2014 International Conference on Computational Science and Computational Intelligence

An Automated Approach to Hardware Performance Monitoring Counters

Fernando G. Tinetti
Comisión de Inv. Científicas Prov. de Bs. As.
III-LIDI, Fac. de Informática, UNLP
1900 La Plata, Argentina.
Email: fernando@info.unlp.edu.ar

Mariano Méndez
III-LIDI, Fac. de Informática
Universidad Nacional de La Plata (UNLP)
1900 La Plata, Argentina.
Email: marianomendez@gmail.com

***Abstract*—Program performance optimization could be a very complex process, even with current software development facilities/tools. An Integrated Development Environment (IDE) usually does not include many aids for optimization and/or performance evaluation. We propose to include performance evaluation through hardware monitoring counters into IDE software. Currently, it is possible to reach hardware monitoring counters via many libraries, and we have also seen that many of those libraries are approximately at the same abstraction level (including the way at which they allow access to the hardware counters). Thus, it is not only possible to include some performance evaluation library into the development process but, also, including specific aids to use some library via *configurable/adjustable code snippets*. We show, as a proof of concept, an Eclipse plug-in to help High Performance Computing (HPC) programmers to access hardware monitoring event counters using PAPI (Performance API). The plug-in is able to automatically include source code to count specific events available via PAPI in sections of source code defined by the programmer. Also, given that the code is automatically included, it would be also possible to remove that code from the release version (for the production environment).**

Keywords*—*Performance, Hardware Counters, High Performance Computing, Fortran

I. Introduction

Programming in the HPC field implies at least optimization since response time should be reduced for solving grand challenge/compute intensive problems [1]. Even when HPC has always been related to parallel computing, the relationship HPC-parallel computing is now stronger, because sequential computing is almost tied to a few GHz mainly due to heat dissipation problems found on current technology [2] [3] [4]. There are well known optimization techniques, and many of them are already implemented in compilers [5] [6] [7]. However, the increasing complexity of (microprocessors') microarchitectures plus the usage of (at least) multiple cores for parallel processing has led to HPC programmers to have into account a large number of guidelines for optimization and parallelization [8] [9] [10]. Development tools in this scenario do not provide enough aids or, at least, have not evolved accordingly.

Most of the optimization and parallelization work is mainly guided through timing experiments [11]. Even when the methodology has been successful, sometimes does not provide enough information about specific bottlenecks and performance penalties to the non-specialist or inexperienced HPC programmer. Even more, timing is a good but indirect measurement of what is going on at the hardware level. Since the availability of hardware performance event counters, the HPC developer has access to invaluable (and otherwise impossible to obtain) hardware performance, thus having more reliable information for the optimization and parallelization work [12].

Even when manufacturers have provided access to event counters and there are specific libraries for access to such counters, the instrumentation process is too detailed/of low abstraction level, quite complex, and prone to error. In the performance evaluation cycle of: a) instrumentation, b) profiling execution, and c) performance analysis of results, we propose to aid the developer by adding the portions of instrumentation source code. We have identified a methodology enabling the programmer to select event counters for automatic inclusion of the necessary (source code) instrumentation and, also, we have implemented that methodology into a specific IDE as a proof of concept. Thus, the scientific programmer can be focused on "what" to analyze (for optimization and parallelization) instead of "how" to carry out the access to hardware monitoring event counters when such counters are part of the analysis.

The organization of the rest of this article is as follows. In Section II we describe hardware counters, their usage, and how they have evolved throughout time. Section III includes a description of the Performance Application Programming Interface (PAPI). The proposed automated methodology and the implemented tool are described in Sections IV and V. In Section VI a Fortran example program has been measured by adding a counter for the number of store and load instructions at runtime. Finally, the conclusions and further work are given in Section VII.

II. Hardware Performance Counters

Hardware counters are a consequence of a hardware design technique called Design For Test (DFT). DFT "is a general term applied to design methods that lead to more thorough and less costly testing" [13]. This technique has been used since the early age of electronics to include a set of testability features in the hardware for fault detection. In the 40's and 50's, engineers used to check some features like the voltage from the analog computers by using some instruments [14].

