CCLRC

Technical Report
RAL-TR-2004-034

Use of Performance Toolkits to analyse a real performance problem

J.V. Ashby

09 December 2004

Council for the Central Laboratory of the Research Councils

Library and Information Services
CCLRC Rutherford Appleton Laboratory
Chilton Didcot
Oxfordshire OX11 0QX
UK
Tel: +44 (0)1235 445384
Fax: +44 (0)1235 446403
Email: library@rl.ac.uk

CCLRC reports are available online at:
http://www.clrc.ac.uk/Activity/ACTIVITY=Publications;SECTION=225;

ISSN 1358-6254

Use of Performance Analysis Toolkits to analyse a real performance problem

John Ashby
Computational Science and Engineering Department
Rutherford Appleton Laboratory
Chilton, Didcot, Oxon OX11 0QX
J.V.Ashby@rl.ac.uk

Abstract

We consider the use of two Hardware Performance Monitoring tools in understanding a performance problem with a computational science code. The tools are assessed for the quality and usability of the information they present. While the underlying problem is not fully explained, sufficient understanding is attained to achieve a partial remedy.

1. INTRODUCTION

Modern high performance computers have complex architectures, with multiple registers, Floating Point Units, cache levels, etc. They frequently employ techniques such as speculative execution and pre-fetching to keep processor pipelines busy. Often they will provide a toolkit which is capable of monitoring the performance of a program at a detailed level. Such a toolkit, often initially developed as a debugging aid at the hardware design stage, enables a programmer to determine such measures as the number of instructions processed, memory accesses from the various levels of cache, activity of the Floating Point Units, etc.

The aim of this report is to investigate the use of performance analysis toolkits in analysing a specific performance problem, namely the discrepancy between the runtimes for a code which uses statically declared arrays and the same code using dynamically allocated arrays. The objectives of this are first to understand where this discrepancy comes from, but equally to evaluate how useful the toolkits are in providing the information that will feed this understanding. This latter part, of course, has two aspects: is the information available and how easy is it to obtain?

Two such toolkits are HPMToolKit on the IBM Regatta machine, HPCx, and SpeedShop for the SGI Origin3000. Each allows in varying degree investigation of the very low-level activity of a program. Since the code exhibits the same problem on both HPCx and Green, the Origin3000 at Manchester, we shall use both and attempt to compare the tools.

2. THE PCHAN BENCHMARK CODE

PCHAN is a CFD program which solves the Navier-Stokes' equations on a topologically regular grid in three dimensions. It employs several large arrays which hold solution variables and their derivatives, as well as various physical quantities related to the domain. As the code is written in Fortran90 it would be desirable to make use of the language feature that allows the dynamic allocation of memory to create these arrays, thus allowing them to be created to just the size required for a particular run, rather than the old Fortran77 way of declaring arrays statically at the largest imaginable size. In Pchan a third course is taken, the code is re-compiled to the appropriate size for each different problem. However, it has been noticed that using dynamic allocation slows down the program considerably, sometimes by a factor a great as 11. Why this should be is difficult to understand from a naïve point of view; memory is memory, whether it is allocated statically or dynamically, and the code should do exactly the same computations on the data.

3. HPMTOOLKIT

IBM's HPMToolKit can be used in two basic modes. In the first an existing, unmodified code is monitored by running it using hpmcount. The -g argument to hpmcount determines which statistics are collected, there is a choice of 61 sets of 8 primary statistics with various derived measures, but duplication means that there are 234 rather than 488 primary statistics (for example the processor cycle count turns up in many sets). The most commonly useful sets are 60, which concentrates on the activity of the Floating Point Units, 5 which gives statistics on memory loads from the outer levels of cache and 56 which deals with level 1 cache access.

In the second mode, HPMToolKit provides a set of user-callable routines which can be inserted into a program to instrument it at a level of granularity chosen by the user.

