Appeared in *Proceedings of the 1994 International Conference on Parallel Processing*, St. Charles, Illinois, Vol. III, pp. 188-192.

A Hierarchical Approach to Modeling and Improving the Performance of Scientific Applications on the KSR1

Eric L. Boyd, Waqar Azeem, Hsien–Hsin Lee, Tien–Pao Shih, Shih–Hao Hung, and Edward S. Davidson
Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan

Abstract

We have developed a hierarchical performance bounding methodology that attempts to explain the performance of loop-dominated scientific applications on particular systems. The Kendall Square Research KSR1 is used as a running example. We model the throughput of key hardware units that are common bottlenecks in concurrent machines. The four units currently used are: memory port, floating-point, instruction issue, and a loop–carried dependence pseudo–unit. We propose a workload characterization, and derive upper bounds on the performance of specific machine-workload pairs. Comparing delivered performance with bounds focuses attention on areas for improvement and indicates how much improvement might be attainable.

We delineate a comprehensive approach to modeling and improving application performance on the KSR1. Application of this approach is being automated for the KSR1 with a series of tools including K–MA and K-MACSTAT (which enable the calculation of the MACS hierarchy of performance bounds), K–Trace (which allows parallel code to be instrumented to produce a memory reference trace), and K–Cache (which simulates inter-cache communications based on a memory reference trace).

1. Introduction

Computer scientists and engineers use performance evaluation as a tool to achieve several different goals. Computer architects are interested in understanding existing and proposed machines in order to improve the design of new machines. The developers of libraries, compilers, and operating systems focus on effective utilization of machine resources. Application developers optimize specific programs by understanding performance bottlenecks. End users may only be interested in choosing the fastest or most cost-effective machines and application packages.

An effective performance evaluation technique can provide insights for each of these groups. Many researchers have evaluated scientific computers by focusing on the expected performance. We believe that the best approach to improving performance for scientific applications is to bound the best achievable performance that a machine could deliver *on a particular code* and then try to approach this bound in delivered performance.

We present a technique for determining and approaching performance bounds for scientific loop-dominated codes, using the Livermore Fortran Kernels [1] as a running example to illustrate the method on the Kendall Square Research KSR1,[1] a shared virtual memory Massively Parallel Processor (MPP) with a Cache–Only Memory Architecture (COMA) [2] [3]. These bounds focus on the latency and bandwidth of specific machine components, particularly memory, instruction issue, and floating-point units, since these units are common bottlenecks.

The KSR1 is built as a group of ALLCACHE engines, connected in a fat tree hierarchy of rings. Up to 34 rings can be connected by a single second-level ring for a maximum configuration of 1088 processors. Each first–level ring has up to 32 processor nodes and up to two ALLCACHE directories. Although 256-processor systems have been built, all of our experiments in Section 4 used a single-ring 32-processor KSR1.

Each KSR1 node contains a 64-bit custom processor with a 20 MHz clock. The basic load/store RISC architecture is enhanced to allow a 2-instruction (VLIW format) issue per clock cycle: one address calculation, branch, or memory instruction and one integer or floating-point calculation instruction. Floating-point multiply-add triad instructions allow a peak performance rating of 40 MFLOPS.

Each node also has a D–subcache, I-subcache, and local cache. Each subcache is 64 sets, 2-way set associative, random replacement, 1 per cycle access rate, 2 cycle access time, with 2KB block (allocation unit) and 64B subblock (transaction unit). The local cache is 32MB, 128 sets, 16-way, LRU, 16KB page (allocation unit), 128B subpage (ring transaction unit). On subcache miss, average local cache access time was found to be 23.4 cycles (allocated block) or 49.2 (unallocated) or 150 to 180 (local cache miss, single ring transaction). [4] [5]

Two tools under development, *K–MA* and *K-MACSTAT*, will automatically generate the MACS performance bounds hierarchy for the KSR1 on loop–dominated applications. Interprocessor communication and cache simulation can be reconstructed by a simulation tool under development, *K–Cache*, using traces generated by a parallel application trace collection tool, *K–Trace*. Software pipelining of loop code is made possible through the use of a retargetable tool, the *Object Code Optimizer* (*OCO*) [6], targeted for the KSR1.

