Automatic Generation of Performance Bounds on the KSR1

Hsien–Hsin Lee   Edward S. Davidson

Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, Michigan 48109
{linear, davidson}@eecs.umich.edu
Phone: (313) 936-2917

ABSTRACT

Performance evaluation techniques have become critical indices for both architecture designers and software developers in recent years. Automatic tools are in demand for characterizing performance easily. In addition, a good performance analyzer should be able to point out the amount of performance loss relative to an ideal, where it occurs, and its causes. In this paper, MACS bounds, a hierarchy of bounds for modeling the performance, will be introduced for loop-dominated scientific code. This model successively considers the peak floating-point performance of a Machine of interest (M), the essential operations in a high level Application code of interest (MA), the additional operations in the Compiler-generated workload (MAC), the compiler-generated Schedule for this workload (MACS), and the actual delivered performance. In ascending through the MACS bounds hierarchy from the M bound, the model becomes increasingly constrained as it moves in several steps from potentially deliverable toward actually delivered performance. Each step from one bound to the next quantifies a performance gap associated with particular causes. The bound methodology will then be applied to a multiple-processor system to predict the run-time bound of a scientific application. We present three automatic bound generators, *K-MA*, *K_MACSTAT* and *MACS_GAP*, based on our performance model on the KSR1 Massively Parallel Processing (MPP) machine. Finally, we will use the performance bound models and tools to experiment on a real parallel scientific application running on the KSR1. The analysis results of our selected loops will be illustrated by the MACS bounds hierarchy, machine utilization, and relative speed-up to uniprocessor. These information will be very useful for improving the performance of the application at the right point.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

Performance evaluation techniques have been evolved as an important issue to assist the design of new computer architectures as well as the development of application software. Both computer architects and programmers need performance evaluation information to build higher performance systems and tune the performance of applications, respectively. Computer architects need investigate the performance of existing machines in order to design next generation machines with faster execution rate and lower communication penalties with a reasonable cost. Software developers focus on the compiler designs, operating systems and applications themselves on some specific machines to utilize the machines' resources more effectively according to the results of performance analysis. End users also rely on the performance evaluation results of standard benchmarks to choose the most favorable machines and/or application codes. For applications of small size, tuning performance may reduce system response time by a few seconds or minutes. However, for applications that consume a large amount of time such as most scientific application programs, tuning the performance may reduce the application execution time from several days or even months to running a day or a week. To improve the performance of scientific applications significantly, concurrent processing as well as performance evaluation and tuning techniques should be applied simultaneously.

Many commercial and research-oriented Massively Parallel Processing (MPP) systems have been built in recent years such as the *Kendall Square Research* KSR1, Cray T3D, Thinking Machine CM-5, Intel Paragon, Stanford Dash, MIT Alewife... etc. To be successful MPP systems must result in an increased system throughput and a decreased running time of several orders of magnitude. More powerful computers are demanded for many scientific applications over a range of many research and industrial fields. Although these MPP systems are both size and generation scalable [3] and the peak performance numbers claimed by the systems manufacturers are also scalable, the actual performance when running even those applications whose problems sizes are getting larger and larger today does not grow linearly with the scaled hardware components. Furthermore, the delivered performance of a scientific application running on an MPP system is much lower than the ideal peak performance claimed by the manufacturers. The actual performance delivered however, is generally much lower than the deliverable performance of an application. The causes of performance degradation between measured and deliverable performance cannot be well understood by considering only the running time of an application on a system. Therefore, computer scientists rely on performance evaluation techniques to investigate and analyze the reasons why and where the performance is degraded, and then to determine and apply effective performance tuning techniques to approach the deliverable performance of an application.

A hierarchical performance bounding methodology was originally developed by our research group in [1, 2]. In [4], this methodology was applied to build a performance model for the *Kendall Square Research* KSR1 machine. The preliminary performance analysis results based on the performance bounding methodology on the KSR1 using the *Livermore Fortran Kernels* 1-12 (LFK1-12) benchmark as an example are illustrated in [4, 5].

In this paper, the KSR1 architecture will be presented in next section. In section 3 and section 4, we will develop the performance bounds model defined in [4, 5] into a more precise model for both sequential and parallel versions of scientific codes running on the KSR1. In section 5, a hierarchi-

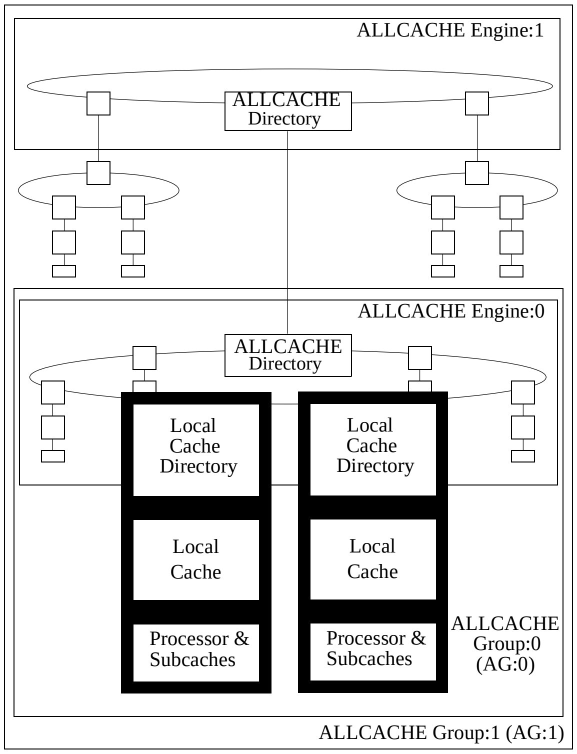


Figure 1: KSR Communication Hierarchy

cal performance gaps model is introduced to delineate specific causes of performance degradation. Section 6 presents two automatic performance bounds generators, *K-MA* and *K-MACSTAT*, that have been developed in this study based on our bounds methodology. An actual scientific application code will be used as an example to illustrate the performance results produced by our tools. Finally, section 9 will discuss future work of our bound methodology and tools for shared-memory as well as distributed-memory (message-passing) machines.

2 KSR1 Architecture

The *Kendall Square Research* KSR1 is a scalable shared-memory MPP system, which is the first commercial Cache-Only Memory Architecture (COMA) system. The KSR1 is built as a group of ALLCACHE[1] engines, connected in a fat-tree hierarchy of unidirectional rings. Each single ring can connect up to two ALLCACHE directories and up to 32 nodes. Each node is comprised of a custom processor and a cache memory. The system memory hierarchy consists only of these caches and disks. Up to 34 rings can be connected by a single second-level ring for a maximum configuration of 1088 nodes. The KSR1 machine provides programmers with a strict sequentially-consistent programming environment under its uniform System Virtual Address (SVA) space and implicit dynamic data migration through the memory management of its ALLCACHE mechanism. The communication hierarchy of the KSR is illustrated in Figure 1.

[1]ALLCACHE is a trademark of the KSR comapny.

2.1 Node Architecture

Each node on a ring of the KSR1 is compromised of a general-purpose 64-bit custom dual instruction issue RISC processor and two levels of private cache. Each processor operates at 20 MHz clock rate. As in most RISC architectures, only load and store instructions are allowed to access memory and the other instructions access the data residing in registers. There are four functional units in each processor cell, the Floating Point Unit(FPU), the Integer Processing Unit(IPU), the Cell Execution Unit(CEU), and the eXternal Input/output Unit(XIU).

The FPU has 64 floating-point registers, each of which is 64 bits wide. The floating-point register file contains three read and two write ports. The floating-point instructions support eight linked triad instructions that allow the processor to execute two floating-point operations in a single clock cycle, giving a peak floating-point rate of 40 MFLOPS. The IPU has 32 fixed-point registers of 64 bits. It performs integer and logical arithmetic on 64-bit integers in these registers. The CEU has 32 address registers which are each 40 bits wide. It fetches all instructions at the rate of 2 instructions per clock cycle, controls instruction flow and executes the address generation, and all load, store operations. The XIU has 64 I/O control and data registers. It performs direct memory access(DMA) and programmed I/O operations.

The processor issues a two-instruction VLIW-format instruction pair in each clock cycle, except when stalled waiting for load/store to complete. One instruction of the pair is issued to FPU/IPU for floating-point, integer arithmetic or logical instructions. The other portion of the pair is issued to CEU/XIU for address calculation, branches, memory reference instructions, or I/O operations. All functional units are fully-pipelined, hence, if the compilers schedule the instructions perfectly without any stalls, the processor can complete two instructions simultaneously in each clock cycle. The processor does not provide any hardware interlocks, thus it is up to the compiler to schedule instructions statically by checking the data dependency. Two dependent instructions must be separated by at least as large number of clock cycles as the latency of this dependent instructions pair. If sufficient independent instructions cannot be found to fill in between this dependent pair, *nop* (no operation) instructions will be inserted by the compiler.

The memory hierarchy for each node is comprised of a 0.5 MB subcache (L1) adjacent to the processor, backed by a 32 MB local cache (L2). The 0.5 MB subcache is further divided into a 0.25 MB data subcache and a 0.25 MB instruction subcache. The subcache is 64 sets, 2-way associative and uses a random replacement, write-back policy. The local cache is 128 sets, 16-way associative with a LRU (Least Recent Used) replacement policy. However, 4-way of the local cache is dedicated exclusively to the operating system; hence, only 12-way of the local cache is actually available to users' programs. The unit of allocation in the subcache is a 2 KB block; the transaction unit between subcache and local cache is a 64-byte subblock. In the local cache, 16 KB pages are allocated and the transaction units between local caches (a ring transaction) is a 128-byte subpage.

2.2 Ring Architecture

The KSR1 system implements a sequentially-consistent shared-address-space programming model, masking the physical distribution to local caches. The data coherence on the ring architecture must

Memory Component	**Memory Size (MBytes)**	**Memory Access Read (Cycles)**
Each Subcache	0.25	2 (1 per clock)
Local Cache	32.0	
(allocated block)		23.4
(unallocated block)		49.2
Remote Cache	32.0 each	
(allocated page AE:0)	(1024 total)	150 – 180
(allocated page AE:1)	(34816 total)	470 – 600 (estimated)

Table 1: KSR1 Memory Size and Read Access Latencies

be maintained by hardware interconnect as well as the coherence protocol. Peak bandwidth of a single ring is 1 Gbyte/second (128 million accesses per second); maximum achievable data transfer bandwidth is reported as 731 Mbyte/second. The unit of transfer on the ring is a 128-byte page plus 16-bytes of leading header information. The measured data access latencies are shown in Table 1.

A snooping protocol is used to locate a subpage within a single-ring system (ALLCACHE Engine:0 (AE:0) only with no ALLCACHE directories). Each subpage transaction makes one complete trip around the ring, allowing each node to monitor all traffic on the ring. For a transaction caused by a local cache miss, the request travels part way around the ring to the first node whose directory determines that it can respond, the request is serviced by the responding node, and the response then completes the ring tour back to the requestor. The coherence protocol can be applied to a two-level ring system straightforwardly. If the ALLCACHE directory determines that the request cannot be satisfied in its lower level ring, then it transfers the request to the higher level ring. The request then proceeds around part of the AE:1 and part of a remote AE:0 in a similar fashion. The response then completes the tour on each of these three rings in the reverse order.

The actual KSR1 coherence protocol has over 200 states which deal with the effects of parallel and pending requests and split transactions. The coherence protocol for a subpage is approximately described by the four-state transaction diagram of Figure 2. These four states for a subpage in a particular local cache are defined as follows.

- **Invalid (I)**: the local cache does not contain a valid copy of this subpage, but does not own the subpage.

- **Copy (C)**: the local cache has a read-only copy of the subpage.

- **Exclusive Owner (EO)**: the local cache owns the subpage and no other cache has a copy of this subpage.

- **Nonexclusive Owner (NO)**: the local cache owns the subpage, and has a valid copy but other caches might also have a copy of this subpage.

The local cache control unit changes the state of a subpage either on receiving a request from the processor (Pread/Pwrite) or on snooping a relevant request on the ring network (Nread/Nwrite).

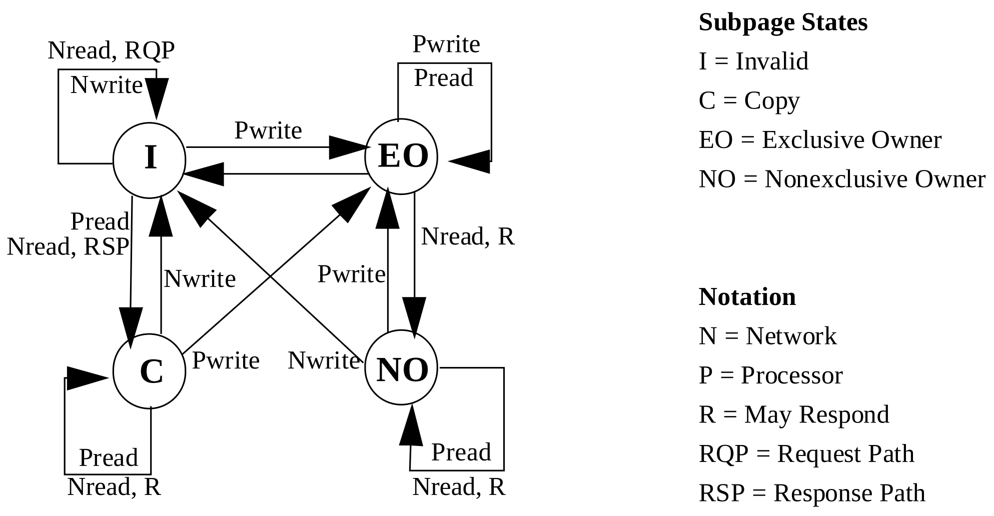


Figure 2: KSR1 Local Cache Coherence Protocol

2.3 Latency-Hiding Techniques

In parallel systems with distributed shared memory, the accesses to remote data have increasingly communication costs as the size of the system is scaled up. These accesses thus become an increasingly severe bottleneck for a high performance system. Accordingly, many scalable multiprocessors rely on some latency-hiding mechanisms to enhance the performance of the system. The KSR provides some latency hiding features in its coherence protocol and instructions.