4. SPEEDSHOP

SpeedShop from SGI similarly provides a coarse level set of tools to be run from a shell and applied to a whole program, and a library of user-callable routines which can be embedded in a code. The data that can be gathered by either of these two modes is more restricted than with HPMToolKit, but this has the advantage that it is more accessible and packaged in a form more digestible to the application programmer. In SpeedShop rather than specifying a set of data (from a choice of 61 such sets) to collect, there are 29 experiments, some of which can be run using more or less frequent sampling or with larger counters. Typical experiments are: usertime which measures cpu time, fpe which collects data on floating point exceptions and a variety of hwc (hardware counter) experiments which sample where in the code the Program Counter is when a particular counter pverflows. For example the dc_hwctime experiment works by incrementing a counter whenever a primary data cache miss occurs. Then every 8009 misses SpeedShop looks to see which part of the program is executing and increments a counter for that block of code. At the end of the run a report can be generated which gives how frequently the Program Counter was found in a partucular routine.

More details of how to use both HPMToolKit and SpeedShop are given in subsequent sections and in references 1-3.

5. HPMTOOLKIT EXPERIMENTS

Initial thoughts on the possible causes of the different speeds centred around compiler optimisations, on the grounds that some automatic optimisation might not be possible in the dynamic case (c.f. the argument that C programs are inherently less optimisable than Fortran(77) ones as the compiler is unable to make assumptions that two pointers in C refer to different locations in memory that can be made about two Fortran variables). However it was found that the difference occurred at all levels of optimisation, even with no optimisation, which effectively ruled out this hypothesis.

To investigate the possible causes further the program was instrumented by inserting calls to a routine which timed blocks of code. This was a relatively crude timer developed for Fortran77 programs many years ago, but for the purposes of this experiment it was adequate. The results show that the bulk of the time used by the program is spent in a routine called `rhs`, and that within that routine the loops which access and update sections of the large rank 4 arrays are the most discrepant, the statically allocated version of the code spending much more time in those loops than the dynamically allocated version. However, this still does not explain why there was such a difference between the static and dynamic versions of these arrays.

The routine in question is at the heart of the PCHAN code. PCHAN is solving the 3d compressible Navier Stokes equations on a logically regular (though spatially irregular) grid using a finite difference scheme with explicit multistep Runge-Kutta time stepping. The grid is distributed between multiple processors. The `rhs` routine calculates the contributions to the right-hand side for the time stepping process and updates at grid points within a processor. Data is shared between processors by calls from the main program to which `rhs` returns.

To look more closely at what differences in execution there might be that would lead to different run-times a non-optimised serial version for a small test problem was built on HPCx. Although PCHAN is a parallel code, the behaviour under investigation is not thought to arise from any parallel interactions such as communications bottlenecks. Thus a small serial case that could be run rapidly and repeatedly was sufficient. This proved to be the case; for one pair of runs the statically allocated executable took 1.490s of cpu time, the dynamic 2.040s. This meant that even running these jobs using the batch queues gave turn-around of only a minute or so.

The first investigation undertaken was to use `hpmcount` to look at various performance measures for both static and dynamic codes. `Hpmcount` is a tool which produces statistics for the overall execution of an uninstrumented program. It supports 61 sets of data, of which three, sets 60, 5 and 56, are thought to be most useful. The first set chosen was set 60, which is the default. This looks at the activity of the Floating Point Units.

Statistic (set 60)	*Static program*	*Dynamic Program*
FPU Divides	3155121	3154826
FPU Fused Multiply Add	81842835	81840228
FPU0 Result	136448327	129283390
FPU1 Result	146338504	152964074
Processor Cycles	1962129510	2679838713
FPU Store Instruction	62971390	62912862
Instructions completed	2204912990	3028609316
LSU Floating Point Load	301954800	302070469

As can be seen, most of these statistics are very similar (again, naively one would expect many of them to be identical; the same program acting on the same data should produce exactly the same number of divides and multiply-adds, for example. That it doesn't suggests that there is a degree of speculative execution which differs slightly between the two versions). For the most part the differences are insignificant. The different apportioning of operations between the two Floating Point Units is probably because the units are having to idle while waiting for data. Clearly the number of processor cycles is different since this is precisely the cpu-time discrepancy under investigation. The different (by about 40%) number of instructions completed is interesting, and again is probably due to speculative execution. It is interesting that this is not reflected in the FPU statistics, so may warrant further investigation.