2. MACS Performance Bounds Hierarchy

The MACS machine-application performance bound methodology provides a series of upper bounds on the best achievable performance and has been used for a variety of loop-dominated applications on vector, superscalar and other architectures [7] [8] and extended to the bounds hierarchy used here in [9]. Four common bottleneck units (memory port, floating-point, instruction issue, and a loop-carried dependence pseudo-unit) are included in the KSR1 model to assess their individual workloads in the application and to examine how well the available parallelism among them is exploited [10]. The hierarchy of bounds equations is based on the peak floating–point performance of a Machine of interest (M), the Machine and a high level Application code of interest (MA), the

[1] The University of Michigan Center for Parallel Computing, site of the KSR1, is partially funded by NSF grant CDA-92-14296.

Appeared in *Proceedings of the 1994 International Conference on Parallel Processing*, St. Charles, Illinois, Vol. III, pp. 188-192.

Compiler-generated workload (MAC), and the actual compiler-generated Schedule for this workload (MACS), respectively.

The M bound for the KSR1 is 0.5 CPF (clocks per floating–point operation), assuming perfect combining of floating–point adds and multiplies into triad instructions and no other limitations on performance.

The MA lower bound on the run time of an application loop counts the essential operations for each selected function unit per inner loop iteration from the high level code of the application. The number of *essential* floating–point arithmetic operations is simply the number of floating–point operations (add, multiply, etc.), reduced by combining them into triads where possible. Counting only the essential memory operations requires inter-iteration dependence analysis. For m iterations of the inner loop, the number of distinct array elements that appear on the left hand side of the assignment statements will be of the form $am+b$. The number of *essential* store operations is defined to be a per iteration. The number of *essential* loads is counted similarly by examining the right side of each statement and counting the distinct array elements that appear on the right side before they appear on the left side of an assignment statement.

The MAC bound is similar to MA, except that it is computed using the actual operations produced by the compiler, rather than only the essential operations counted from the high level code. Thus MAC still assumes an ideal schedule, but does account for redundant and unnecessary operations inserted by the compiler as well as those that might be necessary, but not included in the MA count of essential operations. It removes one degree of freedom from the model by using an actual rather than an idealized workload.

The MACS bound, in addition to using the actual workload, removes another degree of freedom by using the actual schedule rather than an ideal schedule. However, it ignores cache miss stalls, interprocessor communication, and interrupts.

3. Gaps between Performance Bounds

In ascending through the bounds hierarchy from the M bound, the model becomes increasingly constrained as it moves in several steps from potentially deliverable toward actually delivered performance. This approach exposes and quantifies specific performance gaps, as shown in Figure 1, that are extremely useful for identifying bottlenecks in the machine and weaknesses in the compiler. We then individually evaluate, for example, the efficacy of the data flow analysis and the code scheduling phases of the compiler and identify their shortcomings. Restructuring techniques with the greatest potential performance gains can be selected according to which gaps are the largest and their causes. This approach can be implemented within a *goal–directed* compiler for general use.

Gap A (M –> MA) is caused by essential memory operations, issue limitations, loop–carried dependencies, and noncombinable floating–point operations. A large fraction of the avoidable performance loss shown by *gap A* can commonly be attributed to poor reuse of data in the high level application code. Excessive renaming and/or large data bandwidth between loops may introduce unnecessary (yet "essential") memory accesses. Reordering and fusing loops, and economical data structures, are common solutions to reducing *gap A*.

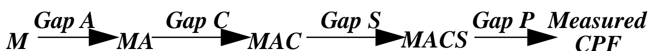

Figure 1: Gaps between performance bound models and measured time

Gap C (MA –> MAC) is caused by instruction set weakness, resource bandwidth limitations, and compiler inefficiencies. Typical factors leading to *gap C* on the KSR1 include redundant instructions (particularly redundant memory accesses), overhead for subroutine calls, and redundant base index registers. Subroutine inlining reduces save/restore overhead. Declaring data in Fortran *common* blocks promotes sharing of base index registers.

Gap S (MAC –> MACS) is caused by hardware and compiler scheduling inefficiencies. Loop unrolling can reduce the number of instructions per loop iteration by reusing registers and reducing overhead. The compiler typically achieves better scheduling with fewer *nop* operations by moving independent instructions of unrolled loop iterations into the slots where no operations existed previously. Typically the KSR1 Fortran compiler unrolls 2, 4, 8, or 16 iterations, limited heuristically by register set size and the size and complexity of the loop body. *Gap S* can be reduced with the help of the KSR1 version of OCO which employs a software pipelining technique known as polycyclic loop scheduling [10].