Nowadays, the DFT concept is applied on modern microprocessors as the broadly known hardware performance

978-1-4799-3010-4/14 $31.00 © 2014 IEEE
DOI 10.1109/CSCI.2014.19

event counters. Paradoxically, the history of Intel's computer hardware counters has been related to unrevealed corporation secrets at least regarding the programming community [15]. The most emblematic and known case is the Intel Pentium set of hardware counters which saw the light of the day, in 1994, as a non documented feature [16][15]. The Intel Pentium processor "has a comprehensive set of performance counters similar to the ones existing in PowerPC and it also has four breakpoint registers for establishing breakpoints. Although these features are not documented, and are only available through a non-disclosure agreement with Intel, Pentium debugging and performance monitoring features have been reverse-engineered and published in [15]" [17]. Intel changed its policy regarding this undocumented registers from "considered Intel confidential and proprietary" Apendix H from [18] to "The developers realized that the utility would be useful for other programmers, so they obtained permission from Intel to distribute the program, as long as the source code was kept secret" [15].

After that incident there has been a shift of perspective regarding the use of DFT features by Intel Corporation, Intel made these features accessible for the programmers community. The first Intel processors which had performance monitoring capabilities were those in the Pentium P5 family, although these capabilities were not officially available to be used until the appearance of Pentiun P6 family. In the P5 Pentium family processors and the P6 Family processors 2 hardware counters were available. In the Intel Core Microarchitecture, 3 function counters per thread can be used. The processors based on Intel NetBurst Microarchitecture have 18 performance counters. The processors based on Intel Sandy Bridge Microarchitecture and processors based on Intel Westmere Microarchitecture have 8 general-purpose performance counters (g.p.p.c.) and 3 function counters per thread (f.c.p.t). Finally, the processors based on Intel Nehalem Microarch have 4 g.p.p.c. and 3 f.c.p.t [19].

Programmers have been using hardware counters for analysing how their programs behave in a specific platform by using the chips monitoring features in two ways. The broadest way is to perform an analysis by using a third party software made for this purpose, such as: perf, perfsuite, perfcrt, perfmon, Oprofile, Vtune, and so forth. The second and less known way, perhaps because it is more complicated and tailored, is performing the analysis by using a specific API/library. Using some library implies instrumentation, i.e. changing the source code to be measured, including calls to some libray such as: PAPI [20], Intel Performance Counter Monitor library [21], Rabbit [22], PCL [23], and so forth. There are, however, a succession of tasks when accessing libraries: start the library, build a set of events to be monitored, start to count, stop to count and, finally, get measurements. All the steps listed up above, are similar in the different libraries, as shown in Figure 1 on two examples, one written in Fortran using PCL and one written in C using PAPI.

