# MODELING AND APPROACHING THE DELIVERABLE PERFORMANCE CAPABILITY OF THE KSR1 PROCESSOR

by

Waqar Azeem

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Engineering
(Computer Science and Engineering)
in The University of Michigan
1993

# TABLE OF CONTENTS

**TABLE OF CONTENTS** i
**LIST OF FIGURES** iii
**LIST OF TABLES** iv
**ACKNOWLEDGEMENTS** v
**CHAPTER**

**I. INTRODUCTION** 1
- 1.1 Problem Definition and Background 1

**II. THE KSR1 PROCESSOR** 3
- 2.1 Functional Units 3
- 2.2 Instruction Scheduling and Timing 4
- 2.3 Scheduling Restrictions 8
- 2.4 Summary of Schedule Conflicts 10

**III. PERFORMANCE BOUNDS MODEL** 12
- 3.1 Formulation of the Bounds Equations 12
- 3.2 The Application Workload and the MA Bound 14
- 3.3 Performance of the compiled code and the MAC and MACS Bounds 15
- 3.4 Sources of Performance Loss 19
  - 3.4.1 Loop 2 20
  - 3.4.2 Loop 6 20
  - 3.4.3 Loop 7 21

**IV. AUTOMATIC CODE RESCHEDULING** 22
- 4.1 The Object Code Optimizer (OCO) 22
- 4.2 Performance of OCO Generated Code 24
  - 4.2.1 Estimating execution time for Loop 6 25
  - 4.2.2 Estimating execution time for Loop 2 26
- 4.3 The Template Packing Problem of the KSR1 26
- 4.4 Shortcomings of List Scheduling for the KSR1 28

**V. APPROACHING THE MA BOUND** 29
- 5.1 Scheduling a Loop Body 29
- 5.2 Redundant Memory References 29
  - 5.2.1 Loop 2 30
  - 5.2.2 Loop 8 33
  - 5.2.3 Loop 6 34
- 5.3 Redundant Register Move Operations 35
- 5.4 Performance of Hand-Coded loops 35

5.5 Steady-State Inner-Loop performance 37

**VI. CONCLUSIONS** **39**

**REFERENCES** **41**

# LIST OF FIGURES

Figure
3.1 % of MA, MAC, MACS bounds achieved by the compiled code 19
4.1 Stages followed by the Object Code Optimizer 23
4.2 Performance of compiler-scheduled vs. OCO-scheduled code 25
4.3 Example code segment 27
4.4 Another example code segment 28
5.1 Compiler-generated code for loop 2 31
5.2 Rescheduled code for loop 2 33
5.3 A section of compiler-generated code for loop 6 34
5.4 A section of rescheduled code for loop 6 34
5.5 Performance results for the modified LFK 1-12 36
5.6 % of MA bound achieved by rescheduled code after discounting outer loop overhead in LFK 2, 4 and 6 38

# LIST OF TABLES

| Table | | |
|---|---|---|
| 2.1 | Timing for some floating point execute instructions | 5 |
| 2.2 | Reservation template for floating point instructions of the type X+Y->Z and X-Y->Z | 6 |
| 2.3 | Reservation template for triad instructions of the form (X-Y)*Z and (X+Y)*Z | 7 |
| 2.4 | Timing for some floating point memory reference instructions | 7 |
| 2.5 | Reservation template for a floating point load | 8 |
| 2.6 | Reservation table for the code segment in example 2.2 | 9 |
| 2.7 | Reservation table for the code segment in example 2.3 | 10 |
| 3.1 | Essential floating point execute and memory operations in LFK 1-12 | 14 |
| 3.2 | The number of clocks required per essential floating point operation for LFK 1-12 | 15 |
| 3.3 | Performance of LFK 1-12 | 16 |
| 3.4 | Actual workload generated by the compiler for LFK 1-12 | 17 |
| 3.5 | Performance of LFK 1-12 compared with the MAC and MACS bounds | 18 |
| 4.1 | Performance of code produced by OCO | 24 |
| 4.2 | Reservation table for code segment in figure 4.3 | 27 |
| 4.3 | Reservation table for issue order of 1, 3, 2, 4 | 28 |
| 4.4 | Reservation table for issue order of 1, 2, 3, 4 | 28 |
| 5.1 | Comparison of hand-coded vs. compiled LFK 1-12 | 36 |
| 5.2 | Performance of LFK 2, 4 and 6 after eliminating outer loop overhead | 37 |

## ACKNOWLEDGEMENTS

I would like to thank my advisor Edward S. Davidson for his patience and able guidance. Daniel Windheiser's comments and suggestions influenced a major portion of this study. I am indebted to him for that and all the consultations regarding OCO. Discussions with Tien-Pao Shih regarding his performance evaluation of the IBM RS/6000 also proved to be invaluable. Chris Reeves, who directs the compiler efforts at KSR, provided helpful insights into some aspects of the KSR1 compiler.

This work was funded in part by a grant from the Ford Motor Company. The University of Michigan Center for Parallel Computing, which is partially funded by NSF grant CDA-92-14296, provided access to the KSR1.

# CHAPTER I
# INTRODUCTION

## 1.1 Problem Definition and Background

Being able to evaluate and predict the performance of an architecture for an application is an extremely important capability which has been the focus of attention of a number of researchers in the scientific computing community. It allows a distinction to be made between the advertised peak performance of a system and its deliverable performance. The peak performance is what the manufacturer claims for an ideal application which fully utilizes a key resource of the system, *e.g.* the floating point unit (MFLOPS), the instruction issue unit (MIPS), or the memory ports (memory bandwidth). Peak performance is typically much higher than the deliverable performance of a real application which may not be ideal. Furthermore actual delivered performance, as provided by benchmarking, is typically much less than deliverable performance since the benchmark code may be poorly compiled. It is therefore important to identify the performance delivering capability of a machine for real applications. Even more important, for improving performance, is the task of isolating the factors that are responsible for the gaps between peak, architecturally deliverable, and actually delivered performance.

We use an analytic machine-application performance bounding methodology to produce a performance model for the KSR1 [1] [2] processor which upper bounds the best achievable performance. This model evaluates the throughput of those units in the architecture that are common performance bottlenecks. Four units are included in this model to assess their individual workloads in the application and to examine how well the available parallelism between functional units is exploited by the application code. These units are: instruction issue, floating-point, memory and a dependence pseudo-unit which models loop carried recurrences. Using this model we can predict the best steady-state loop performance that the KSR1 processor is able to deliver for a given application. A similar performance evaluation strategy has been used for the vector supercomputer Cray systems [3], the superscalar IBM RS-6000 [4] [5], the decoupled access-execute Astronautics ZS-1 [6] [7], and the vector mini-supercomputer Convex C-240 [8].

Since scientific programs are typically loop dominated, we use the first twelve Livermore Fortran Kernels (LFK) to assess the steady-state inner-loop performance of the KSR1. This benchmark set provides us with an opportunity to examine a variety of loops. Each of these is different in character from the others and yet is small enough to be treated individually in detail. A large portion of the execution time is spent in such well-structured inner loops which is an incentive to invest time and effort in isolating performance bottlenecks and discovering generally applicable code optimization techniques for them.

A hierarchy of bounds equations exists [8], where a succession of performance models is developed based on a Machine and high level Application code of interest (MA), the Compiler-generated workload (MAC) and the actual compiler-generated Schedule for this workload (MACS). We make use of this hierarchical approach to individually evaluate the efficacy of the data flow analysis and the code scheduling phases of the compiler and identify their shortcomings. This kind of hierarchical analysis exposes specific performance gaps that are extremely useful for identifying bottlenecks in the machine and weaknesses in the compiler. Restructuring techniques that potentially result in the greatest performance gains are selected according to which gaps are the largest. This approach can be implemented within a compiler for general use.

We present techniques for approaching the performance bound by both automatic and manual code rescheduling. The automatic rescheduling is performed by adapting the Object Code Optimizer (OCO) -- a tool developed by Daniel Windheiser at IRISA, France, for the i860, MIPS and SPARC architectures -- to the KSR1 architecture. For a given application code, this tool selects the relevant reservation table templates and attempts to pack the instructions optimally, using the polycyclic scheduling technique [9] (a.k.a cyclic scheduling or software pipelining [10] [11] [12] [13]). We also make use of hand-coding to create schedules that approach the performance bounds given by our model.

In chapter 2, we introduce the KSR1 processor architecture and discuss scheduling constraints. We present the performance bounds model and the performance of the compiled code for the workload presented by LFK 1-12 in chapter 3. In chapter 4, the automatic code restructuring tool, OCO, is presented along with the performance gains resulting from its use. In chapter 5, we present the approach used to hand code the loops and discuss the steady-state inner loop performance achieved. Chapter 6 states the general conclusions drawn from this work.

# CHAPTER II
# THE KSR1 PROCESSOR

## 2.1 Functional Units [1] [2]

The KSR1 processor is a general purpose 64-bit custom VLSI RISC processor that is capable of executing multiple instructions during each clock cycle. As in most RISC architectures, only the load and store operations reference memory while all other instructions use operands residing in registers. There are four chips in each processor cell: the Floating Point Unit (FPU), the Cell Execution Unit (CEU), the Integer Processing Unit (IPU) and the eXternal I/O Unit (XIU).

The FPU executes all floating point instructions and has 64 floating point registers which are each 64 bits wide. The floating point register file has three read and two write ports. This number of registers is large enough to enable the compiler, in most cases, to keep the operands in the registers for as long as they are alive within a typical scientific inner loop construct, *e.g.* a FORTRAN DO-loop. The FPU instructions include eight linked triad instructions which allow the processor to issue two floating point operations in a single cycle. This is possible when an add (subtract) operation immediately uses the result of a preceding multiply operation or vice versa. Since KSR instructions contain three operand fields, the result of a triad instruction must be stored back into one of the source registers. The clock rate for the processor is 20 MHz, which gives the processor a peak floating point rate of 40 MFLOPS.

