The MHAOTEU Toolset

Jaume Abella Sid Ahmed Ali Touati Alan Anderson* Carlos Ciuraneta
Josep M. Codina Min Dai Christine Eisenbeis† Grigori Fursin Antonio González‡
Josep Llosa Michael O'Boyle§ Andry Randrianatoavina Jesús Sánchez
Olivier Temam¶ Xavier Vera Gregory Watts

Abstract

Memory latency is one of the main reasons for performance degradation of current computers. This problem is exacerbated by the fact that the relative memory latency increases by about 50% per year. On the other hand, there is a lack of tools to help programmers to improve their applications from a memory perspective. This paper presents the MHAOTEU toolset, a set of tools that are build to help the programmers to tune their applications for a better use of the memory hierarchy.

Key words: Cache Memories, Program Optimizations, Performance Analysis.

1 Introduction

Memory performance is becoming an important bottleneck in current microprocessors. Significant research effort has been devoted to propose novel techniques to improve its performance. Some of these techniques require only hardware support while others require some support of the programmer/compiler. Examples of the latter are prefetching, blocking, copying, etc.

This type of optimizations require some knowledge of the behaviour of the program and the architecture. In addition a programmer does not know what performance improvement he/she might expect from optimizing a code. The situacion gets worse due to the lack of tools that allow the programmer to use existing memory optimizations.

The MHAOTEU Project (Memory Hierachy Analysis and Optimization Tools for the End-User) aims at developing a set of tools that will help program developers to tune their applications for a better use of the memory hierarchy. The purpose of this project is to build a set of analysis tools and a set of optimization tools which address the above issues and are practical enough to be exploited in an industrial environment. The focus of the tools are Fortran numerical applications for single processor platforms.

Different analysis techniques have been developed. These techniques provide different ways of analyzing memory hierarchy behaviour. They range from very accurate but slow analysis to extremely fast but not so accurate analysis. To give support to the analysis tools we have also developed a set of instrumentation tools. The toolset also implements many code transformations and optimizations. These transformations are performed automatically under the control of the user.

*alan@epc.co.uk
†Christine.Eisenbeis@inria.fr
‡antonio@ac.upc.es
§mob@dcs.ed.ac.uk
¶Olivier.Temam@lri.fr

16th IMACS World Congress (© 2000 IMACS)

The rest of the paper is organized as follows. The next section presents our approach to integrate the different components in a single tool. The following sections introduce the different components that compose the tool. Section 3 presents the instrumentation modules. Section 4 overviews the different analysis approaches. The transformation module is presented in Section 5 and a software prefetching module is described in Section 6. Finally, Section 7 shows the main conclusions of our work.

2 Software architecture

The Mhaoteu tool began as a diverse set of components on many platforms and required a coordinated drive towards integration. The objective was to allow the tools to interoperate on the users problem.

The original intention was to produce a single all-encompassing tool based on C++ and motif graphics. That was completely unrealistic for tools of such diverse origin and a research budget. Instead we decided to use modern principles of client server engineering. The tools would be grouped on servers, wrapped with a standard interface and communicate across a local network. The command and graphic interfaces would be written in Java for speed of implementation and provide HTML graphics via HTTP to a potentially remote browser as the user interface. Now we were looking at a system of components that could be on one machine or spread around a network. The user could be local or remote.

Related components were grouped in "servers". These were responsible for implementing a standard command language, communicating via TCP/IP sockets across the network and operating their components.

Central to the overall design is the database in which are stored the potentially large performance statistics. The data in the database is extracted or filtered by a JDBC SQL query and displayed in HTML.

A Java program known as Coordinator drove the tools and the database as well as projecting a graphical user interface in the form of a Java applet to a users browser. Coordinator, database, tools and user interface were all separable across a network. The only compromise made was to assume for the moment that bulk data exchanges between the tools would take place via a shared filestore system.

A significant part of the integrative task was to define and enforce common definitions for data interchange between the tools, whether via shared filestore or database tables.

The Coordinator presents a Command window to the user. Tools are selected and instructed through command dialogs and the program is presented in a tree like form to aid navigation. There is a project system to manage the versions of source files as they are experimentally transformed into a more highly optimised state. The source of the program is presented in HTML through a browser window with links to related performance information. The intention is to focus the user on the task and as far as possible disguise the diverse origins and nature of the tool components.

3 Instrumentation component

The purpose of the Instrumentation component is to extract run-time information from the applications to optimize. This information can be used either by the user in order to study the behaviour of an application, or by other tool components in order to improve their accuracy or to obtain traces of the execution.

This component has as input the application source code to study, and as output the source code instrumented with calls to particular subroutines that collect the run-time information. The instrumented code has to be linked with a MHAOTEU library that contains the subroutines and, executed to obtain the desired information.