The KSR coherence protocol enables an *automatic update* on network read requests. If a node sends a read request for a subpage onto the ring, then each local cache that is isolated along the response path between the responder and the requesting node will automatically update an invalid copy if it has one, unless the local cache is too busy.

Communication penalties can be further reduced by using the explicit *prefetch* and *poststore* instructions. The memory and communication activity of a prefetch can go on in parallel with subsequent computation. A successful prefetch moves the requested subpage into the local cache of the requesting processor in advance of the actual demand request so that the processor will not stall. Poststore allows a processor to broadcast an updated subpage by sending the subpage on a complete tour of the ring. Other local caches that have invalid copies update their copies as in automatic update. These two instructions can be automatically generated and inserted into the compiled code by the KSR compiler. The programmers also can explicitly request them to be inserted by means of intrinsic functions.

3 MACS Hierarchical Performance Bounding Model

Conventional claimed peak performance and the performance results of benchmarks on a machine cannot precisely characterize the deliverable performance capability of a specific application running on a particular machine. Given an application code, people may be interested in how fast this application can run on a machine of interest, namely, an upper bound on the achievable performance of the given application on the machine of interest [11]. Since software and hardware causes of performance degradation can be very intricate, it may not be clear which of several potential bottlenecks must be addressed in order to improve a poor performance. Many factors can affect the running time of the application. A compiler's job is to take the source code and compile it into an executable code that is well-optimized for the architecture of the machine that runs it. A poor compiler will generate inefficient codes due to redundant generated instructions or deficient scheduling. The efficiency of the executable code also strongly depends on some inherent properties such as application parallelizability, data locality, data structure layout, distribution of functional units used, communication protocols, etc. To identify the causes of performance deficiency and improve the performance by addressing these causes, we need an analysis of both source code and compiled code to obtain a series of upper bounds that explicitly identify the deliverable performance of the application and the individual contributions of several factors to the degradation in actually delivered performance.

Our research group has developed a hierarchical bounding methodology, called MACS bounds, for computing such bounds [1, 2, 9, 10]. In the following sections, the series of upper bounds will be described and the way in which they impose progressively stricter, less idealized, assumptions regarding the machine, application code, compiler and code scheduler as the bound is tightened from potential toward actually delivered performance.

3.1 M Bound

Most scientific applications contain a large number of floating-point operations within their innermost loops which consume the major portion of the running time. The M bound is defined as the peak floating-point performance of the Machine of interest. To achieve the M bound during some phase of an application, the floating-point functional unit must be kept continuously busy during that phase. The M bound ignores the effect of all instructions other than essential floating-point operations. Thus, the memory access latencies for accessing the data required in computation, the overhead instructions of loop bodies, etc, have no effect on the M bound. Hence, in most cases, it is impossible for application codes to achieve or even closely approach the M bound.

3.2 MA Bound

The M bound is a hardware architecture's peak performance, a unique performance figure for a particular machine, which is independent of the application. It thus has no sensitivity to the nature of a particular application. An application's high level code contains necessary operations of various kinds that must be performed. We define these necessary operations that are apparent by analyzing the application source code as *essential* operations. Suppose one has an ideal compiler that only generates the instructions for these *essential* operations, and an ideal machine that does not require any other operations for proper execution. This idealized machine then dispatches these instructions to their corresponding functional units to execute with full instruction-level parallelism with the assumption that no instruction latencies, memory stalls, or control or data dependencies exist. In particular, it is assumed that floating-point multiply-add triad instructions (if the machine has such instructions) are used wherever possible to combine multiply-add operation pairs, each distinct array element operand that must be loaded is loaded only once, etc. To model general scientific computers, the modeled functional units typically include an instruction issue unit, a floating-point processing unit, a data memory port, and a dependence pseudo-unit. The first three are real hardware constructs. The dependence pseudo-unit is an abstract construct to account for loop-carried data dependencies that occur in applications with recurrence. All other dependence is assumed to be handled ideally by loop unrolling, software pipelining, etc., so that it causes no performance degradation. The MA bound is then defined as the number of clocks per loop iteration that is needed by the Machine's bottleneck functional unit (the busiest unit) for the given Application in order to execute the essential operations assigned to it.

3.3 MAC Bound and MACS Bound

The MAC and MACS bounds include same effects of the compiler that were ignored in the MA bound computation. Given the actual compiled code of an application, we want to assess how fast it might run on the target machine. This information gives two bounds that are more tightly constrained to the performance limit of the actual compiled code, and more closely approach the delivered performance of the application on the target machine.

The MAC bound is defined as the performance bound of a Machine and a high level Application code under an actual Compiler-generated workload on the assumption that a perfect instruction scheduler is used to schedule the compiled code with no stalls. A perfect instruction scheduler always can find independent instructions to fill the latency slots which arise between two dependent instructions. The difference between the MA bound and the MAC bound is that the MAC bound counts the instructions that appear in an actual compiled workload, rather than only essential operations counted from the high level code, to estimate the deliverable performance. Consequently, the MAC bound will add time for all operations, including redundant operations, inserted by the compiler. For simplicity, we refer to all operations in the compiled code that are not included among the essential operations counted in the MA bound, as "redundant" operations, even though some of them may be necessary due to implementation details of the machine or the application that are not modeled in the MA bound.

The MACS bound is computed by using actual compiler-generated Schedule for the workload, instead of an assuming perfect instruction scheduler as in the MAC bound computation. The code that runs on the target machine may experience latency or resource conflict delays imposed dynamically by the hardware or statically by the compiler whenever there are resource conflicts or data dependencies between two instructions that are scheduled too closely.

However, the MACS bound still ignores some effects that are difficult to model with simple equations. These include cache miss penalties, inter-processor communications, OS interrupts, and I/O operation overheads. The MACS bound thus presents a performance bound on the minimum run time of an actual machine code as generated and scheduled by the compiler.

4 The MACS Bounds Model on the KSR1

In this section, the formulation and computation of the MACS bound hierarchy described in previous section is developed for the KSR1 processor. Previous work on defining bounds equations and analyzing the *Livermore Fortran Kernels* (LFKs) 1- 12 benchmarks on the KSR1 can be found in [4, 5]. We will extend the original model into a more precise one to more accurately and completely characterize the performance bounds of application codes. The performance bounds are expressed clock cycles per floating-point operation (CPF) or the number of clock cycles spent per iteration of innermost loop (CPL). The bottleneck units considered for the KSR1 are those we described in Section 3.2 (issue, floating-point, memory port, and dependence units).

4.1 KSR1 M and MA Bounds

The M bound is defined as the machine peak floating-point performance. The floating-point triad instructions of the KSR1 processor allow two floating-point operations to be finished in each clock cycle, resulting in a processor peak performance of 0.5 CPF. For computing CPL values, the Total Number of essential Floating-point operations per iteration (*TNF*), as shown in Equation 4.1, is required. The parameters f_a, f_m, f_{ma}, f_{misc} represent floating-point add/subtract, floating-point multiply/divide, all floating-point triad instructions, and other miscellaneous floating-point operations provided by KSR1 processor such as comparison, ceiling/floor instructions, negation, square root approximation, ..., etc, respectively. Therefore, the M bound in CPL is as shown Equation 4.3:

$$TNF = f_a + f_m + 2 \times f_{ma} + f_{misc} \tag{4.1}$$

$$M\ Bound\ (in\ CPF) = 0.5 \tag{4.2}$$

$$M\ Bound\ (in\ CPL) = 0.5 \times TNF \tag{4.3}$$

From the MA bound, the bandwidths of each of the four potential bottleneck units described in Section 3.2 are used to characterize the performance. Timing equations for each of these units are developed, which specify the minimum number of clock cycles required to process one iteration of a given loop through that unit.

The floating-point unit can execute one instruction per clock cycle including the eight linked multiply-add triad instructions. The lower bound, t_f, for the floating-point unit, as shown in Equation 4.4 , is the sum of the number of combinable essential triad instructions (f_{ma}), the number of uncombinable essential multiply (f_m), add (f_a) and the other floating-point operations (f_{misc}), in one iteration of a loop.

$$t_f = f_{ma} + f_m + f_a + f_{misc} \tag{4.4}$$

The memory part of the KSR1 is kept busy for at least one clock cycle for each floating-point load and floating-point store. As shown in Equation 4.5, the lower bound, t_m, of the memory unit is modeled by the number of essential floating-point loads (l_{fl}), and stores (s_{fl}), in one iteration of a loop.

$$t_m = l_{fl} + s_{fl} \tag{4.5}$$

The instruction issue unit can issue an instruction pair on each clock cycle. One of the pair goes to the FPU/IPU, while the other one goes to the CEU/XIU, as discussed in Section 2.1. Floating-point stores and linked triad instructions conflict on a register port, FPU{C}, that prohibits these two instructions to be executed simultaneously, thus constraining instruction issue. In Equation 4.6 below, x is the number of branch overhead instructions that enter the FPU/IPU, typically including a loop index decrement instruction and a compare instruction to set or clear the conditional code of the branch; y is for those entering CEU/XIU and typically includes a branch instruction and an increment instruction for each base address register for accessing array elements. Since loops can be unrolled either by the compiler or by hand, only a fraction $(1/k)$ of the branch overhead instructions will contribute to the time for each iteration of the original loop, where k is the degree of the unrolling. The issue unit lower bound, t_i as shown in Equation 4.6, is the maximum of the essential floating-point memory accesses plus the normalized branch overhead, the essential number of multiply-add combinable and uncombinable floating-point operations plus the normalized branch overhead, and the essential instructions that use the FPU{C} register port in one iteration of the loop.

$$t_i = max((l_{fl} + s_{fl} + y/k), (f_{ma} + f_a + f_m + f_{misc} + x/k), (f_{ma} + s_{fl})) \tag{4.6}$$

The lower bound of the dependence pseudo-unit, t_d as shown in Equation 4.7, is the total latency cycles required to traverse the longest loop-carried dependence in the program dependence graph. This unit is used to model the effect of recurrence, *i.e.* a result of one iteration depends on the corresponding result of a previous iteration. t_d is computed as the sum of the latencies of the operations in one tour of the longest cycle divided by the number of iterations in the cycle; t_d is 0 if there is no loop-carried dependence.

$$t_d = \frac{total\ latency\ cycles\ required\ to\ traverse\ a\ recurrence\ cycle}{number\ of\ iterations\ in\ the\ cycle} \tag{4.7}$$

The lower bound, t_l, on the run time of a processor for one iteration of a loop is the maximum number of clock cycles spent in any one functional unit. Consequently, tl is the minimum cycles per loop iteration for a processor based only on a machine model and the essential operations in the source-level application code. t_l is the MA bound in CPL, as shown in Equation 4.8. However, it is obvious that t_i dominates t_f and t_m, so the bottleneck unit is the instruction issue unit unless a loop-carried dependence exists with $t_d > t_i$. The MA bound in CPF is obtained by dividing t_l by the total number of essential floating-point operations, as shown in Equation 4.9.

$$MA\ Bound\ (in\ CPL) = t_l = max(t_f, t_m, t_i, t_d) = max(t_i, t_d) \tag{4.8}$$

$$MA\ Bound\ (in\ CPF) = t_l/TNF = max(t_i, t_d)/TNF \tag{4.9}$$

Many aspects of performance are ignored, or assumed to be ideal, in MA bound computation including i) factors that limit the concurrency among machine units, ii) the inability to pack the reservation templates of the individual instruction in a loop body tightly into a reservation table, iii) an imperfect compiler that generates inefficient machine codes, iv) resource contention, register spilling, cache misses and inter-nodal communication, v) instructions other than floating-point, memory reference, or branch overhead, and vi) residue code is not within the innermost loop. Thus many opportunities exist to improve actual code performance that does not reach the MA bound. Because the MA bound does not consider the effects of compiler and clock rate, this bound is also useful for comparing the inherent architecture-level performance of an application executed on different machines' architectures.

4.2 KSR1 MAC and MACS Bounds

Computing the MAC bound for the KSR1 processor is quite similar to the process of computing MA bound. The difference is that we count instructions in the actual compiled code for the MAC bound instead of only essential operations. It is assumed that there is an ideal code scheduler while computing the MAC bound, *i.e.* that each generated instruction can be completed in one clock cycle. Hence we count all generated instructions, except *nop*, for each functional unit in the same manner as in computing the MA bound. The equations for t_m of memory unit, t_f of the floating-point unit, and t_d of the dependence pseudo-unit remain unchanged. However, redundant or other necessary but previously ignored operations generated by the compiler will add extra clock cycles to the parameter values in these equations. In the IPU, CEU, or XIU, the compiler might also possibly generate redundant or necessary operations that we ignored while computing the MA bound. The equations for computing t_f, t_m, and t_d remains unchanged, but the results could be different from those in MA bound due to the effect of the compiler. The equation for t_i is rewritten as in Equation 4.10. The MAC bound equations in CPL and CPF shown in Equation 4.11 and 4.12 are the same as the MA bound equations. Note that the value of TNF is unchanged so that CPF bounds consistently reflect cycles per *essential* floating-point operation.