Turning to set 5 which deals with data loads from level 2 and level three caches we find the following:

Statistic (set 5)	*Static program*	*Dynamic Program*
Data load from L3	1754460	1917725
Data load from memory	66080	48016
Data load from L3.5	0	0
Data load from L2	9977907	12067761
Data load from L2.5 shared	18	46
Data load from L2.75 shared	0	0
Data load from L2.75 modified	0	0
Data load from L2.5 modified	29	28

To understand the levels of cache referred to here requires a knowledge of the machine architecture. Each processor (CPU) has 64kb of level one cache. There are two processors on a chip and these share 1440kb of level 2 cache. Four chips (thus eight processors) are packaged together in a multi-Chip module (MCM) and they share 128Mb of level 3 cache. It is possible (in a multi processor program) for a process to access level 3 cache on a different MCM from its own. This is referred to as level 3.5 cache. Since we have a serial program here it is a relief that no L3.5 references are made. Similarly a process can load from L2 on a different MCM (L2.75) or on a different chip on the same MCM (L2.5). We see a few L2.5 loads here which is odd, given the serial program, but these probably represent system activities, perhaps as the program is migrated around the CPUs available to the serial batch queue. Otherwise there are more L3 and L2 loads for the dynamic code that for the static, but fewer from memory, and so the overall effect is probably neutral in terms of time. These discrepancies may be worth pursuing.

The final set of those recommended by the HPCx team is set 56 which covers level 1 cache and Translation lookaside buffer (TLB, a table which maps effective to real memory addresses) misses:

Statistic (set 56)	*Static program*	*Dynamic Program*
Data TLB misses	62712	94979
Instruction TLB misses	1113	1353
L1 D cache load misses	7880247	10566145
L1 D cache store misses	24172886	23600076
Processor cycles	1964229969	2668406206
Instructions completed	2204912080	3028609423
L1 D cache store references	156168768	150600749
L1 D cache load references	648202041	1208903357

Three discrepant measures immediately spring out of this experiment: Data TLB misses, L1 cache load misses and references. (The processor cycles and instructions completed are increased, but as before this is an artefact of the longer processing time. Many of the completed instructions may be no-operation instructions while the processor waits for the data identified by a TLB miss to be fetched.) These clearly warrant further investigation.

6. SPEEDSHOP EXPERIMENTS

Similar measures are available from SpeedShop on the SGI Origin. However, only one counter can be examined at a time, for example the floating point instruction counter is checked using the `gfp_hwc` (graduated floating point hardware counter) experiment. Data is accumulated each time a counter overflows a value, by default 32771, and a note is made of where in the program the overflow occurs. In this way it is possible to build up a picture routine by routine.

The results of a `gfp_hwc` experiment are shown below:

Counts(static)	*%(static)*	*%(static)*	*Counts(dynamic)*	*%(dynamic)*
rhs	62035503	29.8	69441749	31.6
ent_euler	35917016	17.2	40013391	18.2
d1eta	23070784	11.1	23267410	10.6
d1xi	17204775	8.3	17172004	7.8
d1z	16385500	7.9	16451042	7.5
d2eta	5767696	2.8	5636612	2.6
pdns3d	4751795	2.3	6980223	3.2
d2z	4686253	2.3	4620711	2.1
d2xi	4686253	2.3	4784566	2.2
init_tch	393252	0.2	426023	0.2
calcdt	327710	0.2	360481	0.2
deta	262168	0.1	229397	0.1
dxi	163855	0.1	163855	0.1
statacc	98313	0	98313	0

As with the IBM, the floating point counts are very similar for the two programs, and any discrepancies are most likely to be the result of speculative execution.

The SGI architecture has only two levels of cache rather than the three of the IBM. The data cache miss counter is examined using the `dc_hwc` experiment.

Counts(static)	*%(static)*	*%(static)*	*Counts(dynamic)*	*%(dynamic)*
rhs	21429214	43.8	29975853	50.3
ent_euler	6951458	14.2	8207894	13.8
d1xi	6091251	12.5	6479268	10.9
d1eta	4799914	9.8	5216673	8.8
d1z	3812421	7.8	3611227	6.1
d2xi	1689619	3.5	1714255	2.9
d2eta	1451471	3	1609552	2.7
d2z	1135309	2.3	1315973	2.2
pdns3d	1030606	2.1	808882	1.4
period_x	186823	0.4	188876	0.3

Counts(static)	*%(static)*	*%(static)*	*Counts(dynamic)*	*%(dynamic)*
period_z	141657	0.3	133445	0.2
bc_ch	45166	0.1	41060	0.1
init_tch	28742	0.1	39007	0.1
calcdt	20530	0	18477	0
pad_comp_bump	16424	0	22583	0

For the most part these results are consistent between the two programs with the exception of the greatly increased (30%) number of primary data cache misses in rhs when dynamic memory is used. Again this is consistent with the results on the IBM, suggesting a similar mechanism is slowing down the dynamic memory version on both machines.