The CEU has 32 address registers, each 40 bits wide, and is the control unit of the processor. It fetches an instruction pair from memory during each clock cycle and executes all load, store, address arithmetic and branch instructions. It performs address generation, and is responsible for handling branches to insure the integrity of the control flow.

The IPU has 32 registers, each of which is 64 bits wide. It performs arithmetic and logical operations on 64 bit integers in these registers. Finally, the XIU performs external I/O and DMA by providing a 30 MB/sec pathway to peripheral devices. The XIU has 64 I/O control and data registers.

The KSR1 processor can issue two instructions in a single clock cycle. Code is compiled into instruction pairs and emitted in a VLIW type format. The right instruction of a pair goes to either the CEU or the XIU, and the left instruction of the pair goes to either the FPU or the IPU. Thus it is possible to initiate an address calculation, memory instruction, branch, or I/O instruction on the CEU/XIU side in each cycle, while on the FPU/IPU side a floating point or integer execute instruction can be initiated in each cycle. Due to the presence of linked triad instructions, it is possible to initiate two floating point operations in one clock cycle. Two flavors of *nop* instructions exist; one for each side of the instruction pair. The *finop* is the no operation instruction for the FPU/IPU side, while the *cxnop* is its counterpart on the CEU/XIU side.

All functional units are fully pipelined, *i.e.* each functional unit can start a new operation on every clock cycle. Since most functional units have multiple stages, the availability of the results of certain operations has an associated multi-cycle latency. The processor does not have any hardware interlocks[1], so it is up to the compiler to schedule all the instructions statically, while ensuring that two dependent instructions are separated by a number of clock cycles that is at least as large as the latency for that dependent pair. If no independent instructions are found to fill this gap, *nop* instructions are inserted by the

[1] The KSR1 processor stalls only on a cache miss.

compiler.

All execute and control flow instructions on the KSR1 are performed between registers. The KSR's three-address instruction set architecture allows reuse of operands that already reside in registers. However, in the case of linked triad instructions, one of the source registers must also be used as the destination register, which may necessitate some copying or reloading of operands for later use in some of the loops we examined. Performance may be degraded by these added operations.

Each KSR processor has two levels of private cache. Level 1 consists of a 0.25 MB *data subcache* and a 0.25 MB *instruction subcache*. The subcache is 2-way associative and uses a random replacement, write-back policy. Each processor contains four cache control units (CCU) that manage the local cache. The CCU allocates space in the subcache on a block basis. The unit of data transfer from the second level cache is a subblock. A block is 2048 bytes and contains 32 subblocks, each of which is 64 bytes. The data subcache is large enough to eliminate the effect of cache misses from our timing measurements on the workload in this study.

The second level cache, called the *local cache*, is 32 MB in size. It is 16-way set associative and the replacement policy is LRU. The unit of allocation in the second level cache is a 16KB virtual memory page. Remote caches, the local caches of other processors, are accessed through a unidirectional, slotted ring interconnect and data is transferred between these caches in units of 128 byte subpages. Each ring can have up to 32 processors and communication with the caches of processors on other rings takes places though a higher level ring. This hierarchical ring interconnect structure is expected to provide scalability for large systems.

## 2.2 Instruction Scheduling and Timing [1]

Each instruction makes use of certain operand sources and execution resources. The sources refer to operand registers and the ports through which they are read. The resources are the functional units, register write ports and memory load and store ports. Instructions in execution reserve required resources and source register read ports for their exclusive use. The placement of instructions in a schedule must satisfy scheduling restrictions that insure that *i)* no resource or source register read port is reserved by two or more instructions at the same time, and *ii)* two dependent instructions are separated by a distance no less than the latency of that instruction pair. This section defines the exclusive reservations and latencies associated with particular instructions. The following section illustrates some scheduling restrictions.

The next instruction pair from the schedule is issued in each cycle, unless the processor is stalled. Instruction scheduling is carried out assuming that no stalls occur; a legal schedule will remain legal in the presence of stalls. A *nop* appears in the left and/or right side of the pair wherever no new work is scheduled for the functional units. KSR uses the term *launch* to refer to the first cycle in execution of instructions that are not *nops*. Thus in each cycle two instructions are issued and those that are not *nops* are launched some constant number of cycles later. Henceforth we refer only to launch times and define all scheduling restrictions relative to the launch times. Cycles are numbered consecutively beginning from cycle 0. Instruction pairs of a schedule, *i.e.* a trace of the static listing of the code are numbered likewise so that the non-*nop* instructions of a pair, i, are launched in cycle i.

The timing of the use of each source and resource for an instruction is specified by a triple of the form [*delay, cycles, source restriction*]. If an instruction is launched in cycle *i*, then *delay* and *cycles* indicate that the resource or the read port (A, B, or C) of the source register file is used exclusively from cycle number *i+delay* through *i+delay+cycles-1*. The *source restriction* applies only to the A, B, and C sources and dictates that the source register contents may not be altered by another instruction during cycles *i+delay* through *i+delay+source restriction*. This means that the source register contents may be altered in or before cycle *i+delay-1* by a logically preceding instruction, or in or after cycle *i+delay+source restriction+1* by a logically succeeding instruction. This restriction dictates that a source register's contents cannot be changed until the instruction can no longer be affected by an exception, and insures the capability to revert to the machine state before the instruction. Since the source restriction applies only to source registers, it is absent from the timing description for resources.

*Result delay* is used to specify the *latency* between two dependent instructions, *i.e.* the minimum separation between their launch cycles that is sufficient to avoid read after write (RAW) hazards. Suppose that an instruction launched in cycle j, has a delay parameter value of $delay_j$ associated with one of its sources. Suppose further that the value of that source is the result of an earlier instruction i, launched in cycle i. The correct value will be read by instruction j if and only if $i+result\ delay_i < j+delay_j$. The *latency* for this pair of instructions then is $result\ delay_i+1-delay_j$, and this latency is satisfied if and only if $j-i \geq latency$.

Table 2.1: Timing for some floating point execute instructions

| Instruction | FPU{A,B} Source | FPU{C} Source | FPU Mult. Resource | FPU Add Resource | FPU Result Resource | Result Delay |
|---|---|---|---|---|---|---|
| fadd8 | [0,1,1] | -- | -- | [0,1] | [1,1] | 1 |
| fmul8 | [0,1,1] | -- | [0,1] | -- | [1,1] | 1 |
| fsmd8 | [0,1,3] | [2,1,1] | [2,1] | [0,1] | [3,1] | 3 |
| fsdm8 | [0,1,3] | [2,1,1] | [2,1] | [0,1] | [3,1] | 3 |
| fmas8 | [0,1,3] | [2,1,1] | [0,1] | [2,1] | [3,1] | 3 |
| fmds8 | [0,1,3] | [2,1,1] | [0,1] | [2,1] | [3,1] | 3 |

The first entry in table 2.1 [1][2] is for a floating point add instruction of the form X+Y -> Z. The X and Y operands which are used for the add operation are read from the register file in cycle 0 which is the same cycle in which the instruction is launched. The first two parameters in the FPU{A,B} source triple indicates that the floating point A and B ports are used in cycle 0 and remain busy for one cycle. The *source restriction* implies that the values in the registers that are being read can not be changed by any later instruction until cycle 2 (*delay* + *source restriction* = *0* + *1*) or later. The FPU add resource is also used in cycle 0, as indicated in the FPU add resource column. The floating point adder has two stages, but is fully pipelined and can be used in the very next cycle (cycle 1) by another instruction. Since an exclusive reservation of the first stage guarantees that no conflicts can occur in the second stage in the next cycle, the FPUAdder resource refers only to the first stage. The second stage reservation is thus implicit. Other fully pipelined resources are handled similarly. One of the two FPU register file write ports is used in cycle 1, to write the result of the instruction to the destination register. The *result delay* indicates that the Z result register cannot be reserved as the source operand register of a later instruction, j, until cycle 2 (*result delay*

[2] Chapter 10 Execute Instructions: pp 10-22, 10-42, 10-46, 10-50, 10-59, 10-57

+ *1*) or later, *i.e.* $j + delay_j \geq i + 2$. The second instruction is a floating point multiply of the form X*Y -> Z. This entry is identical to the first except that the FPU multiply unit is used instead of the add unit. The floating point subtract, fsub8 (X-Y -> Z), is not shown because usage of the resources and read ports of the source register file is exactly the same as for the floating point add instruction. Table 2.2 is a reservation template [4], [13] for the floating point add and subtract instructions. The columns correspond to clock

Table 2.2: Reservation template for floating point instructions of the type X+Y->Z and X-Y->Z

| | Cycle 0 | Cycle 1 |
|---|---|---|
| Issue 1 | x | |
| FPU {A,B} | x | |
| FPU{C} | | |
| FPU Adder | x | |
| FPU write | | x |

cycles while the rows list the relevant resources and source register file read ports. Issue 1 is the left issue unit used for FPU/IPU instructions. FPU {A,B} and FPU{C} sources are the three floating point register file read ports while FPU write is one of the two FPU register file write ports. Each resource or read port reservation required by the instruction is indicated by an x entered in the corresponding cell of the template.