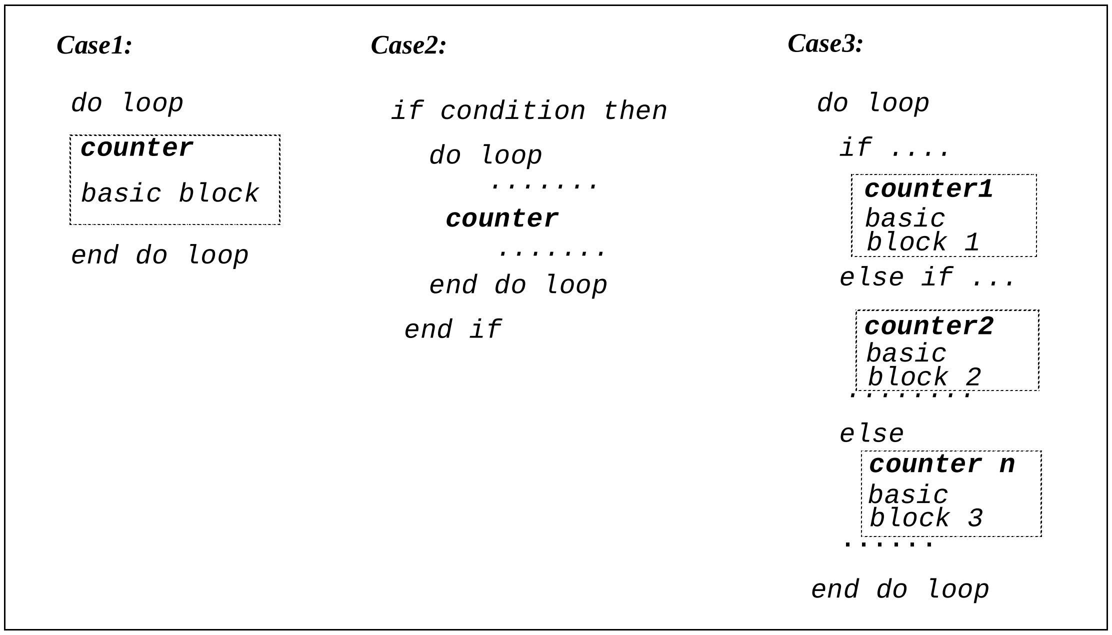
Since the KSR1 compiler produces VLIW-format instructions pairs in the generated assembly code, with FPU/IPU instructions in left column and CEU/XIU instructions in right column, it is straightforward to obtain the MAC bound simply by counting the number of operations except *nop* in the left and right columns of the compiled assembly code separately for an innermost loop, and the number of instructions that use the FPU{C} register port, then taking the largest count divided by the degree of unrolling, k, as the MAC bound of that loop in CPL.

$$\begin{aligned} t_i \;=\; & max((t_f + \; Other\ IPU\ except\ finop + x/k), \\ & (t_m + \; Other\ CEU/XIU\ except\ cxnop + y/k), \\ & (f_{ma} + s_{fl})) \end{aligned} \tag{4.10}$$

$$MAC\ Bound\ (in\ CPL) = t_l = max(t_f, t_m, t_i, t_d) = max(t_i, t_d) \tag{4.11}$$

$$MAC\ Bound\ (in\ CPF) = t_l/TNF = max(t_i, t_d)/TNF \tag{4.12}$$

Computing the MACS bound for KSR1 processor is also straightforward. Because no hardware interlocks are implemented on KSR1 processor, the compiled code is statically scheduled for execution correctness. When no other instructions are available to be filled in the latency slots between two instructions with data dependence, *nop* (no operation instruction) instructions are automatically inserted by the compiler. All the latency cycles due to data dependence requirements are thus resolved by the compiler. The number of clock cycles, MACS bound in CPL, required for one



```
Case1:
do loop
  counter
  basic block
end do loop
```

```
Case2:
if condition then
   do loop
       .......
     counter
       .......
   end do loop
 end if
```

```
Case3:
do loop
  if ....
     counter1
     basic
     block 1
  else if ...
     counter2
     basic
     block 2
    ........
  else
     counter n
     basic
     block 3
   ......
end do loop
```

Figure 3: Inserting counters in loops

iteration of a loop as scheduled by the compiler can therefore be calculated directly from the static listing of the assembly code. The MACS bound equations for an inner loop are specified as follows.

$$MACS\ Bound\ (in\ CPL) = (Compiled\ Assembly\ Code\ Length)/k \tag{4.13}$$

$$MACS\ Bound\ (in\ CPF) = (Compiled\ Assembly\ Code\ Length)/(k \times TNF) \tag{4.14}$$

The difference between the MACS bound and the actual run time is caused by cache stall and system overhead.

4.3 Run-Time Performance Bounds on the KSR1 Single Processor

The MACS bounds models on the KSR1 introduced in previous sections all focus on one iteration of an innermost loop. They relate the efficiency of each loop to a series lower bounds. However, an MA bound associated with a particular loop in the application may not have much effect on the overall performance, since this loop may only execute for a small number of iterations. For calculating the run-time bound of an entire application, it is necessary to combine the bounds information based on one iteration of each loop weighted by the corresponding number of iterations obtained from some instrumented profiling tools. Since such a profiling tool is not available on the KSR1, the number of iterations can be obtained from the static loop bounds given by source program itself or by inserting counters in basic blocks of the application as shown in Figure 3. A basic block is a sequence of consecutive statements in which flow of control enters only at the beginning and exits only at the end without any possibility of branching within. If there is more than one basic block in an innermost loop due to conditional branches, separate counters are inserted as illustrated in Case 3 of Figure 3.

Assume that C_{l_i} is the iteration count and B_{l_i} is the MACS bound for the ith basic block in loop body l, where b_l is the number of total basic blocks inside loop l and m is the total number of loops in the program. The entire run-time bound for an application, denoted by RB, can be computed as Equation 4.15.

$$RB(MACS, sequential) = \sum_{l=1}^{m}\sum_{i=1}^{b_l}(B_{l_i} \times C_{l_i}) \tag{4.15}$$

This weighted summation is the total run-time bound of an application and its terms yield the distribution of the total run-time bound over the individual loop blocks. This information can guide program restructuring or compiler optimization to focus on improving the performance of those loops with largest run-time bounds.

4.4 Run-Time Performance Bounds on the KSR1 Multiple Processors

In this section, the MACS bound is applied to predict the deliverable performance of multiple processors executing a single application on the KSR1 system. The code of a parallelized program can be divided into two parts: *i*) code in parallelized regions and *ii*) code in sequential regions. The run-time bound of a parallelized application is obtained by collecting the MACS bound of each region separately and then combining and computing the results. The Seq_RB_n value is defined as the run-time bound of the nth sequential section of the program and can be computed as Equation 4.16. The run-time bound calculation for parallel loops will be discussed in section 4.4.1 and 4.4.2 as a function of the workload distribution. In section 4.4.3, we will combine the run-time bounds obtained from serial and parallel sections to get $RB(MACS, parallel)$, the final run-time bound of the entire parallel application.

$$Seq\_RB_n = \sum_{i=1}^{b_n}(B_{n_i} \times C_{n_i}) \tag{4.16}$$

4.4.1 Run-time Bound of a Parallel Loop for a Well Balanced Workload

A well balanced execution workload occurs when an equal number of iterations is assigned to each available processor. For example, if N is the total number of iterations of a particular loop, each iteration takes equal time, and p processors are involved in the execution of this loop, then each processor executes N/p iterations. Conceptually, with perfect balance, no processor is idle during the parallel execution. $Para\_RB_k$ for a well balanced workload is the best possible performance that a parallelized loop can achieve. Let p be the total number of processors allocated to the execution of the application. The formula is given in Equation 4.17, where k stands for the kth parallelized loop:

$$Para\_RB_k = \frac{\sum_{k}\sum_{i=1}^{b_k}(B_{k_i} \times C_{k_i})}{p} \tag{4.17}$$

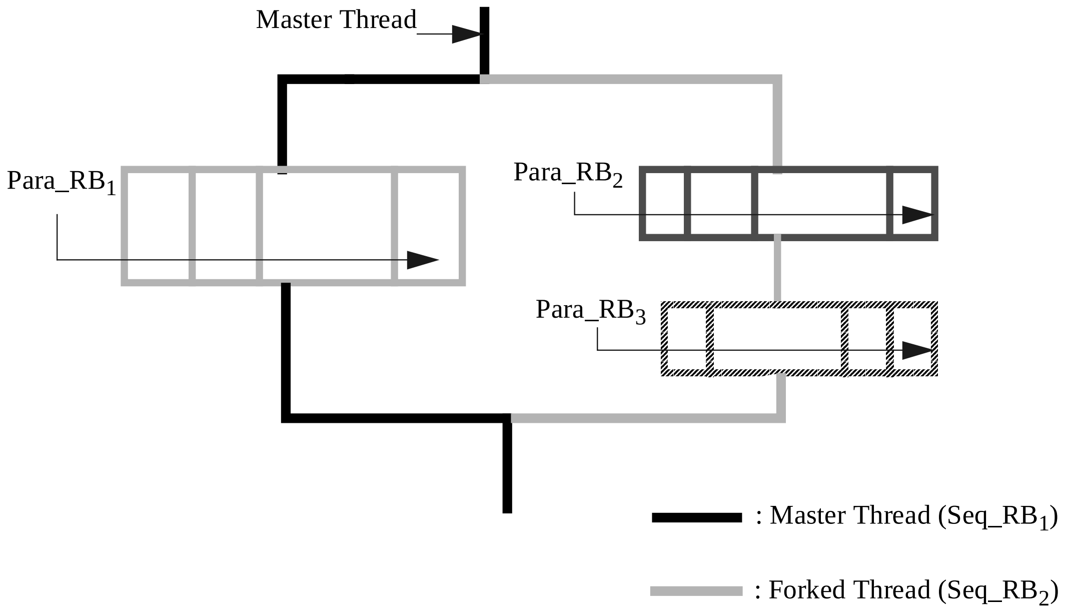


Figure 4: An example of execution pattern

4.4.2 Run-time Bound of a Parallel Loop for an Actual Workload Distribution

In most cases, the distribution of the workload of the loops in parallel regions is not so ideal as described in Section 4.4.1. Both static and dynamic strategies for workload allocation find it difficult to make every allocated processor busy all the time during parallel execution. Load imbalance generally exists when the number of iterations in a parallel region cannot be equally distributed onto the p processors and/or loops contain conditional branches that cause variable execution time per iteration. Thus, under these common circumstances, the bound in Equation 4.17 is often quite loose. A tighter bound for a particular parallel loop k can be obtained by collecting the number of iterations dispatched to each active thread in loop k, computing the MACS bound for each thread individually and taking the maximum of the MACS bounds of those threads as the final run-time bound of loop k. The formula is given in Equation 4.18, where superscript t stands for the tth thread and RB_k^t is the run-time bound of the tth thread in the kth parallelized loop. This $Para\_RB_k$ will portray the deliverable parallel run-time bound of the kth parallelized loop under an actual distribution of workload by the system.

$$Para\_RB_k = MAX^t(RB_k^t | RB_k^t = \sum_i (B_{k_i} \times C_{k_i}^t)) \tag{4.18}$$

4.4.3 Run-time Bound for an Application

All combinations of nesting of parallel directives are allowed in the KSR1 system for exploiting higher degrees of parallelism. Figure 4 illustrates some of the possibilities for nested parallel constructs. The master thread first forks into two section blocks and each section block contains

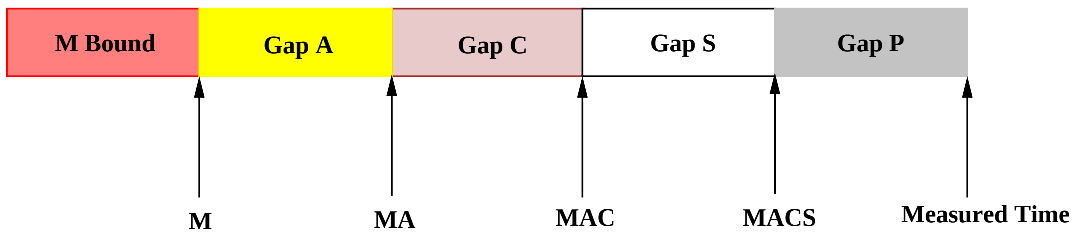


Figure 5: Gaps between the MACS bounds hierarchy and measured time

parallel regions with their own multiple threads. To compute the total run-time bound for an application, it is necessary to combine the run-time bounds of every sequential region, Seq_RB_n, and parallel region, $Para\_RB_k$, using the entire execution pattern of that region. The maximum run-time bound of a thread in an outermost synchronization region will determine the final run-time bound of that synchronization region. The run-time bound of the master thread in Figure 4 is shown as Equation 4.19.

$$RB(MACS, parallel) = MAX((Para\_RB_1 + Seq\_RB_1), (Para\_RB_2 + Para\_RB_3 + Seq\_RB_2)) \tag{4.19}$$

5 Gaps Between Performance Bounds

In ascending through the MACS bounds hierarchy from the M bound, the model becomes increasingly constrained as it moves step by step from potentially deliverable toward actually delivered performance. Each step quantifies a performance gap associated with particular causes. We make use of this hierarchical approach on a processor to individually evaluate the efficacy of the data flow analysis and the code scheduling phases of the compiler and identify their shortcomings. This method of hierarchical analysis exposes specific quantified performance gaps, as illustrated in Figure 5. The sizes of the gaps provide extremely useful information for identifying bottlenecks caused by the machine itself, by weakness in the compiler, and by implicit communications. Data restructuring, program reordering or proper data access sequencing techniques that promise the greatest potential performance gains can then be determined and applied according to which gaps are the largest. This gap-closing approach can be implemented within a *goal-directed compiler* for general purposes. The key causes of each gap are listed below.

- **Gap A (M to MA)**: Excessive loads/stores or loop bookkeeping per floating-point operation in the high level application code;

- **Gap C (MA to MAC)**: Weak compiler or hardware restrictions which add nonessential instructions;

- **Gap S (MAC to MACS)**: Weak compiler scheduling algorithms or hardware restrictions that preclude a tight schedule;

- **Gap P (MACS to Measured CPF)**: Unmasked stalls in the critical path due to communication, cache misses, and interrupts.

Note that the Gaps A, C, and S involve the performance of a single processor, while Gap P involves the interaction of processors and private memories, and internodal communication.

