Assembly Code Analysis Using Stochastic Process Algebra

Lamia Djoudi and Leïla Kloul

PRiSM, Université de Versailles, 45, Av. des Etats-Unis, 78000 Versailles
{ladj,kle}@prism.uvsq.fr

Abstract. Currently compilers contain a large number of optimisations which are based on a set of heuristics that are not guaranteed to be effective to improve the performance metrics. In this paper, we propose a strategy which allows us the analysis and the choice of the best optimisation, by focusing on the hot part of an assembly code. In our approach, for each optimisation applied, the code of the hot loop is extracted and its dependency graph generated. Finally, and in order to select the best optimisation, the generated graphs are analytically analysed using stochastic process algebra.

Keywords: Assembly code, Code optimisation, Data dependencies graph, Stochastic process algebra, Performance analysis.

1 Introduction

Due to complex interactions between hardware mechanisms and software dynamic behaviours, scientific code optimisation at compile time is an extremely difficult task. State-of-the-art compilers are still challenged to achieve high performance independently from runtime or micro-architectural parameters. As regard the application, the knowledge of its behaviour implies a serious effort and a high degree of expertise.

Currently compilers contain a large number of optimisations which are based on a set of heuristics that are not guaranteed to be effective to improve the performance metrics. Achieving high performance relies heavily on the ability of the compiler to exploit the underlying architecture and the quality of the code it produces. This requires a full understanding and a full control of the application behaviour the compiler has when compiling the application to provide a precise diagnostic about the success or failure of an optimisation if applied. It also requires the ability to transmit to the compiler certain architecture characteristics to make it choose the right optimisation to apply.

Before starting to optimise an application, the programmer must identify the main factors limiting the performances of its application. For that, two types of code analysis techniques can be used: static and dynamic techniques.

Static techniques extract opportune information from the program analysis (source, assembly or binary code). These techniques are usually faster than dynamic analysis but less precise.

© Springer-Verlag Berlin Heidelberg 2008

Dynamic techniques require a dynamic execution of the program (real execution or simulation). Profiling a program consists in collecting opportune information during its execution in order to guide efficient optimisations (dead code elimination, prefetching, instruction scheduling, memory layout transformation, ...). These optimisations can be applied either by transforming the initial source code or the assembly code, or by re-compiling it guided by the collected information. Several dynamic techniques can be used, among which we have instrumentation and sampling.

– Instrumentation inserts instructions to collect information. Several instrumentation methods exist: source modification, compiler injected instrumentation, binary rewriting to get an instrumented version of an executable, and binary translation at runtime. Instrumentation adds code to increment counters at entry/exit function, reading hardware performance counters, or even simulate hardware to get synthetic event counts. The instrumentation runtime can dramatically increase the execution time such that time measurements become useless. It may also result in a huge code. Moreover it does not help in finding the bottlenecks if there are any. MAQAO[4] and EEL[10] are examples of tools which are based on the instrumentation.
– Sampling consists in taking measuring points during short time intervals. The validity of the results depends on the choice of the measures and their duration. Prof [11] and GProf [7] are examples of sampling-based tools.

Obtaining dynamic information on the behaviour of a program is relatively complex. In one hand the application size is increased and on the other hand the number, the complexity and the interactions of the transformations (optimisations) to apply are important. Moreover the validity of these information depends on the input parameters of the program.

However certain information like the execution time or the iteration number can be obtained only dynamically and they have an impact on the effectiveness of the optimisations. As a large application may execute for hours and sometimes even days, if a developer is focusing on the implementation or the tuning of a key computation, it will be far more efficient and less cumbersome to run just the key computation, isolated from the rest of the program.

In this paper, we propose a strategy which allows the analysis and the choice of the best transformation, by focusing on the hot part of a code. It is well known that loop optimisation is a critical part in the compiler optimisation chain. A routinely stated rule is that 90% of the execution time is spent in 10% of the code. The main characteristic of our strategy is the assembly code isolation. We isolate the hot loop from a large application for the purpose of performance tuning. We are interested in the assembly code because it is the natural place to observe performance. It is close enough to the hardware and it is possible to check the job done by the compiler.

Our approach is based on: (1) a static analysis of the code to predict the data dependency, (2) a dynamic analysis to select the hot loop in code, (3) a code isolation to extract the hot loop from a large application in order to apply different transformations with different input data and finally (4) an analytical modelling to select the best transformation.

While we use the MAQAO tool for code analysis, we have chosen to use PEPA [8] as the performance modelling technique. Our objective is the investigation of the impact of each of the transformations on the execution time of the code. The results obtained are compared to those obtained when executing the code and it is shown that our approach based on PEPA achieves comparable results, but at a much lower cost.

Structure of the paper: in Section 2, we give a brief overview of the tool MAQAO and the formalism PEPA before describing our approach. In Section 3, we present a case study and show how to apply our approach. Section 4 is dedicated to the numerical results. We finally conclude, in Section 5, with a discussion about the possible extensions of our work.