Under the MHAOTEU toolset we have developed three different types of instrumentation. They use the ICTINEO experimental compiler [1].

High Level Instrumentation is the tool used by the Dynamic Analysis and the GRW [3]. It instruments the source code in order to obtain memory traces of the execution and information about the non-scalar memory references. For instance, for each memory instruction one may obtain the base address and the offset.

Profiler is the instrumentation used by the SPLAT tool [6] in order to improve the accuracy of the static memory analysis. This component instruments the source code in order to obtain the number of iterations per loop, and the number of executions per memory instruction.

Program 102.swim, Procedure name=calc2

Cache level	Num. of accesses	Total misses	Load misses	Write misses	Num. of conflicts	Num. of capacity	Num. of compulsory
0	36,797,600	20,063,043 - **54.52%**	9,540,733 - **47.55%**	10,522,310 - **52.44%**	4,256,620 - **21.21%**	15,806,423 - **78.78%**	0 - **00.0%**
1	20,067,513	1,173,725 - **05.84%**	504,865 - **43.01%**	668,860 - **56.98%**	0 - **00.0%**	1,173,725 - **100.0%**	0 - **00.0%**
TLB	36,797,600	14,200 - **00.03%**	6,461 - **045.5%**	7,739 - **054.5%**	0 - **00.0%**	0 - **00.0%**	0 - **00.0%**

Loop misses (and miss ratio)

Loop	0	1	TLB
0 (src)	1 - **04.98%**	1 - **08.51%**	0 - **00.0%**
1 (src)	20,003,412 - **99.70%**	1,149,354 - **97.92%**	9,010 - **63.45%**
2 (src)	36,500 - **00.18%**	21,790 - **01.85%**	5,150 - **36.26%**
3 (src)	23,050 - **00.11%**	2,550 - **00.21%**	30 - **00.21%**

Loop hierarchy and execution time

Depth	Loop Nr.	Begin	End	Exec. time
0	0 (src)	515	578	0
1	1 (src)	526	575	0
0	2 (src)	585	602	0
0	3 (src)	610	627	0

Figure 1: Profiler

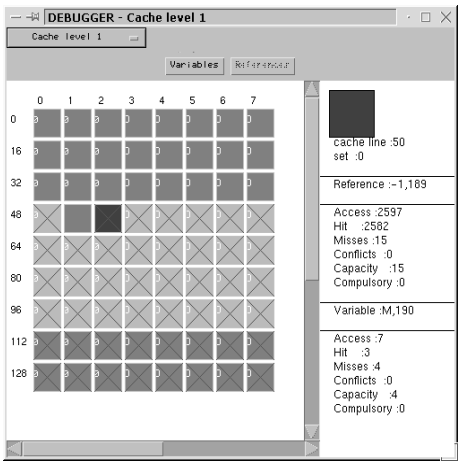

Figure 2: Debugger

Loop Timing is the instrumentation tool to obtain the execution time spent by the application in each loop. This information is not required by other components but, is very useful for the end user in order to focus the optimizations in the relevant parts of the application.

4 Analysis tools

4.1 Memory System Profiling

The purpose of this component is to provide a memory-wise program execution profile. The tool is focused on collecting cache performance information. This memory system profiler provides hit/miss ratio and associated stall cycles for several components of the memory system: all cache levels and TLB for a given memory system architecture. This tool can be connected to Ictineo (in the MHAOTEU platform) which provides source-code traces or an object-code tracer from the public domain like ATOM for Alpha processors. These profiling facilities are used to evaluate the nature of misses: spatial, temporal, capacity, conflict, compulsory... to help finding the nature of performance degradations, etc. (see Figure 1 for an example showing the statistics displayed). Unlike current analysis tools, the scope of profiling is not only the whole code but also procedure or loop nest constructs since optimizations are usually performed at that level.

4.2 Memory Performance Debugging

While profiling can be used to summarize the behavior of different code constructs (routines, loop nests, statements, references...) and locate memory bottlenecks, it provides little assistance to explain the occurrence of these bottlenecks and thus to find solutions. This problem can be compared to program optimization: profiling is used to find time-consuming code sections, but if the cause of performance degradations does not clearly appear in the source code, debugging must then be used to execute the program step and by step and to understand the workings of the bottleneck. The *cache performance debugger* serves the same purpose for memory performance evaluation. It is a graphical tool based on Java that shows, reference by reference, the inner workings of a cache for small code sections, as well as numerous statistics to guide the analysis (see Figure 2 for a view of the tool). This component can then be used to understand the workings of memory bottlenecks.

4.3 SPLAT

The SPLAT tool consists of a static locality analysis enhanced with very simple profiling data, which results in a negligible slowdown. This feature allows the tool to be used for highly time-consuming applications and to include it as a step in a typical iterative analysis-optimization process.