Gap P (MACS –> Measured CPF) results from subcache miss penalties and context switches. Cache simulation enables the visualization of data movement in the memory hierarchy. *Gap P* can be reduced by restructuring data reference patterns in order to maximize data reuse in the subcaches and local caches. Common techniques include loop blocking, loop fusion, domain decomposition, affinity regions, and prefetch and poststore instructions. *Gap P* becomes critical in parallel code, although uniprocessor performance is still important. Two tools discussed in Section 5 (K–Trace and K–Cache) will provide insights into the penalties for cache misses and communication that typically account for the majority of *gap P* in parallel applications. *Gaps A*, *C*, and *S* involve the performance of a single processor; *Gap P* involves intranode cache effects and internodal communication for parallel applications.

4. MACS Model for the KSR1

4.1. KSR1 MA, MAC, and MACS Bounds

The MA performance bound model for the KSR1, in clock cycles per inner loop iteration, t_l, is:

$$t_l = \mathrm{Max}\ (t_i,\ t_f,\ t_m,\ t_d) = \mathrm{Max}\ (t_i,\ t_d) \qquad (1)$$

where each term in the *Max* expression is the number of busy cycles per iteration in the corresponding unit, as defined below. The MA bound is computed in units of clocks per floating–point operation (CPF) by dividing t_l by the total number of essential floating–point operations per iteration, $TNF = f_a + f_m + 2*f_{ma}$.

The KSR1 processor can issue one instruction per clock cycle to either the floating–point (FPU) or integer (IPU) unit and one to either the CEU or XIU (load, store, address arithmetic, branch, and I/O instructions). In the bounds equation, f_{ma} counts the number of essential floating–point multiply–add triad operations, and f_m and f_a count the noncombinable multiply and add operations. Floating–point stores and linked triad instructions conflict on a register port (FPU{C}) which indirectly constrains instruction issue. Unrolling a loop k times reduces branch overhead by a factor of $(1/k)$. FPU/IPU branch overhead (x) typically includes a loop index decrement

Appeared in *Proceedings of the 1994 International Conference on Parallel Processing*, St. Charles, Illinois, Vol. III, pp. 188-192.

instruction and a compare instruction to set/clear the conditional code for the branch, and has a value of 2. CEU/XIU instructions include floating–point loads and stores, $l_{fl} + s_{fl}$, and branch overhead, y, which typically counts one increment instruction for each essential base address register and one branch instruction. Thus

$$t_i = \mathrm{Max}\,((l_{fl} + s_{fl} + y/k), (f_{ma} + f_a + f_m + x/k), (f_{ma} + s_{fl})) \quad (2)$$

Both the floating–point unit and the data subcache are fully pipelined, so the issue unit subsumes the bound for the floating–point unit, $t_f = f_{ma} + f_m + f_a$, and the memory unit $t_m = l_{fl} + s_{fl}$, as shown in Equation (1) above.

The loop–carried dependence pseudo–unit is a fictitious unit that models the time, t_d, for the longest loop–carried dependence as the sum of the latencies of the operations in one traversal of the cycle divided by the number of iterations in one traversal. In the absence of a loop–carried dependence, t_d is 0.

While computing the t_l bound we have ignored *i*) factors that may limit concurrency between the machine units, *ii*) the inability to pack the reservation templates of the individual instructions in a loop body tightly into a reservation table, *iii*) cache misses, internodal communication, register spilling and other operations introduced by the compiler, and *iv*) time for code that is not in the inner loop, loop start-up time, system overhead and contention. Therefore it is possible for an optimal schedule to exhibit performance that does not reach the MA bound.

The MA bounds (in CPF) for LFKs 1-12 are calculated in Table 1. The f_{ma} operation in loop 5 is of the form X*(Y-Z), where Z is the result of the previous iteration. Thus $t_d = 4$, the triad–to–triad latency. Loop 11 has a floating–point add operation of the form X+Y where X is the result of the previous iteration. Thus $t_d = 2$, the add–to–add latency. Only LFK 7 has t_i, and hence t_l, affected by FPU{C} source conflicts.

The twelve kernels were compiled for the KSR1 using the –O2 option of version 1.1.3 of the Fortran compiler which does loop unrolling in addition to other global optimizations.