The *fsmd8* entry in table 2.1 is for a linked triad instruction "subtract and multiply to the destination" of the form (X-Y)*Z -> Z, *i.e.* the multiply operation uses the result of the add operation that immediately precedes it. The X and Y operands which are used for the first operation (the subtract in this case) are read from the FPU register file in cycle 0, *i.e.* the same cycle in which the instruction is launched. As above, the A and B ports are used in cycle 0 and remain busy for one cycle. The *source restriction* indicates that the values in the registers being read cannot be changed by a later instruction until cycle 4 (*delay* + *source restriction* = *0* + *3*). Thus the *delay* and *cycles* entries for FPU{A,B} and FPU{C} sources describe the usage of the three floating point register file read ports. The FPU add unit is used in cycle 0 and remains busy for one cycle. The third operand Z is read from the read port C of the FPU register file in cycle 2 and the FPU multiply resource is used in the same cycle to multiply the result of the previous subtract by this operand. The register used to read this operand must not be changed by a later instruction until cycle 4, since *delay* + *source restriction* = 2 + 1. The floating point multiply unit, like the add unit, consists of two pipeline stages and remains busy for one cycle, *i.e.* during cycle 2. The result of the instruction is written back to the destination register in cycle 3 using one of the two FPU register file write ports. The *result delay* indicates that the Z result register cannot be reserved as the source operand register of a later instruction, j, until cycle 4 (*result delay* + *1*). Table 2.3 is the reservation template for this instruction.

The next entry in table 2.1 is for *fsdm8*; a linked triad instruction "subtract from destination and multiply" of the form (X-Y)*Z -> X. Therefore a floating point operation (X-Y)*Z can be represented by either `fsmd8.tr %f1, %f2, %f3` or `fsdm8.tr %f1, %f3, %f2` where %f1, %f2 and %f3 are floating point registers holding the values Y, X and Z respectively. The third operand is the destination register for both instructions. The suffix 8.tr implies that the instructions operate on 8 byte operands and

Table 2.3: Reservation template for triad instructions of the form (X-Y)*Z and (X+Y)*Z

| | Cycle 0 | Cycle 1 | Cycle 2 | Cycle 3 |
|---|---|---|---|---|
| Issue 1 | x | | | |
| FPU {A,B} | x | | | |
| FPU{C} | | | x | |
| FPU Mult. | | | x | |
| FPU Adder | x | | | |
| FPU write | | | | x |

trap on an exception. Both instructions produce the same result; the destination register is Z (%f3) for fsmd8.tr while it is X (%f2) for fsdm8.

The last two entries in table 2.1 are *fmas8* "multiply and subtract from destination" of the form X-(Y*Z) -> X, and *fmds8* "multiply by destination and subtract" of the form (X*Y)-Z -> X. The former triad allows only one operand register (X) to be used as the destination while the latter allows either X or Y to be used as the destination since multiplication is commutative. The subtract (s) can be changed to an add (a) to produce four more triad instructions. Note that any of the three operand registers can be selected as the destination register (possibly by rewriting the expression using the commutativity of the add or the multiply) except that the (X-Y)*Z result cannot use the Y operand register as the destination, the (X*Y)-Z instruction cannot use the Z operand as the destination register and the X-(Y*Z) can only use the X operand as the destination register.

Table 2.4: Timing for some floating point memory reference instructions

| Instruction | CEU {A,B} Source | FPU {C} Source | Load/Store Resource | Result Delay |
|---|---|---|---|---|
| ld8 %f | [0,1,0] | -- | [2,1] | 2 |
| st8 %f | [0,1,0] | [1,1,0] | [2,1] | 0 |

Table 2.4 [1][3] describes the use of each source and resource by the floating point load and store instructions. Both of these instructions use a CEU register as an address register that is read using the CEU{A,B} source. Instruction launch takes place in cycle 0 and the address register is read in the same cycle. The CEU register read port is kept busy for one cycle and it is legal to change the value in the CEU register in cycle 1or later. The store instruction reads the value to be stored from the floating point register in cycle 1, and reserves the FPU{C} read port during that cycle[4]. The load/store resource for the store is the memory write port which is used in cycle 2 for one cycle. The *result delay* for the store is specified as 0, which means that the result of the store can be reloaded by a load instruction launched in cycle 1 or later. Thus store instruction's *latency* is 1 (*result delay + 1 = 0 + 1*).

For the load instruction, as in the store, a CEU register is used as an address register, the CEU read

[3] Chapter 9 Memory Reference Instructions: pp 9-3, 9-10

[4] The store instruction uses a particular FPU register file read port, {C}, rather than any one of the three.

port is kept busy for one cycle, and the value in the CEU register may be changed in cycle 1. The load/store resource is used in cycle 2 and remains busy for one cycle. This resource for a load is the FPU register file write port used to write the contents of the memory location to the destination register and the *result delay* is 2. Note that the memory read port reservation is unspecified here, but no specification is needed since the KSR1 stalls on read miss and no conflicts can occur at the memory read port. Table 2.5 is the reservation template for a floating point load. Issue 2 is the issue unit used for CEU/XIU instructions. For simplicity, we have not shown the CEU{A,B} read port in the reservation templates since we have not observed any resource contention there.

Table 2.5: Reservation template for a floating point load

| | Cycle 0 | Cycle 1 | Cycle 2 |
|---|---|---|---|
| Issue 2 | x | | |
| FPU write | | | x |

## 2.3 Scheduling Restrictions

The complex resource reservations and delay parameters for different instruction classes make optimal code scheduling for the KSR by hand a very tedious task. We illustrate a few scheduling restrictions for the KSR1 processor in the following examples.

### Example 2.1: Triad, Store Sequence Illegal

Consider the following code segment:

```
Cycle 0:        fmad8.tr    %f15, %f20, %f0    ;   cxnop

Cycle 1:        finop                          ;   st8       %f9, 24008 (%c10)
```

The first instruction is a triad which is launched in cycle 0. The registers %f15 and %f20 are multiplied and the result is added to %f0 which is also the destination register. From table 2.1, the FPU{C} resource usage specification is [2,1,1]. This means that the read port {C} will be used in cycle 2 for one cycle, and that the value in the register %f0, read using this port cannot be changed by any later instruction before cycle 4 (*delay* + *source restriction* = *2* + *1*). The instruction in cycle 1 is a store, and from table 2.4 the store instruction's usage of the FPU{C} source to read the value in %f9 is [1,1,0]. Therefore the store will also require use of the {C} read port in cycle 2, one cycle after it is launched. Since no resource can be used by more than one instruction in the same cycle this leads to a conflict between the two instructions. The schedule presented above is therefore illegal and will produce incorrect results. Since the usage of the FPU{C} source is the same for all linked triad instructions, this prevents floating point store instructions from being launched in a cycle that immediately follows the launch of a linked triad instruction.

### Example 2.2: Triad, Load, Diad Sequence Illegal

Another schedule which has been observed to be illegal is illustrate by the following code segment:

```
Cycle 0:        fmad8.tr    %f3, %f5, %f8      ;   cxnop

Cycle 1:        finop                          ;   ld8       32040 (%c10), %f11

Cycle 2:        fmul8.tr    %f12, %f14, %f16   ;   cxnop
```

Table 2.6 is the reservation table for this code segment. The reservation table is mechanically constructed by placing the reservation template for each instruction in the reservation table beginning in the column at which the instruction is launched. The columns correspond to clock cycles and each row shows the usage of a resource or register read port. The reservations of the triad, load and the multiply instructions are indicated by the numbers 0, 1 and 2, which correspond, respectively, to the cycle during which the corresponding instruction is launched. Issue 1 is the launch unit for the FPU/IPU side of the instruction pair, while issue 2 is the one used to launch instructions belonging to the CEU/XIU side. FPU{A,B} and FPU{C} sources are the three floating point register file read ports and FPU write1 and write 2 are the two write ports in the floating point register file. The floating point multiplier and adder are also shown.

Table 2.6: Reservation table for the code segment in example 2.2

| | Cycle 0 | Cycle 1 | Cycle 2 | Cycle 3 |
|---|---|---|---|---|
| Issue 1 | 0 | | 2 | |
| Issue 2 | | 1 | | |
| FPU {A,B} | 0 | | 2 | |
| FPU{C} | | | 0 | |
| FPU Mult. | 0 | | 2 | |
| FPU Adder | | | 0 | |
| FPU write 1 | | | | 0 |
| FPU write 2 | | | | 1 (2) |

The triad has a *delay* parameter of 3 associated with the FPU result resource, and therefore it uses the write port in cycle 3. The load, which is launched in cycle 1, takes 2 cycles to complete and also uses a write port in cycle 3. Therefore, both the floating point register file write ports are busy in cycle 3. Since the multiply operation is launched in cycle 2, it completes in the very next cycle and it also requires use of a register write port in cycle 3, but both write ports are already busy. This leads to a collision in a four column reservation table for the schedule shown above. Thus a triad, load and a diad (two operand floating point instruction) cannot be launched in consecutive cycles.

Note that the three independent instructions of this example could actually be issued in two cycles: the triad and load in cycle 0 and the diad in cycle 1. In this case the load and the diad would use two different FPU write ports in cycle 2.

## Example 2.3: Illegal Triad, -, Diad Sequences

The schedule presented in the following code segment is illegal because two instructions require the use of a single functional unit at the same time.

```
Cycle 0:        fmad8.tr    %f9, %f2, %f10     ;   cxnop

Cycle 1:        finop                          ;   cxnop

Cycle 2:        fadd8.tr    %f8, %f12, %f13    ;   cxnop
```

Table 2.7 is the reservation table for this code segment, where the reservation of the sources and resources by the instructions is indicated by a number corresponding to their launch cycle. The triad instruction is launched in cycle 0 and the add is launched in cycle 2. Since the triad and add instructions both attempt to reserve the FPU Adder in cycle 2, the reservation table for this code segment has a conflict and the schedule will produce incorrect results.