As a check, on the IBM the secondary data cache statistics were similar for the two versions. On the SGI secondary data cache misses can be counted using the `dsc_hwc` experiment.

Counts(static)	*%(static)*	*%(static)*	*Counts(dynamic)*	*%(dynamic)*
rhs	2395466	61	2736197	63.6
ent_euler	1110356	28.3	1073807	25
pdns3d	153532	3.9	149602	3.5
d1xi	80565	2.1	92879	2.2
d1eta	38383	1	58819	1.4
d2z	26331	0.7	47553	1.1
d1eta	25807	0.7	52007	1.2
d2z	5502	0.1	6026	0.1
d2eta	5240	0.1	2751	0.1
calcdt	2096	0.1	2227	0.1

The results are broadly similar, certainly as similar as those for the IBM..

Finally the IBM experiments pointed strongly at the translation lookaside buffer misses as a large factor on slowing down the dynamically allocated code. This has a SpeedShop experiment also, `tlb_hwc`.

Counts(static)	*%(static)*	*%(static)*	*Counts(dynamic)*	*%(dynamic)*
ent_euler	40606	50.5	38293	7
rhs	30840	38.3	499608	91.8
pdns3d	2056	2.6	1028	0.2
d2xi	1285	1.6	514	0.1
d1xi	1285	1.6	1028	0.2
d1eta	1285	1.6	514	0.1

Here the difference between the two codes is strong, much stronger than on the IBM and clearly localised to the `rhs` routine, the same routine where much of the extra time was being spent during

the early timing experiments.

To probe further we need to instrument the code at a lower level to see if we can identify lines of source which might be giving rise to the differences.

With SpeedShop this involves placing calls to a routine `ssrt_caliper_point` through the code. This routine takes two arguments, 1 (or .TRUE.) and an identifying string. The compiled code must be linked against `libss`. Then the experiment is run using `ssrun` as before. Results are viewed using `prof -caliper` [n1,]n2 which analyses the segment of program execution between the n1-th and the n2-th caliper point. In addition -lines will expand the analysis on a line-by-line basis. For our experiment we chose to put a caliper point at the start of the `rhs` routine. This routine is called 15 times during the execution phase and so the caliper point appears as points 1 to 15. The salient output (keeping only those lines contributing more than 2% of the total number of TLB misses) from prof -caliper 1,15 -lines is, for the static case:

```
-----------------------------------------------------------------------
Line list, in descending order by function-time and then line number
-----------------------------------------------------------------------
    counts   %  cum.%  samples  function (dso: file, line)

     6413  8.2  10.4      121  ent_euler (TEST-STATIC.x: ent_euler.f, 128)
     2597  3.3  13.7       49  ent_euler (TEST-STATIC.x: ent_euler.f, 131)
     8692 11.1  31.1      164  ent_euler (TEST-STATIC.x: ent_euler.f, 156)
    11448 14.6  48.8      216  ent_euler (TEST-STATIC.x: ent_euler.f, 333)
```

The last two of these (which account for the major part of the TLB misses in this version) are both references to `Q(i,j,k,l)`, the large array which came under suspicion from the simple timing experiments.

For the dynamic case:

```
-----------------------------------------------------------------------
Line list, in descending order by function-time and then line number
-----------------------------------------------------------------------
    counts   %  cum.%  samples  function (dso: file, line)

     24592 4.4  11.6      464  rhs (TEST-DYNAMIC.x: rhs_3d.f, 955)
     18020 3.2  14.9      340  rhs (TEST-DYNAMIC.x: rhs_3d.f, 976)
     17225 3.1  22.0      325  rhs (TEST-DYNAMIC.x: rhs_3d.f, 1088)
     40121 7.2  33.9      757  rhs (TEST-DYNAMIC.x: rhs_3d.f, 1343)
     18815 3.4  38.4      355  rhs (TEST-DYNAMIC.x: rhs_3d.f, 1387)
     37842 6.8  45.1      714  rhs (TEST-DYNAMIC.x: rhs_3d.f, 1404)
     20405 3.7  50.8      385  rhs (TEST-DYNAMIC.x: rhs_3d.f, 1448)
     39379 7.0  57.8      743  rhs (TEST-DYNAMIC.x: rhs_3d.f, 1467)
     20617 3.7  62.4      389  rhs (TEST-DYNAMIC.x: rhs_3d.f, 1511)
     35245 6.3  68.7      665  rhs (TEST-DYNAMIC.x: rhs_3d.f, 1528)
     20988 3.8  74.4      396  rhs (TEST-DYNAMIC.x: rhs_3d.f, 1572)
     13621 2.4  76.8      257  rhs (TEST-DYNAMIC.x: rhs_3d.f, 1603)
     34132 6.1  86.6      644  rhs (TEST-DYNAMIC.x: rhs_3d.f, 1617)
     30369 5.4  92.0      573  rhs (TEST-DYNAMIC.x: rhs_3d.f, 1636)
```