The average CPF of a set of applications can be used to calculate their harmonic mean performance as follows:

$$\mathrm{HMEAN\ (MFLOPS)} = \mathrm{CPU\ clock\ rate\ (MHz)} / \mathrm{Avg.\ CPF} \quad (3)$$

The harmonic mean performance of the compiled code is 10.05 MFLOPS (1.99 CPF) while MA is 14.93 MFLOPS (1.34 CPF) for LFKs 1-12, hence the compiled code achieves 67.34% of the MA bound performance (calculated by dividing the CPF of the bound by the CPF of the actual application). All kernels but LFK 2 and 6 achieve at least 60% of their MA bound performance, while loops 10 and 12 achieve over 90%. Loop 6, at 29.93%, is the furthest away from MA. As k increases, the k–dependent term in the MA bound becomes negligible. The average CPF of the MA bounds of the first 12 LFKs would then be 1.24, corresponding to 16.13 MFLOPS. The compiled code achieves 62.31% of this bound.

The MAC bound calculation for the KSR1 is similar to the MA calculation. However all the actual compiler–generated instructions are counted in the assembly code, and the t_i formula is changed to:

$$t_i = \mathrm{Max}\,((l_{fl} + s_{fl} + (\text{other CEU/XIU}) + y'), (f_{ma} + f_a + f_m + (\text{other FPU/IPU}) + x'), (f_{ma} + s_{fl})) \quad (4)$$

The "other CEU/XIU" and "other FPU/IPU" terms count instances of all other types of instructions (except nops). Each term is calculated by counting the actual number of instructions of the corresponding type that appear in the unrolled loop and dividing by the degree of unrolling, k. In particular, $x' = x/k$ and $y' = y/k$.

Since the KSR1 assembly code is statically scheduled, the compiler must insert explicit *nop* instructions in the code to insure that data dependence requirements between instructions are satisfied. The MACS bound for one iteration of a loop, t_l, is thus the number of lines in the static listing of the assembly code divided by the degree of unrolling, k, and dividing by TNF to get CPF.

Table 2 (changed parameter values appear in **bold**) shows the MAC bound calculation. Counts have been divided by k; k, x/k, and y/k are taken from Table 1. The "other CEU/XIU" column lists those instructions on the CEU/XIU side of the issue that are not included in y and are neither floating–point loads nor stores. In the kernels examined, these instructions copy an IPU register to a CEU register. "Other FPU/IPU" = 0, as no such instructions exist in the kernels examined, except for floating–point moves which are implemented and counted as floating–point adds.

LFK	f_a	f_m	f_{ma}	l_{fl}	s_{fl}	t_d	x	y	t_i	k	MA Bound (CPF)	Compiled (CPF)
1	0	1	2	2	1	0	2	2	3 + 2/k	8	0.6 + 0.4/k	0.96
2	0	0	2	4	1	0	2	3	5 + 3/k	8	1.25 + 0.75/k	3.09
3	0	0	1	2	0	0	2	2	2 + 2/k	8	1 + 1/k	1.32
4	0	0	1	2	0	0	2	3	2 + 3/k	8	1 + 1.5/k	1.55
5	0	0	1	2	1	4	2	2	3 + 2/k	8	2.5; k=1 2.0; k>1	2.43
6	0	0	1	2	0	0	2	3	2 + 3/k	8	1 + 1.5/k	4.07
7	0	0	8	3	1	0	2	2	10; k=1 9; k>1	4	0.625; k=1 0.56; k>1	0.92
8	6	0	15	9	6	0	2	3	21 + 2/k	1	0.58 + 0.06/k	1.01
9	1	0	8	10	1	0	2	2	11 + 2/k	4	0.65 + 0.12/k	0.8
10	9	0	0	10	10	0	2	2	20 + 2/k	2	2.22 + 0.22/k	2.46
11	1	0	0	1	1	2	2	2	2 + 2/k	8	2 + 2/k	2.88
12	1	0	0	1	1	0	2	2	2 + 2/k	8	2 + 2/k	2.46