Table 2.7: Reservation table for the code segment in example 2.3

| | Cycle 0 | Cycle 1 | Cycle 2 | Cycle 3 |
|---|---|---|---|---|
| Issue 1 | 0 | | 2 | |
| Issue 2 | | | | |
| FPU {A,B} | 0 | | 2 | |
| FPU{C} | | | 0 | |
| FPU Mult. | 0 | | | |
| FPU Adder | | | 0(2) | |
| FPU Write 1 | | | | 0 |
| FPU Write 2 | | | | 2 |

An *fadd* or *fsub* instruction may not be launched two cycles after a triad if the triad uses the FPU Adder for its second operation. Similarly, an *fmul* cannot be launched two cycles after a triad that uses the FPU Multiplier for its second operation.

## 2.4 Summary of Schedule Conflicts

A KSR1 schedule that issues no more than one FPU/IPU and one CEU/XIU instruction per cycle is legal if and only if it satisfies each of the following four conditions:

(1) **Resource conflicts**: No resource is reserved by more than one instruction at any time. This condition can easily be checked by *i)* placing the reservation template for each instruction in a reservation table beginning in the cycle at which the corresponding instruction is launched, and *ii)* checking the reservation table to make sure that each cell has only a single entry. Note that there are two FPU register write ports, but a reservation template does not specify which of these ports is to be used by the instruction. Consequently, there is no reservation conflict as long as no more than two FPU register write ports are reserved in any cycle.

(2) **Write-after-read hazards**: The *source restrictions* are all satisfied, *i.e.* no register is written by a later instruction until the *source restrictions* for all previous instructions that use that register as a source have lapsed. This condition must be checked for each source register of each instruction. Consider an instruction that is launched in cycle i and uses source register R with *delay+source restriction* = S. This *source restriction* is violated if and only if there is an instruction launched at cycle j ($j \geq i$) that uses R as a destination register with a *delay* of D where $i+S \geq j+D$.

(3) **Read-after-write hazards**: The latencies are all satisfied, *i.e.* for a dependent pair of instructions, i and j ($j > i$), where the result register of i is a source register of j with $delay_j$ then the *latency*

for this pair of instructions is satisfied if and only if $j - i > \textit{result delay}_i - \textit{delay}_j$.

(4) **Write-after-write hazards**: The logically earlier instruction of a pair of instructions with the same destination register must reserve the register file write port in the reservation table before the logically later instruction.

Note that conditions 2 and 3 above cannot be checked simply in the reservation table alone since the *source restriction* and *result delay* parameter values are not explicitly shown in the reservation table. They could easily be added, but this would require a reservation table row for each register of each register file in the machine.

From example 2.1 above it is apparent that a resource conflict at the FPU{C} read port will always occur when a triad instruction is followed by a store instruction launched one cycle later. From example 2.2 it is seen that a triad, load, diad sequence (launched in successive cycles) will have a resource conflict that attempts to place three reservations of the two FPU register file write ports in the same cycle. From example 2.3, it is seen that a triad followed by a diad instruction launched two cycles later that uses the same floating-point arithmetic function unit as the second operation of the triad will cause a resource conflict at that function unit. In our experience with the workloads of this study, these were the only resource conflicts that arose.

# CHAPTER III
# PERFORMANCE BOUNDS MODEL

## 3.1 Formulation of the Bounds Equations

This model produces a lower bound on the run time of a loop program by taking into account the most common performance bottlenecks in an architecture, and evaluating their effect by counting the essential number of operations per inner loop iteration from the high level code in the application [6]. The number of floating point arithmetic operations is simply the number of floating point operations (add, multiply, etc.) that appear in the source listing of one inner loop iteration. Operation pairs that can be combined into triads must also be recognized. 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 $a$; the minimum number that is required to be performed during each iteration. The number of essential loads can be 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 [4].

The performance model bounds from below the number of clock cycles that must be spent for each iteration of the inner loop, $t_l$. It considers several potential bottleneck units in the machine, and uses the bandwidth of these machine units to characterize machine performance. These are: the floating point unit, the memory unit, the instruction issue unit, and a dependence pseudo-unit. Timing equations for each of these units are developed to specify the minimum number of clock cycles required to process one iteration of a given loop through that unit.

The dependence unit is a fictitious unit, but provides a convenient method of modeling the effect of the number of clock cycles, $t_d$., required to traverse the longest recurrence cycle in the dependence graph. For each loop-carried dependence cycle, the sum of the latencies of the operations in one traversal of the cycle is divided by the number of iterations in one traversal. Then $t_d$ is set to the maximum of this value over all recurrence cycles. In the absence of a recurrence cycle, $t_d$ is 0. In general,

$$t_d = \text{Total latency per iteration of a loop-carried dependence} \tag{3.1}$$

The floating point unit in the KSR1 processor is capable of starting the execution of one new instruction per clock cycle. The instructions executed per iteration in the floating point unit contribute to $t_f$. The essential number of instructions is set equal to the sum of the number of combinable floating point multiply-add operations, the number of non-combinable floating point multiply operations, and the number of non-combinable floating point add operations. The number of each of these in one iteration of the loop is $f_{ma}$, $f_m$, and $f_a$, respectively. Whenever possible, the add (subtract) and multiply operators that appear in the high level code are combined into one of the eight triad instructions of the KSR1 and included in $f_{ma}$ count.

$$t_f = f_{ma} + f_m + f_a \tag{3.2}$$

The effect of the floating point memory operations is modeled by $t_m$. The memory unit is kept busy for at least one clock cycle by each floating point load and each floating point store. The number of essential loads and stores per iteration of the inner loop is $l_{fl}$ and $s_{fl}$, respectively. Thus

$$t_m = l_{fl} + s_{fl} \qquad (3.3)$$

The instruction issue unit can issue two instructions during each clock cycle. One of these instructions goes to either the FPU or the IPU (floating point and integer instructions), while the other goes to the CEU or the XIU (load, store, address arithmetic, branch, and I/O instructions). Therefore, the number of clock cycles spent per iteration issuing instructions will be at least equal to the maximum of the FPU/IPU and the CEU/XIU instructions. The number of clock cycles per inner loop iteration accounted for by the instruction issue unit is characterized by $t_i$. Since loops can be unrolled, either by the compiler or by hand, only a fraction of the branch overhead instructions, which appear only once in each iteration of the unrolled loop, will contribute toward $t_i$. This fraction is 1/k where k is the degree of unrolling. As k increases, this fraction asymptotically approaches 0. A register port conflict in the floating point register file, which constrains concurrent scheduling of floating point stores and linked triad instructions, is also taken into account when computing $t_i$. Thus $t_i$ is the maximum over three components: the total number of essential instructions issued per iteration on the FPU/IPU side, the total number of essential instructions issued per iteration on the CEU/XIU side, and the essential number of uses of the FPU{C} source during each iteration. The FPU{C} source is used for one clock by each linked triad instruction and each store instruction. Thus

$$t_i = \max\left((l_{fl} + s_{fl} + y/k),\ (f_{ma} + f_a + f_m + x/k),\ (f_{ma} + s_{fl})\right) \qquad (3.4)$$

In (3.4), $x$ is the branch overhead per unrolled iteration of the loop on the FPU/IPU side of the issue. It typically includes a loop index decrement instruction and a compare instruction to set/clear the conditional code for the branch and has a value of 2. In the same equation, $y$ is the branch overhead per unrolled iteration of the loop on the CEU/XIU side of the issue. It typically includes an increment instruction for each base address register and a branch instruction and has a value of 1 plus the number of base address registers needed. Both the $x$ and $y$ terms are normalized to a *per original iteration* overhead by dividing by the degree of unrolling $k$.

Each inner loop iteration of an application will spend a certain number of clock cycles performing essential operations in each of these units. To perform an iteration, the processor will require at least as much time as the busiest of its units. The time spent in a steady-state iteration of a loop is at least the maximum over each of these machine units of its minimum busy time. The number of clock cycles per (rolled) iteration of the inner loop in the steady-state is at least $t_l$, where

$$t_l = \max(t_i, t_f, t_m, t_d) \qquad (3.5)$$

It is obvious that $t_i$ dominates both $t_f$ and $t_m$, making the processor's instruction issue unit the bottleneck that determines $t_l$, unless a recurrence cycle exists with $t_d > t_i$.

While computing the $t_l$ bound we have ignored *i)* factors that may limit concurrency between the machine units, *ii)* the inability to pack reservation templates in a reservation table, *iii)* cache misses, 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 bound.

## 3.2 The Application Workload and the MA Bound

In this study, the first twelve loops from the Livermore Fortran Kernels [14] are used to illustrate our approach to performance evaluation and performance improvement for the KSR. The application workload is initially determined from the high level code by counting the number of essential operations for each iteration of the inner loop. This workload is similar to one that would be generated by a compiler that completely unrolled the loops and used an optimal dataflow analysis technique leading to maximum register reuse among the loop iterations. Table 3.1 shows the essential operations per inner loop iteration for the first twelve LFKs.

Table 3.1: Essential floating point memory and execute operations in LFK 1-12

| LFK | fa | $f_m$ | $f_{ma}$ | $l_{fl}$ | $s_{fl}$ |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 2 | 2 | 1 |
| 2 | 0 | 0 | 2 | 4 | 1 |
| 3 | 0 | 0 | 1 | 2 | 0 |
| 4 | 0 | 0 | 1 | 2 | 0 |
| 5 | 0 | 0 | 1 | 2 | 1 |
| 6 | 0 | 0 | 1 | 2 | 0 |
| 7 | 0 | 0 | 8 | 3 | 1 |
| 8 | 6 | 0 | 15 | 9 | 6 |
| 9 | 1 | 0 | 8 | 10 | 1 |
| 10 | 9 | 0 | 0 | 10 | 10 |
| 11 | 1 | 0 | 0 | 1 | 1 |
| 12 | 1 | 0 | 0 | 1 | 1 |

Since only the essential floating point operations from the high level application code are counted and used to compute the bound, the resultant machine-application (MA) bound ignores the effect of the extra operations introduced by the compiler. It also does not account for scheduling effects and register spilling. Since only two loops, 5 and 11, have loop-carried dependence, where the result of one iteration depends upon the corresponding result of the previous iteration, the value of $t_d$ is zero for all other loops. The $f_{ma}$ operation in loop 5 is of the form X*(Y-Z), where the result of one iteration is the Z input for the next. Thus a consecutive pair of these triad instructions has a latency of 4 which is the $t_d$ for this loop-carried dependence cycle. Loop 11 has a floating point add operation of the form X+Y where the result of one iteration is the X input for the next. A consecutive pair of these instructions has a latency of 2 which is the value of $t_d$ for the loop-carried dependence cycle in loop 11.