2 The Approach

The approach we propose allows bridging the gap between the code analysis provided by MAQAO and the PEPA-based analytical modelling for a formal analysis of the code performances. Before developing this approach, we present the main characteristics of MAQAO and give a brief overview of PEPA.

2.1 MAQAO

MAQAO [4] stands for Modular Assembly Quality Analyzer and Optimizer. The concept behind this tool is to centralise all low level performance information and build correlations. As a result, MAQAO produces more and better results than the sum of the existing individual methods. Additionally, being based after the compilation phase allows a precise diagnostic of compiler optimisation successes and/or failures. MAQAO provides several options among which we have:

- MAQAOPROFILE[5] is an option which allows us to give a precise weight to all executed loops, therefore underscoring hotspots. Correlating this information provides the relevant metrics:

  (i) the hotpath at run-time which passes through the whole program and where the application spends the most of its time.
  (ii) the monitoring trip count is very rewarding. By default most of compiler optimisations target asymptotic performance. Knowing that a loop is subjected to a limited number of iterations allows us to choose the optimisations characterised by a cold-start cost.
- Static Analyser: MAQAO's static module extracts the entire code structure and expresses it using a set of graphs: Call Graphs (CGs), Control Flow Graphs (CFGs) and Data Dependencies Graphs (DDGs). Computing the DDGs is a key issue to determine critical path latency in a basic block and perform instructions re-scheduling or any code manipulation technique. It also allows an accurate understanding of dynamic performance hazards

and determines the shortest dependency that corresponds to the overlapping bottleneck.

– MAQAOADVISOR proposes different transformations to apply on the hot loops, at the code source level, to improve the performances of the code. The program is then compiled again and the new assembly code is resubmitted to MAQAO for a new analysis of the code.

2.2 PEPA

In PEPA a system is described as an interaction of *components* which engage, either singly or multiply, in *activities*. These basic elements of PEPA, components and activities, correspond to *states* and *transitions* in the underlying Markov process. Each activity has an *action type*. Activities which are private to the component in which they occur are represented by the distinguished action type, τ. The duration of each activity is represented by the parameter of the associated exponential distribution: the *activity rate*. This parameter may be any positive real number, or the distinguished symbol $\top$ (read as *unspecified*). Thus each activity, a, is a pair (α, r) consisting of the action type and the activity rate respectively. We assume a countable set of components, denoted $\mathcal{C}$, and a countable set, $\mathcal{A}$, of all possible action types. We denote by $\mathcal{A}ct \subseteq \mathcal{A} \times \mathbb{R}^+$, the set of activities, where $\mathbb{R}^+$ is the set of positive real numbers together with the symbol $\top$.

PEPA provides a small set of combinators which allow expressions to be constructed defining the behaviour of components, via the activities they undertake and the interactions between them.

Prefix $(\alpha, r).P$: This is the basic mechanism for constructing component behaviours. The component carries out activity (α, r) and subsequently behaves as component P.

Choice $P + Q$: This component may behave either as P or as Q: all the current activities of both components are enabled. The first activity to complete, determined by a *race condition*, distinguishes one component, the other is discarded.

Cooperation $P \underset{L}{\bowtie} Q$: Components proceed independently with any activities whose types do not occur in the *cooperation set* L (*individual activities*). However, activities with action types in the set L require the simultaneous involvement of both components (*shared activities*). When the set L is empty, we use the more concise notation $P \parallel Q$ to represent $P \underset{\emptyset}{\bowtie} Q$.

The published stochastic process algebras differ on how the rate of shared activities are defined. In PEPA the shared activity occurs at the rate of the slowest participant. If an activity has an unspecified rate, denoted $\top$, the component is *passive* with respect to that action type. This means that the component does not influence the rate at which any shared activity occurs.

Hiding P/L: This behaves as P except that any activities of types within the set L are *hidden*, i.e. they exhibit the unknown type τ and can be regarded as an internal delay by the component. These activities cannot be carried out in cooperation with another component.

Constant $A \overset{def}{=} P$: Constants are components whose meaning is given by a defining equation. $A \overset{def}{=} P$ gives the constant A the behaviour of the component P. This is how we assign names to components (behaviours).

The evolution of a model is governed by the structured operational semantics rules of the language. This gives rise to a continuous-time Markov chain which can be solved to obtain a steady-state probability distribution from which performance measures can be derived.

2.3 The Proposed Approach

When the assembly code of an application is submitted to MAQAO for instrumentation, the hot loops of the program are selected and the corresponding Data Dependencies Graphs (DDGs) are generated, one for each hot loop. These graphs are computed with intra and inter iteration dependencies and are enriched with static cycles estimated by compiler.

In our approach, and in order to investigate the impact of each transformation on the code performances, we propose to extract, after each transformation applied, the hot loops of an application using the DDGs. Based on these graphs which provide only static information on the transformation applied, we build a PEPA model for each new version of a selected loop.