III. PAPI: Performance Application Programming Interface

Using and measuring the processors monitoring features has been "an imprecise art" [24] for many years. The main reasons probably are: poor hardware documentation, unavailability to the programmer, portability, and so forth [25].

```
PROGRAM PCL_test
!declarations
INCLUDE 'pclh.f'
INTEGER counter_list(1), flags, res
INTEGER*8 i_result, descr
REAL*8 fp_result
INTEGER PCLquery,PCLstart,PCLstop
EXTERNAL PCLquery,PCLstart,PCLstop

res = PCLinit(descr)
IF(res .NE. PCL_SUCCESS)
   WRITE(*,*) 'error in PCLinit'

!count for user mode
flags = PCL_MODE_USER

!count cycles
counter_list(1) = PCL_CYCLES

!check if event is available
res = PCLquery(descr,counter_list,1,flags)
IF(res .EQ. PCL_SUCCESS)THEN

  !start counting
  IF(PCLstart(descr, counter_list&
    , 1, flags) .NE. PCL_SUCCESS) THEN
    WRITE(*,*) 'problem starting events'&
    ,counter_list(1)
  END IF

!do some work

!stop counting
  IF(PCLstop(descr,i_result,fp_result,1)&
     .NE. PCL_SUCCESS) THEN
    WRITE(*,*) 'probl. stopping events'&
     ,counter_list(1)
  ELSE
    WRITE(*,*) 'result: ', i_result
  END IF

ELSE
  !event not supported
  WRITE(*,1001) counter_list(1),res
ENDIF

res = PCLexit(descr)
IF(res .NE. PCL_SUCCESS)
   WRITE(*,*) 'error in PCLexit'

STOP
END PROGRAM
```

```
#include <papi.h>
#include <stdio.h>

#define NUM_FLOPS 10000

main()
{
int retval, EventSet=PAPI_NULL;
long_long values[1];

/* Initialize the PAPI library */
retval=PAPI_library_init(PAPI_VER_CURRENT);
if (retval != PAPI_VER_CURRENT) {
  // Error
}

/* Create the Event Set */
if (PAPI_create_eventset(&EventSet)
     != PAPI_OK) handle_error(1);

/* Add Total Instructions Executed to
our Event Set */
if (PAPI_add_event(EventSet, PAPI_TOT_INS)
    != PAPI_OK) handle_error(1);

/* Start counting events in the
Event Set */
if (PAPI_start(EventSet) != PAPI_OK)
    handle_error(1);

// Do some work

/* Read the counting events in
the Event Set */
if (PAPI_read(EventSet, values)!=PAPI_OK)
    handle_error(1);

printf("After reading counters: %lld\n",
       values[0]);

/* Stop the counting of events in
 the Event Set */
if (PAPI_stop(EventSet, values)!=PAPI_OK)
    handle_error(1);

printf("After stopping the counters:
 %lld\n",values[0]);
}
```

Fig. 1. Example of PCL and PAPI Usage

PAPI, also known as Performance Application Programming Interface, provides two ways to approach hardware counters [20][24]. The first one is called the "high-level interface" which allows programmers to acquire simple measurements easily. This interface only allows the user to start, stop and show measurements taken for a specific set of events [20]. The second one, called the "low level-interface", allows programmers to perform complex measurements by introducing the abstract concept of the EventSet that allows to access to platform specific events to be measured [20] [25]. We will focus on the Fortran library but our work is also applicable to the C library.

IV. An Automated Approach to Hardware Counter

We have analysed how PAPI counters should be introduced manually in Fortran source code by using the low-level interface. After some few examples we noticed that there is a clear process that enables the automated Fortran source code instrumentation. This process can be described as:

1) Initialize PAPI Library
2) Create the set of events
3) Set counters
4) Start counting
5) Stop counting
6) Print Results

Currently, this process has to be implemented directly by the programmer with the possibility of adding at least new bugs related to the measurement process. Another level of

complexity is introduced because the availability of PAPI hardware counters depends on the platform [25]. We propose the automation of this process by integrating PAPI counter into a modern IDE (Integrated Development Environment) like Eclipse [26]. In addition, we propose a tool with the ability to detect the PAPI library availability and the list of counters which are available to the platform. We have designed two Eclipse plug-ins, the first one to interact between the IDE and the operating system. The second one to integrate platform specific available hardware counters to a Fortran Program. The second plug-in should insert the necessary Fortran source code to measure a set of specific counters, previously selected from the editor by the programmer. We will base our second plug-in on Photran [27], an advanced multiplatform integrated development environment (IDE) for Fortran based on Eclipse [26]. From the beginning, Photran was designed to support restructuring (called refactorings in Photran), and much of its development effort has been focused on providing a robust restructuring infrastructure. We have used Photran's infrastructure that supports source code transformations for developing our Eclipse plug-in.

V. The Tool

The tool has been divided into 2 sets of plug-ins. In this section we will offer a thorough description of these two eclipse plug-ins.