Table 3.1 and (3.1)-(3.5) are used to calculate the $t_l$ bound in table 3.2. This bound is in units of clocks per inner loop iteration (CPL). Note that only LFK 7 has $t_i$, and hence $t_l$, affected by FPU{C} source conflicts. The MA bound is computed by converting $t_l$ to units of clocks per floating point operation (CPF), *i.e.* by dividing $t_l$ (in CPL) by the total number of essential floating point operations per iteration (TNF). TNF is calculated as shown in (3.6). Since they are based only on essential operations and are computed in units of CPL and CPF, respectively, without regard to clock rate, $t_l$ and the MA bound can be used for performance comparisons to other architectures while factoring out clock rate and compiler effects. CPF will be used as the unit of performance throughout the rest of the paper.

TNF (Total number of essential floating point operations per iteration) = $f_a + f_m + 2*f_{ma}$ (3.6)

MA Bound (in CPF, Clock cycles per essential floating point operation) = $t_l$ / TNF (3.7)

Table 3.2: The number of clocks required per essential floating point operation for LFK 1-12

| LFK | $t_f$ | $t_m$ | x | y | $t_i$ | $t_l$ | MA Bound |
|---|---|---|---|---|---|---|---|
| 1 | 3 | 3 | 2 | 2 | 3 + 2/k | 3 + 2/k | 0.6 + 0.4/k |
| 2 | 2 | 5 | 2 | 3 | 5 + 3/k | 5 + 3/k | 1.25 + 0.75/k |
| 3 | 1 | 2 | 2 | 2 | 2 + 2/k | 2 + 2/k | 1 + 1/k |
| 4 | 1 | 2 | 2 | 3 | 2 + 3/k | 2 + 3/k | 1 + 1.5/k |
| 5[a] | 1 | 3 | 2 | 2 | 3 + 2/k | 5.0; k=1<br>4.0; k>1 | 2.5; k=1<br>2.0; k>1 |
| 6 | 1 | 2 | 2 | 3 | 2 + 3/k | 2 + 3/k | 1 + 1.5/k |
| 7 | 8 | 4 | 2 | 2 | 10; k=1<br>9; k>1 | 10; k=1<br>9; k>1 | 0.625; k=1<br>0.56; k>1 |
| 8 | 21 | 15 | 2 | 3 | 21 + 2/k | 21 + 2/k | 0.58 + 0.06/k |
| 9 | 9 | 11 | 2 | 2 | 11 + 2/k | 11 + 2/k | 0.65 + 0.12/k |
| 10 | 9 | 20 | 2 | 2 | 20 + 2/k | 20 + 2/k | 2.22 + 0.22/k |
| 11[b] | 1 | 2 | 2 | 2 | 2 + 2/k | 2 + 2/k | 2 + 2/k |
| 12 | 1 | 2 | 2 | 2 | 2 + 2/k | 2 + 2/k | 2 + 2/k |

a. $t_d = 4$
b. $t_d = 2$

## 3.3 Performance of the Compiled Code and the MAC and MACS Bounds

The first twelve loops 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. Table 3.3 shows the values of *k* used by the compiler for each loop, the performance of the compiled code in CPF, and the percentage of the performance indicated by the MA bound that is achieved by the compiled code. This was computed as follows:

% of bound achieved = CPF (bound) / CPF (actual) * 100 (3.8)

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

Harmonic Mean (MFLOPS) = CPU clock rate (MHz) / Average CPF (3.9)

The harmonic mean performance of the compiled code for the first twelve loops is 10.05 MFLOPS, while the MA bound, using the average CPF from table 3.3, indicates a performance potential of 14.93 MFLOPS. All but LFK 2 and 6 achieve at least 60% of the bound, while loop 10 achieves 92.68%

Table 3.3: Performance of LFK 1-12

| LFK | k | MA Bound (CPF) | Compiled (CPF) | % of bound achieved |
|---|---|---|---|---|
| 1 | 8 | 0.65 | 0.96 | 67.77 |
| 2 | 8 | 1.34 | 3.09 | 43.36 |
| 3 | 8 | 1.12 | 1.32 | 84.84 |
| 4 | 8 | 1.19 | 1.55 | 76.77 |
| 5 | 8 | 2.0 | 2.43 | 82.3 |
| 6 | 8 | 1.19 | 4.07 | 29.93 |
| 7 | 4 | 0.56 | 0.92 | 60.87 |
| 8 | 1 | 0.64 | 1.01 | 63.37 |
| 9 | 4 | 0.68 | 0.80 | 85.0 |
| 10 | 2 | 2.28 | 2.46 | 92.68 |
| 11 | 8 | 2.25 | 2.83 | 79.5 |
| 12 | 8 | 2.25 | 2.46 | 91.46 |
| Average | -- | 1.34 | 1.99 | 67.34% |

of the bound. Loop 6, at 29.93% of the performance bound is the furthest away from its idealized minimum run time. If we were to use an infinitely large value of k instead of the value generated by the compiler, the k-dependent term in the MA bound for all loops would become negligible. The average CPF for the MA bounds of the first 12 LFKs would then be 1.24, which corresponds to 16.13 MFLOPS. The compiled code achieves 62.31% of this bound.

To investigate the reasons for the gap between the MA bound and the delivered performance, it is important to isolate the performance degrading factors. The gap could be the result of inefficient code due to a weakness in the compiler, or in the bounds model due to ignoring interactions among units, scheduling conflicts, outer loop code, start-up transients, or system level effects such as cache misses and operating system activity. To explain some of the factors that contribute to this gap a succession of bounds is developed [8] based on the machine and application of interest (MA, as above), the compiler-generated workload (MAC), and the actual compiler-generated schedule of this workload (MACS). This series of increasingly constrained performance models exposes specific performance gaps by successively binding factors that were previously idealized.

The machine-application-compiler (MAC) bound is similar to the MA bound, 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 the MAC bound still ignores scheduling effects, 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 estimate of essential operations. Thus it removes one degree of freedom from the model by using an actual rather than an idealized workload. It still, however, assumes an idealized schedule for the compiler-generated workload. If the performance of the code is considerably closer to the MAC bound than it was to the MA bound, it can be deduced that the compiler-generated workload causes a significant portion of the gap between the MA bound and the delivered performance.

Table 3.4: Actual workload generated by the compiler for LFK 1-12

| LFK | k | $f_a$ | $f_m$ | $f_{ma}$ | $l_{fl}$ | $s_{fl}$ | other CEU /XIU | $t_f$ | $t_m$ | $t_i$ | $t_l$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 8 | 0 | 1 | 2 | 2.13 | 1 | 0 | 3 | 3.13 | 3.38 | 3.38 |
| 2 | 8 | 0 | 0 | 2 | 5 | 1 | 0 | 2 | 6.0 | 6.38 | 6.38 |
| 3 | 8 | 0 | 0 | 1 | 2 | 0 | 0 | 1 | 2 | 2.25 | 2.25 |
| 4 | 8 | 0 | 0 | 1 | 2 | 0 | 0.13 | 1 | 2 | 2.5 | 2.5 |
| 5 | 8 | 0 | 0 | 1 | 2.13 | 1 | 0 | 1 | 3.13 | 3.38 | 4.0 |
| 6 | 8 | 0.63 | 0 | 1 | 2.13 | 1 | 0.13 | 1.63 | 3.13 | 3.63 | 3.63 |
| 7 | 4 | 2.75 | 0 | 8 | 4.5 | 1 | 0 | 10.75 | 5.5 | 11.0 | 11.5 |
| 8 | 1 | 10 | 0 | 15 | 15 | 6 | 0 | 25 | 21 | 27 | 27 |
| 9 | 4 | 2 | 1 | 7 | 10 | 1 | 0.25 | 10 | 11 | 11.75 | 11.75 |
| 10 | 2 | 9 | 0 | 0 | 10 | 10 | 0.5 | 9 | 20 | 21.5 | 21.5 |
| 11 | 8 | 1 | 0 | 0 | 1.13 | 1 | 0 | 1 | 2.13 | 2.38 | 2.38 |
| 12 | 8 | 1 | 0 | 0 | 1.13 | 1 | 0 | 1 | 2.13 | 2.38 | 2.38 |

Table 3.4 shows the actual workload generated by the compiler (k is the degree of unrolling introduced and the instruction counts have been normalized to the original iteration of the inner loop by dividing by k). The branch overhead values, x and y, from table 3.2 are used since they remain unchanged. The increased entries are shown in bold type. The number of non-combinable floating point add operations, $f_a$, changes in LFK 6, 7, and 8 because of the introduction of floating point move instructions. A copy operation from one floating point register to another is implemented as a floating point add on the KSR1. The compiler also failed to find one pair of combinable multiply-adds in LFK 9. There are a number of "nonessential" loads, and in LFK 6 there is a "nonessential" store. The "other CEU/XIU" column lists those operations on the CEU/XIU side of the issue that are not included in y and are neither loads nor stores. These operations are instructions to copy an IPU register to a CEU register. They contribute to $t_i$ if the CEU/XIU issue term is the maximum term in $t_i$. For the MAC bound, $t_i$ is calculated as follows:

$$t_i = \max\left((l_{fl} + s_{fl} + \text{other CEU/XIU} + y/k),\ (f_{ma} + f_a + f_m + x/k),\ (f_{ma} + s_{fl})\right) \qquad (3.10)$$

