different techniques were used: forking and OpenMP. MicroLauncher entirely integrates the forking technique. Pinning each process to a different core

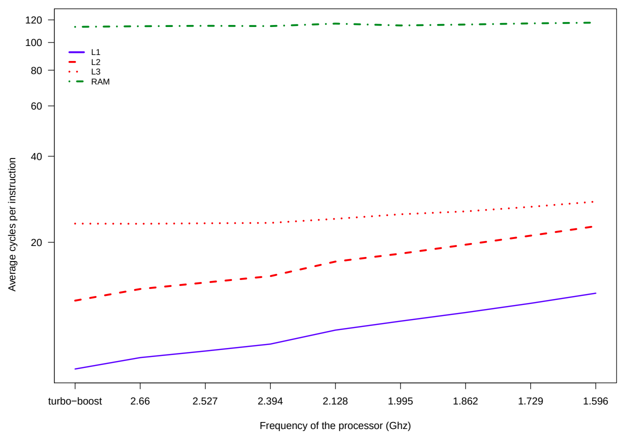

Figure 13. Average cycles per load (using movaps) by modifying the frequency of the processor

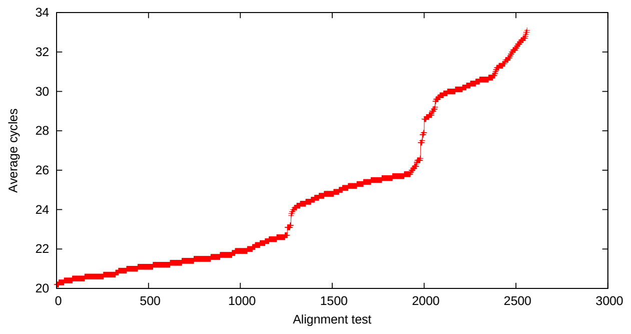

Figure 15. Average cycles per iteration on a 8-array traversal with a 8-core execution

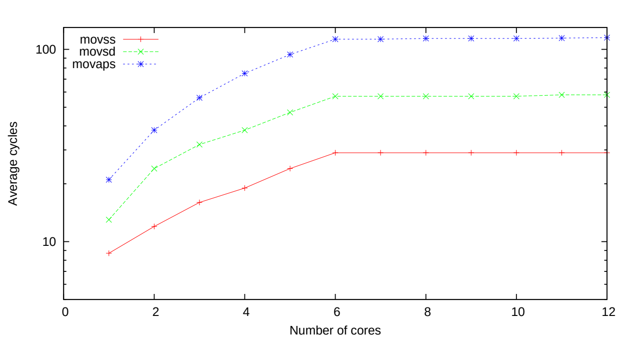

Figure 14. Average cycles per iteration depending on the instruction used and the number of cores

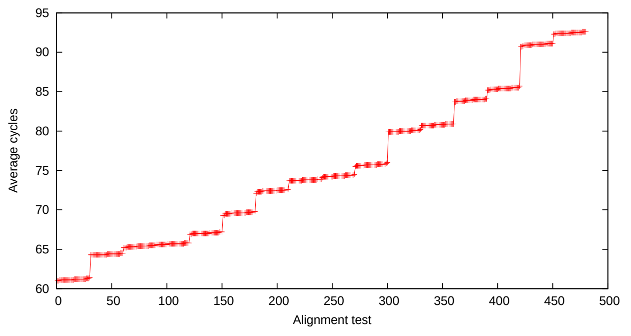

Figure 16. Average cycles per iteration on a four array traversal with a 32-core execution

and launching the same sequential program. Thereby simulating a typical HPC profile: one process on each core, each performing the same type of workload. For the second technique, MicroLauncher lets the OpenMP runtime pin the threads on each separate core. The remainder of the section presents MicroLauncher's usage in the parallel paradigm.

5.2.1 Handling Process Forking

Forking the same process exposes the memory access saturation of an architecture. Consider Figure 14, it shows the latency evolution in a logarithmic scale of an 8 load array access from an array residing in RAM. The breaking point for the dual-socket Nehalem machine is six cores. Under six cores, the latency is not greatly affected; over six cores, there is no longer a single change in the latencies. In order to obtain peak performance from the architecture, over six cores, it is best to have some cores handling memory movement and others processing data.

5.2.2 Considering Alignment

When considering alignments, MicroLauncher tests a variety of alignment settings for each allocated array. One study tested the variation of performance depending on the alignment of each array. Figure 15 illustrates these experiments on a 32-core machine but only using eight cores.