A. PAPI Integration Plug-in

This first plug-in has been designed to integrate PAPI library into the Eclipse Environment. For this purpose the PAPI Integration plug-in will provide the following new features:

- Determine the PAPI version.
- Show the PAPI version to the user.
- Gather the available PAPI events to count in the working platform.
- Show the available PAPI events to count in the working platform.

In order to implement these features two plug-in projects called papi.core and papi.core.ui have been created [28]. The project called papi.core contains the core classes to implement the required functionality, including the class called PAPI whose implementation is to act as a wrapper to interact with the papi-avail command. In addition, a class called PapiEvent has been implemented for modelling each PAPI event.

The second project will contain the classes to implement the required user interface to use papi.core classes by extending eclipse using the extension points. In other words, papi.core.ui will implement the new eclipse user interface extension to interact with the papi.core classes by extending the eclipse main Source menu, creating two new commands, adding the new commands to the eclipse's Source menu, and creating two new command handlers [28].

To determine and show the PAPI Version a new Eclipse command has been implemented by extending the Eclipse Command Extension Point [29][28]. In order to make it available in the eclipse workbench, a new Eclipse command Handler has been developed: VersionHandler class. This class will interact with the papi.core classes to gather the information about the PAPI library availability and its installed version. Once all this information has been retrieved, successfully or not, the information will be communicated to the Eclipse workbench [29].

A new Eclipse Command has been implemented in order to Gather and show the available PAPI events to count in the working platform [25]. This command is handled by the class ShowAvailablePapiEventsHandler which interacts with the papi.core plug-in to obtain the list of PAPI events available for this machine [28]. Once the information is available to be shown, it is displayed as an HTML file in a new Eclipse editor.

B. PAPI Photran Integration Plug-in

The Photran Integration Plug-in is responsible for gathering the set of PAPI events available in the working machine from the user interface, and then to adding these events to a piece of Fortran source code selected in the Fortran editor. This Plug-in is implemented in the papi.core.photran plug-in project as a source code transformation including an InputDialog for selecting the counter to add to the source code (as a new eclipse editor action). The most complex component of this plug-in is the way source code transformation is implemented. Our approach is using a rewritable AST (Abstrac Syntax Tree) to represent the Fortran source code [30]. Photran infrastructure allows to apply precondition checking in order to determine the correctness of the selected source code (e.g. a transformation is not made if the user selects source code withou executable statements). After these checks have been performed a dialog will be presented to the user in order to select events from the available Papi events in the platform. This dialog will contain two list boxes. the first listing available PAPI events, and the second one is a list for keeping the selected events to add to the source code, as shown in Figure 3.

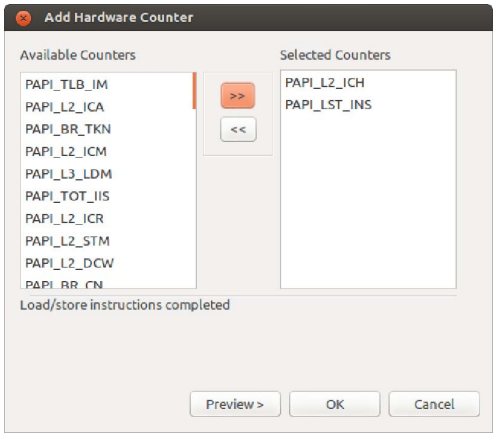

Fig. 2. A dialog implemeted for selection available PAPI events to measure.

The next step will be focused on building the necessary Fortran source code to initialize the PAPI library:

```
! initialize PAPI Library
chkflg = PAPI_VER_CURRENT
call PAPIF_library_init(chkflg)
if (chkflg .ne. PAPI_VER_CURRENT) then
```

```
    print *, 'Error PAPI Library out of date'
    call abort
  end if
```

and, after that, an event set must be created:

```
 ! create the eventset
  event_set = PAPI_NULL
  call PAPIF_create_eventset(event_set, check)
  if (check .ne. PAPI_OK) then
     print *, 'Error in subroutine PAPIF_create_eventset'
     call abort
  end if
```