Table 1: Calculation of the MA Bound

LFK	f_a	f_m	f_{ma}	l_{fl}	s_{fl}	other CEU/XIU	t_d	t_i	MAC Bound (CPF)	MACS Bound (CPF)
1	0	1	2	**2.13**	1	0	0	3.38	0.68	0.93
2	0	0	2	**5**	1	0	0	6.38	1.59	2.59
3	0	0	1	2	0	0	0	2.25	1.12	1.25
4	0	0	1	2	0	**0.13**	0	2.5	1.25	1.31
5	0	0	1	**2.13**	1	0	4	3.38	2.0	2.31
6	**0.63**	0	1	**2.13**	**1**	**0.13**	0	3.63	1.81	3.56
7	**2.75**	0	8	**4.5**	1	0	0	11.25	0.70	0.89
8	**10**	0	15	**15**	6	0	0	27	0.75	0.97
9	**2**	**1**	**7**	10	1	**0.25**	0	11.75	0.69	0.76
10	9	0	0	10	10	**0.5**	0	21.5	2.39	2.39
11	1	0	0	**1.13**	1	0	2	2.38	2.38	2.75
12	1	0	0	**1.13**	1	0	0	2.38	2.38	2.38

Table 2: Calculation of the MAC and MACS Bounds

The number of noncombinable floating–point add operations, f_a, changes in LFK 6, 7, and 8 because of the introduction of floating–point move instructions. The compiler failed to find one pair of combinable multiply-adds in LFK 9. Many compiled kernels in-

Appeared in *Proceedings of the 1994 International Conference on Parallel Processing*, St. Charles, Illinois, Vol. III, pp. 188-192.

clude *nonessential* loads, and in LFK 6 there is a *nonessential* store.

The compiler-generated workload for loop 3 does not have any nonessential operations, and therefore the MAC bound is the same as the MA bound, as seen by comparing Tables 2 and 3. These two bounds are also the same for loop 5 due to the fact that t_d dominates the time spent in all other modeled machine units. The MAC bounds for loops 1, 4, 9, 10, 11 and 12 show only a small change from the MA bound, indicating that the workload produced for the bottleneck unit of the bound is close to the set of essential operations. These slight changes appear in loops 4, 9 and 10 due to the small fraction added to t_i by "other CEU/XIU" instructions, and in loops 1, 11, and 12 due to the slight increase in the number of loads, l_{fl}.

A large gap (*gap C*) between the MA and MAC bounds is evident in LFKs 2, 6, 7, and 8. In LFK 2, the compiler fails to identify all of the redundant loads, despite unrolling eight times. In LFK 6, the compiler fails to use a scalar as a reduction variable, thus introducing a nonessential store instruction. (This has been fixed in the latest compiler release.) In LFK 7, the compiler introduces extra FPU move instructions to save reusable values (overwritten due to a hardware restriction that requires one of a triad's source registers to be used as a result register) despite the fact that the FPU/IPU instruction stream is already the kernel bottleneck and some reloads can be masked. In LFK 8, the compiler again introduces redundant move instructions and nonessential loads.

All of the loops, except loops 2, 4, and 6, achieve at least 94% of the MACS bound performance. For loops 10 and 12 the MACS bound is the same as the MAC bound. This implies that the schedule for the compiler-generated workload was optimal. This is confirmed by the observation that the CEU/XIU portion of the instruction issue unit is the bottleneck, and the code on this side does not have any *nops*. However, the percentage of MACS performance achieved by the compiled code is 97.15% and 96.54% for loops 10 and 12, respectively. The remaining performance gap could be due to two reasons: the timing measurements are slightly perturbed due to system overhead, and only $\lfloor n/k \rfloor \times k$ iterations of a loop are executed in the unrolled part of the inner loop. The remaining $n - \lfloor n/k \rfloor \times k$ iterations are executed in a stub which is not unrolled and has a higher branch overhead associated with it. It is also likely that the schedule in the stub would be suboptimal, even though the schedule for the unrolled section is optimal.

4.2. KSR1 Performance Improvement

Given the insights produced by the MACS hierarchy gaps between the MA bound and the measured performance it is possible to tune the delivered single processor performance by modifying the assembly code produced by the compiler. Hand coding of the inner loops of each kernel elevated the KSR1 processor performance from 67.34% to 87.58% of the MA bound performance, as shown in Figure 2. [10]

The MA bound is clearly unattainable by modifying only the inner loops of LFK 2, 4, and 6 since they contain significant outer loop overhead. To assess this, we have developed a technique for measuring delivered steady–state inner loop performance. [8] If this steady–state performance is used as the metric for "delivered performance" for the hand–coded loops, the highest bars in Figure 2 are obtained: an average steady–state delivered performance of 94.83% of MA. LFK8 poses a very difficult template packing problem which even hand–coding could not solve. Each other loop achieved a steady–state inner loop performance greater than 90% of the MA bound performance.