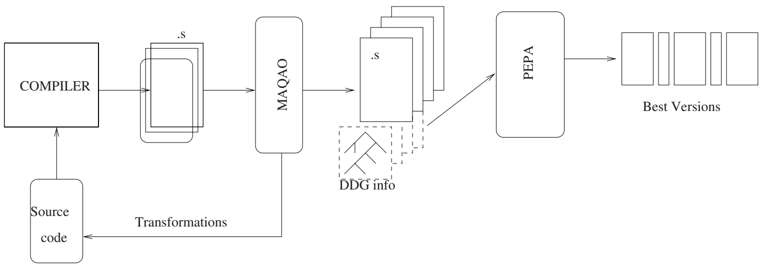


Fig. 1. Building PEPA models from the DDGs of a selected loop

In this paper, three code transformations are investigated. Each time the DDG of the selected loop is extracted and the corresponding PEPA model built. The first transformation, called *noUnroll* transformation, consists in adding to the hot loop, at the source level, a command which forces the compiler to not unroll the code. The second transformation consists in reducing the definition interval of the loop index. We call it the *index inversion* transformation. Finally the last transformation investigated consists in breaking the data dependencies of the code. In this case, an instruction of the hot loop is decomposed in several instructions to execute several iterations separately.

In the following, using a case study we show how we apply each of the transformations on the original code of a selected loop, the corresponding generated DDG and the PEPA model.

3 Case Study

To investigate the impact of the selected transformations on the execution time of the code, we consider the CX3D application. CX3D is an MPI application used to simulate Czochralski crystal growth [2], a method applied in the silicon-wafer production to obtain a single crystal from semiconductors, metals, ... It covers the convection processes occurring in a rotating cylindrical crucible filled with liquid melt.

The program of the C3XD application contains several hot loops. The one we select and extract is the inner loop of the following code where $Imax = 31$, $Kmax = 91$ and $Nmax = 41$.

```
DO  I = 2,  Imax
  DO  K = 2,  Kmax
    DO  N = 2,  Nmax
      Pnew = Max(Pnew, DP(I, K, N))
      CONTINUE
```

In the following we apply each of the three transformations described above on the inner loop of this code. In each case, we present the generated DDG of this loop and for each graph, we describe its corresponding PEPA model.

3.1 The Original Code

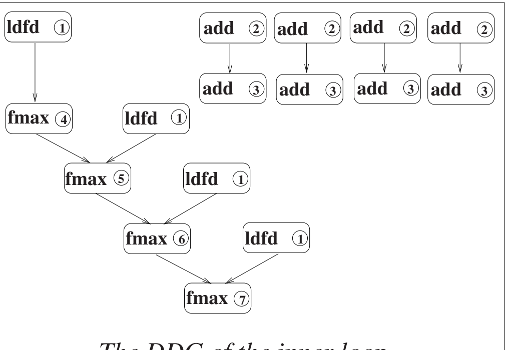
To start, we consider the original version of the loop and generate its corresponding DDG using MAQAO (Figure 2). Each node of the graph contains the assembly instruction to be executed and a number.

Three types of instructions are used: *add*, *ldfd* and *fmax*. Instruction *ldfd* represents a floating point memory access instruction. It is an $8-byte$ float point load instruction. Instruction $fmax$ determines the maximum numeric value of its arguments (floating-point maximum) [9].

The numbering in the graph specifies the order in which the instructions have to be executed. According to this numbering, several instructions have to be

```
DO I=2, Imax
  DO K=2, Kmax
    DO N=2, Nmax
      Pnew = MAX (Pnew, DP(I,K,N)
      CONTINUE
```

The original code



The DDG of the inner loop

Fig. 2. The original code and the DD Graph of selected loop

executed at the same time. For example, all loading instructions $ldfd$ have to be executed at the same time, at stage 1 of the execution process. Similarly, four *add* instructions are to be executed simultaneously at stage 2 and stage 3. Unlike the loading instructions, instructions $fmax$ have to be executed sequentially.

In the following, we present the PEPA model of the graph of the selected loop.

The PEPA Model. This model consists of five components types. These components are built such that their activities match the instructions in the graph and their ordering. Note that because some instructions appear several times in the graph and to avoid any ambiguity in the PEPA model, we number the corresponding activities.

– Component *Register* plays the role of an instruction register which specifies the next instruction to execute. It allows us to respect the instruction order given by the graph as well as the simultaneous execution of certain instructions. Action type *load* models the loading instruction $ldfd$ and action $fmax_4$ models the first $fmax$ instruction to be executed at stage 4 of the graph. Actions add_i, $i = 2, 3$, model the instructions *add* to be executed at stages 2 and 3 respectively.

$$Register \overset{def}{=} (load, r).Register_1 \qquad Register_2 \overset{def}{=} (add_3, \top).Register_3$$
$$Register_1 \overset{def}{=} (add_2, \top).Register_2 \qquad Register_3 \overset{def}{=} (fmax_4, s_4).Register$$

– Component $Maxi$ models the left path of the graph, from $(ldfd, 1)$ to $(fmax, 7)$. Action types $fmax_i$, $i = 5, 6, 7$ model the three last $fmax$ in the path.

