Enabling Data Structure Oriented Performance Analysis with Hardware Performance Counter Support[*]

Karl Fürlinger[1,2], Dan Terpstra[2], Haihang You[2], Phil Mucci[2], and Shirley Moore[2]

[1] Computer Science Division,
EECS Department,
University of California at Berkeley
`fuerling@EECS.Berkeley.EDU`

[2] Innovative Computing Laboratory,
Department of Electrical Engineering and Computer Science,
University of Tennessee
{`karl,terpstra,you,mucci,shirley`}`@eecs.utk.edu`

Abstract. An interesting and as of yet under-represented aspect of program development and optimization are data structures. Instead of analyzing data with respect to code regions, the objective here is to see how performance metrics are related to data structures. With the advanced performance monitoring unit of Intel's Itanium processor series such an analysis becomes possible. This paper describes how the hardware features of the Itanium 2 processor are exploited by the perfmon and PAPI performance monitoring APIs and how PAPI's support for address range restrictions has been integrated into an existing profiling tool to achieve the goal of data structure oriented profiling in the context of OpenMP applications.

1 Introduction

Parallel performance analysis is traditionally the domain of scientific application developers. With the increasingly widespread adoption of multicore CPUs, the number of developers that have to optimize their applications for parallel performance can be expected to increase significantly.

The most common approaches for performance analysis are profiling and tracing, the former often being preferred due to its lower overheads and more easily comprehensible results and the latter finding most use for message passing applications. Most current tools, profiling as well as tracing, deliver their data correlated to the program source code, e.g., percentage of execution time spent in a particular function, number of bytes transferred in a particular MPI call, or cache misses incurred due to a particular program statement.

[*] This work was partially supported by US DOE SCIDAC grant #DE-FC02-06ER25761 (PERI) and NSF grant #07075433 (SDCI).

An aspect of program development that has received less attention, mostly due to limited hardware support, is the dimension of data structures. The objective here is to see how performance metrics are related to data structures in addition to source code regions. With the advanced performance monitoring unit of Intel's Itanium processor series such an analysis becomes possible.

This paper describes how the hardware features of the Itanium 2 processor are exploited by the perfmon and PAPI performance monitoring APIs and how PAPI's support for address range restrictions has been integrated into an existing profiling tool to achieve the goal of data structure oriented profiling in the context of shared memory applications.

The rest of this paper is organized as follows: In Sect. 2 we describe the basic hardware capabilities of the Itanium 2 processor and how they are made available through PAPI. We then describe how we integrated the address range restriction capabilities into an existing profiling tool for OpenMP and made them available to the user. In Sect. 3 we demonstrate the application of the extended tool to a simple example application. In Sect. 4 we describe related work in the area and in Sects. 5 and 6 we describe our plans for continuing our work and conclude, respectively.

2 Data Structure Oriented Profiling

This section describes the hardware capabilities for data address range restricted monitoring and how they are made available through the layers of perfmon2 and PAPI in a profiling tool for OpenMP applications. perfmon2 is a generic low-level interface to the Performance Monitoring Unit of modern microprocessors which is currently implemented for Itanium, IA-32, x86-64, and PPC64 architectures [9]. PAPI is a cross-platform interface to the hardware counters supported by the performance monitoring unit (PMU) that includes portable routines as well as a standard set of performance metrics [1]. PAPI is layered on top of perfmon2 on perfmon2-supported architectures.

2.1 Hardware Capabilities and Their Availability through Perfmon and PAPI

Event counting on the Itanium 2 processor can be qualified by a number of conditions, including instruction address, opcode matching, and data address ranges. Of the roughly 475 native events available on the Itanium 2, 160 of them are memory related and can be counted with data address specification in place. In addition to data address restrictions, the Itanium 2 supports restrictions with respect to instruction addresses, which is however not discussed in this paper.