5.1 Gap A (M $\longrightarrow$ MA Gap)

Gap A is caused by essential memory accesses, loop overheads, loop-carried dependencies, and some floating-point operations that cannot be combined as triad instructions. Application programmers are not expected to write any code to run at the architectural peak performance. An adequate job for a programmer is to write and arrange his code reasonably efficiently so that a strong optimizing compiler might approach the peak performance. High level code programmers will bear the responsibility of reducing *Gap A*. Poorly reused data in high level application code is a common cause of performance loss exhibited by the *Gap A*. Excessive data renaming statements (e.g. $a(i) = b(i)$) and/or large data bandwidth between loops may introduce redundant memory accesses that appear as essential accesses in the high level code. Reordering loops and/or exploiting the opportunities for fusing loops, procedure inlining, ...,etc. , are feasible techniques for reducing *Gap A* so as to preserve the data not to be spilled out from registers as long as possible. More explicit data flow analysis such as global data dependency analysis should be done at the source code level in order to have a consistent program state and an efficient program restructuring approach. Some of these techniques can be carried out by a compiler or precompiler; others still remain the responsibility of the programmer.

5.2 Gap C (MA $\longrightarrow$ MAC Gap)

The contribution of *Gap C* is due to hardware restrictions and weaknesses of the compiler. Some of the compiler optimization schemes need to be developed for tuning the compiled codes to run more efficiently. Basically, reducing *Gap C* requires reducing the number of critical path instructions in the machine codes by eliminating nonessential operations. For analyzing the effect of the KSR compiler, we have developed two tools, named *K-MA* and *K-MACSTAT*, for generating the performance bounds automatically. using the information given by the tools, we analyzed some application codes and address some common causes of *Gap C*, described below.

• **Redundant instructions**: There are some redundant memory accesses and other redundant operations in the compiled codes. Redundant memory accesses that result in performance degradation should definitely be eliminated. Both redundant floating-point operations and redundant memory accesses can be observed by comparing the results generated by *K-MACSTAT* with the results generated by *K-MA*. Eliminating the redundant operations must be done by hand thus far.

• **Overhead for subroutine calls**: There are many loads and stores performed when a subroutine call occurs. They are used to save the current program state in memory when the subroutine is called and then reload this state back to the registers when it returns. This memory access overhead resulting from subroutine calls can be avoided by *procedure inlining.* Inlining a procedure is a process of replacing a subroutine call or function reference with the text of the subroutine or function.

Case 1:

```
program main
common /area1/ x(n), y(n),z(n)
real x(n), y(n), z(n)
....
call A(x,y,z)
...
end program
...............
subroutine A(xx,yy,zz)
real xx(n), yy(n), zz(n)
....
do loop k=1, n
xx(k) = yy(k) + zz(k)
..
end subroutine
```

Case 2:

```
program main
common /area1/ x(n), y(n),z(n)
real x(n), y(n), z(n)
....
call A
...
end program
...............
subroutine A
common /area1/ xx(n), yy(n), zz(n)
real xx(n), yy(n), zz(n)
....
do loop k=1, n
xx(k) = yy(k) + zz(k)
..
end subroutine
```

Figure 6: Different versions of code using a different number of base registers

• **Excess base index registers**: The data arrays declared in a common block should share a common base register if the offsets for all elements are within the offset range of the memory operations in the assembly code. However, it is usual for programmers to write high level code by employing the method of passing arguments when performing a subroutine call. Thus, those data arrays declared in the common blocks of callers will not be re-declared in a subroutine body. As a result, they are indeed arranged in consecutive memory space at run time, but the compiler does not know this fact and assigns extra base index registers for them during compile time. These extra base index registers will introduce some redundant memory accesses due to loading each base index, and will increase the length of the compiled code by the need to increment each of them. These extra operations will degrade performance to some degree. This problem can also be reduced by *procedure inlining*. Once the relevant procedures are inlined, the compiler is able to find which arrays are declared in which common blocks and thus share base registers where possible, thereby saving the extra instructions and registers for storing and updating the base indices.

The following example in Figure 6 shows how the programming methods for high level code affect the efficacy of the compiler. The difference between these two code versions are shown by **bold** statements. In Case 1, while compiling the body of subroutine A, the subroutine call is implemented by passing the arguments. The compiler will not recognize that array variables xx, yy, and zz are located in the common memory space. It thus produces three distinct base index registers for xx, yy, and zz, respectively. In case 2, while compiling the body of subroutine A, the compiler can distinguish the fact that array variables xx, yy, and zz are actually located consecutively in the memory address space due to the common variables declaration. It therefore assigns only one base index register for these three array variables instead of three, provided that all of the offsets are within the offset range of the memory operations in the assembly code.

5.3 Gap S (MAC $\longrightarrow$ MACS Gap)

Gap S is caused by resource conflicts and compiler scheduling inefficiencies. There are two popularly known scheduling strategies for optimizing the compiled code to reduce the *Gap S*.

• **Loop unrolling**: Loop unrolling can reduce the number of instructions per loop iteration by reusing the registers. It also reduces base index register increments, branch, and counter update code by performing them only once per unrolled loop. It achieves better scheduling with fewer *nop* operations by moving independent operations of unrolled loop iterations into the slots where no operations existed previously. The KSR Fortran compiler can do loop unrolling automatically by turning on the option *-O2* for code optimization. The degree of unrolling way possibly be limited by register set size and may depend on the size and complexity of the code. The compiler typically unrolls 2, 4, 8, or 16 iterations using a heuristic. Increasing the degree of unrolling will reduce the start-up and termination overheads.

• **Polycyclic loop scheduling**: A re-targetable tool, called *OCO* (Object Code Optimizer) in [4], can reschedule the KSR compiled code by using a polycyclic loop scheduling algorithm, also known as software pipelining.

5.4 Gap P (MACS $\longrightarrow$ Measured CPF/CPL Gap)

Gap P results from cache miss penalties, internodal communication stalls, and context switches. Cache and internodal communication simulation enable the visualization of data structure movement in the memory hierarchy. Two tools, *K-Trace* and *K-Sim*, developed by our group for the KSR1 will provide insights into cache and communication behavior. *Gap P* can be reduced by restructuring data reference patterns [12] in order to maximize the data reuse in the subcache and local cache levels to reduce the cache misses and internodal communication stalls. Common techniques include loop fusion, loop blocking, loop interchange [13], and special functions such as the *prefetch/poststore* instructions of the KSR1 [6]. *Gap P* is usually critical in parallel machines, especially in shared-memory systems, due to their inherent and implicit data manipulation.

6 Automatic Generation of MACS Bound on the KSR1

6.1 *K-MA*

Based on the high level application code, the *K-MA*[2] tool calculates the number of essential floating-point operations and load/store operations, estimates the cache miss ratio, and computes the achievable performance bound of a FORTRAN do-loop program for the KSR1 system. *K-MA* generates the MA bound based on the model described in Section 4.1. Since the MA bound model does not consider cache behavior, *K-MA* assumes a simplified cache model that combines an approximation of the reference window model (the set of data ideally stored in cache for later use) with a software cache (*i.e.* a cache whose storage and replacement strategy is governed by

[2]*K-MA* was developed by Tien-Pao Shih and Ashish Mehra

software). This simplified model provided background for estimating the cache miss bound for a software cache. Although this does not simulate the actual cache performance of the KSR1, it can give an indication of the working set size.

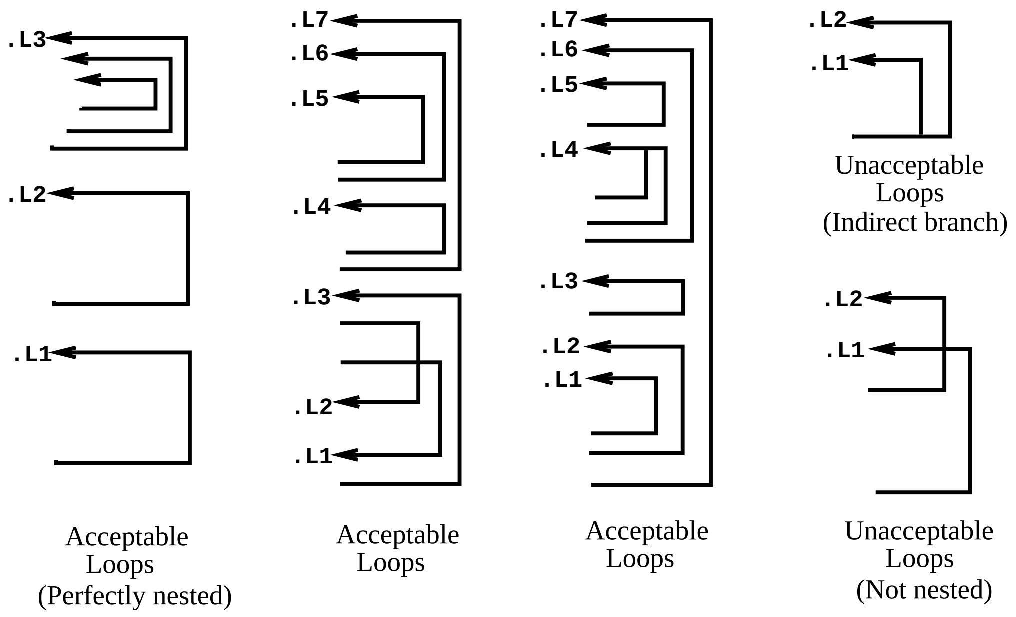
K-MA was implemented in *SIGMA*, a tool kit for building parallelizing compilers and performance analysis systems. *SIGMA* builds a database of a C or FORTRAN source program, which contains needed information such as the expression trees and dependence vectors. Then *K-MA* backtracks through the expression trees to compute the parameters used in MA bound, computes reference windows for all dependent reference pairs in loop, and puts the windows to cache. Finally, *K-MA* reports statistics for the MA bound parameters and cache hit ratio. Currently *K-MA* only evaluates innermost loops and ignores all internal branches. It assumes a typical scientific application that is dominated by floating-point loads, stores, adds, and multiplies. In addition, *K-MA* can provide a more accurate cache hit/miss ratio if given a model of the cache hierarchy instead of assuming a software cache.

6.2 *K-MACSTAT*

K-MACSTAT was developed in this study for automatically generating the parameter values used in MAC and MACS bounds equations. It is a single pass forward-scanning program, which reads a section of the KSR assembly code as its input, scans the code, and analyzes the loops inside, then generates the parameters for computing the MAC and MACS bounds. Parameter statistics include the number of each type of instruction found in each basic block. *K-MACSTAT* accepts perfectly nested loops and most cases of imperfectly nested loops (with forward branches inside the loop body) as its inputs. Some cases of acceptable and unacceptable loop constructs are illustrated in Figure 7. Basically, general nested loops written in well-structured programming styles are all accepted by *K-MACSTAT*.

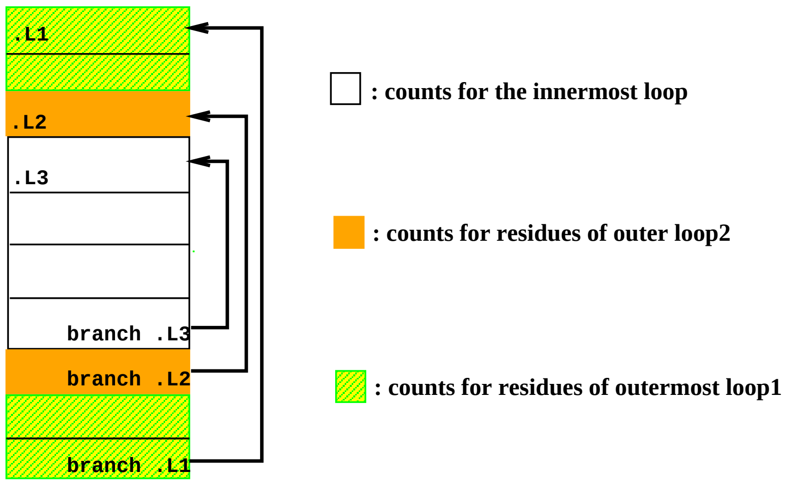
K-MACSTAT reports statistics for each loop in a designated region of interest. The statistics include the nesting relations for each loop, FPU/IPU operations except *nop*, CEU/XIU operations except *nop*, total length of the KSR compiled code including *nop*, floating-point operations, f_a, f_m, f_{ma}, f_{misc}, and memory access operations, *load* and *store*. The statistics for innermost loops are reported in their entirety. Statistics for outer loops report only on code not contained in their inner loops, *i.e.* the *residue* code. The residue code of a nested loop is exemplified in Figure 8.

Forward branches usually appear inside loops and often complicate the computation of the MACS bounds. However, the bounds statistics for each branch part within an innermost loop are made available for calculating the run-time bound. *K-MACSTAT* has been designed to recognize forward branches within innermost loops. Statistics for code spanned by forward branches including multiple forward branches or a single branch with loop unrolling are reported separately along with the statistics of the surrounding loop. Intersecting forward branches are also recognized and reported. For obtaining the final run-time bound, users have to combine the information given by the number of iterations executed during the run time for each basic block (including innermost loops with or without their corresponding forward branch blocks) as well as its corresponding bound parameters. The *K-MACSTAT* satistics of an innermost loop containing forward branches within is illustrated in Figure 9.