$$Maxi \overset{def}{=} (load, \top).Maxi_1 \qquad Maxi_3 \overset{def}{=} (fmax_6, s_6).Maxi_4$$
$$Maxi_1 \overset{def}{=} (fmax_4, \top).Maxi_2 \qquad Maxi_4 \overset{def}{=} (fmax_7, s_7).Maxi$$
$$Maxi_2 \overset{def}{=} (fmax_5, s_5).Maxi_3$$

– Components $Comp_i$, $i = 5, 6$, each of these components models a branch, $Comp_1$ for branch $(ldfd, 1)$ to $(fmax, 5)$ and $Comp_2$ for $(ldfd, 1)$ to $(fmax, 6)$.

$$Comp_i \overset{def}{=} (load, \top).Comp'_i \qquad Comp'_i \overset{def}{=} (fmax_i, \top).Comp_i$$

– Component $Comp_{j,k}$, $j = 1, \ldots, Nmax$ and $k = 1, 2$ models the last branch of the main part of the graph, that is from $(ldfd, 1)$ to $(fmax, 7)$. As this $fmax$ is the last instruction to be executed in the whole graph, $Comp_{j,k}$ allows us to model also the iteration process of the inner loop in the code. If $1 \leq j \leq Nmax - 1$, we have:

$$Comp_{j,1} \overset{def}{=} (load, \top).Comp_{j,2} \qquad Comp_{j,2} \overset{def}{=} (fmax_7, \top).Comp_{j+1,1}$$

Finally when $j = Nmax$, we have:

$$Comp_{Nmax,1} \overset{def}{=} (load, \top).Comp_{Nmax,2}$$
$$Comp_{Nmax,2} \overset{def}{=} (fmax_7, \top).Comp_{1,1}$$

– Component *Adder* models the behaviour of a sequence $(add, 2)$ to $(add, 3)$ in the graph. The four sequences of this type in the graph have to synchronise on both instructions. Therefore, these sequences are aggregated in the PEPA model to a single component with the same activities and the same rate.

$$Adder \overset{def}{=} (add_2, a_2).Adder' \qquad\qquad Adder' \overset{def}{=} (add_3, a_3).Adder$$

The whole model equation is:

$$CodeO \overset{def}{=} (Adder \underset{K_1}{\bowtie} (Register \underset{K_2}{\bowtie} (Maxi \underset{K_3}{\bowtie} (Comp_1 \underset{K_4}{\bowtie} (Comp_2 \underset{K_4}{\bowtie} Comp_{1,1})))))$$

where the cooperation sets are defined as $K_1 = \{add_2, add_3\}$, $K_2 = \{load, fmax_4\}$, $K_3 = \{load, fmax_5, fmax_6, fmax_7\}$ and $K_4 = \{load\}$.

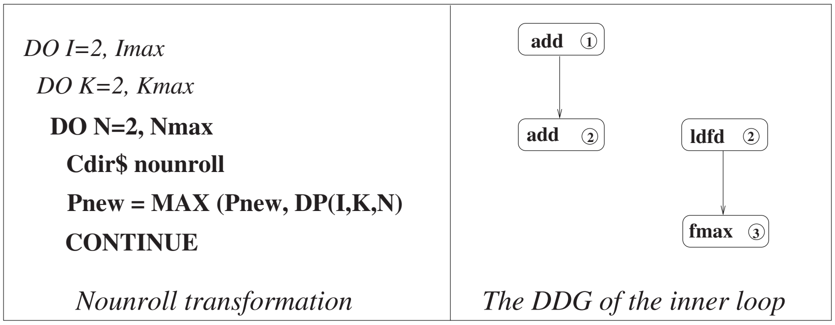
3.2 The noUnroll Transformation

The *noUnroll* transformation consists in introducing a command to force the compiler to not unroll the program. We use command "*Cdir$ nounroll*" as shown in Figure 3. The consequence of such a command is the generation of a much smaller dependency graph. Like in the previous graph, each node consists of the assembly instruction and a number specifying its position in the execution process. This graph specifies that two instructions, *add* and *ldfd*, must be executed simultaneously at stage 2.

The PEPA Model. As the DDG of the selected loop is simpler in this case, the corresponding PEPA model is also simpler. It consists of three components.