Once the event set has been created each PAPI event to be used must be added to the event set. These events have been selected from the User Interface so they must be included one by one:

```
  ! set counters to <should be replaced by the event descr.>
  event_code = <should be replaced by the event name>
  call PAPIF_add_event(event_set, event_code, check)
  if (check .NE. PAPI_OK) then
     print *, "Abort After PAPIF_add_events: ", check
     call abort
  endif
```

After all the selected events have been inserted in the source code, the source code needed to start the counting process should be added as well:

```
  ! start counting
  call PAPIF_start(event_set, check)
  if(check .ne. PAPI_OK) then
     print *, 'Abort after PAPIF_start: ', check
     call abort
  endif
```

The automated source code generated must be inserted at the starting point from the selected Fortran statement sequence. There are two further steps to be performed. The first one is to stop the measurement process:

```
  ! stop counting
  call PAPIF_stop(event_set, values, check)
  if (check .ne. PAPI_OK) then
     print *, 'Abort after PAPIF_stop: ', check
     call abort
  endif
```

The second one is to print results:

```
  print *, 'Number of < event name>:  ', values(1)
  print *, 'Number of < event name>:  ', values(2)
     ...      ...      ...      ...
  print *, 'Number of < event name>:  ', values(<N>)
```

Before all these changes are be applied they will be shown in a diff view (Figure 3), allowing the user to confirm changes or cancel them. Once changes are confirmed, the original AST structure of the program will be rewritten by adding new AST nodes corresponding to the new Fortran code required to make the program capable of using the PAPI library.

VI. Using the Tool

A matrix multiplication Fortran program has been used to test the tool, i.e. to transform the source code by inserting instrumentation code. The source code is listed in Figure 4:

As an example we will focus on measurement of conditional branch instructions taken in the three nested do statements. First, the Fortran source code is selected and PAPI Hardware Counter –> Add Hardware Counters menu option is selected, as shown in Figure 5. Once the Event Selection

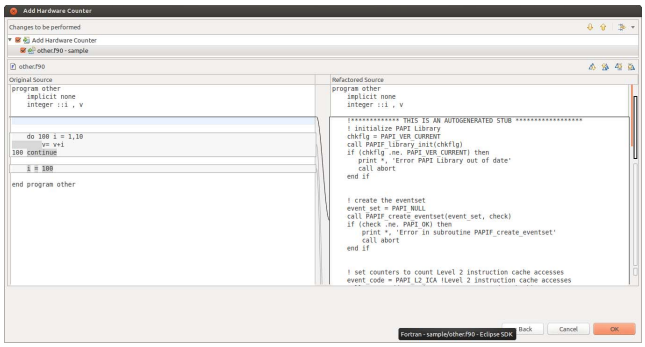

Fig. 3. Diff view before applying changes.

```
program matmul
 integer               :: row,col,i,j,k
 integer,allocatable   :: a(:,:),b(:,:), c(:,:)

 row=3000
 col=3000
 allocate(a(1:row,1:col))
 allocate(b(1:row,1:col))
 allocate(c(1:row,1:col))
  ! init matrices a and b such that c(i, j) = n
  do i = 1,row
    do j=  1,col
      a(i,j) = 1
      b(i,j) = 1
      c(i,j) = 0
      c(i,j) = 0
    end do
  end do
  do i=1,row
    do j=1,col
     do k=1,col
          c(i, j) =c(i, j) + a(i, k) * b(k, j)
      end do
    end do
  end do
 deallocate(a)
 deallocate(b)
 deallocate(c)
end program  matmul
```

Fig. 4. Matrix Multiplication Frotran Program.