To specify the data address range qualification, four pairs of special registers are available. The starting and ending addresses cannot be specified exactly, since the hardware representation relies on powers-of-two bitmasks. The perfmon library used by PAPI tries to optimize the alignment of these power-of-two regions to cover the addresses requested as effectively as possible with the four sets of registers available. Perfmon first finds the largest power-of-two address region completely contained within the requested addresses. Then it finds successively smaller power-of-two regions to cover the errors on the high and low end of the requested address range. The effective result is that the actual range specified is always equal to or larger than and completely contains the requested range, and can occupy from one to four pairs of address registers. In some cases this can result in significant overcounts of the events of interest, especially if two active data structures are located in close proximity to each other. This may require that the developer insert some padding structures before and/or after a particular structure of interest to guarantee accurate counts (although the padding may introduce additional perturbations).

The PAPI team has implemented a generalized interface for data structure and instruction range performance instrumentation beginning with the PAPI 3.5 release. Since PAPI is a platform-independent library, care must be taken when extending its feature set so as not to disrupt the existing interface or clutter the API with calls to functionality that is not available on a large subset of the supported platforms. To that end, the PAPI developers elected to extend an existing PAPI call, `PAPI_set_opt()`, with the capability of specifying starting and ending addresses of data structures or instructions to be instrumented. The `PAPI_set_opt()` call previously supported functionality to set a variety of optional capability in the PAPI interface, including debug levels, multiplexing of eventsets, and the scope of counting domains. This call was extended with two new cases to support instruction and data address range specification: `PAPI_INSTR_ADDRESS` and `PAPI_DATA_ADDRESS`. To access these options, a user initializes a simple option specific data structure and calls `PAPI_set_opt()` as illustrated in the code fragment below:

```
...
option.addr.eventset = EventSet;
option.addr.start = (caddr_t)array;
option.addr.end = (caddr_t)(array + size_array);
retval = PAPI_set_opt(PAPI_DATA_ADDRESS, &option);
...
```

The user creates a PAPI eventset and determines the starting and ending addresses of the data to be monitored. The call to `PAPI_set_opt()` then prepares the interface to count events that occur on accesses to data in that range. The specific events to be monitored can be added to the eventset either before or after the data range is specified.

It is important that the user has some way to know what approximations have been made, so that appropriate corrective action can be taken. For instance, to isolate a specific data structure completely, it may be necessary to pad memory before and after the structure with dummy structures that are never accessed. To facilitate this, `PAPI_set_opt()` returns the offsets from the requested starting and ending addresses as they were actually programmed into the hardware. If the addresses were mapped exactly, these values are zero.

2.2 Data Structure Monitoring Support in ompP

ompP is a profiling tool for OpenMP applications designed for Unix-like systems. ompP differs from other profiling tools like gprof [4] or OProfile [7] in primarily two ways. Firstly, ompP is a measurement based profiler and does not use program counter sampling. The instrumented application invokes ompP monitoring routines that enable a direct observation of program execution events (like entering or exiting a critical section). An advantage of the direct approach is that the results give exact counts, rather than sampled values, and hence they can even be used for correctness testing.

The second difference lies in the way of data collection and representation. While general profilers work on the level of functions, ompP collects and displays performance data in the user model of the execution of OpenMP events [5]. For example, the data reported for critical section contains not only the execution time but also lists the time to enter and exit the critical construct (enterT and exitT, respectively) as well as the accumulated time each thread spends inside the critical construct (bodyT) and the number of times each thread enters the construct (execC). An example profile for a critical section is given in Fig. 1.

```
R00002 main.c (20-23) (unnamed) CRITICAL
 TID      execT      execC      bodyT     enterT      exitT     L3_MISSES
   0       1.00          1       1.00       0.00       0.00       534 513
   1       3.01          1       1.00       2.00       0.00       534 733
   2       2.00          1       1.00       1.00       0.00       535 420
   3       4.01          1       1.00       3.01       0.00       535 062
 SUM      10.02          4       4.01       6.01       0.00     2 139 728
```

Fig. 1. Profiling data delivered by ompP for a critical section. execC denotes the execution count, enterT, exitT, bodyT, and execT are timing data in seconds for entering, exiting, executing the body of the critical section. execT is the sum of all other reported times.