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. 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

Table 3.5: Performance of LFK 1-12 compared with the MAC and MACS bounds

| LFK | Compiled (CPF) | MAC bound (CPF) | % of MAC bound achieved | MACS bound (CPF) | % of MACS bound achieved |
|---|---|---|---|---|---|
| 1 | 0.96 | 0.68 | 70.84 | 0.93 | 96.35 |
| 2 | 3.09 | 1.59 | 51.46 | 2.59 | 83.82 |
| 3 | 1.32 | 1.12 | 84.84 | 1.25 | 94.7 |
| 4 | 1.55 | 1.25 | 80.65 | 1.31 | 84.51 |
| 5 | 2.43 | 2.0 | 82.3 | 2.31 | 95.06 |
| 6 | 4.07 | 1.81 | 44.47 | 3.56 | 87.5 |
| 7 | 0.92 | 0.72 | 78.26 | 0.89 | 96.53 |
| 8 | 1.01 | 0.75 | 74.26 | 0.97 | 96.04 |
| 9 | 0.8 | 0.69 | 86.25 | 0.76 | 95.0 |
| 10 | 2.46 | 2.39 | 97.15 | 2.39 | 97.15 |
| 11 | 2.88 | 2.38 | 82.47 | 2.75 | 95.49 |
| 12 | 2.46 | 2.38 | 96.54 | 2.38 | 96.54 |
| AVG | 1.99 | 1.38 | 69.34% | 1.84 | 92.46% |

in loops 4, 9 and 10 due to the small fraction added to $t_i$ by "other CEU/XIU" instructions. Only a slight increase in the number of loads, $l_{fl}$, is the reason why the MA and MAC bounds are close for loops 1, 11 and 12. Loops 2, 6, 7, and 8 show a significant gap between the MA and MAC bounds due to a variety of factors.

Another bound which can be used to evaluate the scheduling effects is the machine-application-compiler-scheduler (MACS) bound. The MACS bound imposes a limit on the minimum run time of a particular schedule generated by the compiler for some application. It is useful in explaining a portion of the gap between the MAC bound and delivered performance by considering the actual schedule of the compiled code, rather than the idealized schedule of the MAC bound. The MACS bound thus exposes weaknesses in the compiler's scheduler and/or the unachieveability of the idealized schedule. Calculating the MACS bound for the KSR1 processor is very straightforward since the code is statically scheduled. The compiler must insert explicit *nop* instructions in the code to insure that data dependence requirements between instructions are satisfied. Thus, the number of clock cycles required (except for cache stalls) for one iteration of a loop, $t_l$, can be counted directly from the static listing of the assembly code and the degree of unrolling, k. This count can be can be normalized to CPF by using (3.7) or to MFLOPS by using

(3.9). Loops for which the performance of the compiled code is not close to the MACS bound performance need to be investigated further to find out the reasons for the remaining performance gaps.

All of the loops, except loops 2, 4, and 6, achieve at least 94% of the MACS bound. 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

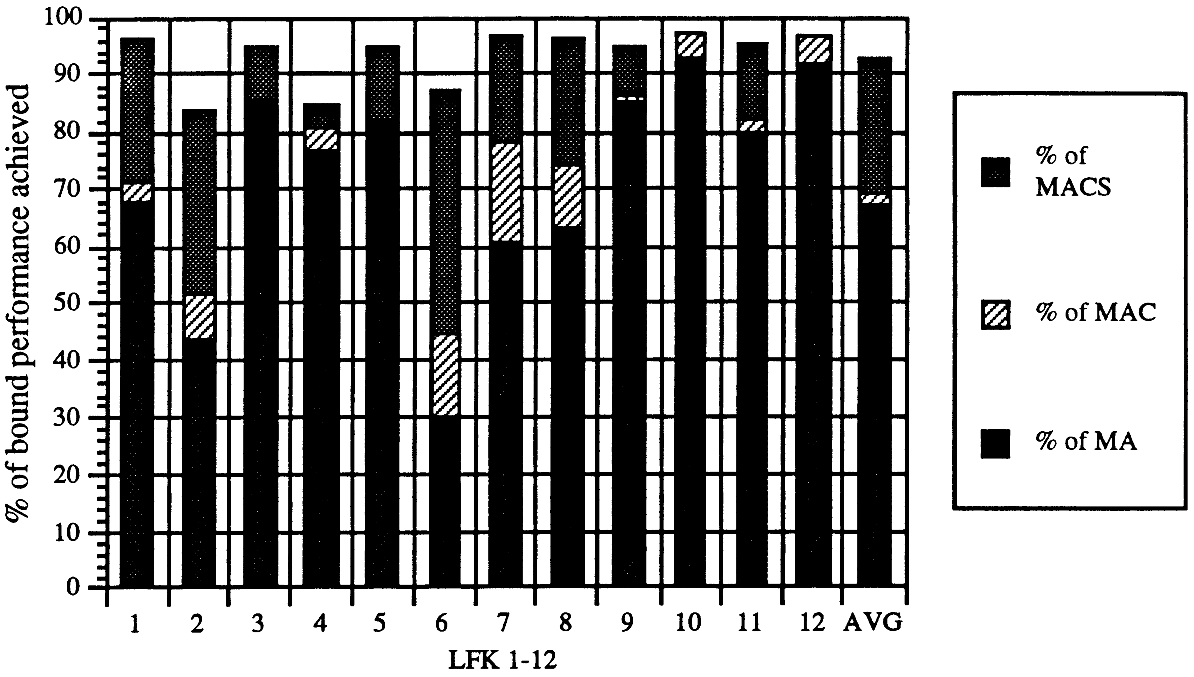

Figure 3.1: % of MA, MAC and MACS bound performance achieved by compiled code

instruction issue unit is the bottleneck, and the code on this side does not have any *nops*. However, the percentage of the MACS bound achieved by the compiled code is 97.15 and 96.54 for loops 10 and 12, respectively. The remaining 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 sub-optimal, even though the schedule for the unrolled section is optimal. The performance gap between the compiled code's performance and the MACS bound for loops 2, 4, and 6 is large enough to warrant further investigation. This could be due to the presence of some outer-loop overhead in these loops; since the bound assesses only steady-state inner-loop performance, delivered performance would be less than the performance predicted by the MACS bound if there is outer-loop overhead. Steady-state inner-loop performance measurement is discussed in more detail in section 5.5.

## 3.4 Sources of Performance Loss

It is obvious from table 3.5 and figure 3.1 that some of the performance loss occurs due to the fact that the number of operations generated by the compiler exceeds the number of operations in the essential set. The compiler's inability to recognize operand reuse and to retain such operands in the register file leads to the introduction of redundant memory operations in the schedule, elevating the run time. Although loop unrolling assists the compiler in recognizing reuse, some redundant operations remain.

For loops 1, 2, 6, and 8 a gap ranging from 21.78% to 43.03% of the measured run time is observed between the MAC and the MACS bounds. This implies that the schedule produced by the compiler for the compiler-generated workload is far from ideal. However, loops 2, 6, 7 and 8 also have a considerable number of redundant operations, as seen from the gap ranging from 8.1% to 17.4% of the measured run time between their MA and the MAC bounds. It is possible for the scheduling task to become more complicated if these redundant operations introduce new dependencies among operations within and between iterations. The scheduling task might have been simpler if redundant operations were not present in the code. In the following sections, some of the loops are examined more closely to explain the presence of redundant operations. Code rescheduling is considered in chapter IV and chapter V.

## 3.4.1 Loop 2

The high level code for the inner loop is

```
DO 2 k = IPNT+2 , IPNTP , 2

            i = i+1

2           X(i) = X(k) - V(k)*X(k-1) - V(k+1)*X(k+1)
```

The loop index increment for each loop is 2, necessitating four loads per iteration: the k and k+1 elements of each of the arrays X and V. However, the value loaded in a register for X(k+1) during an iteration can be reused as X(k-1) during the next iteration. Therefore the essential number of loads required for one iteration of the loop is 4. The compiler produces 5 loads for each iteration by generating a memory access for each reference to X(k-1). Apparently the compiler does not have enough information to identify dependent memory accesses, even though the loop has been unrolled eight times. This hints at a weak dataflow analysis technique which gives rise to a considerable gap between the actual and expected performance.

## 3.4.2 Loop 6

The code for this loop is as follows:

```
DO 6 i = 2 , n

DO 6 k = 1 , i-1

            W(i) = W(i) + B(i,k) * W(i-k)

6 CONTINUE
```

All results are written to the same location in the W array during each iteration of the inner loop. This clearly does not require any store operations inside the loop. The compiler still generates redundant memory stores, although it might not do so if W(i) was replaced by a scalar. This would be consistent with the behavior observed in loop 3 which performs an inner product where the source code accumulates the result to a scalar in the inner loop. The compiler avoided producing redundant stores for loop 3.

The only floating point instruction in this loop is a linked triad instruction. The linked triad, like all other instructions of the KSR1, is a three-address format register-to-register instruction. The destination register is the same as the third source register in this instruction format, which is W(i) in this case. The same register can be used in the triad instruction in the next iteration of the loop, but the compiler moves the new value in the destination register to another register, and makes use of the target of the move instruction in the next iteration. This move could simplify the task of coloring the interference graph in some situations, but appears to be unnecessary here.