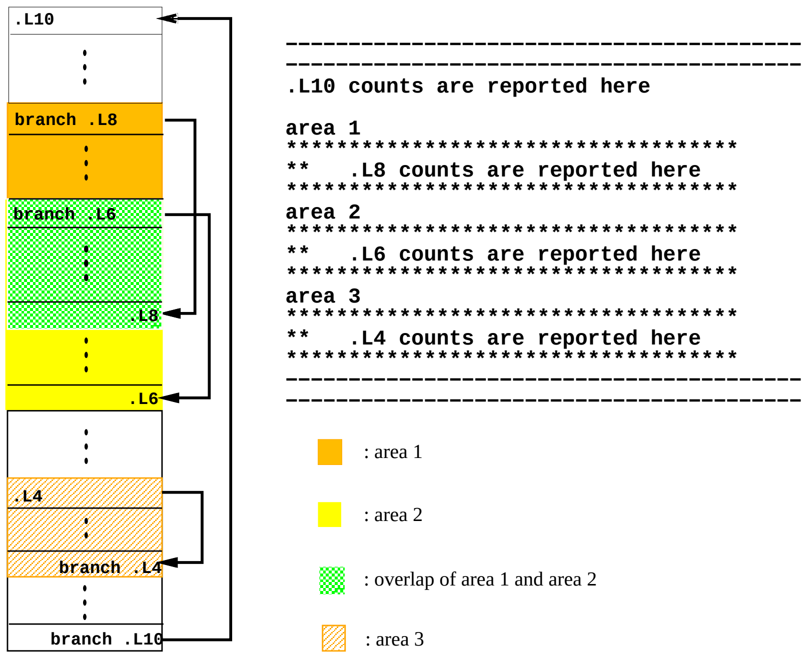
.L3
.L2
.L1
Acceptable Loops (Perfectly nested)
.L7
.L6
.L5
.L4
.L3
.L2
.L1
Acceptable Loops
.L7
.L6
.L5
.L4
.L3
.L2
.L1
Acceptable Loops
.L2
.L1
Unacceptable Loops (Indirect branch)
.L2
.L1
Unacceptable Loops (Not nested)

Figure 7: Acceptable and unacceptable loops for *K-MACSTAT*



.L1
.L2
.L3
branch .L3
branch .L2
branch .L1
: counts for the innermost loop
: counts for residues of outer loop2
: counts for residues of outermost loop1

Figure 8: Residues of nested loops



```
-----------------------------------------
-----------------------------------------
.L10 counts are reported here

area 1
************************************
**    .L8 counts are reported here
************************************
area 2
************************************
**    .L6 counts are reported here
************************************
area 3
************************************
**    .L4 counts are reported here
************************************
-----------------------------------------
-----------------------------------------
```

Figure 9: Statistics of an innermost loop with forward branches

An example of a complete scanning mechanism of *K-MACSTAT* is illustrated in Figure 10. After the last step shown in Figure 10, the results of bounds parameters for each basic block and residues associated with each outer loop are calculated and reported. Note that redundant floating-point operations, redundant memory accesses, and inefficient scheduling can be observed by comparing *K-MACSTAT* results with those of *K-MA* (*i.e. Gap C* and *S*). Eliminating redundant operations and rescheduling the code currently must be done by hand.

A FORTRAN subroutine, *MACS_GAP*, was developed for automatically generating an average MACS bound and its hierarchical performance gaps as described in section 5 for parallel programs running on the KSR1. We collect the MACS bounds information from the tools we developed. M bound is collected from the machine's peak floating-point performance. MA bound is derived from *K-MA* and MAC and MACS bounds is derived from *K-MACSTAT*. These bounds information, the number of active processors, and the counts of conditional branches that are taken, will be used as inputs to *MACS_GAP* during application run time. Then *MACS_GAP* will derive the hierarchical performance gaps for each thread that actively executes in a particular loop, and calculate the normalized[3] average time of *Gap A, C, S*, and *P* for that loop. It also reports the number of execution clock cycles needed for one iteration of each loop.

[3]Normalized to measured application run time.

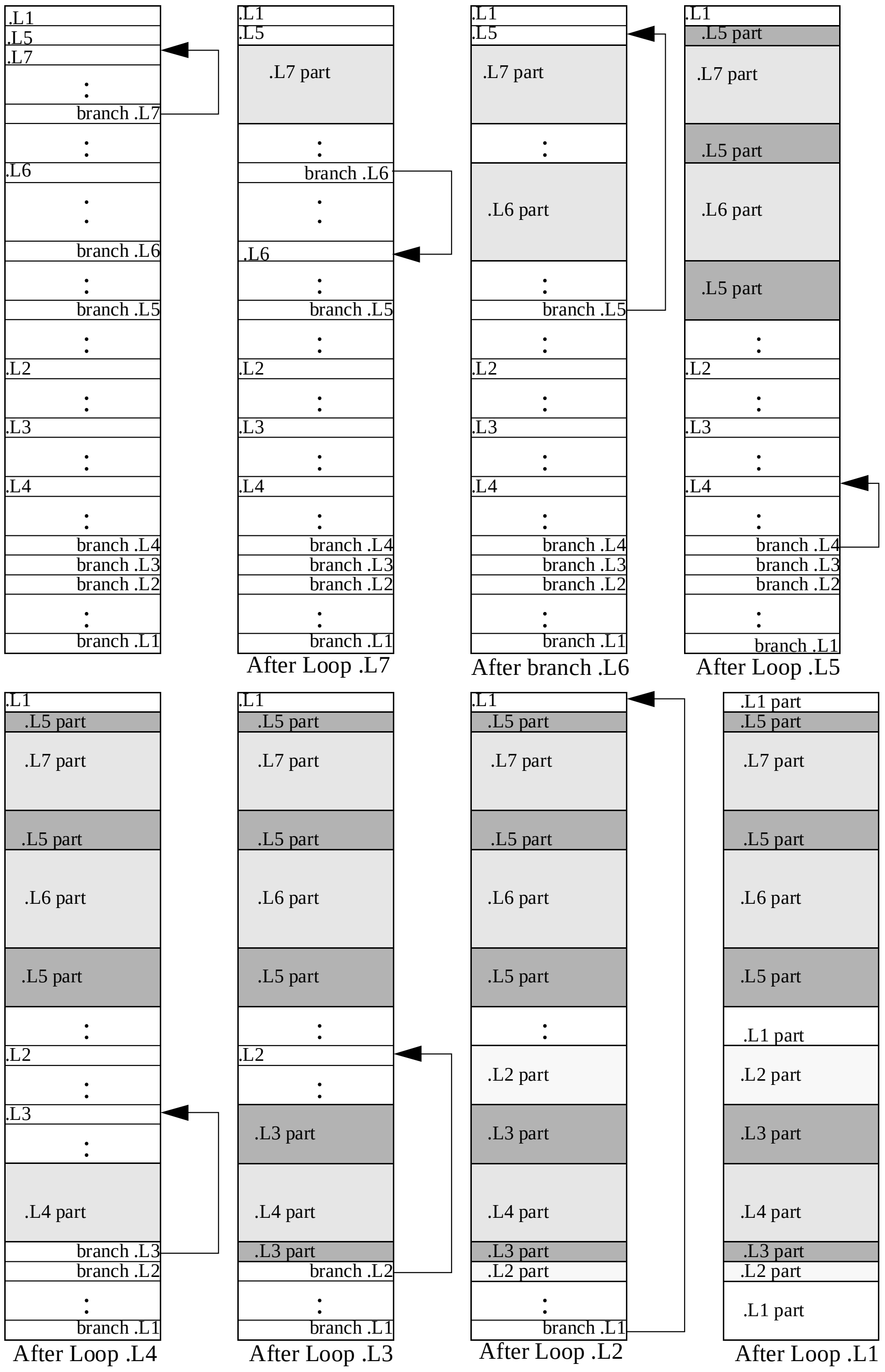


Figure 10: Program scanning by *K-MACSTAT*

7 Experiments and Discussion

An actual parallel application code running on the KSR1 was used to demonstrate the effectiveness of the hierarchical performance bounds and the evaluation tools, *K-MACSTAT* and *MACS_GAP*. In this section, the MACS bounds hierarchy methodology will be applied to the threads of a parallel application and the tools will be used to generate the bounds and gaps values. The application code is a parallelized scientific computing program, developed by the Ford Motor Company, that performs a finite element simulation of a car crash. The program simulates a few-millisecond car crash which is divided into many thousands of time steps, in which each time step executes the main loop of the program once. Thus, to complete the entire computation of the application, many thousands of cycles of the main loop are run. The finite element data set for this program contains many thousands of nodes and elements and a node connection array to maintain the information about the element-node connection relationship. The first iteration of this program is used for the purpose of initializing the variables, *i.e.* loading the data elements into cache memories and initializing their values. The results we have in the following discussion were collected from the 220th iteration, which we believe is representative of steady-state. The best achieved application speed-up that we measured was obtained on 28 processors of the KSR1.

The whole program is comprised of several hundred subroutines, each subroutine is comprised of FORTRAN do-loops with primarily floating-point operations inside. From the profile report produced by the *gprof* (a standard UNIX profiling tool), the 13 most time-consuming loops of the sequential version code were selected for our experimental analysis. The code was run on different numbers of processors including 1, 5, 10, 15, 20, and 28 and the selected 13 loops were analyzed using the gaps model of the MACS hierarchical bounds. We inserted counters for the 13 loops, collected the number of iterations including the number of times each conditional statement is satisfied for each processor, and calculated the average run time bound per iteration by taking a weighted average of the run time bounds for each loop. The normalized bar charts of Figures 11 through 17 illustrate the hierarchy of performance gaps for these loops. From each normalized bar, the ratio of the M, MA, MAC, and MACS time bounds as a fraction of measured run time are shown. Each gap contribution to actual run time is clearly visible. We can thus pay primary attention to the largest gaps in order to gain substantial improvement of total performance.

In Figure 11, *Gap P* takes up more than 50% of the run time, except for *loops 7, 9,* and *10.* Because this is a uniprocessor run, *Gap P* is essentially comprised of subcache and local cache misses on a single processor. These voluminous numbers of cache misses and their associated stalls could be explained by the fact that the total data set used in this program is too large to fit into one local cache, even though the local cache size is 32 MB. Whenever a main loop iteration ends and another one is started, most of the needed cache lines may have been replaced. In this case, the data that has been accessed in the same code regions of previous simulation time steps cannot be found in the local cache and needs to be reloaded by replacing some other currently resident data. In short, *Gap P* in a single-processor run results from capacity misses and collision misses. *Gap S* is most visible on *loops 3, 7, 9,* and *10.* For these loops, some improvement may result from a more efficient instruction scheduling strategy. Looking into the assembly codes and removing redundant instructions inserted by the compiler for some loops, such as *loops 2, 5, 7, 9,*

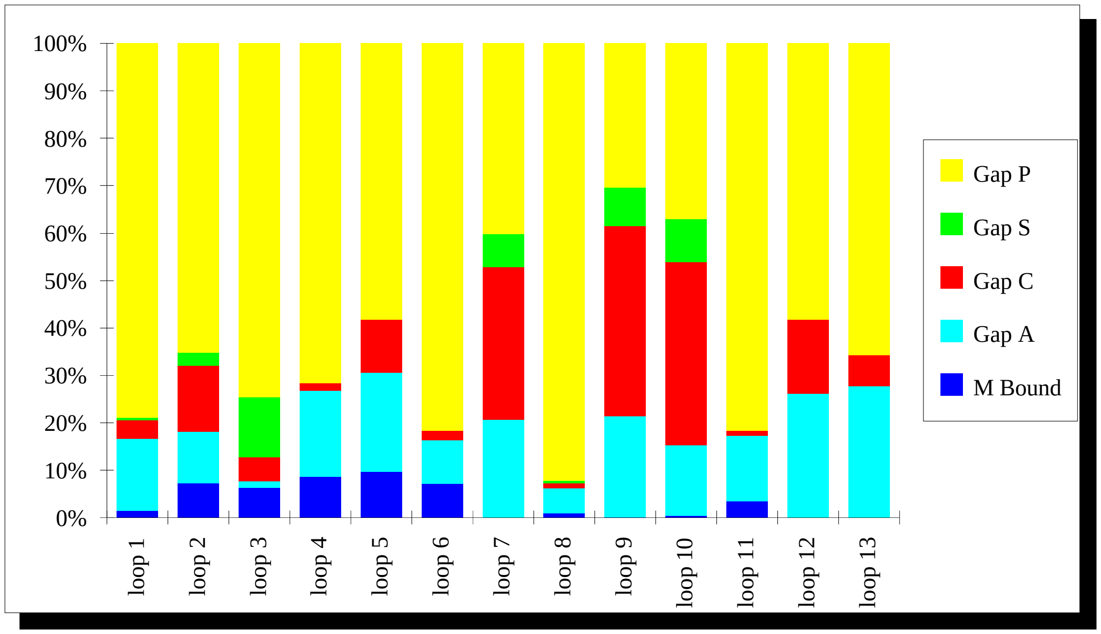


Figure 11: MACS gaps for 1 processor on the KSR1

10, and *12*, will be able to reduce the effect of *Gap C*. *Gap A* in most of the loops, especially *loops 12* and *13*, may result from an excessive number of load/stores in assignment statements and/or uncombinable floating-point instructions in the high level code. Code restructuring to reduce data moves and promote register locality may help reduce *Gap A*.

The experimental results for multiple-processor runs on the KSR1 are shown in Figures 12 through 17. Increasing the number of processors results in increasing the total capacity of the caches. The effect of capacity misses that occur in the uniprocessor case should be reduced or even eliminated by increasing the number of processors up to a certain amount. However, the penalties due to capacity misses appear to be compensated for by the internodal communication penalties introduced by multiple-processor execution. Whenever there is a read-write data element that is shared among two or more processors, inter-processor communication, resulting from coherence misses, becomes inevitable. The measured network latencies for a remote data access are listed in Table 1 of Section 2.2. We can also analyze the performance gaps for the multiprocessor in the way we described in the preceding section.