Dialog is shown, the PAPI_BR_TKN is selected and added to the selected events list, as shown in Figure 6 for the event PAPI_LST_INS. A preview is shown as in Figure 7 and, finally, the changes are applied to the source code as shown in Figure 8. The programmer selects the code to be monitorized,

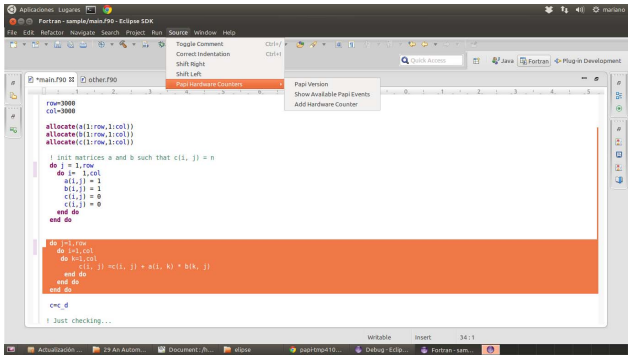

Fig. 5. Selecting source code to be transformed.

selects the specific events (from those available, which depend on the architecture and PAPI), and accepts/rejects the actual

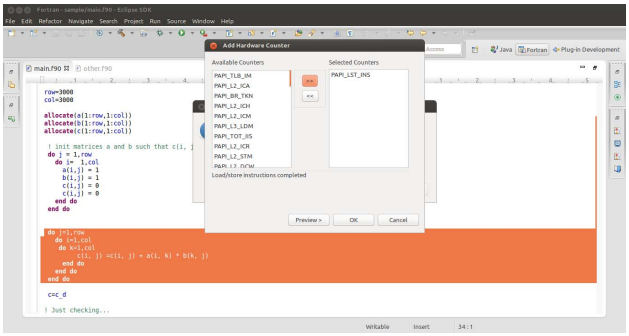

Fig. 6. Selecting PAPI Events to Measure.

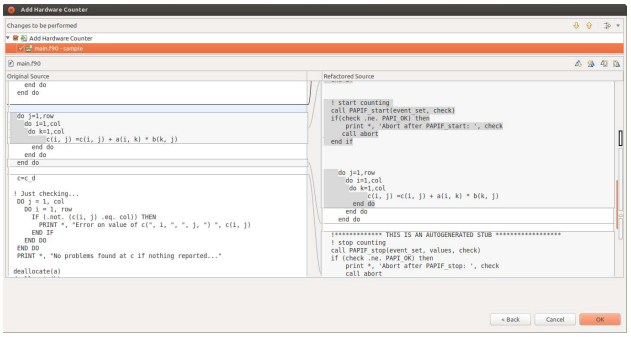

Fig. 7. Diff View Before Applying Changes.

source code transformation. After following the previous steps, the resulting program is as follows:

```
program matmul
 implicit none
 integer, parameter  :: NUM_EVENTS =1
 integer, dimension(NUM_EVENTS)  :: event_set
 integer*8, dimension(NUM_EVENTS)    :: values
 integer :: check
 integer::event_code
 integer               :: row,col,i,j,k
 integer,allocatable :: a(:,:),b(:,:),c(:,:)

 row=100
 col=100
 allocate(a(1:row,1:col))
 allocate(b(1:row,1:col))
 allocate(c(1:row,1:col))
  ! init matrices a and b such that c(i, j) = n
 do i = 1,row
   do j=  1,col
     a(i,j) = 1
     b(i,j) = 1
     c(i,j) = 0
     c(i,j) = 0
   end do
 end do

 !********* THIS IS AN AUTOGENERATED STUB **************
 ! initialize PAPI Library
 chkflg = PAPI_VER_CURRENT
 call PAPIF_library_init(chkflg)
 if (chkflg .ne. PAPI_VER_CURRENT) then
    print *, 'Error PAPI Library out of date'
    call abort
 end if

 ! create the eventset
 event_set = PAPI_NULL
 call PAPIF_create_eventset(event_set, check)
 if (check .ne. PAPI_OK) then
 print *, 'Error in subroutine PAPIF_create_eventset'
 call abort
```