The static information is aimed at computing the different types of misses that will happen during the execution. Compulsory misses require to compute the intrinsic reuse of data. Capacity misses require in addition to compute

the volume of data referenced by each loop iteration. Finally, conflict misses are identified by computing interferences among data references.

The profiling consists of just the number of executions of each basic block, which is a facility provided by many current compilers. From this information, the number of executions of each memory instruction and the average number of iterations of each loop can be derived. This profiling information is highly valuable in order to improve the accuracy of the tool.

This static and dynamic information is used as an input to the locality analyzer. The locality analysis is divided into three phases: (i) reuse phase, (ii) volume phase, and (iii) interference phase. The first phase identifies all the reuse exhibited by the program. This information is the basis for computing misses. In particular, compulsory misses do not require any additional analysis: they consist of all references without any reuse. The volume phase is targeted to identify capacity misses. Finally, the interference phase computes the conflict misses.

SPLAT has been compared with techniques based on simulation, showing that it is highly accurate for numeric codes. In addition the SPLAT tool provides a detailed evaluation of the reuse exhibited by a program, quantifying and qualifying the different types of misses either globally or detailed by program sections, data structures, memory instructions, etc.

4.4 Cache Miss Equations

Cache Miss Equations [5] are a very accurate analytical model of the cache memory. They describe the cache behavior by means of diophantine equations, which allows us to use mathematical techniques to compute the locality of each memory reference. For instance, by solving CME one could compute the different types of cache misses that each reference will cause. Even though the computation cost of generating CME is a linear function of the number of references, to solve them is a NP-Hard problem and thus trying to study a whole program may be unfeasible.

CME allow us to study each reference in a particular iteration point independently of all other memory references. Deciding whether a reference causes a miss or a hit for a given iteration point is equivalent to deciding whether it belongs to the polyhedra defined by the CME. The total number of cache misses can be computed by analyzing all iteration points [7, 2].

Our implementation is based on estimating the result of the CME by means of sampling techniques. This technique is very fast and accurate, and the error can be bounded with a given confidence. For example, for a loop nest with one million of iteration points, studying only 1000 iteration points allows us to know the number of misses with an error less than 5% with a 95% confidence.

We have also developed efficient techniques to count the number of integer points inside the polyhedra defined by the CME. By exploiting some intrinsic properties of the particular types of polyhedra generated by CME, we reduce the complexity of the algorithm, which results in very high speed-ups. We show that the proposed technique can compute the miss ratio of most SPECfp95 benchmarks just in a few seconds on a typical workstation. This opens the possibility to include this analysis framework in production compilers in order to support many optimizations.

4.5 Generalize Reference Windows

Reference windows have been primarily introduced by Gannon, Jalby and Gallivan in [4] for designing a criterion intended to evaluate locality properties of a code and guiding optimizing transformations. In today's language reference window evaluates the number of capacity misses. It is therefore more adapted to the management of software-controlled local memories.

We have extended reference windows in order to take into account also conflicts misses, i.e. these misses caused by numerous reuse of the same cache line set. The main idea is to consider each set of the cache as some local memory and evaluate the reference window associated to this set. Typically in a direct-mapped cache, if the size of this generalized reference window (GRW) is more than 1 at some program point for some cache set then some miss will occur eventually after that point. For instance consider the following sequence of memory accesses.

```
DO I=0,99
  X (I+1)
  X (I)
  X (I - 1)
ENDDO
```

We assume a cache line size of 8 words, and 32 sets in the cache. Figure 3 shows the behaviour of GRW size over time and over sets. It can be observed that the window size is never larger than 2. This means that if associativity is more

than 2 no cache misses, apart compulsory misses, will occur.

Current status of work is the generation of equations that define the general reference window from some loop and fixed program point. We are exploring both static and dynamic approaches for computing GRW. We expect that a static approach gives parameterized formula for GRW and hence guides optimization. However GRW computation gives rise to a number of theoretical, possibly open, problems. A graphical dynamic approach gives a very useful feeling to the programmer about data locality behaviour of the code. It can act as a performance debugging tool, which is under development.

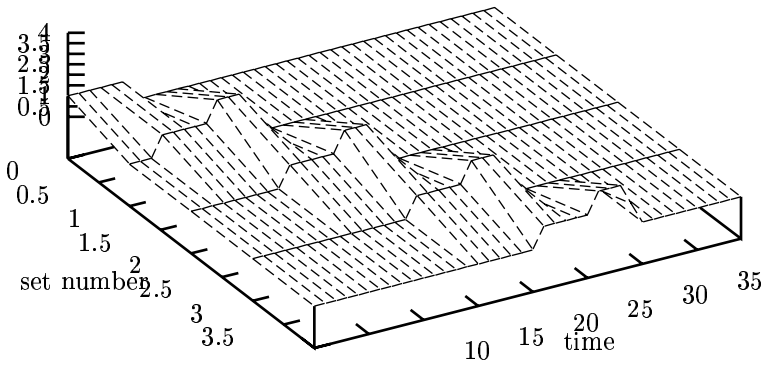

Figure 3: Evolution of the reference window over time and sets.