To assess the effect of increasing the number of processors, p, we can use the *machine utilization factor* to observe the efficiency variation among different numbers of processors for a particular loop. A *machine utilization factor* (MUF) is defined as the percentage of total run time that a CPU was kept busy during the execution of an application. Since the execution time consumed in computation and communication has been isolated in our hierarchical MACS bounds model, the MUF of a loop is equal to the MACS bound percentage of the total execution time as shown in

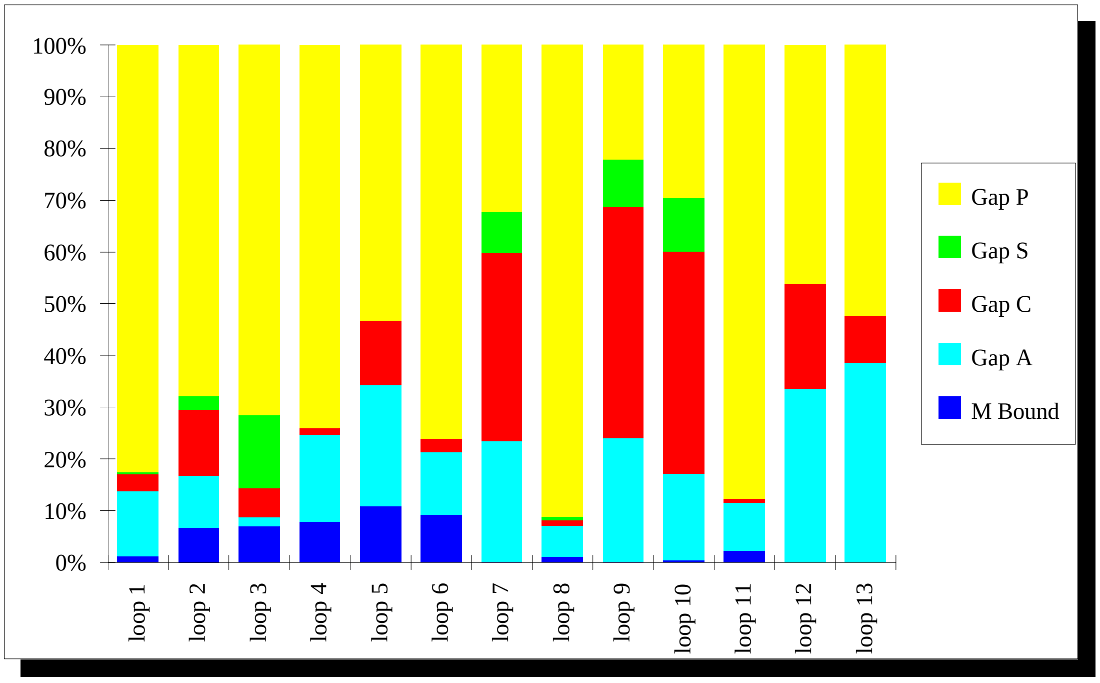


Figure 12: MACS gaps for 5 processors on the KSR1

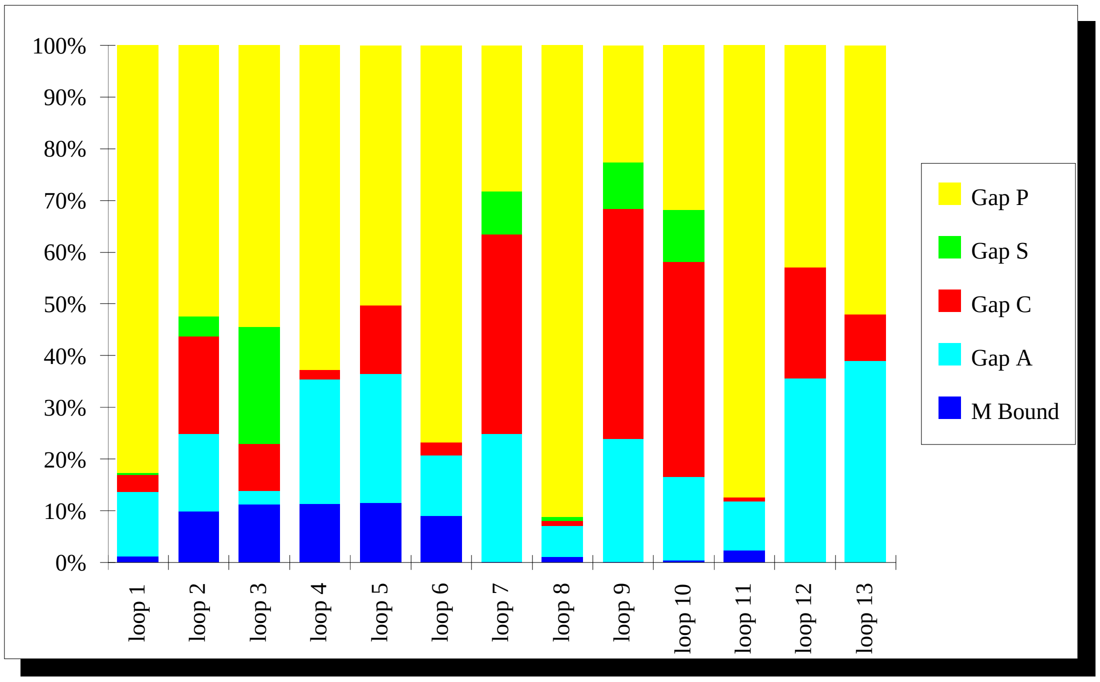


Figure 13: MACS gaps for 10 processors on the KSR1

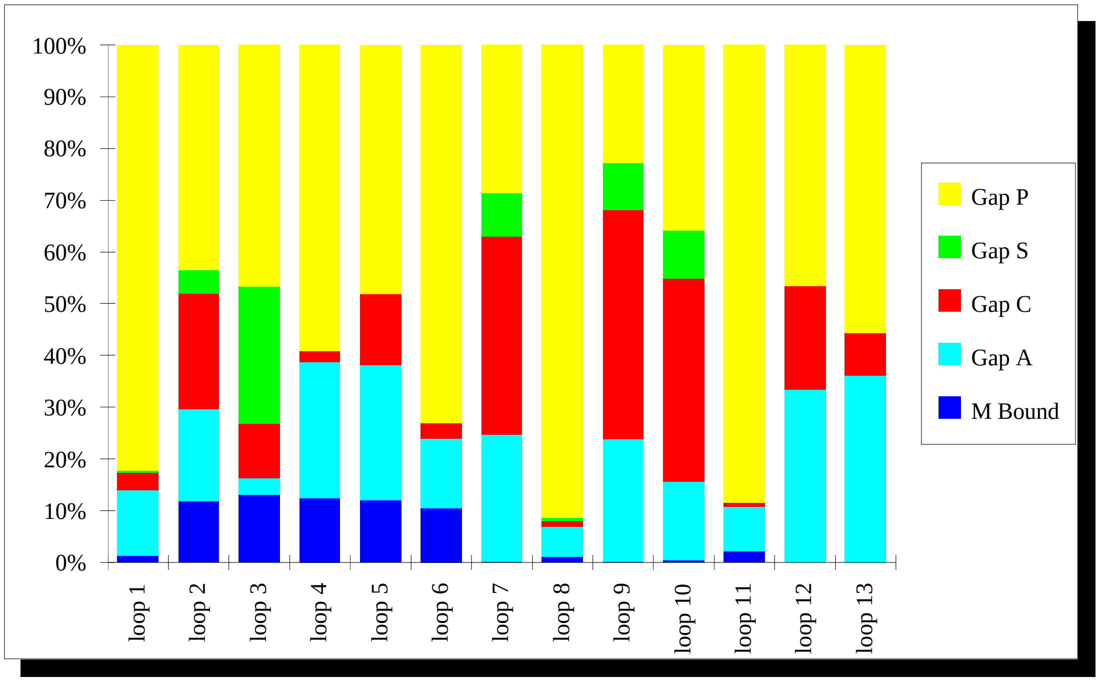


Figure 14: MACS gaps for 15 processors on the KSR1

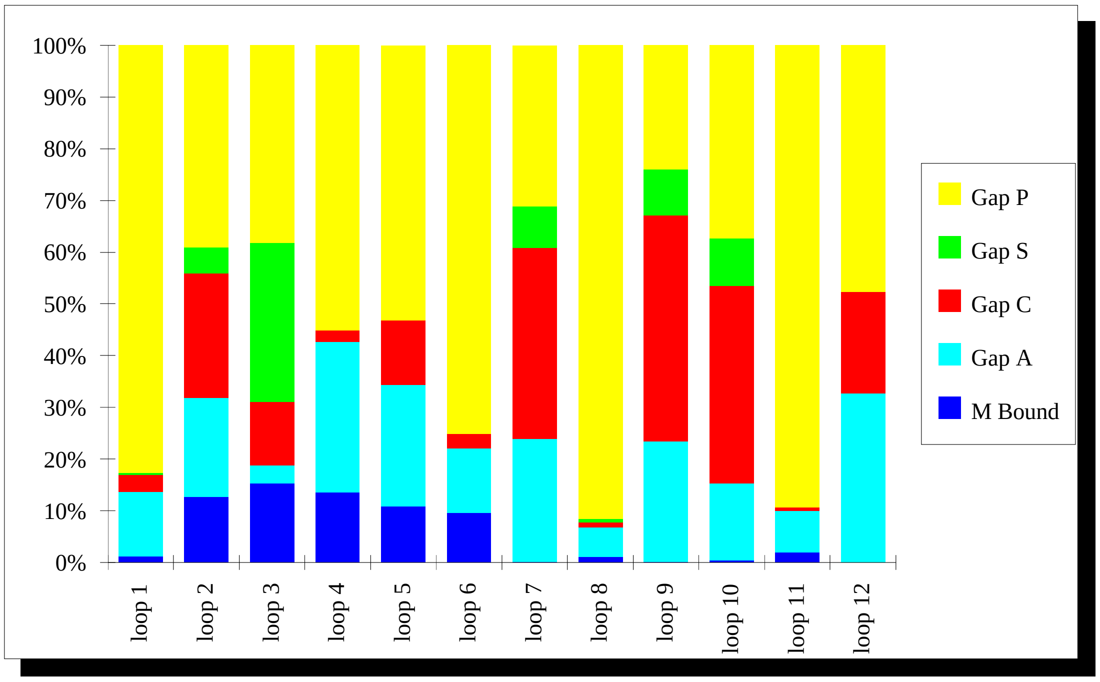


Figure 15: MACS gaps for 20 processors on the KSR1

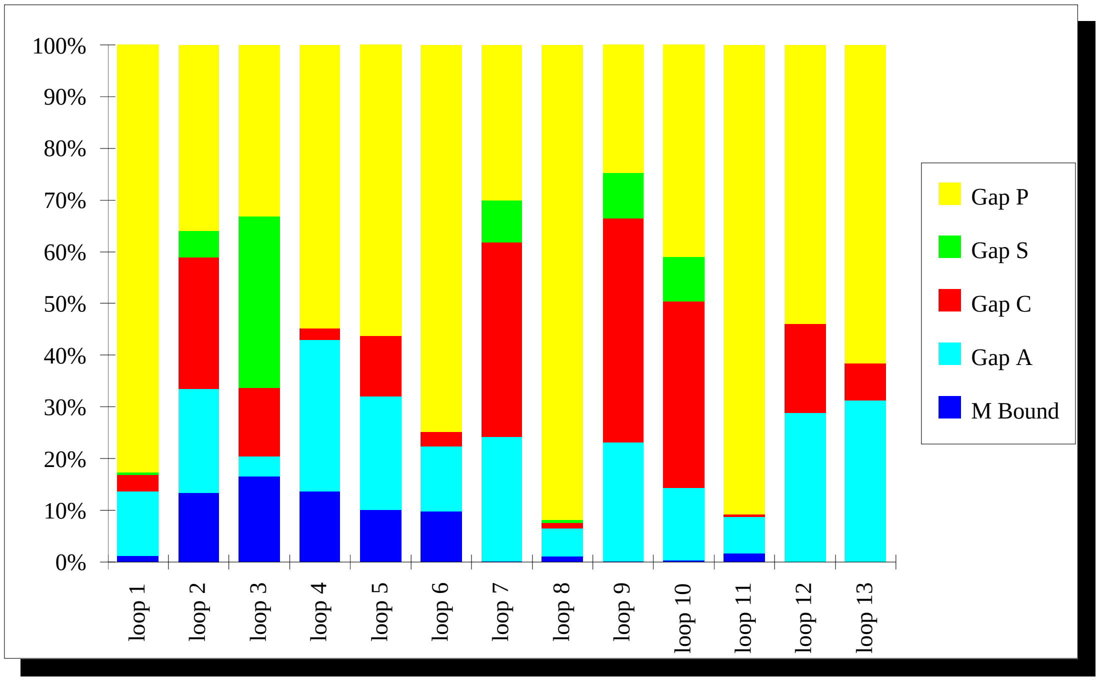


Figure 16: MACS gaps for 25 processors on the KSR1

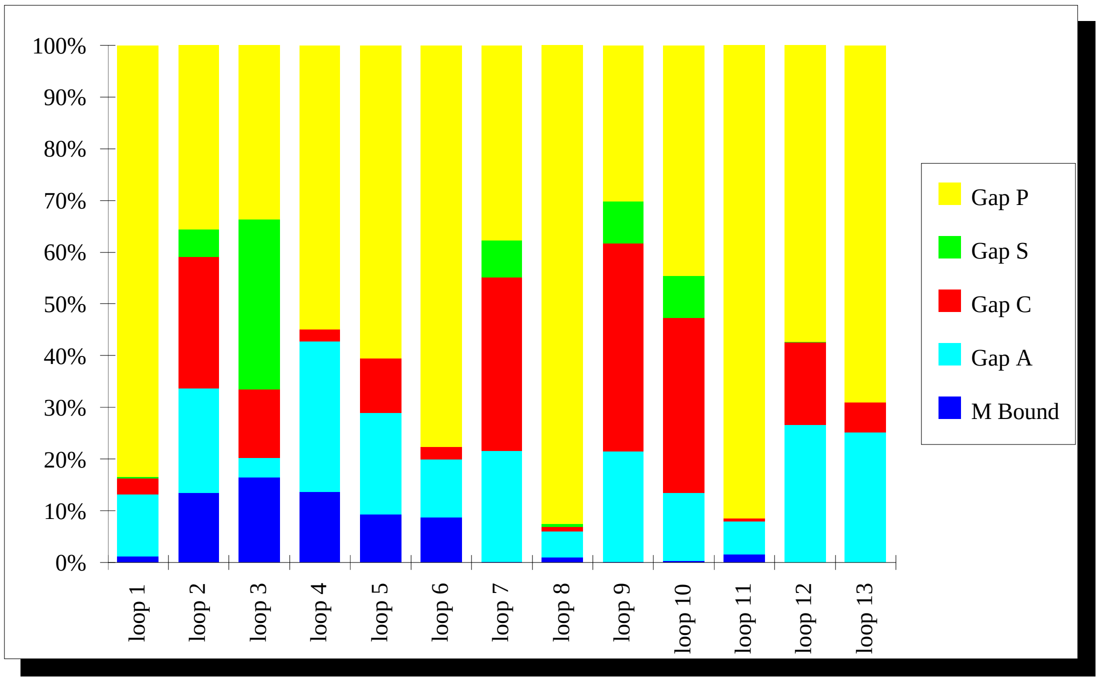


Figure 17: MACS gaps for 28 processors on the KSR1

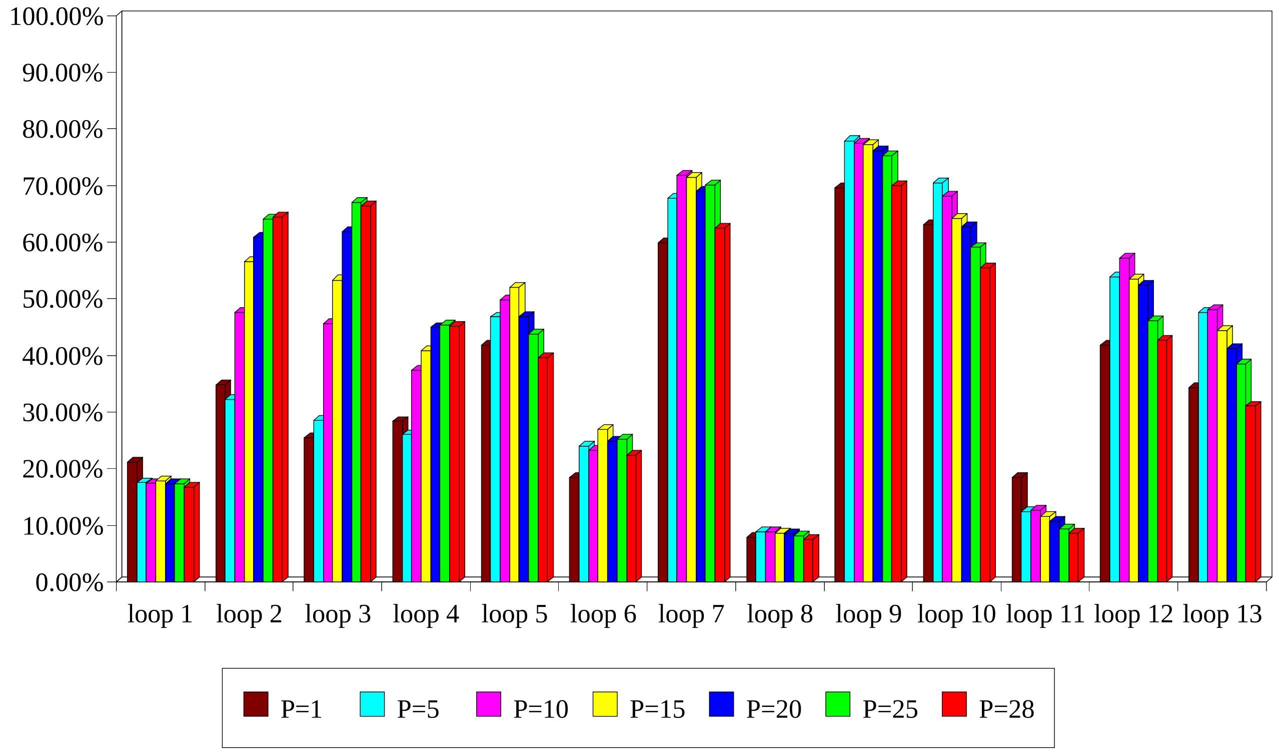


Figure 18: CPU-averaged MUF for the 13 loops on the KSR1

Equation 7.1. The average MUF over all processors for each of the 13 loops is plotted in Figure 18.

$$machine\ utilization\ factor\ (MUF) = \frac{t_{MACS}}{t_{MACS} + t_{GAP\_P}} = 1 - (Gap\_P)\% \qquad (7.1)$$

It is obvious from the data shown in Figure 18 that the MUF is not a simple function of the number of processors. The MUF often improves as p is increased beyond one; however, it degrades in many cases when the number of processors increases beyond some value. This value varies for different loops. Most loops also show other non-monotonic behaviors at same point. For a given loop to utilize the machine with its best efficiency, we can select the number of processors, p, with the maximum MUF. However, if further speed-up is needed, users may have to apply more processors and simply suffer the loss in the processor efficiency. Compromises may have to be made if there are data locality problems with using a different number of processors for different loops. The speed-up for each of the 13 loops, shown in Figure 19, is obtained from the ratio of the measured run time with a particular number of processors to that of uniprocessor. Some speed-up values, e.g. for *loops 12* and *13*, are greater than the number of processors due to the effect of a large number of cache capacity misses in uniprocessor execution.

In summary, MUF and speedup plots (Figures 18 and 19) are useful for choosing the most effective number of processors for an existing code and the bounds hierarchy bar charts (Figures 11

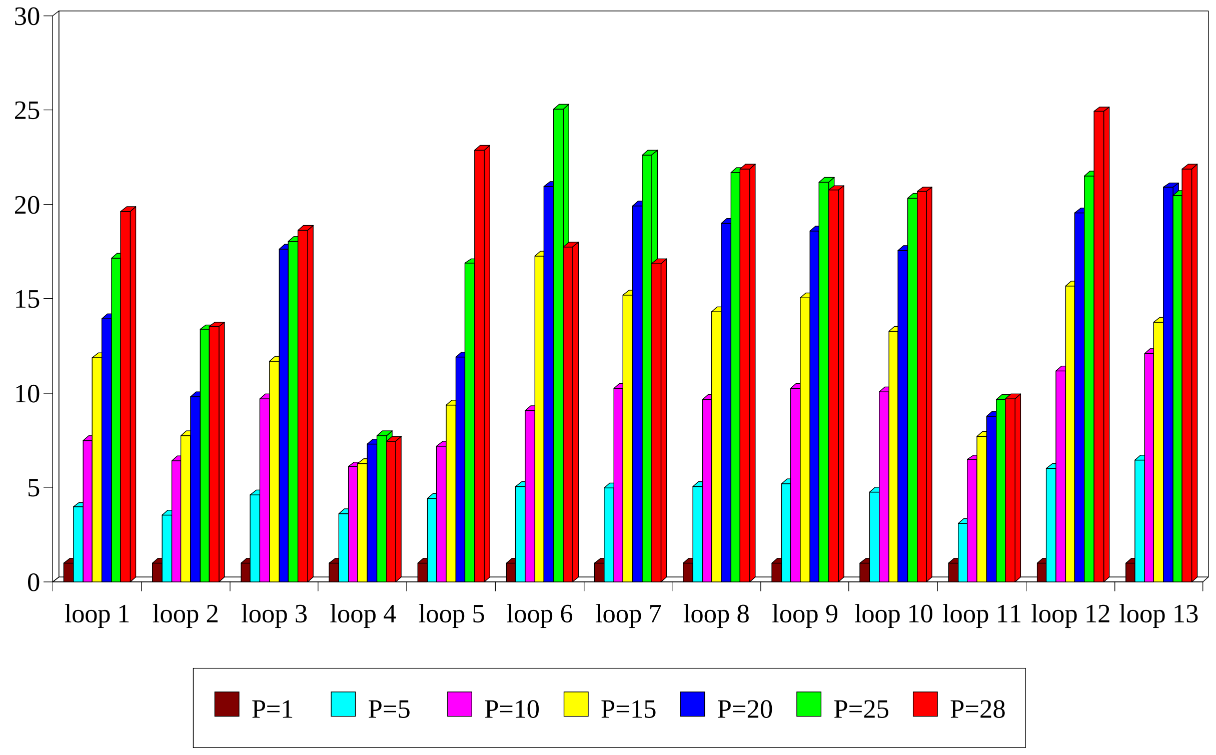


Figure 19: Speed-up of the 13 loops for different number of processors on the KSR1

through 17) are useful for gap closing to improve performance and achieve more scalability of the code to larger numbers of processors.

8 Conclusion

We have presented an overview of the KSR1 architecture, a precise description of the MACS hierarchical performance bounds for single and multiple KSR processor runs, and a discussion of the principle causes and cures for the various performance gaps illuminated by the MACS bound hierarchy. We also demonstrated experimental evidence of the use of the bound model and the automatic evaluation tools we developed on a sophisticated parallel application running on the KSR1. We can determine from the distribution of performance bounds and gaps that are quantitatively shown by the normalized bar charts how large a portion of the run time is contributed by each source, such as the high level code, compiler, code scheduler, cache misses and internodal communications. Then we can use this understanding to select and apply available techniques to reduce the specific performance gaps. Because communication including local cache misses and internodal communication is more critical in shared-memory systems like the KSR1, the tools were designed to illuminate the number of clock cycles spent for communication per iteration of a particular loop. We also presented an analysis concept on machine utilization and quality of loops parallelism, which can give us more insights in improving the processor efficiency as well as selecting an effective number of processors for a parallel application.

9 Future Work

From the experimental results shown in Section 7, a rather significant portion of time is spent in *Gap P*. Most of the execution time of the application is consumed in communication rather than in computation. For understanding *Gap P*, more explicit information on communication protocols, data allocation strategies and data transactions is demanded for further investigation of the performance gap.

9.1 Shared-Memory Multiprocessor Systems