The benchmark program is a single strided traversal of a number of arrays. In the case of Figure 15, there are four arrays accessed with a stride one and *movss* instructions. The Y-axis shows the number of cycles per iteration; the X-axis shows various alignment configurations tested, upwards of 2500. The figure shows, for *movss* accesses, there is a variation of 20 to 33 cycles. The number of cycles per iteration is significantly dependant of arrays.

Memory saturation is exposed in Figure 16 where the plot line represents a 32-core execution of a benchmark program. The program contains a four array traversal with the *movss*

Unroll factor	OpenMP time (in s)	Seq. time (in s)
1	9.42	18.30
2	9.36	16.97
3	9.34	15.19
4	9.32	14.57
5	9.31	14.53
6	9.31	14.39
7	9.31	14.60
8	9.31	14.60

Table 2. Execution time of the OpenMP and the sequential versions of a **movss** unrolled version

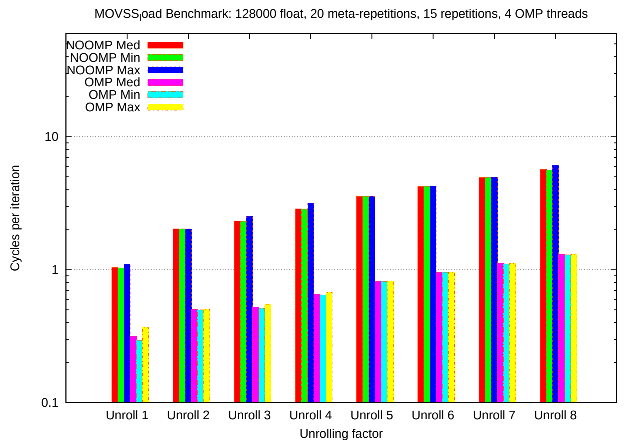

Figure 17. Cycles per iteration for unrolled versions of memory loads using movss for sequential and OpenMP for an array of 128k elements

instructions, the figure shows performance variations from 60 to 90 cycles per iteration with such a configuration.

5.2.3 Considering OpenMP

On the OpenMP side, Figures 17 and 18 show the number of cycles per iteration of a program using *movss* instructions. The difference between the two figures is the size of the array traversed. Comparing the minimum and maximum values obtained across ten runs shows the stability of the results. Moreover, the cycle number comparison per iteration on an OpenMP version and a sequential one is shown. As opposed to the previous figures, the OpenMP ones have a logarithmic scale. On the four core architecture, Table 2 shows the execution time of the benchmark program. Unrolling achieves a significant performance gain for the sequential version. It is not true in the OpenMP setting due to the overhead of the parallel setup. Finally, the OpenMP 128k version has a significantly better performance gain compared to the six million version.

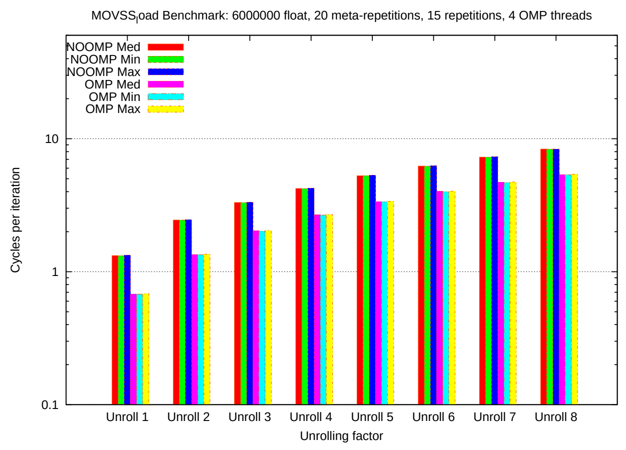

Figure 18. Cycles per iteration for unrolled versions of memory loads using movss for sequential and OpenMP for an array of six million elements

6. Related Works

Profiling systems help understand the behavior of targeted applications. However, due to their complexity, it is often difficult to determine why programmers do not achieve peak-performance. Profilers such as gprof [12], Vtune [19], Scalasca [10], or Maqao [7] enable users to comprehend bottlenecks and unknowns with the application, often without insight as to why the program is not performing as expected.

There has been work in program generation whether by compilers [4, 8, 16], multi-level program generators [11], or source to source compilers [17, 18] benchmark automatic generation. However, these compilers or generators create programs for performance or characterization purposes. They do not try to create thousands of variations to determine the effect of these changes on the underlying architecture. Code specialization is also dependent on the input [15] but is too specific to a particular domain and application. Comparatively, MicroCreator defines the type of benchmark program exactly disregarding an existing application. Though some x86 architecture knowledge may be required, the generation and its automatic fashion easily allow any user to understand architecture characteristics.

Executing programs in a closed environment can be handled by a simulator [13, 20]. Simulators are never perfectly matched to real architectures and, as they evolve quickly, simulators often get further away from actual architectures.