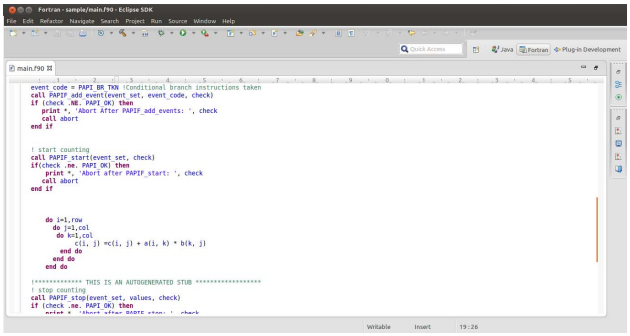

Fig. 8. Tranformed Source Code.

```
 end if

 ! set counters to count Conditional branch instructions taken
 event_code = PAPI_BR_TKN !Conditional branch instructions taken
 call PAPIF_add_event(event_set, event_code, check)
 if (check .NE. PAPI_OK) then
    print *, 'Abort After PAPIF_add_events: ', check
    call abort
 end if

 ! start counting
 call PAPIF_start(event_set, check)
 if(check .ne. PAPI_OK) then
 print *, 'Abort after PAPIF_start: ', check
    call abort
 end if

     do i=1,row
       do j=1,col
        do k=1,col
             c(i, j) =c(i, j) + a(i, k) * b(k, j)
         end do
       end do
      end do

 !********* THIS IS AN AUTOGENERATED STUB **************
 ! stop counting
 call PAPIF_stop(event_set, values, check)
 if (check .ne. PAPI_OK) then
     print *, 'Abort after PAPIF_stop: ', check
     call abort
 end if
 print *, 'Number of Conditional branch instructions
    taken:  ', values(1)

 deallocate(a)
 deallocate(b)
 deallocate(c)
end program  matmul
```

The source code transformation is successful, providing the conditional branch instructions taken at runtime.

VII. Conclusions and Further Work

In this article we have identified some repetitive steps necessary for hardware counters monitoring performance evaluation. Furthermore, we have selected the PAPI library for gathering hardware counters values. Based on these general steps we have built a tool which automates the use of the PAPI library to collect data from a specific platform hardware counters. Our tool has been built as an Eclipse plug-in, and determines which hardware counters are available on the working platform. In addition, the tool allows the user to select a set of the hardware events and to automatically add the necessary instrumentation source code to the program to

be monitored. We have found some similarities among other different monitoring libraries (e.g. PCL, Rabbit) in the way they should be introduced in the source code. Further research will expand the plug-in capabilities in order to work with these other libraries. Furthermore, we have worked with the PAPU low-level approach, but a new feature could be added to the plug-in to include the PAPI high-level approach.

REFERENCES

[1] NSF-ACCI Task Force, "The nsf-acci task force on grand challenges, national science foundation advisory committee for cyberinfrastructure task force on grand challenges final report," Mar. 2011. [Online]. Available: https://www.nsf.gov/cise/aci/taskforces/TaskForceReport_Grand-Challenges.pdf

[2] H. Sutter, "The free lunch is over: a fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, Mar. 2005, (update 2009). [Online]. Available: http://www.gotw.ca/publications/concurrency-ddj.htm

[3] S. Y. Borkar, P. Dubey, K. C. Kahn, D. J. Kuck, H. Mulder, S. S. Pawlowski, and J. R. Rattner, "Platform 2015: Intel processor and platform evolution for the next decade," 2005, intel Platform 2015 White Paper. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.92.1433

[4] F. G. Tinetti and M. Méndez, "Fortran legacy software: Source code update and possible parallelisation issues," *ACM Fortran Forum*, vol. 31, no. 1, Apr. 2012. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2179280.2179281

[5] J. A. Bilmes, K. Asanovic, C. Chin, and J. Demmel, "Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology," in *Proceedings of the International Conference on Supercomputing*, A. SIGARC, Ed., Vienna Austria, 1997.

[6] R. C. Whaley and J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the SC98 Conference*, I. Publications, Ed., Orlando USA, 1998.

[7] A. V. Aho, M. S. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Prentice Hall, 2006.