Because the data movements in a COMA system are totally handled by hardware, with at most a software assist, communication behavior must be observed through hardware monitoring, hierarchical cache simulation and/or communication analysis. Communication penalties on the KSR1 may consist of subcache misses, internodal communication for local cache misses, idle processors due to barrier synchronizations, data multicasts for maintaining data coherence, network accesses contention, ..., etc. There are two experimental tools developed by our research group for the purpose of simulating the communication behavior. *K-Trace*[4] is a tracing tool for the KSR1 multiple processor system. It can be instrumented within a program region of interest and generates an individual memory accesses trace for each processor that executes in that region. Having an explicit full trace of memory accesses on an application code, we can perform cache simulations at various levels. *K-LCache* was developed to analyze the data communication between processors on a KSR ring based on some assumptions for a simplified local cache model. It reports a lower bound and an upper bound on local cache misses.

Further subdividing *Gap P* can allow construction of a hierarchical performance model of communication penalties for shared-memory multiprocessor systems. Performance degradation caused from load imbalance, excess data migration, and inefficient data layout should be further investigated and understood according to the subdivision of communication performance. A complete hierarchical performance bound model combining computation with internodal communication would help us to understand more about the performance loss in an entire system.

9.2 Message-Passing Multicomputers

In a message-passing system, communication between processors relies on explicit passing of messages by the software. All communication behavior is thus exposed explicitly to high level code programmers. Therefore, it becomes the responsibility of programmers of these systems to specify the necessary communication transactions for every data element in the high level code, *i.e.*

[4] *K-Trace* and *K-LCache* were developed by Shih-Hao Hung

programmers must know when and where a data element is updated and where and when it needs to be sent to those nodes that own stale data copies in order to achieve a consistent execution of a program. This work, all needed to be done by hand, is very elaborate and prone to error. A systematic method is proposed here to simplify the work on specifying the routines for sending and receiving messages, where they need to be used and what data they need to transmit. Through this method, programmers will find it much easier to write a message-passing code from trace results collected by a shared-memory system.

Suppose that a shared-memory version code is available to run on a particular shared-memory multiprocessor system and a parallel tracing tool is also provided by the same system. From the full traces generated by running an application on some particular number of processors, each accessed memory address can be associated with its corresponding data element, and the accessing region of code for each request processor can be discovered automatically. Each accessing region could be either a sequential execution region or a synchronization region between barrier check-in and check-out of a parallel execution. The explicit data structure movements needed for a message-passing code will therefore be totally known from simply observing the data accesses behavior reported by *K-Trace*. A new tool is being developed for forward tracking the multi-thread traces based on each data item recorded in trace. For each data element, either the full data-flow path or the data migration that occurs between distinct processors will be reported. Every data element movement reported by this tool becomes an essential data movement in a message-passing system.

Packing as many data elements together as many as possible and sending them in one message would reduce the number of message and the control information needed to be transferred to a minimum. Deriving an efficient algorithm for merging the data elements into a minimum number of messages is one goal of our future work. In this manner, we hope to be able to port parallel shared-memory application codes efficiently to message-passing systems in a highly automated manner.

References

[1] 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*, pp.1166-1178, August, 1993.

[2] 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*, pp.203-212, May, 1993.

[3] G. Bell, "Ultracomputers: A Teraflop Before Its Time," *Communications of the ACM*, pp.27-47, August, 1992.

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

[5] E. L. Boyd, W. Azeem, H-H. Lee, T-P. Shih, S-H. Hung, E. S. Davidson, "A Hierarchical Approach to Modeling and Improving the Performance of Scientific Applications on the KSR1,"

Proceedings of International Conference on Parallel Processing, vol.III, pp.188-192, August, 1994.

[6] *KSR1 Principles of Operation*, Kendall Square Research Corp., October, 1992.

[7] "Kendall Square Multiprocessor: Early experiences and performance", Oak Ridge National Laboratory Technical Report ORNL/TM-12065, April, 1992.

[8] K. Hwang, Advanced Computer Architecture: Parallelism, Scalability, Programmability, McGraw-Hill Inc., 1993.

[9] W. Meleis, "Reducing Memory Bandwidth Requirement in Scientific Code," Directed Study, University of Michigan, 1992.

[10] W. H. Mangione-Smith, "Performance Bounds and Buffer Space Requirements for Concurrent Processors," Ph.D. dissertation, University of Michigan, 1991.

[11] U. Banerjee, S-C. Chen, D. J. Kuck, R. A. Towle, "Time and Parallel Processor Bounds for Fortran-Like Loops," *IEEE Transactions on Computers*, pp.660-670, September, 1979.