– Component $Register_i$ $1 \leq i \leq Nmax$: as in the previous model, this component plays the role of an instruction register. However, it also allows modelling the $Nmax$ iterations of the modelled loop. For $1 \leq i < Nmax$, we have:

$$Register_i \overset{def}{=} (add, \top).Register'_i \qquad Register'_i \overset{def}{=} (fmax, \top).Register_{i+1}$$



```
DO I=2, Imax
 DO K=2, Kmax
  DO N=2, Nmax
   Cdir$ nounroll
   Pnew = MAX (Pnew, DP(I,K,N)
   CONTINUE
```

Fig. 3. The code with no unroll and the DD Graph of selected loop

The last iteration of the loop is given by $i = Nmax$, such that:

$$\begin{aligned} Register_{Nmax} &\stackrel{def}{=} (add, \top).Register'_{Nmax} \\ Register'_{Nmax} &\stackrel{def}{=} (fmax, \top).Register_1 \end{aligned}$$

– Component $Process_1$ models the sequence of add instructions. However, as the second add in the sequence has to be executed at the same time as $ldfd$, both are modelled using activity add_load.

$$Process_1 \stackrel{def}{=} (add, a).Process'_1 \qquad Process'_1 \stackrel{def}{=} (add\_load, l).Process_1$$

– Component $Process_2$ models the $ldfd$ and $fmax$ sequence of instructions.

$$Process_2 \stackrel{def}{=} (add\_load, \top).Process'_2 \qquad Process'_2 \stackrel{def}{=} (fmax, f).Process_2$$

The whole model equation is given by:

$$CodeN \stackrel{def}{=} (process_1 \underset{\{add\_load\}}{\bowtie} process_2) \underset{\{add, fmax\}}{\bowtie} Register$$

3.3 Index Inversion Transformation

This transformation consists in exchanging the inner loop index with the outer loop index which is smaller. Thus the number of iterations of the inner loop becomes smaller (31 instead of 41). Consequently, the generated graph is a simple two node graph. This dramatic reduction in the graph size, compared to the one of the original code, is due to the fact that as $Nmax$ is now smaller, the CPU does not need to use the $L3$ memory cache, it uses the $L2$ memory cache only.

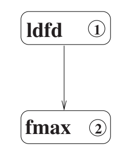
The PEPA Model. For this version of the program, the PEPA model consists of two components $Process$ and $Register_i$, $1 \leq i \leq Nmax$.

– Component $Process$ models the behaviour described by the graph. That is loading the data before executing instruction $fmax$.

$$Process \stackrel{def}{=} (load, l).Process_1 \qquad Process_1 \stackrel{def}{=} (fmax, f).Process$$

```
DO N=2, Nmax
 DO K=2, Kmax
  DO I=2, Imax
   Pnew = MAX (Pnew, DP(I,K,N)
   CONTINUE
```

Indices I and N inversion



The DDG of the inner loop

Fig. 4. Code with index inversion and the DD Graph of selected loop

– Component $Register_i$ models the iteration process.

$$\begin{aligned}
Register_i &\overset{def}{=} (fmax, \top).Register_{i+1} && if\ \ 1 \le i < Nmax \\
Register_i &\overset{def}{=} (fmax, \top).Register_1 && if\ \ i = Nmax
\end{aligned}$$

The model equation consists of the cooperation of the two components of the model over activity $fmax$.

$$CodeI \overset{def}{=} Process \underset{\{fmax\}}{\bowtie} Register_1$$

3.4 Breaking the Dependencies

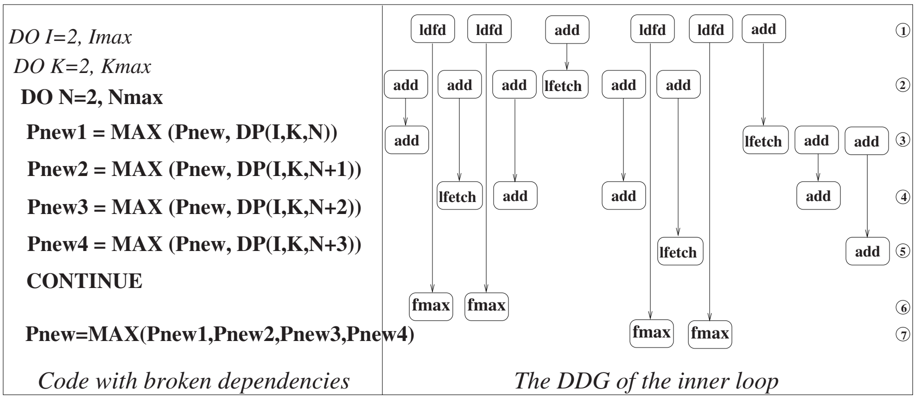
The last transformation investigated consists in breaking the data dependencies of the code. In this case, the instruction of the inner loop is decomposed in four instructions in order to execute four iterations in one (see Figure 5).

Note that $lfetch$ instruction is used to explicitly prefetch data into the L1, L2, or L3 caches [9].

The PEPA Model. As the DDG generated after this transformation is bigger than the previous one, the PEPA model is also bigger. It consists of eleven components in which a combined name for an activity implies the simultaneous execution of the instructions behind this name.