Profiling data in a style similar to that shown in Fig. 1 for critical sections are delivered for each OpenMP construct, with the columns (execution times and counts) depending on the particular construct. Furthermore, ompP supports the query of hardware performance counters through PAPI [1] and the measured counter values appear as additional columns in the profiles. In addition to OpenMP constructs that are instrumented automatically using Opari [8], a user can mark arbitrary source code regions such as functions or program phases using a manual instrumentation mechanism. Function calls can be automatically instrumented with compilers that support this feature.

Profiling data are displayed by ompP as flat profiles and as callgraph profiles, giving both inclusive and exclusive times in the latter case. As an advanced productivity feature, ompP performs overhead analysis in which four well-defined overhead classes (synchronization, load imbalance, thread management, limited parallelism) are quantitatively evaluated.

To support data structure oriented profiling, mechanisms that allow the user to specify the address range of data structures as well as changes in the result reporting are required. Several methods for specifying the data structure to be monitored have been implemented in `ompP`. First, for global and statically allocated data structures, a data structure's symbol name can be supplied when invoking `ompP`. For static and global data structures the name can be associated with the correct virtual address start and size of the symbol by invoking the `nm` tool provided that the debug information is contained in the binaries. In addition to this automatic conversion from symbol names to address ranges, the user can manually specify name, data address start and range through environment variables. For example:

```
# export OMPP_DATA_ADDR=0x6000000000022f80
# export OMPP_DATA_SIZE=34000
# export OMPP_DATA_NAME=tilearray
# ./ompp myprogram
```

For dynamically allocated memory or stack variables, the above mechanisms are neither adequate nor convenient. For this reason, `ompP` also provides a programmatic way to select the address range, size and name to monitor. A developer can annotate the source code with direct calls to set up the monitoring. For example:

```
double vec[1234];
...
ompp_papi_set_drange(vec, sizeof(vec), "vec")
```

will set up monitoring for the `vec` array. Alternatively it is possible to use source code annotations in the form of pragmas (in C/C++) or special comments (FORTRAN) that are translated to `ompp_papi_set_drange()` calls by the Opari preprocessor, for example:

```
#pragma pomp inst data(vec, sizeof(vec), "vec")
```

The latter technique has the advantage that has no compile- or link time dependency on `ompP` remains because the annotations can just be ignored by compilers.

The data reporting side of `ompP` has been changed to account for the additional data available through the address range restriction. The header section of `ompP`'s profiling report lists the address range restrictions that are in effect and the offsets that occur due to imprecise coverage of the range with Itanium 2's four coarse mode counters as shown below. This information is important when interpreting the profiling reports to exclude the possibility to falsely attribute data to neighboring data structures.

```
Data Address    : 0x6000000000022f80
Data Size       : 34000 (0x84D0)
Data Name       : tilearray
Data Start Offs.: 0
Data End   Offs.: 0
```

For the bookkeeping of profiling data, ompP introduces execution overhead in the measured application that is directly proportional to the number of region enter or exit operations monitored. For reasonably "well behaved" applications, such as the SPEC OpenMP benchmarks, `ompP`'s monitoring overhead usually is below five percent of execution time with PAPI enabled. We could not observe additional overhead using Itanium's data address range restriction feature over using the counters without this feature enabled.

3 Experimental Evaluation

In this section we show a simple application example of the data structure monitoring capabilities of `ompP`. The program fragment shown in Fig. 2 implements a very simple matrix × vector multiplication $Ax = b$ and the purpose is to show how the memory system related events can be restricted to only those occurring for the matrix A as an example. Consider this program fragment in Fig. 2 and note that data range monitoring is set up for the `a` array with the proper size.

```
  integer n, i, j
  parameter( n=12000 )

  double precision a(n,n)
  double precision b(n)
  double precision c(n)
...
!$POMP INST DATA( a , n*n*8 , "a" )

!$OMP PARALLEL DO private(i,j)
  DO i = 1, n
    c(i) = 0.0;
    DO j = 1, n
      c(i)=c(i)+a(i,j)*b(j)
    END DO
  END DO
...
```

Fig. 2. Matrix × vector multiplication example for data address range restrictions