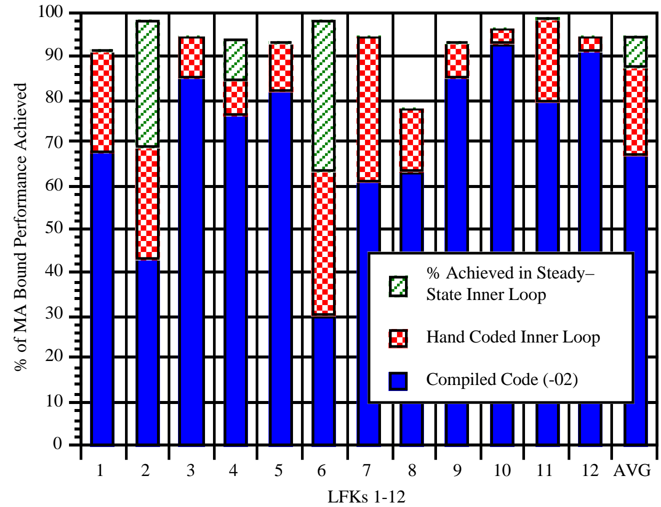

Figure 2: % of MA Bound Performance Achieved

5. Evaluation Tools for the KSR1

We are developing a series of tools to further facilitate the performance analysis of the KSR1. MA, MAC, and MACS performance bounds for the KSR1 can be generated by the tools K–MA and K-MACSTAT. When coupled with an understanding of interprocessor communication given by K–Trace and K–Cache, these tools provide an application programmer or compiler writer the ability to ferret out specific sources of performance degradation on the KSR1. Code restructuring, aided when needed by a version of OCO targeted for the KSR1, can then proceed in a more efficient goal-directed manner.

5.1. *K–MA* and *K-MACSTAT*

Currently K–MA calculates MA for inner loop bodies that contain no internal branches. K–MA assumes a typical scientific application that is dominated by floating–point loads, stores, adds, and multiplies. The tool can be extended to handle branching by using a profile methodology and it can be extended to more general code segments by adopting an integer performance model.

For a given source code segment, K–MA employs a two step process. SIGMA, a tool kit for building parallelizing compilers and performance analysis systems [11], builds a database which contains needed information such as expression trees and dependence vectors. Then K–MA backtracks through the expression tree to compute the number of essential floating-point operations, groups memory operations based on their dependence vectors and computes the number of essential load/stores, and computes the length of the maximally–weighted dependence cycle. It is then straightforward to calculate the workload of each of the four units and the performance bound of the loop for given machine parameters.

K-MACSTAT is a single pass, forward–scanning tool that generates the parameters used in the MAC and MACS bounds. It reports statistics for each loop in a designated region of interest. Statistics for outer loops report only on code not contained in the inner loops, i.e. the residue code; statistics for code spanned by forward branches are reported separately. Using TNF and t_d values

Appeared in *Proceedings of the 1994 International Conference on Parallel Processing*, St. Charles, Illinois, Vol. III, pp. 188-192.

calculated by *K-MA* and profile–generated frequencies for all conditional branches, MAC and MACS are then derived by taking weighted averages of the component statistics.

5.2. *K–Trace* and *K–Cache*

K–Trace instruments the assembly code of an application and generates memory traces of the code on the KSR1. Since K–Trace directly modifies assembly programs without any other input from the compiler, it is independent of the compiler and can thus be used to trace C, Fortran, and assembly programs on the KSR1.

Difficulties encountered in creating tracing tools for the KSR1, or porting them from other platforms, are caused by the fact that the KSR1 has no interlocks, all constants and subroutines are referred to via a table of constant pointers, no symbol table is generated for optimized code, the identification of global variables is difficult in KSR1 assembly codes, and KSR does not officially support assembly language programming.

After instrumentation, the modified program is linked with K–Trace run time routines and executed. An execution trace is produced for each processor including address, access type, and a pointer to the assembly code listing for each memory reference. Synchronization events in the program are also recorded in the trace. The instrumentation instructions do not affect processor state (registers and condition codes), so the output results of an instrumented program are identical to those of the original program.