– Component $Register$ plays the role of the instruction register which states the next instruction to be executed in a program. Following the numbering used in the graph, the order of instructions is the following.

$$\begin{aligned}
Register\ \ &\overset{def}{=} (ldfd\_add_1, l).Register_1 \\
Register_1 &\overset{def}{=} (lfetch\_add_2, d_2).Register_2 \\
Register_2 &\overset{def}{=} (lfetch\_add_3, d_3).Register_3 \\
Register_3 &\overset{def}{=} (lfetch\_add_4, d_4).Register_4 \\
Register_4 &\overset{def}{=} (lfetch\_add_5, d_5).Register_5
\end{aligned}$$



```
DO I=2, Imax
 DO K=2, Kmax
  DO N=2, Nmax
   Pnew1 = MAX (Pnew, DP(I,K,N))
   Pnew2 = MAX (Pnew, DP(I,K,N+1))
   Pnew3 = MAX (Pnew, DP(I,K,N+2))
   Pnew4 = MAX (Pnew, DP(I,K,N+3))
   CONTINUE

   Pnew=MAX(Pnew1,Pnew2,Pnew3,Pnew4)
```

Fig. 5. The code with broken dependencies

$$Register_5 \overset{def}{=} (fmax_6, f_6).Register_6$$
$$Register_6 \overset{def}{=} (fmax_7, f_7).Register$$

Note that the index of an activity refers to the execution stage number.

– Component $Process_1$ models sequence $(ldfd, 1)$ to $(fmax, 6)$ which appears twice in the graph. As both sequences must synchronise on both instructions, a single component models both sequences.

$$Process_1 \overset{def}{=} (ldfd\_add_1, \top).Process_1' \qquad Process_1' \overset{def}{=} (fmax\_6, \top).Process_1$$

– Component $Process_2$ models sequence $(add, 1)$ to $(lfetch, 2)$.

$$Process_2 \overset{def}{=} (ldfd\_add_1, \top).Process_2' \qquad Process_2' \overset{def}{=} (lfetch\_add_2, \top).Process_2$$

– Component $Process_3$ models sequence $(ldfd, 1)$ to $(fmax, 7)$ which appears twice in the graph. As both sequences must synchronise on both instructions, a single component models both sequences.

$$Process_3 \overset{def}{=} (ldfd\_add_1, \top).Process_3' \qquad Process_3' \overset{def}{=} (fmax_7, \top).Process_3$$

– Component $Process_4$ models sequence $(add, 1)$ to $(lfetch, 3)$.

$$Process_4 \overset{def}{=} (ldfd\_add_1, \top).Process_4' \qquad Process_4' \overset{def}{=} (lfetch\_add_3, \top).Process_4$$

– Component $Process_5$ models sequence $(add, 2)$ to $(add, 3)$.

$$Process_5 \overset{def}{=} (lfetch\_add_2, \top).Process_5' \qquad Process_5' \overset{def}{=} (lfetch\_add_3, \top).Process_5$$

– Component $Process_6$ models sequences starting with $(add, 2)$ and finishing with add or $lfetch$ at stage 4. The three sequences can be modelled using a single component.

$$Process_6 \overset{def}{=} (lfetch\_add_2, \top).Process_6' \qquad Process_6' \overset{def}{=} (lfetch\_add_4, \top).Process_6$$

– Component $Process_7$ models sequence $(add, 2)$ to $(lfetch, 5)$.

$$Process_7 \overset{def}{=} (lfetch\_add_2, \top).Process_7' \qquad Process_7' \overset{def}{=} (lfetch\_add_5, \top).Process_7$$

– Component $Process_8$ models sequence $(add, 3)$ to $(lfetch, 4)$.

$$Process_8 \overset{def}{=} (lfetch\_add_3, \top).Process_8' \qquad Process_8' \overset{def}{=} (lfetch\_add_4, \top).Process_8$$

– Component $Process_9$ models sequence $(add, 3)$ to $(add, 5)$.

$$Process_9 \overset{def}{=} (lfetch\_add_3, \top).Process_9' \qquad Process_9' \overset{def}{=} (lfetch\_add_5, \top).Process_9$$

– Component $Iteration_i$, $1 \leq i \leq Nmax$: it models the iteration process of the loop. If $1 \leq i < Nmax$, we have:

$$Iteration_i \overset{def}{=} (ldfd\_add_1, \top).Iteration_{i,1} \qquad Iteration_{i,1} \overset{def}{=} (fmax_7, \top).Iteration_i$$

When $i = Nmax$, we have

$$Iteration_{Nmax} \overset{def}{=} (ldfd\_add_1, \top).Iteration_{Nmax,k}$$
$$Iteration_{Nmax,1} \overset{def}{=} (fmax_7, \top).Iteration_1$$

The complete model equation is:

$$CodeC \overset{def}{=} Iteration_1 \underset{M_1}{\bowtie} (Register \underset{M_2}{\bowtie} (Process_1 \underset{M_3}{\bowtie} (Process_2 \underset{M_4}{\bowtie} (Process_3 \underset{M_3}{\bowtie} (Process_4 \underset{M_3}{\bowtie} (Process_5 \underset{M_6}{\bowtie} (Process_6 \underset{M_7}{\bowtie} (Process_7 \underset{M_8}{\bowtie} (Process_8 \underset{M_5}{\bowtie} Process_9))))))))$$

where the cooperation sets are defined as $M_1 = \{ldfd\_add_1, fmax_7\}$, $M_2 = \{ldfd\_add_1, lfetch\_add_2, lfetch\_add_3, lfetch\_add_4, lfetch\_add_5, fmax_6, fmax_7\}$, $M_3 = \{ldfd\_add_1\}$, $M_4 = \{ldfd\_add_1, lfetch\_add_2\}$, $M_5 = \{lfetch\_add_3\}$, $M_6 = \{lfetch\_add_2, lfetch\_add_3\}$, $M_7 = \{lfetch\_add_2, lfetch\_add_4\}$ and $M_8 = \{lfetch\_add_5\}$.