Here we see that `Q(i,j,k,l)` is involved in almost all these lines of code. The odd one out, though, is line 1603 which involves another rank 4 array, `WX`. This may be a clue. Consider line 955:

```
          q(i,j,k,2)=q(i,j,k,2)
     &    +dx_dxi(i,j)*(wx(i,j,k,19)*((wx(i,j,k,11)*y_etaodet(i,j)
     &                                 -wx(i,j,k,12)*y_xiodet(i,j))
     &                               +(-wx(i,j,k,8)*x_etaodet(i,j)
     &                                 +wx(i,j,k,9)*x_xiodet(i,j)))
     &                  +wx(i,j,k,17)*((wx(i,j,k,30)*y_etaodet(i,j)
     &                                 +wx(i,j,k,11)*res2(i,j)
     &                                 -wx(i,j,k,28)*y_xiodet(i,j)
     &                                 +wx(i,j,k,12)*res6(i,j))
     &                               +(-wx(i,j,k,24)*x_etaodet(i,j)
     &                                 +wx(i,j,k,8)*res4(i,j)
     &                                 +wx(i,j,k,22)*x_xiodet(i,j)
     &                                 +wx(i,j,k,9)*res8(i,j))) )
```

and line 1303:

```
          dw1(i,j,k)=wx(i,j,k,20)*wx(i,j,k,2)
     &                +wx(i,j,k,17)*wx(i,j,k,10)
```

The similarity between the two lines is the large strides around the `WX` array, particularly in multiplying together two elements of the array, which means that the calculation cannot be reordered by the compiler to maximise data locality. Once again, though, the fundamental question is why is the computation less efficient when dynamic storage is used?

7. FURTHER HPMTOOLKIT EXPERIMENTS

Before investigating that further, though, we shall try to gain similar information from HpmToolkit. As with SpeedShop, this involves placing subroutine calls in the code and linking against two libraries. It is also necessary to include a header file in any file that contains calls to the library and pass the file through a preprocessor such as cpp. The four routines used in Fortran are: `f_hpminit` to initialise the library, `f_hpmterminate` to finish profiling, `f_hpmstart` which starts profiling on a block of code and `f_hpmstop` which ends a block of profiling. (The equivalent C routines are: `hpmInit, hpmTerminate, hpmStart` and `hpmStop`.) The process of instrumenting the `rhs` code is tedious as the routine consists of nearly 100 basic blocks (loops, etc.), each of which must be individually tagged.

Running the instrumented code produces a `*.viz` file and a file prefixed `perfhpm`. The latter is an ascii file which can be viewed in the usual ways and which contains similar information to that given by `hpmcount` for the whole program, but for each individual block of code. The `.viz` file is a binary file that can be read by a graphical visualisation tool, `hpmviz`. In a multi-process program, each process would produce its own `.viz` file and all of these could be visualised at once. This means that the performance measurements and metrics for a given code block can be displayed for all processes. However, is is not possible to display the measurements for all code blocks, which restricts the usefulness of the tool in this study. It is possible to run both the static and dynamic memory versions and to display the measurements for a particular block in each version. Given that we have established that the important measures to consider are TLB misses, L1 cache misses and references, we can restrict the visualisation tool to show these measures.

Rather than plough through all 98 code sections looking for discrepancies, we can concentrate on the sections identified by SpeedShop as of particular interest. Here the ability to compare static and dynamic code section is very useful. In the table below the code line number refers to the line identified by SpeedShop, the actual line number in the instrumented code is different.