Code wrappers are also often suggested to test benchmark programs but their simplicity often makes them unusable in the general case. MicroLauncher allows generic programs as well as assembly programs. Its options allow multi-core execution with automatic synchronization and OpenMP execution. It handles the array allocation with automatic alignment check and comparison.

Though the idea of benchmarking is not new [1, 14], work generally focused on known benchmark suites [2, 3, 6]. Using those benchmark suites, it is possible to infer the underlying architecture's components and the code's impact. The empirical approach provides a view of performance variations and obtainable peak performance. For benchmarking reasons, executing the program in a controlled state is critical for distinguishing small variations of performance. Specific benchmark suites exist such as P-RAY [1]. P-Ray studies multi-core architectures and works on studying each element of the multi-core realm. The MicroTools provide a means to study code variations in the application realm. By creating variations of the generated code and runtime decisions, the MicroTools' data provides a means to study around a given point. The study allows the user to detect whether the variations have an impact performance or not and, if so, determine which variation is optimal.

7. Conclusion and Future Work

MicroCreator and *MicroLauncher* present a novel way to automate the understanding of the underlying architecture. MicroCreator creates variations of a described program in order to evaluate variations in performance or power utilization. It uses a simple XML format and generates assembly code accepted by MicroLauncher. MicroLauncher allows the execution of a microkernel program in a stable environment. It is essential to evaluating differences between similar but not equivalent microprograms.

Microtools give an input on the performance and power utilization of a given architecture with almost no knowledge needed. The gathered information helps better understand the architecture or write applications. Performance issues in the applications are characterizable with these types of studies. The tools are entirely independent of the underlying architecture and can directly use the same creator input files to perform the same benchmark tests.

Future work consists in allowing greater possibilities for MicroCreator such as fully supporting every OpenMP/MPI constructs. The plugin system can be further deployed to augment the tweaking of MicroCreator, directly rewriting greater parts of the MicroCreator without having to modify a single line of the tool's code. These two elements greatly add to the possibilities and potential of the tool. For MicroLauncher, future work consists in fully supporting the execution of parallel programs.

Finally, automating both sides of the analysis workflow is an on-going effort. On one side, applications drive MicroCreator's generated code to test variations around the application's hotspots. Such a technique allows the MicroTools to find the optimal code and runtime environment variation around an existing real-world application. On the other side, data-mining techniques allow to process the MicroTools data generated in order to automate the analysis. Both together form a cohesive solution to application characterization around the two focal tools: the MicroTools.

References

[1] *P-RAY: A Suite of Micro-benchmarks for Multi-core Architectures*. In the Proc. of the International Workshop on Languages and Compilers for Parallel Computing, July 2008.

[2] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. The nas parallel benchmarks 2.0. 1995.

[3] Pointer Intensive Benchmark.

[4] Intel Corporation. Intel compiler.

[5] Intel Corporation. Using the rdtsc instruction for performance monitoring.

[6] Standard Performance Evaluation Corporation.

[7] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J-T. Acquaviva, and W. Jalby. Exploring Application Performance: a New Tool For a Static/Dynamic Approach. In *lacsi*, Santa Fe, NM, October 2005.

[8] GNU Gcc. Gnu gcc.

[9] GNU Gcc. Gnu gcc plugin system.

[10] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, April 2010.

[11] Robert Glück and Jesper Jørgensen. An automatic program generator for multi-level specialization. *Lisp Symb. Comput.*, 10:113–158, July 1997.

[12] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. *SIGPLAN Not.*, 39:49–57, April 2004.

[13] Jeffrey Heid. x86sim: A simulation tool for the intel x86 architecture, 1999.

[14] W. Jalby, C. Lemuet, and X. Le Pasteur. A New Set of Microbenchmarks to Explore Memory System Performance for Scientific Computing, 2004. International Journal of High Performance Computing Applications.

[15] Minhaj Ahmad Khan, H. P. Charles, and D. Barthou. Languages and compilers for parallel computing. chapter An Effective Automated Approach to Specialization of Code, pages 308–322. Springer-Verlag, Berlin, Heidelberg, 2008.

[16] LLVM. The llvm compiler infrastructure.

[17] Eric Petit, François Bodin, Guillaume Papaure, and Florence Dru. Poster reception - astex: a hot path based thread extractor for distributed memory system on a chip. In *SC*, page 141. ACM Press, 2006.

[18] Daniel J. Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10:215–226, 2000.

[19] J. Reinders. *VTune performance analyzer essentials: measurement and tuning techniques for software developers*. Engineer to Engineer Series. Intel Press, 2005.

[20] SimpleScalar. Simplescalar.