### 3.4.3 Loop 7

Loop 7 is an equation of state fragment, and presents a number of opportunities for data reuse across iterations. The source code is

```
      DO 7 k = 1, n
      X(k) =    U(k) + R * ( Z(k) + R*Y(k))) +
                T* (U(k+3) + R*(U(k+2) + R*U(k+1)) +
                T* (U(k+6) + R*(U(k+5) + R*U(k+4))))
    7 CONTINUE
```

There are 8 triad instructions for each iteration of the loop which account for all the floating-point execute instructions in the loop. The compiler recognizes that some of the operands already reside in registers and can be reused in later iterations. However, in some cases the operand which is to be reused later can get overwritten by a result due to the restriction posed by the three address instruction format which requires the use of one of the source registers as the result register. To avoid the penalty of an extra memory access, the compiler introduces instructions to move a reusable operand between registers before its use. There is a total of 11 move instructions in the loop body which is unrolled four times; almost three per iteration. These moves lead to further congestion on the FPU/IPU side of the instruction issue, which is already a bottleneck for this loop. Code rescheduling to eliminate this problem is discussed in section 5.3.

# CHAPTER IV
# AUTOMATIC CODE RESCHEDULING

## 4.1 The Object Code Optimizer (OCO)

The Object Code Optimizer was developed by Daniel Windheiser at IRISA, France, for the i860, MIPS and SPARC architectures. It is a re-targetable tool which can be used for any architecture in the same general class. It reschedules code by using the cyclic scheduling algorithm presented in [9]. The terms polycyclic scheduling, cyclic scheduling and software pipelining are often used to describe similar code generation techniques. Cyclic scheduling is derived from Patel's work on introducing delay into a reservation table to achieve higher performance from hardware pipelines with reused resources [15]. It was developed by Rau [10] and Hsu [12] using the reservation table approach. All loop iterations have identical code schedules in cyclic scheduling. A new iteration begins every *initiation interval* (II, in clock cycles), quite commonly before all the instructions belonging to prior iterations have been issued. A *modulo reservation table* (MRT) [12] containing II columns[1] is used to keep track of all iterations that are executing concurrently during II successive clock cycles. Only the resources where conflicts might occur need be considered and each of these is assigned a unique row in the MRT. The reservation of all other resources is implicit. Each instruction is defined by a resource reservation template. Resource reservation templates for different classes of KSR instructions are discussed in section 2.2.

Cyclic scheduling of loop dominated code is most useful when a processor implementation has long latencies that can be masked by proper scheduling. For loops with acyclic dependence graphs, this scheduling technique can guarantee that the most heavily used resource is fully utilized after an initial start-up transient, provided that the machine has a sufficient number of registers and instruction reservation templates whose shapes can be tightly packed [3] [4].

OCO accepts as input a segment of assembly code for the target architecture. The input code can also contain some optional directives which among other things can be used to identify those loops that require restructuring. The tool is given a description of the target processor's resources (functional units, number of registers, etc.) and its instruction set. This machine description is a set of resource reservation templates, one for each class of instructions. The reservation templates are specified in a lisp-like format and include sufficient information for OCO to be able to determine source restrictions and instruction latencies. The instruction specification conforms to the assembly instruction syntax.

This machine description is used to build an internal database. Then the assembly code for an application of interest is read in and an internal representation is constructed in a list format. A control flow graph is then built for each procedure in the input file. Procedures are identified by directives that are inserted in the object code before it is given to the tool as input. A use-definition (UD) analysis ensues to form in-out sets as follows. Consider a node in the control flow graph that defines a value for some variable. Consider all paths in the graph that begin at that defining node for that variable, contain no other defining nodes for that variable, and end at a node that uses that variable. The value is *live* during all nodes on such paths, including the defining node and the final use node. For a node N, the set *in(N)* is the set of values from logically preceding statements that are live when the program flow reaches N. In OCO, this analysis is not sensitive to control flow and does not distinguish between information coming into a basic

[1] Hsu uses an MRT that is the transpose of the table described here.

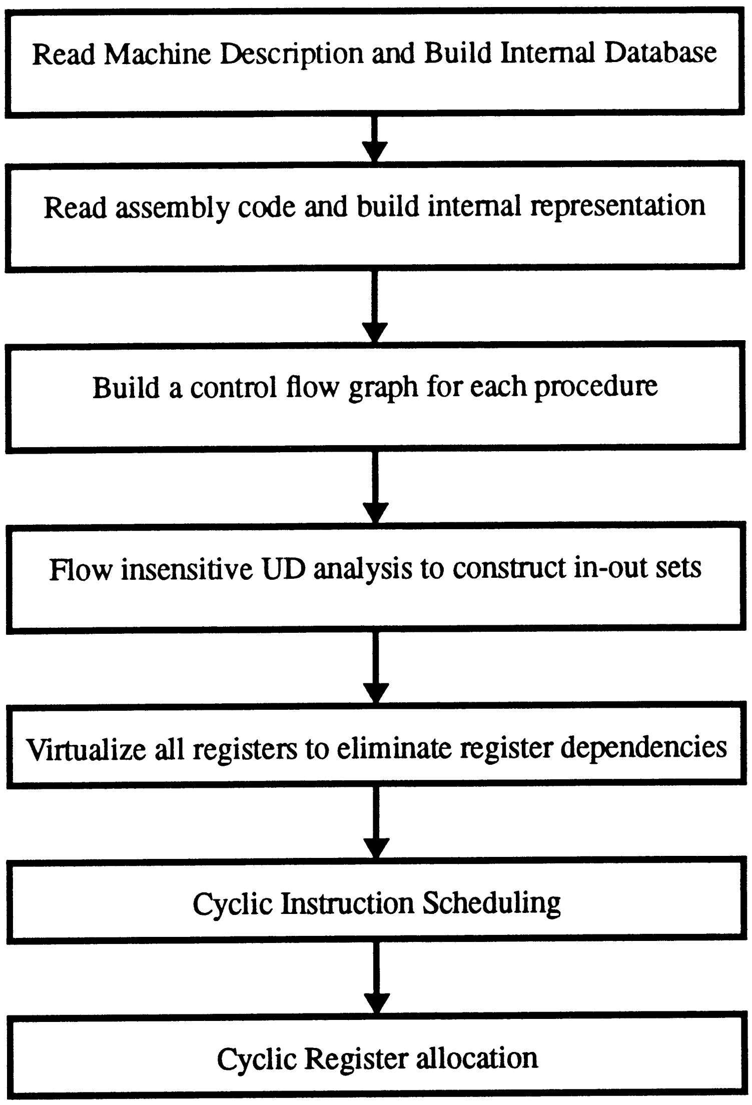
Read Machine Description and Build Internal Database
Read assembly code and build internal representation
Build a control flow graph for each procedure
Flow insensitive UD analysis to construct in-out sets
Virtualize all registers to eliminate register dependencies
Cyclic Instruction Scheduling
Cyclic Register allocation

Figure 4.1: Stages followed by the Object Code Optimizer

block from two different paths in the control flow graph. Similarly, *out(N)* is the set of all values that are live when N is completed [16].

Physical registers are then replaced by virtual registers, thereby transforming the loops to conform to the single assignment condition, *i.e.* each distinct value is assigned to a unique virtual register. Cyclic schedules are built using modulo reservation tables (MRTs) for do loops that do not contain conditional statements or function calls. Then cyclic register allocation of live values to physical registers is done using the technique presented in [17].

Every cyclic schedule has a *prologue*, an *epilogue*, a *kernel* and some *compensation code* associated with it. The *prologue* and *epilogue*, also known as transients, are necessary to fill and empty the pipeline, respectively. The *kernel* is the steady-state MRT scheduled code that is executed repeatedly after the *prologue* and before the *epilogue*. Loop unrolling may be applied by OCO before the cyclic scheduling is done. If the loop is unrolled k times, some *compensation code* is needed for those executions that perform a number of iterations that is not a multiple of k. The compensation code executes up to k-1 iterations so that the number of iterations executed in the cyclic schedule is always a multiple of k. OCO generates the *prologue*, *kernel*, and *epilogue* code, but the *compensation code* is not produced.

## 4.2 Performance of OCO Generated Code

The bounds model assumes parallelism in the instruction stream and full utilization of the busiest functional units. The heuristic approach to cyclic scheduling used by OCO attempts to achieve maximal utilization of the busiest functional units. OCO may be expected to produce schedules that approach the MAC bound by rescheduling the KSR compiler-generated code. If the input to OCO contains only the essential operations for the busiest functional unit, a schedule generated by OCO might approach the MA bound.

A reservation template for each class of KSR instructions was written to allow the use of this tool

Table 4.1: Performance of code produced by OCO

| LFK | Compiled (CPF, k=1) | MAC Bound (CPF, k = 1) | OCO (CPF) | % of MAC achieved by OCO |
|---|---|---|---|---|
| 1 | 2.45 | 1.2 | 1.22 | 98.36 |
| 2 | 3.06 | 2.25 | 2.72[a] | 82.76 |
| 3 | 2.57 | 2.0 | 2.02 | 99.0 |
| 4 | 2.71 | 2.5 | 2.57 | 97.26 |
| 5 | 4.6 | 4.0 | 4.02 | 99.5 |
| 6 | 4.25 | 3.0 | 4.05[a] | 74.09 |
| 7 | 1.66 | 0.75 | 0.76 | 98.68 |
| 8 | 1.01 | 0.72 | 0.99 | 72.72 |
| 9 | 1.33 | 0.88 | 0.95 | 92.63 |
| 10 | 3.06 | 2.44 | 2.48 | 98.39 |
| 11 | 6.08 | 4.0 | 4.02 | 99.5 |
| 12 | 6.09 | 5.0 | 5.03 | 99.4 |
| AVG | 3.24 | 2.40 | 2.57 | 93.42% |

a. Estimated performance

for the KSR1. The first 12 LFK loops were compiled with loop unrolling turned off[2] to allow a fair comparison between cyclic scheduling and loop unrolling. The branch overhead per iteration will be larger in this case as compared to the case where the loops are unrolled k times, because the overhead is incurred once each iteration instead of once every k iterations. Thus the MAC bounds calculated for this workload differ from the MAC bounds shown previously for the unrolled loops.