Code line	*Dynamic*			*Static*		
	TLB misses	*L1 cache misses*	*L1 cache store*	*TLB misses*	*L1 cache misses*	*L1 cache store*
955	65	284588	363596	191	189105	348910
976	649	66311	372252	637	61081	370198
1088	233	63938	348347	318	61002	369845
1343	1256	276132	364454	1000	243318	367697
1387	1261	159226	363291	722	100991	367388
1404	153	232101	373558	267	195084	347308
1448	953	181083	348987	713	107205	349663
1467	914	347169	365134	1050	221982	372192
1511	758	258745	364033	467	222363	349893
1528	455	316624	373603	571	201802	350062
1572	905	256629	348703	985	226566	347621
1603	395	12656	570341	700	10131	580341
1617	782	270266	352375	695	186638	348694
1636	604	224108	363800	674	214766	347822

From this table it appears that the story is not as straightforward on the IBM as it appeared to be on the Origin. The L1 cache stores are comparable (to about 5%) according to these results and can be ignored. What emerges from a study of the TLB and cache misses is that for some sections of code the static code has fewer TLB misses than the dynamic code, while for other sections the opposite applies. Where the former is the case, the dynamic code has more L1 cache misses, and vice versa. There seems to be a trade-off between TLB and cache misses. There are typically 100 times fewer TLB misses than cache misses, but TLB misses are very expensive.

8. A SMALL TEST PROGRAM.

We have now gone about as far with the performance monitoring tools as possible. To examine further the problem we can use the information gathered so far to produce a small test program which will encapsulate the behaviour and, it is to be hoped, allow deeper exploration to be made. Our test program consists of a loop over i, j and k as in the original pchan code, with accesses to various elements of a rank 4 array, thus:

```
            wx(i,j,k,1) = 0.16667*(wx(i-1,j,k,2)
     &                              + wx(i+1,j,k,9)
     &                              + wx(i,j-1,k,17)
     &                              + wx(i,j+1,k,13)
     &                              + wx(i,j,k-1,33)
     &                              + wx(i,j,k+1,28))
```

Unlike in pchan the test problem uses a six-point finite difference star. The fourth index on the right-hand side is chosen to stride through the whole array as happens in pchan. The matrix is initialised randomly through Fortran90's RANDOM_NUMBER routine. This should not affect the reproducibility of results as the number of floating point operations is fixed. By using #DEFINES

and passing the program through cpp before compilation the WX array can be either statically or dynamically allocated. For a WX array of dimension (61,201, 61, 42) a linux pentium pc with the Lahey-Fujitsu compiler takes 8.32 cpu-seconds in static mode and 8.86 cpu-seconds with dynamic allocation. This is not as great a discrepancy as observed before, but it may well be that initialisation is a much larger relative component of this code than of pchan. The same code run on the IBM Regatta can only accommodate an array of dimension (46,151,46,42), and the times are 5.86s (static) and 6.96s (dynamic). For the larger array on the SGI Origin 300 the times are 22.58s (static) and 24.56s (dynamic).

If it is the case that the large strides through the array caused by the fourth index are responsible for the discrepancy, then it would be instructive to see if reducing those strides has an impact. One way would be to move the fourth index to the first position as in Fortran the first index is the most rapidly varying when stepping sequentially through memory. A neater solution which means little editing (and hence fewer possibilities of mistakes) is to declare a derived type for WX. For the dynamic memory case we have:

```
TYPE SOLVEC
  REAL*8, DIMENSION (42) :: SOLUTION
END TYPE SOLVEC
TYPE(SOLVEC), DIMENSION (:,:,:), ALLOCATABLE :: WX
```

```
wx(i,j,k)%solution(1) = 0.16667*(wx(i-1,j,k)%solution(2)
&                              + wx(i+1,j,k)%solution(9)
&                              + wx(i,j-1,k)%solution(17)
&                              + wx(i,j+1,k)%solution(13)
&                              + wx(i,j,k-1)%solution(33)
&                              + wx(i,j,k+1)%solution(28))
```

In this case the timings on the linux pc drop dramatically to 2.24s (static) and 2.52s (dynamic). On the IBM the reduction is less dramatic, but interestingly both static and dynamic take the same time, 5.12s. On the SGI the reduction is as good as on the pc, 6.28s (static) and 6.38s (dynamic). These results are presented in tabular form below.