5 Transformation tools

In order that users may improve their code once they have tracked down a memory hierarchy problem, they must have a means of sensibly changing their code. A transformation toolbox based on a simple GUI is currently supported which allows users to select a wide range of loop and data modifying transformations. These include loop distribution, array padding, loop tiling, global index reodering, loop splitting etc. A transformation may be selected and if legal, applied to the code. If a particular transformation has a negative effect, there is an undo operation to return the code to its original state.

5.1 Transformation Infrastructure

At the heart of the transformation infra-structure is the MARS internal representation which is an extended linear algebraic framework. This is a high level abstraction of the key components of the program namely iteration spaces, arrays and access functions. Iteration spaces and the index domain of the arrays are represented as integer lattice points within convex polyhedra, stored as an ordered set of linear constraints. Access functions are stored as linear transformations on the iteration vector. This representation allows high-level rapid analysis of program structure and the simple application of high level transformations (and checking of validity) via the transformation toolbox. For instance unimodular and non-singular loop transformations, are implemented as a simple matrix multiplication with the appropriate loop constraint matrix and enclosed access matrices. Data transformations such as global index reordering, array padding etc. can also be similarly achieved, this time by multiplication of the appropriate index domain and all corresponding accesses. The key benefit of this formulation for user directed optimisation, is that we may apply long sequences of transformations without making the transformed program unanalysable. Uniform program representations will become increasingly important when we wish to deal with more optimisations based on more complex transformations.

5.2 Integrating analysis and Transformation

At present the tool allows the user, in effect, to navigate an optimisation space. We are currently investigating how static and dynamic analysis may be examined in order to suggest good optimisations to the user. Furthermore, we wish to have the facility that certain standard optimisation heurestics may be automatically tried, possibly overnight, removing the burden form the programmer.

6 Software Data Prefetching

Software data prefetching is a technique suported by new processors but not enough exploited by the programmers. The concept of software prefetching is basically to add new memory instructions in the original code. With these new instructions we can split apart the request of the data and the use of the data, while finding enough parallelism to keep the processor busy in between. To hide the latency within a single thread, the request of data has to be performed far in advance of the use of the data in the execution stream. This requires the ability to predict what data is needed ahead of time. Software prefetch requires explicit prefetch instructions to move data into the cache. The challenges of software-controlled prefetching include the fact that some criteria is needed to insert the prefetches into the code, and also that the new prefetch instructions involve some amount of execution overhead.

To be sure that prefetches are not unnecessary we need to know which dynamic references misses in the cache. To determine the references to prefetch we use the locality analysis performed by the SPLAT tool since it is the fastest analysis in the toolset.

Once the locality analysis is performed, the Ictineo compiler provides us information about the low-level code in order to insert the prefetch instructions far enough in advance. We also consider the information about the stride of the loop and the factor that multiplies the loop index variable to be sure that a cache line is prefetched only once.

7 Conclusions

In this paper we describe the MHAOTEU Toolset, a set of tools to help the end users to tune their applications in order to improve performance from the memory hierarchy's point of view. The tool provides analysis modules to indicate the user where memory performance degradations occur in applications and explain the possible causes. They allow a programmer to use existing memory optimizations without requiring the user to perform them by hand. Finally we expect that the tool will suggest possible optimizations to the user in the next release.

References

[1] Eduard Ayguadé, Cristina Barrado, et al. *A uniform internal representation for high-level and instruction-level transformations.* UPC, 1995.

[2] Nerina Bermudo, Xavier Vera, Antonio Gonzalez, and Josep Llosa. Optimizing cache miss equations polyhedra. In *4th Workshop on Interaction between Compilers and Computer Architectures*, 2000.

[3] Christine Eisenbeis, William Jalby, Daniel Windheiser, and François Bodin. A strategy for array management in local memory. *Mathematical Programming*, 63:331–370, 1994. Special Issue on Applications of Discrete Optimization in Computer Science.

[4] Kyle Gallivan, William Jalby, and Dennis Gannon. On the problem of optimizing data transfers for complex memory systems. In *1988 ACM International Conference on Supercomputing*, pages 238–253, St. Malo, France, June 1988.

[5] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: an analytical representation of cache misses. In *ICS97*, 1997.

[6] F.J Sánchez and A. González. Fast, flexible and accurate data locality analysis. In *Procs. of PACT'98*, October 1998.

[7] Xavier Vera, Josep Llosa, Antonio Gonzalez, and Carlos Ciuraneta. A fast implementation of cache miss equations. In *Compilers for Parallel Computers*, 2000.