The execution of this application on a 4-way Itanium 2 ("Madison Processor") SMP machine with 1.3 GHz, 3 MB third level cache and 8 GB of main memory resulted in a profiling report that shows the address range specification in place:

```
Data Address     : 0x6000000000026f40
Data Size        : 1152000000  (0x6DDD000)
Data Name        : a
Data Start Offs.: 159552  (0x26f40)
Data End   Offs.: 55800000  (0x35370c0)
```

Note the extra covering of the array due to imprecise mode. After padding the array at the beginning and at the end (with a similar sized pad1 and pad2 array, in this case) and setting counters to measure load instructions (PAPI_LD_INS) we get the profile shown below for the main parallel loop. Note that these numbers are exactly what one would expect to see: Each of the four threads works on a sub-matrix of size $3\,000 \times 12\,000 = 36\,000\,000$ and has to bring this into registers. Also note that the loads for the other data structures are not included.

```
R00002 main.f (30-37) LOOP
 TID       execT       execC       bodyT   exitBarT      PAPI_LD_INS
   0        4.66           1        4.66       0.00       36 000 000
   1        5.24           1        5.24       0.00       36 000 000
   2        5.23           1        5.23       0.00       36 000 000
   3        4.91           1        4.91       0.00       36 000 000
 SUM       20.04           4       20.04       0.00      144 000 000
```

Next we analyzed the memory system performance of the code by measuring the number of level three misses, as shown in the following profile:

```
R00002 main.f (30-37) LOOP
 TID       execT       execC       bodyT   exitBarT        L3_MISSES
   0        5.01           1        5.01       0.00       33 735 611
   1        5.24           1        5.24       0.00       35 447 540
   2        5.22           1        5.22       0.00       35 470 957
   3        5.14           1        5.14       0.00       35 004 338
 SUM       20.61           4       20.61       0.00      139 658 446
```

Evidently, a very high percentage of the loads fail the L3 cache and have to go to memory. Looking at the source code of the multiplication operation the reason is obvious. The array is accessed with a stride of the matrix dimension and not linearly, leading to very poor cache reuse. Changing the two indexes of the matrix multiplication we arrive at a version with much better cache performance: Also note the improved execution time (0.64 seconds vs. 5.20 seconds).

```
R00002 main.f (30-37) LOOP
 TID       execT       execC       bodyT   exitBarT        L3_MISSES
   0        0.64           1        0.64       0.00        2 250 582
   1        0.64           1        0.64       0.00        2 250 741
   2        0.64           1        0.64       0.00        2 250 734
   3        0.64           1        0.64       0.00        2 250 730
 SUM        2.54           4        2.54       0.00        9 002 787
```

While the shown example is trivial, it demonstrates the main benefit of the data address range restriction: contributions of individual elements can be singled out from the overall summed hardware counter values measured and hence another dimension in which performance analysis can be conducted (that along different data structures) opens up. If, for example, the same matrix `a` was accessed in several routines, or parallel loops, a restriction to `a` would show if it is accessed with similar efficiency in all routines. A wrong indexing in one place would stick out in comparison to the other cases, something that might go unnoticed in the wealth of counter data produced by the overall routine, if the restriction to the array was not in place.

4 Related Work

The value of analyzing programs with a focus on data structures has been recognized before. There are proposals [6,2] for new or extended hardware monitoring units that allow a more detailed analysis of cache behavior of data structures such as uncovering the reason for eviction of cache lines.

With respect to actual hardware, the Itanium 2 processor is the only microprocessor available today that allows data structure oriented profiling to be used in practice. Although the `dprof` tool [10,3] exploits these capabilities with goals similar to ours, the techniques used are different. `dprof` uses the Itanium's EARs (event address registers) that can sample load operations, their code and data addresses and latency. This allows basically all data structures to be monitored simultaneously. The association between virtual addresses and data structures is accomplished by using the executable's debug information and by capturing `malloc()` calls. In comparison, our technique can only monitor one data structure at a time, but the results obtained are exact and not subject to sampling inaccuracy. Also, the EARs only monitor load operations and the association to which level of the memory hierarchy satisfied the load has to be done by analyzing the latency, while our technique allows us to monitor all events related to the memory subsystem.