Machine	***Static Memory***		***Dynamic Memory***	
	Array	***Derived Type***	***Array***	***Derived Type***
Linux PC	8.32	2.24	8.86	2.52
IBM Regatta	5.86	5.12	6.96	5.12
SGI Origin 300	22.58	6.28	24.56	6.38

Clearly, even if this is not the cause of the discrepancy between the speeds of dynamic and statically allocated codes, this rearrangement may be of great benefit to the pchan code generally.
In order to see if it is solely the memory strides that influence the speed, the test code has also been rewritten to use 42 individual rank 3 arrays. Declared (or allocated) sequentially, these should occupy the same or similar memory locations as the original rank 4 array. The results are instructive:

Machine	*Rank 4 (42 last)*		*Rank 3 Arrays*		*Derived types*		*Rank 4 (42 first)*	
	Static	*Dynamic*	*Static*	*Dynamic*	*Static*	*Dynamic*	*Static*	*Dynamic*
Linux PC	8.32	8.86	38.28	59.75	2.24	2.52	2.21	2.66
IBM Regatta	5.86	6.96	10.2	11.72	5.12	5.12	5.44	5.61
SGI Origin 300	22.58	24.56	71.52	145.08	6.28	6.38	7.32	8.63

The dynamic/static discrepancy is smaller than observed in the full code, but still present. The use of separate rank three arrays slows the code significantly on both the PC and the Origin, and in the latter case the static/dynamic discrepancy is greatly enhanced. The use of derived types (or of changing the array ordering which in memory placement terms is equivalent) does appear to give a large speed advantage on the PC and the Origin, and a slight advantage on the IBM. To investigate this further a version of the full PCHAN code was built using derived types for the three rank-4 arrays.

9. A NEW VERSION OF PCHAN

In the light of the results given above a new version of PCHAN was produced in which the three rank-4 arrays were replaced by rank 3 arrays of derived types, the members of which were rank 1 arrays of appropriate sizes. This approach was chosen in preference to changing the order of indices in the arrays as it was possible to automate a large part of the editing process using emacs. Where necessary the order of loops was also changed to ensure that memory accesses were as local as possible.

In one area of the code special routines had been provided for the IBM machine which used the rank 4 arrays in ways designed to make best use of cache. These routines were broken by the use of derived types, but fortunately the simple equivalents were still present in the code and were used for the experiments below.

Machine	*Original version*		*Derived Types Version*	
	Static	*Dynamic*	*Static*	*Dynamic*
IBM Regatta P690+				
SGI Origin 3000	1995.84	12272.08	4548.14	4513.59

Curiously the statically allocated verision with derived types goes slower than the original version, but the increased speed of the dynalmic version means that they are both running at equivalent speeds.

10. CONCLUSIONS

In this report we have used two Performance Monitoring tools, HPMToolKit and SpeedShop, to investigate the origin of the discrepancy between the speed achieved by a code using static memory allocation (at compile time) and dynamic allocation (at run time). The combined use of the two codes pointed to excessive Translation Lookaside Buffer misses in the dynamic case, centring round accesses of large rank 4 arrays. Using this information we were able to rewrite the code to achieve similar performance in both dynamic and static cases, though with a loss of speed in the static case.

Our other objective was to review the usability of the tools. Neither has the full functionality that would make their use particularly simple and effective: for example the ability to compare performance metrics for code blocks/procedures in SpeedShop rather than for the same block on different processes would be of much more value inlooking for hot spots in a code. Speedshop is perhaps more intuitive and provides better information for the application programmer, though HPMToolKit is more comprehensive and in skilled hands could be a powerful tool.

Neither tool is a magic bullet, and both require a relatively deep understanding of the underlying architecture and the ability to design and execute additional experiments (including writing small test programs) to fully understand and remedy performance problems.

11. REFERENCES

1. Using the Hardware Performance Monitor Toolkit on HPCx , Joachim Hein, http://www.hpcx.ac.uk/research/hpc/technical_reports/HPCxTR0307
2. Hardware Performance Monitor (HPM) Toolkit, Luiz DeRose, http://www.hpcx.ac.uk/support/documentation/IBMdocuments.HPM.html
3. http://www.sgi.com/products/software/irix/tools/prodev.tml