The timing behavior of the instrumented program and the original program will differ as a result of run time dilation introduced by instrumentation. This dilation may not affect the accuracy of traces in a uniprocessor run because the memory references are recorded in the same order as in the original program. For a multiprocessor run on the KSR1 shared memory system, the timing of memory references among a set of processors is very important because the order of references to the same global address on different processors determines the explicit interprocessor communication, invalidates, and opportunities for automatic updates. The execution of a parallel program on the KSR1 is not deterministic due to system interference, the use of random replacement in the subcache, etc. As a result, the order of the instruction executions and memory references in the parallel program is not deterministic. The uncertainty of parallel execution is limited by the barrier synchronizations found in parallel programs. Modifying the KSR1 Presto library enables the recording of synchronization events.

The traces generated by K–Trace can be input to K–Cache, also under development, to simulate subcaches, local caches, and communication traffic. K–Trace and K–Cache can be used in combination to investigate *Gap P*. Assuming that the barrier synchronizations properly synchronize the high–level application code, K–Cache independently reconstructs the interactions of each processor with its subcache and local cache, and the hierarchical ring interconnect until a synchronization is reached. All processors then perform the accumulated invalidates at that time before continuing past the synchronization. K–Cache will flag occurrences of faulty synchronization, i.e. when some processor writes into a subblock that some other processor reads between the same pair of successive synchronizations. By this means, K–Cache can simulate individual processor portions of the trace simultaneously on distinct processors of a parallel system, and the memory accesses can be sufficiently well–ordered among the nodes.

6. Conclusion

We have presented an overview of the KSR1 architecture, a description of the MACS performance bound model, and a discussion of the principle causes and cures for the various performance gaps illuminated by the MACS bound methodology. We have demonstrated the MACS performance modeling technique on part of the Lawrence Livermore Fortran Kernel benchmark suite, and shown that it is possible to utilize the insights obtained to achieve a high percentage of the MA bound for each kernel examined. We have outlined a suite of tools (K–MA, K–MACSTAT, K–Trace, and K–Cache) that will automatically calculate the MACS bound hierarchy and model internodal communication.

7. References

[1] F. H. McMahon, “The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range,” Technical Report UCRL-5375, Lawrence Livermore National Laboratory, December, 1986.

[2] KSR1 Principles of Operation, Kendall Square Research Corporation,” Waltham, MA, 1991.

[3] KSR1 Technical Summary, Kendall Square Research Corporation,” Waltham, MA, 1992.

[4] E. L. Boyd, E. S. Davidson, “Communication in the KSR1 MPP: Performance Evaluation Using Synthetic Workload Experiments,” *Proceedings of the 1994 International Conference on Supercomputing*, July, 1994.

[5] D. Windheiser, E. L. Boyd, E. Hao, S. G. Abraham, E. S. Davidson, “KSR1 Multiprocessor: Analysis of Latency Hiding Techniques in a Sparse Solver,” *Proceedings of the 7th International Parallel Processing Symposium,* April, 1993, pp. 454-461.

[6] D. Windheiser, *Data Locality and Fine Grain Parallelism Optimization*, Ph.D. thesis, Irisa INRIA–RENNES, 1992. (Available only in French).

[7] W. H. Mangione–Smith, S. G. Abraham, E. S. Davidson, “A Performance Comparison of the IBM RS/6000 and the Astronautics ZS–1,” *Computer*, January, 1991, pp. 39-46.

[8] W. H. Mangione–Smith, T–P. Shih, S. G. Abraham, E. S. Davidson. “Approaching a Machine–Application Bound in Delivered Performance on Scientific Code,” *IEEE Proceedings*, August, 1993, pp. 1166-1178.

[9] E. L. Boyd, E. S. Davidson, “Hierarchical Performance Modeling with MACS: A Case Study of the Convex C-240,” *Proceedings of the 20th International Symposium on Computer Architecture*, May, 1993, pp. 203-212.

[10] W. Azeem. “Modeling and Approaching the Deliverable Performance Capability of the KSR1 Processor,” University of Michigan, Technical Report, CSE-TR-164-93, June, 1993.

[11] D. Gannon, J. K. Lee, B. Shei, S. Sarukai, S. Narayana, N. Sundaresan, D. Atapattu, F. Bodin, “SIGMA II: A Tool Kit for Building Parallelizing Compilers and Performance Analysis Systems,” *Proceedings of the IFIP WG 10.3 Workshop on Programming Environments for Parallel Computing*, Edinburgh, April, 1992, pp. 17-36.