[8] V. Eijkhout, "Introduction to high performance scientific computing," 2011. [Online]. Available: http://www.tacc.utexas.edu/eijkhout/istc/istc.html

[9] S. Chellappa, F. Franchetti, and M. Püschel, "How to write fast numerical code: A small introduction," in *Generative and Transformational Techniques in Software Engineering II*, ser. LNCS 5235, R. Lämmel and J. a. S. Joost Visser, Eds., Braga Portugal, 2008, pp. 196–259. [Online]. Available: http://link.springer.com/book/10.1007/978-3-540-88643-3/page/1

[10] R. Hyde, *Write Great Code, Volume 2: Thinking Low-Level, Writing High-Level*, 1st ed. No Stach Press, 2006.

[11] R. C. Whaley and A. M. Castaldo, "Achieving accurate and context-sensitive timing for code optimization," *Software: Practice and Experience*, vol. 38, no. 15, pp. 1621–1642, Dec. 2008.

[12] F. G. Tinetti and S. M. Martin, "Sequential and shared and distributed memory parallelization in clusters: N-body/particle simulation," in *24th IASTED International Conference on Parallel and Distributed Computing and Systems*, ser. PDCS, A. Press, Ed., Las Vegas USA, Dec 2012.

[13] J. F. Wakerly, *Digital design principles and practices*. Prentice-Hall, Inc., 1989.

[14] E. Gould, "Serial replacement maintenance philosophies and multiple-failure diagnostic strategies: a marriage of multiple-fault integrity and common cause sensibility," in *AUTOTESTCON, 97. 1997 IEEE Autotestcon Proceedings*. IEEE, 1997, pp. 446–454.

[15] T. Mathisen, "Pentium secrets," *Byte magagize*, 1994.

[16] M. Schmit, "Optimizing pentium code," *Dr Dobb's Journal-Software Tools for the Professional Programmer*, vol. 19, no. 1, pp. 40–49, 1994.

[17] J. Carreira, H. Madeira, J. G. Silva *et al.*, "Xception: Software fault injection and monitoring in processor functional units," *Dependable Computing and Fault Tolerant Systems*, vol. 10, pp. 245–266, 1998.

[18] Intel, *Pentium Processor Family Developer's Manual Volume 3: Architecture and Programming Manual*, Mt. Prospect, IL, USA., 1995.

[19] I. Intel, "Intel 64 and ia-32 architectures software developers manual," *Volume 3[a]*, 2013.

[20] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *Proc. Department of Defense HPCMP Users Group Conference*, 1999.

[21] "Intel Performance Counter Monitor - A better way to measure CPU utilization," http://software.intel.com/enus/articles/intel-performance-counter-monitor/.

[22] D. Heller, "Rabbit: A performance counters library for intel/amd processors and linux," *Scalable Computing Laboratory, Ames Laboratory, USDOE, Iowa State University. http://www. scl. ameslab. gov/Projects/Rabbit*, 2000.

[23] R. Berrendorf and H. Ziegler, *Pcl-the performance counter library: A common interface to access hardware performance counters on microprocessors*. FZJ-ZAM, 1998.

[24] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *International Journal of High Performance Computing Applications*, vol. 14, no. 3, pp. 189–204, 2000.

[25] K. London, S. Moore, P. Mucci, K. Seymour, and R. Luczak, "The papi cross-platform interface to hardware performance counters," in *Department of Defense Users Group Conference Proceedings*, 2001, pp. 18–21.

[26] "The eclipse foundation, eclipse.org home," http://www.eclipse.org/.

[27] "Photran, an Integrated Development Environment and Refactoring Tool for Fortran," http://www.eclipse.org/photran/.

[28] E. Gamma and K. Beck, *Contributing to Eclipse: principles, patterns, and plug-ins*. Addison-Wesley Professional, 2004.

[29] E. Clayberg and D. Rubel, *Eclipse: building commercial-quality plug-ins*. Addison-Wesley Reading, 2004, vol. 2.

[30] J. L. Overbey and R. E. Johnson, "Generating rewritable abstract syntax trees," in *Software Language Engineering*. Springer, 2009, pp. 114–133.