5 Future Work

We plan to explore the data address oriented profiling approach along several directions in the future. First, to overcome the limitation of being able to monitor only one address range at a time, we plan to combine the current direct measurement technique with sampling in the following way: The developer will be enabled to specify several data structures to be monitored and those will be kept in a list. In addition, a PAPI overflow event can be set up (either based on the elapsed CPU cycles or on the memory-system related event to be monitored) and on each overflow the monitoring switches to another data structure. A comparison with the EAR-based approach as realized in `dprof` will show which of the two approaches delivers better results in terms of accuracy and ease of use.

A second direction in which a more detailed investigation is warranted is the area of the interaction of threads and data structure. In the current implementation, the same address range restriction is set up for each thread in `ompP`, while a separate event set (and range) is supported by PAPI. With globally allocated data structures shared across threads this is not a problem, but separate data structures per thread are needed to fully support analysis of privatized arrays in OpenMP.

We plan to validate our data-structured oriented approach to performance analysis by using it with applications having known problems with data access. This investigation will compare the ease of our approach with the labor-intensive manual instrumentation, modeling, and analysis that has been used to find and fix these problems. Following this comparison, we plan to use our approach to detect similar problems in large-scale applications.

6 Conclusion

We have discussed the hardware capabilities of the Intel Itanium 2 processor for data address range restrictions and how those capabilities, which are available through the PAPI interface, are used in a profiling tool for OpenMP applications for data structure oriented profiling. From the user's perspective, a central issue is the ability to specify the data items to be measured which we have addressed by offering code markup methods as well as runtime specification through environment variables. An appealing property is the exactness of the delivered results – for load and store operations, for example, the measured data usually fit exactly the expected results. Also, all memory system related events (160) are available for measurement.

There are also some shortcomings with the data address range restrictions as presented here. The most severe issue is the inexact coverage of the data address range due to hardware restrictions which leads to offsets at the beginning and end of the covered range. Padding is needed to move the monitored data structure into an area surrounded by inactive regions of memory that will not disturb the measurement. Naturally, such a padding could have adverse effects on the application's performance and defeat the purpose of discovering if and which padding is required for optimization purposes. To circumvent this issue we plan to investigate a mode where the range is not selected to be minimally including, but maximally fitting the requested range. That would solve problems of wrong attribution at the expense of less accurate results.

References

1. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.J.: A portable programming interface for performance evaluation on modern processors. Int. J. High Perform. Comput. Appl. 14(3), 189–204 (2000)
2. Buck, B.R., Hollingsworth, J.K.: A new hardware monitor design to measure data structure-specific cache eviction information. International Journal of High Performance Computing Applications 20(3), 353–363 (2006)
3. Gaugler, T.: Ein Werkzeug zur Untersuchung des Cacheverhaltens von Datenstrukturen mittels Ereigniszählern. Diplomarbeit, Universität Karlsruhe (2005)
4. Graham, S.L., Kessler, P.B., McKusick, M.K.: Gprof: A call graph execution profiler. SIGPLAN Not. 17(6), 120–126 (1982)
5. Itzkowitz, M., Mazurov, O., Copty, N., Lin, Y.: An OpenMP runtime API for profiling. The OpenMP ARB as an official ARB White Paper (accepted), `http://www.compunity.org/futures/omp-api.html`

6. Kereku, E., Li, T., Gerndt, M., Weidendorfer, J.: A data structure oriented monitoring environment for fortran openMP programs. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, pp. 133–140. Springer, Heidelberg (2004)
7. Levon, J.: OProfile, A system-wide profiler for Linux systems, `http://oprofile.sourceforge.net`
8. Mohr, B., Malony, A.D., Shende, S.S., Wolf, F.: Towards a performance tool interface for OpenMP: An approach based on directive rewriting. In: Proceedings of the Third Workshop on OpenMP, EWOMP 2001 (September 2001)
9. Perfmon2, the hardware-based performance monitoring interface for linux, `http://perfmon2.sourceforge.net/`
10. Tao, J., Gaugler, T., Karl, W.: A profiling tool for detecting cachecritical data structures. In: Kermarrec, A.-M., Bougé, L., Priol, T. (eds.) Euro-Par 2007. LNCS, vol. 4641, pp. 52–61. Springer, Heidelberg (2007)