4 Numerical Results

As one of the sensitive performance measures for our application is the time required to complete the execution of the selected loop, we have used the HYDRA analyser [3] to compute the cumulative passage-time distribution function for completing a hot loop. To translate the PEPA model into an HYDRA input file, we have used Imperial PEPA Compiler (IPC) [1].

The parameters values we have used in our experiments are reported in Table 6. For these values, the cumulative passage-time distribution function for completing the execution of the selected loop is given in Figures 7 and 8.

Original		NoUnroll		Inversion		Breaking Dep.	
Rates	Values	Rates	Values	Rates	Values	Rates	Values
s_1, s_2, s	3	l	1.25	f	7.5	d_1, d_2, d_3, f_1, l	7.5
s_3	2.143	f	15	l	1.25	d_4	1.667
a_1, r	7.5	a	7.5			f_2	3
a_2	1.5						

Fig. 6. The parameters values ($nanoseconds^{-1}$)

Figures 7 and 8 show that, on the three transformations investigated, only the *index inversion* transformation has a positive impact on the execution time of the selected loop. For all values of the iteration number $Nmax$, the time required to complete the execution is smaller when applying this transformation. Thus, when $Nmax = 12$ (Figure 7 left), we can see that, in the worst case, with the original code the loop will be completed in 10 *nanoseconds* while when using the *index inversion*, the completion time is only 5 *nanoseconds*.

Similarly, when $Nmax = 40$ (Figure 8 right), it will require about 30.10^{-9} *seconds* to complete the loop in the the worst, instead of 10 *nanoseconds* with the *index inversion*.

These figures show also that it is far much better to keep the loop in its original version than using any other transformation as both the *noUnroll* and *breaking dependencies* transformations may increase the execution time, the worst being the *breaking dependencies* transformation.

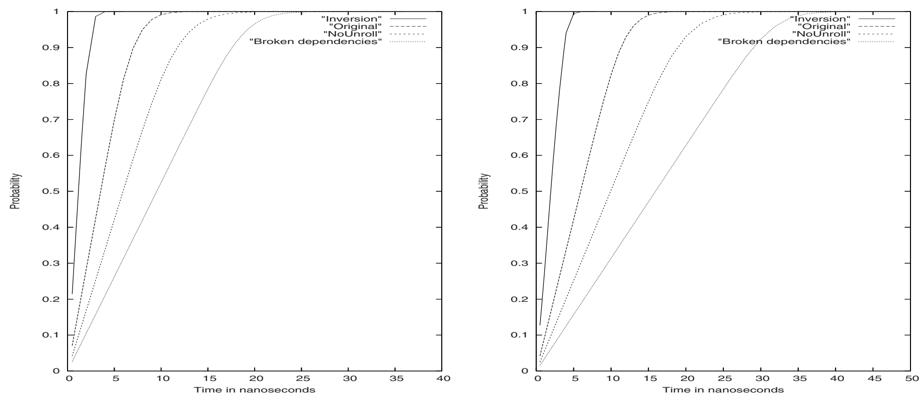


Fig. 7. Cumulative passage-time distribution function for $Nmax = 12$ and $Nmax = 20$

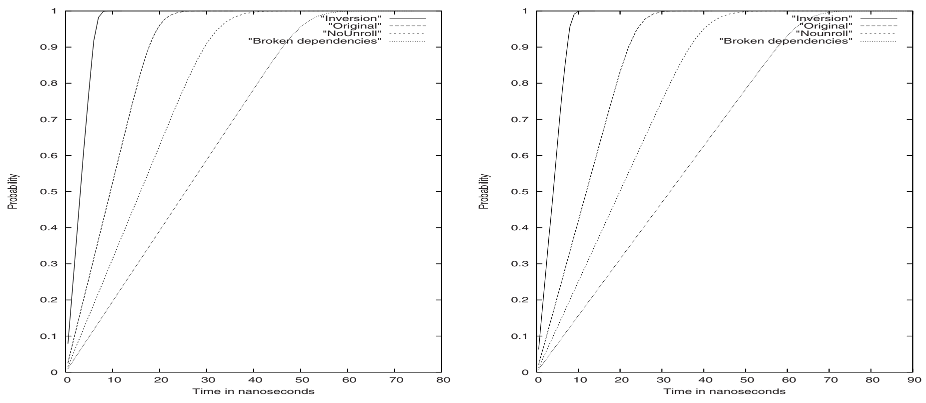


Fig. 8. Cumulative passage-time distribution function for $Nmax = 32$ and $Nmax = 40$

In the following we compare these results of the analytical model to the execution times of the loop on a BULL Itanium 2 Novascale system, 1.6GHz, 3MB of L3. The codes were compiled using Intel ICC/IFORT 9.1.