[12] T-P. Shih, E. S. Davidson, "Grouping Array Layouts to Reduce Communication and Improve Locality of Parallel Programs," submitted to *1994 International Conference on Parallel and Distributd Systems.*

[13] H. Zima, B. Chapman, Supercompilers for Parallel and Vector Machines, Addison-Wesley, ACM Press, New York, 1991

APPENDIX: *K-MACSTAT*

A Algorithm

K-MACSTAT is one of the automatic tools developed in our hierarchical *MACS* performance bounding methodology that attempts to model and explain the performance of loop-dominated scientific applications on particular systems. *K-MACSTAT* is a single pass forward scan program written in *awk* script for generating the parameters used in MAC and $MACS$ bounds computation for the KSR machines automatically. *K-MACSTAT* reads in a section of KSR assembly code as its input, scans the code, analyzes the loops inside, and generates the counts which will be applied in computing MAC and $MACS$ bounds.

A.1 Determine a branch and the correlation of branches

It is supposed that the codes as inputs are well-structured, all backward branches should be considered as defining loops.

When a label is read from the line of the given codes, it is assigned an index, *nc*, and the label name will be recorded, then this program starts to count the number of operations for the future scanned lines. When a label of a backward branch operation is encountered, its label for branch target is compared with the indexed labels that have been recorded. If a match is found, *i.e.* the branch is a backward branch with a target within the scanned area of the code, a new index term, *rnc*, and a new set of variables is created and associated with this *rnc*. These variables will assign the current counts for the target label in them for this loop. An example is illustrated in figure 20.

The relations among all loops with nested structures can be determined by examining both *rnc* and *nc* values. The *nc* of one loop is less than or equal to that of another one, which means this loop begins before the latter one. If the former loop has a larger *rnc*, then it indicates the latter one finishes before the former one. Therefore, we can recognize the latter one is an inner loop of the former one. Our algorithm is shown in figure 21.

At a given point in the scan, there is a set of counts for each label (*nc*) encountered thus far. Each set of counts reports the number of each type of instructions found between the associated label and the current line of the scan.

A.2 Determine an innermost loop

A variable called *label_complete(i)* is employed for indicating whether any loop whose label is associated with $nc = i$. has completed. *Label_complete* is initialized to the $NULL$ state when label i is encountered during the scan. When a branch taken back to label i is found, *label_complete(i)* is assigned to the *YES* state. A loop is innermost if the backward branch which terminates the loop has a target with index i and *label_complete(j)* $= NULL$ for each j that is greater than or equal to i. This check is done before *label_complete(i)* can be changed to *YES* by this loop. If any label has

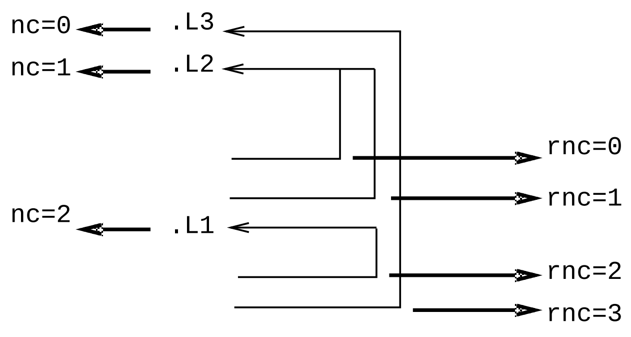


Figure 20: Target labels of a multi-nested loop

```
for (i=0; i<rnc-1; i++)
begin
  for (j=i+1; j<rnc; j++)
      begin
        if (nc_of_label[j] <= nc_of_label[i]) then
           loop[i] is an inner loop of loop[j]
        end if
      end
end
```

Figure 21: Algorithm for determining correlation of nested loops

```
tmp_begin = current_loop_begin[rnc]
  for (m=0; m<rnc; m++) /* checking those completed loops */
  begin
    if (tmp_begin < any_completed_loop_end[m] &&
        tmp_begin > any_completed_loop_begin[m]) then
          current_loop[rnc] is an intercross loop
  end
```

Figure 22: Algorithm for identifying non-nested loops

been ever branched backwards, it indicates there are some inner loops existing inside this checked loop. If none has been branched backwards, it means this loop is the first one to be branched backwards during the interval of the line issuing the label and the current scanned line. Thus, it can be determined as an innermost loop.

A.3 Identify non-nested loops

For identifying a non-nested loop which is unacceptable in the *K-MACSTAT*, it is necessary to check whether the target label of the current scanned branch lies inside any of the other loop bodies. Accordingly, two variables, *real_loop_begin[rnc]* and *real_loop_end[rnc]*, are introduced to keep the starting line number and the ending line number for a completed loop thus far. If the line number of the target label of the current scanned backward branch is within the range of the starting line number and the ending line number of any completed loops, then this current loop can be determined as an non-nested loop. The algorithm is depicted in figure 22.

A.4 Delay Slot

The branch squashing technique is used in KSR architecture. This idea is to allow the branch to indicate that the instructions in the delay slots should be aborted if the branch is mispredicted. The number of delay slots is two on the KSR, therefore, in the *K-MACSTAT* implementation, we cannot ignore the next two successor instruction pairs following those branch instructions since they might be within the execution stream.

The instruction pairs in the delay slot are always to be executed for unconditional branches such as *jmp*. In consequence, *K-MACSTAT* always counts the next two successor instruction pairs as they have to be executed for the unconditional branch instructions of the KSR. For conditional branches, there are three types, quash-never(qn), quash-true(qt), and quash-false (qf). Because what we are interested in are the instructions inside a loop, when a backward branch is taken, it means that the loop continues. Thus, it is supposed that the backward branch for a loop body is always taken for counting the actual instructions inside a loop, then *K-MACSTAT* counts the next two successor instruction pairs while quash-false or quash-never branches are taken and does not

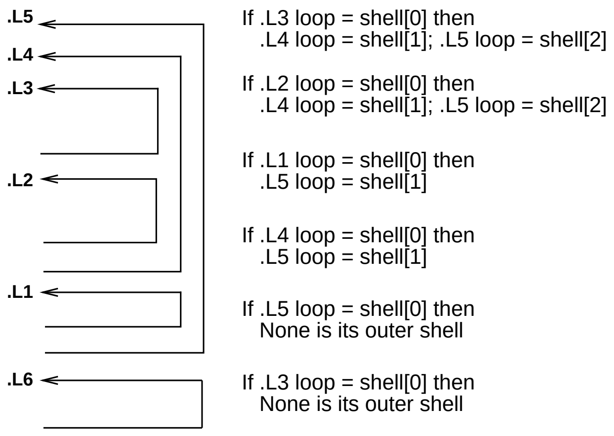

.L5
.L4
.L3
.L2
.L1
.L6
If .L3 loop = shell[0] then
.L4 loop = shell[1]; .L5 loop = shell[2]
If .L2 loop = shell[0] then
.L4 loop = shell[1]; .L5 loop = shell[2]
If .L1 loop = shell[0] then
.L5 loop = shell[1]
If .L4 loop = shell[0] then
.L5 loop = shell[1]
If .L5 loop = shell[0] then
None is its outer shell
If .L3 loop = shell[0] then
None is its outer shell

Figure 23: Statistics report of multiple loops in distinct shells

count the next two successor instruction pairs while quash-true branches are taken.

When a branch is encountered, it is necessary to identify the branch type for later use in order to count those particular types of instructions such as f_a, f_m, f_{ma},.., etc., which occur within the successors instruction pairs after the conditional branch instructions. Then we continue to scan the next instruction pair and check if its previous instruction or previous second instruction contains an unconditional branch, a quash-never or a quash-false branch. If any of them exists, the type of the current instruction has to be identified, categorized and added into our current count. Two variables are needed to preserve the branch types since the delay slot is only two.

A.5 Generate the statistics for the residue parts of nested loops

The whole span of code can be imagined as containing multiple shells from the innermost loops to outermost loops. An outer shell relative to the current processing shell whose index for shell is 0 is indexed by increment the shell index of the current processing shell. In figure 23, an example is illustrated for the shell structures of a multiple loops.

Since we record the count of every loop body for whole span of the codes no matter whether it is an innermost loop or not. To compute the counts of the parameters for the residues of a non-innermost nested loop, we have to subtract the count of the codes of those loops lying inside it. While each branch instruction of a loop is being processed, we subtract the counts of the outer loops containing this processed loop by the count of this processed loop codes. An example of the mechanism can be found in figure 10 of the context of this paper.

The calculations are proceeded in the order of *rnc* after all of the loops inside the given codes have already been identified. Each time we subtract the codes of the current processing loop body. (we do nothing with the innermost loops.) Because of proceeding in the *rnc* order, the innermost loops will always be processed before its outer loops are processed. Therefore, we can derive the counts for each shell from inside out. The algorithm is shown in figure 24.

```
for (i=0; i<rnc-1; i++)
begin
  sh=0
  for (j=i+1; j<rnc; j++)
  begin
    if (nc_of_label[j] <= nc_of_label[i]) then
    begin
      tmp_loop_shell[sh]=j
      /* for recording the indice of outer loops of the currently processed loop */
      sh = sh + 1
    end
  end
  for (k=0; k<sh; k++)
  begin
    sh_out = tmp_loop_shell[k]
    count_of_parameters[sh_out] = count_of_parameters[sh_out] - count_of_parameters[i]
  end
end
```

Figure 24: Algorithm for enumerating residue parts of the codes

A.6 Report conditional forward branches in innermost loops

In usual, there are some conditional branches such as *if-then-else* statements showing up inside a loop body. These branches will complicate us to compute the MAC and MACS bounds, especially while loops are unrolled by the optimized compiler. Therefore, it would be helpful for us to compute the bounds if the results of the enumeration of these conditional branches inside loops are reported. In computing the MAC and $MACS$ bounds, those innermost loops will be the most interesting to us, thus *K-MACSTAT* will report only the forward branch parts inside those innermost loops. Whenever a target label of a branch instruction has notyet been encountered before, it is defined and identified as a forward branch. Then it starts to count the set of the parameters depending on the branch types until its target label has been reached. If it is a quash-true branch, it starts to enumerate immediately after this branch instruction. If it is a unconditional jump, quash-never, or quash-false branch, it starts the counting for two instruction delays due to the two delay slots in the KSR architecture. The results will be reported within the innermost loop body which it belongs to and divided into several areas if there are more than one branch inside the loop due to multiple forward branches or single forward branch with loop unrolling. An example is provided in figure 9 of the context of this paper.

B Usage and restrictions

B.1 Usage

Two scripts version of *K-MACSTAT* are provided. One is written in *Awk* script while the other one is in *Perl* script. First of all, compile the high-level code parts which you intend to analyze into a KSR assembly code file.

For the *Awk* version, execute the following command line after your UNIX prompt.

```
sigmund% awk -f MACSTAT.awk target.s > targer.MACS
```

For the *Perl* version, execute the following command line.

```
sigmund% MACSTAT.perl target.s > target.MACS
```

Then the output will be directed into the designated file, `target.MACS`.

B.2 Restrictions

Subroutine calls which are not inlined into the compiled assembly code will be ignored in the bounds collection by *K-MACSTAT*. If you wish to acquire the actual loop bounds of the innermost loops containing subroutine calls, you have to compute the bounds from the lowest level of the call tree in the application, then accumulate the bound statistics hierarchically. Another restriction upon unacceptable loop structures was discussed in Section 6.2.

B.3 Output reading

Figure 25 shows an output example generated by *K-MACSTAT*. The first part reports the loops correlation, and the main part reports the *MACS* counts for each loop. At the end of the output, the number of loops encountered will be reported. In this example, as reported in the first line of figure 25, there is a perfectly nested pair of loops as reported in the first line. The inner loop is labeled as .L37 while the outer loop is labeled as .L19. There is a forward branch labeled .L17 within inner loop .L37. If this branch is taken in execution, the *MACS* counts in the forward branch region (listed as `Forward branch area 1` which is skipped when .L17 is taken) have to be subtracted from the *MACS* counts of loop .L37. Notice that `Load` and `Store` in the output are for all categories of load and store instructions while `Sfl` and `Lfl` only account for the number of floating-point load and store instructions.

```
.L37 is an inner loop of .L19

Listed by loop completed order inside the machine code
--------------------------------------------------------------------------
Loop labelled .L37-- Branch Region = 0 --
--> An inner-most loop
FPU/IPU instructions except nop = 43
CEU/XIU instructions except nop = 96
KSR compiled codes length = 107 instructions

Fa = 8 Fm = 0 Fma = 4 Fmisc/Other FPU = 12 Lfl = 28 Sfl = 20
Load = 28 Store = 20

Floating-point operations = 28
Memory accesses = 48
Fma+Sfl (Conflict on FPUC)= 24

Forward branch area 1
Forward Branch exists in backward branch region -- 0 --
*************************************************************************
*Target Label = .L17 99 107
* FPU/IPU instructions except nop = 3
* CEU/XIU instructions except nop = 5
* KSR compiled codes length = 7 instructions
* Fa = 0 Fm = 0 Fma = 1 Fmisc/Other FPU = 1 Lfl = 2 Sfl = 3
* Load = 2 Store = 3
*
* Floating-point operations= 3
* Memory accesses = 5
* Fma+Sfl (Conflict on FPUC)= 4
*************************************************************************
--------------------------------------------------------------------------
Loop labelled .L19-- Branch Region = 1 --
--> A residue of an outer loop
FPU/IPU instructions except nop = 52
CEU/XIU instructions except nop = 101
KSR compiled codes length = 114 instructions

Fa = 7 Fm = 0 Fma = 4 Fmisc/Other FPU = 9 Lfl = 25 Sfl = 17
Load = 25 Store = 17

Floating-point operations = 24
Memory accesses = 42
Fma+Sfl (Conflict on FPUC)= 21
--------------------------------------------------------------------------
Number of loops = 2
```

Figure 25: Output example of *K-MACSTAT*