As specified by the DDGs seen in Section 3, some instructions have to be executed at the same time. This is possible on a non parallel machine like Itanium 2 because its processor is built around the EPIC (Explicitly Parallel Instruction Computing) architecture which main feature is the coupling between software and hardware. In this case the compiler is given the task of arranging the code in a certain way to benefit from the parallelism in order to simplify the hardware architecture of the processor.

Table 9 summarises the execution times obtained for our loop. The results are given for different values of the iteration number $Nmax$ and the three transformations. The times reported in this table are higher than the ones obtained with

Loop trip	Original	NoUnroll	Inversion	Breaking Dep.
12	134	135	61	309
20	404	405	132	689
32	787	788	265	1279
40	938	1052	372	1739

Fig. 9. Execution times (*nanoseconds*)

the analytical model. The main reason is related to the rates used in the PEPA models. These rates are computed using the number of cycles the compiler has estimated for each assembly instruction before the code execution. During the execution these estimated numbers may not be the ones used, and be higher because of the processor speed, its availability, the inputs/outputs, or the memory size.

However, like the results of the analytical model, the results in Table 9 show that the *index inversion* transformation leads to the best execution times. Moreover, like previously, they show that it is better to use the original code than the *breaking dependencies* or the *noUnroll* transformation.

Clearly these results lead to the same conclusions as the ones obtained using our approach. However, our approach allows us to investigate the impact of different transformations on a selected loop quicker than a direct execution of the code. Indeed, if the results reported in Table 9 are the execution times of just the loop, the whole program of the CX3D application had to be executed. And the execution of this program takes several minutes using MAQAO.

5 Conclusions

In this paper we have proposed an approach which allows investigating the effectiveness of several optimisations on the execution time of the code.

The stem of our work is the diagnostic that in scientific computing a consequent fraction of the execution time is the time spent in loops with a limited number of iterations. We come out with a novel method for quickly evaluating different transformations in order to select the best one. The technique is faster than simulation and instrumentation because we isolate and evaluate just the hot loops.

In the future, we plan to extend our approach in two important ways. First, we are interested in developing an interface between MAQAO and PEPA, in order to provide the user with an automatic formal approach. We also plan to propose an infrastructure to cooperate with dynamic analysis when we have large input data that lead to cache misses. Indeed, while our approach is fairly straightforward, it does rely on a host of program analysis available in Itanium 2 compiler. In the case of large input data, performance may be unpredictable even if only two parameters are taken into account. In the future, we propose to combine analytical modelling with the tracing of cache behaviour.

References

1. Bradley, J.T., Dingle, N.J., Gilmore, S.T., Knottenbelt, W.J.: Extracting passage times from PEPA models with the HYDRA tool: A case study. In: Proceedings of the Nineteenth annual UK Performance Engineering Workshop, pp. 79–90 (2003)
2. Czochralski, J.: Z. Phys. Chem. 92 (1918)
3. Dingle, N.J.: Parallel Computation of Response Time Densities and Quantiles in Large Markov and Semi-Markov Models, PhD. Thesis, Imperial College, London (2004)
4. Djoudi, L., Barthou, D., Carribault, P., Lemuet, C., Acquaviva, J.-T., Jalby, W.: MAQAO: Modular Assembler Quality Analyzer and Optimizer for Itanium 2. In: The 4th Workshop on EPIC architectures and compiler technology, San Jose (2005)
5. Djoudi, L., Barthou, D., Tomaz, O., Charif-Rubial, A., Acquaviva, J., Jalby, W.: The Design and Architecture of MAQAOPROFILE: an Instrumentation MAQAO Module. In: The 6th Workshop on architectures and compiler technology, San Jose (2007)
6. Djoudi, L., Noudohouenou, J., Jalby, W.: MAQAOAdvisor: A MAQAO Module For Detecting Analyzing And Fixing Performance Problem. In: The 1st International Workshop on Tools Infrastructures and Methodologies for the Evaluation of Research Systems, Austin (2008)
7. Graham, S.L., Kessler, P.B., Mckusick, M.K.: Gprof: A call graph execution profiler. SIGPLAN Not. 17(6), 120–126 (1982)
8. Hillston, J.: A Compositional Approach to Performance Modelling, PhD. Thesis, University of Edinburgh (1994)
9. Intel Itanium 2 Processor Reference Manual For Software Development and Optimization, 251110-002 (April 2003)
10. Larus, J.R., Schnaar, E.: EEL: Machine-Independent Executable Editing. In: PLDI 1995: Proceedings of the ACM SIGPLAN conference on Programming language design and implementation, La Jolla, California (June 1995)
11. Hill, M.: "prof", Unix Programmer's Manual, Section 1. Bell Laboratories, NJ (January 1979)