Initially, the schedules produced for only 5 of the 12 loops came close to the MAC bound with k=1. These were loops 3, 4, 5, 10 and 12. One major reason why the other loops suffered is that in the absence of any directive information, OCO makes the conservative, worst-case assumption that all loads

[2] The -O1 compiler option was used which performs all global optimizations.

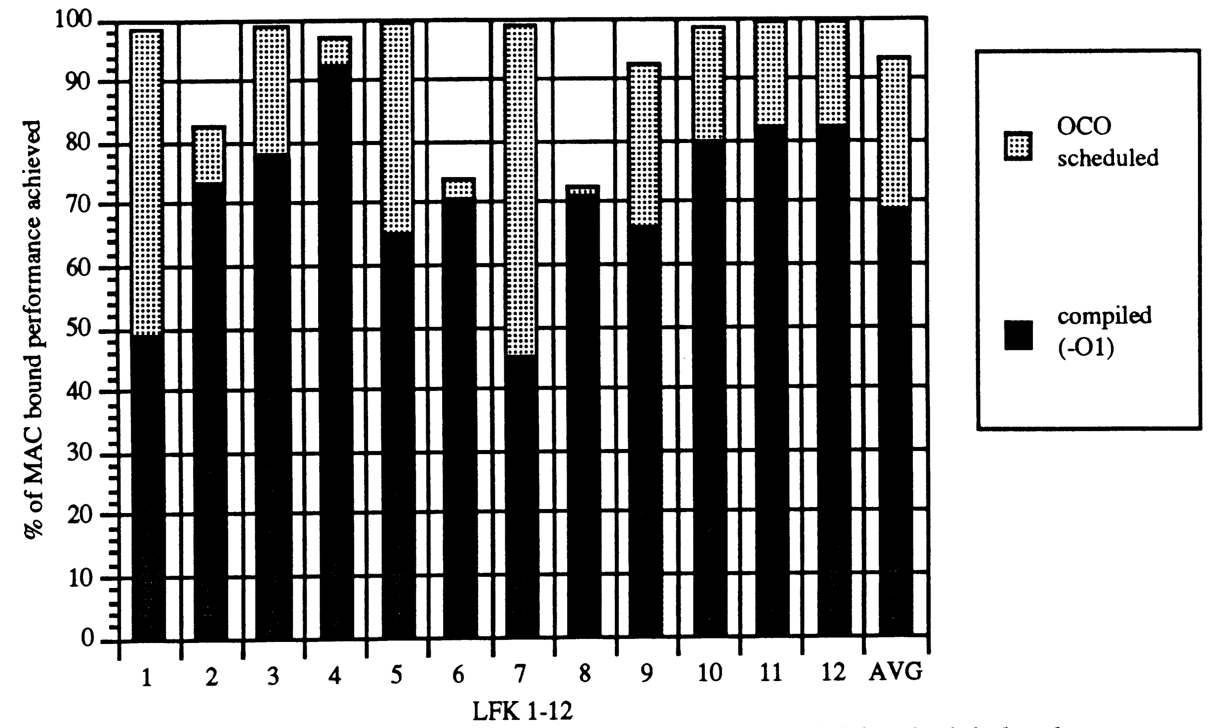

Figure 4.2: Performance of compiler-scheduled vs. OCO-scheduled code

and stores could reference the same memory location. Thus it produced sub-optimal schedules because it was unable to move any loads from later iterations above stores from previous iterations to fill empty slots in the modulo reservation table. It is, however, possible to use OCO directives to distinguish between dependent and independent memory references. In this mode of operation, all references that are not explicitly stated to be dependent are considered independent, there by providing the tool with the necessary degree of freedom to move operations across iterations when it fills in the reservation table slots.

Table 4.1 shows the performance of the code after inclusion of directives to indicate dependent memory references. For 8 of the 12 loops (loops 2, 6, 8 and 9 being the exception), near optimal schedules were produced that reached at least 97% of the ($k = 1$) MAC bound. On the average loops 1-12 displayed a 1.27 times speedup over the compiler scheduled code and achieved 93.42% of the MAC bound. Loops 2 and 6 step through a triangular iteration space, and since the compensation code for the new schedules was not generated, it was not possible to measure the performance of these loops from their execution times. Instead, we conservatively assumed that the compensation code would require as many cycles per iteration as the loop produced by the compiler without unrolling, and calculated the number of iterations executed in the cyclic-scheduled code and in the compensation stub. Each instruction pair in these codes was assumed to take one cycle -- an accurate assumption considering that the small data set size causes no subcache misses, and therefore no processor stalls. Given this information it was fairly straightforward to estimate the time needed to execute the loop.

### 4.2.1 Estimating Execution Time for Loop 6

As for all loops, the number of inner loop iterations executed in the loop is some function of program parameter n. For loop 6, the total number of inner loop iterations, summed over all n executions of the inner loop is $n \times (n-1)/2$. The cyclic-scheduled loop is unrolled twice by OCO, *i.e.* each pass through the MRT in steady-state starts two iterations of the loop. Therefore, when n is even there is no compensation code; when n is odd, one iteration is executed in the stub. Compensation code is thus required for $\lceil n/2 \rceil$ executions of the n executions of the inner loop, and therefore $\lceil n/2 \rceil$ iterations will be executed in the compensation code.

Let C be the number of clock cycles required for one iteration in the compensation code, where C is set equal to the number of cycles required in the steady-state for one iteration of the compiled loop with k=1. Let T be the total application run time (in clock cycles) per iteration for the compiled loop (k = 1), and II be the initiation interval for the cyclic-scheduled loop. As seen in section 3.3, loop 6 exhibits some overhead which contributes to the total application run time. This overhead, O, is equal to T-C. Some additional cycles, P, are required for executing the prologue and epilogue. Then the total number of clock cycles required for n executions of the loop is

$$\left(\frac{n\times(n-1)}{2}-\left\lceil\frac{n}{2}\right\rceil\right)\times II+\left(\left\lceil\frac{n}{2}\right\rceil\times C\right)+\left(\frac{n\times(n-1)}{2}\times O\right)+(n\times P)$$

This performance estimate yields 4.05 CPF for the cyclic-scheduled loop 6, which achieves 74.09% of the MAC bound (k = 1) performance. In the absence of any overhead, *i.e.* if we ignore O in the above expression, the cyclic-scheduled loop's estimated performance would be 3.17 CPF which achieves 94.6%of the MAC bound (k = 1) performance.

### 4.2.2 Estimating Execution Time for Loop 2

The total number of outer loop iterations for an entire run of the loop is $\lfloor log_2 n \rfloor$ [5], where n is a program parameter. For each iteration of the outer loop, a certain number of iterations of the inner loop is performed. This number is determined by another factor, say m, which is dependent upon the initial value of n. In particular

$$\text{for } i = 2 \text{ to } \lfloor log_2 n \rfloor, \qquad m_i = \left\lfloor\frac{m_{i-1}}{2}\right\rfloor \quad \text{and } m_1 = n$$

For each outer loop iteration, i, the number of iterations of the inner loop is $\left\lceil\frac{m_i-2}{2}\right\rceil$.

For a given value of n, the number of iterations with an odd inner loop limit can be calculated from the two relations presented above. Since the loop 2 is also unrolled twice by OCO, loop executions with an odd loop limit will perform one iteration in the compensation code, while all other iterations will execute in the cyclic-scheduled code. Overhead is calculated and included in the estimate as for loop 6. This performance estimate yields 2.72 CPF for the cyclic-scheduled version of loop 2, which achieves 82.72% of the MAC (k = 1) bound performance. If we ignore overhead, the estimated performance would be 2.47 CPF which is 91.1% of the MAC (k = 1) bound performance.

If only the steady-state inner-loop performance for loops 2 and 6 is considered, the harmonic mean performance of LFK 1-12 would achieve 96.98% of the MAC bound performance.

## 4.3 The Template Packing Problem of the KSR1

Loop 8 presents us with a situation where it is not possible to generate a tightly packed schedule that meets the k = 1 MAC bound because it is not possible to place the instruction templates without overlap in a 26 column reservation table. Consider the task of scheduling a floating point add (*fadd*) instruction which follows two floating point multiply-add (*fmad*) instructions. The triad instructions are issued in consecutive cycles, and we wish to schedule the add instruction as early as possible. Assume that there are no data dependencies between these instructions and for the sake of simplicity we shall assume that the CEU/XIU side issues only *nops*. Thus we have the following instruction pairs:

```
(0)        fmad8.tr    %f7, %f4, %f6      ;   cxnop

(1)        fmad8.tr    %f2, %f9, %f12     ;   cxnop

(2)        fadd8.tr    %f3, %f5, %f3      ;   cxnop
```

Figure 4.3: Example code segment

Table 4.2 is a reservation table, where we indicate the use of resources by instructions by entering each instruction pair number in the relevant cells. The triads are launched in cycles 0 and 1, they reserve the multiply unit immediately for one cycle, but wait one cycle before making use of the floating point add resource. Thus the add resource is in use in cycles 2 and 3. The add instruction needs to reserve the add resource in the same cycle it is issued in, and therefore must be delayed until cycle 4. The bounds model would predict a bound of 3 clock cycles (or 0.6 CPF) for this code segment since no resource is in use for more than three clock cycles. However, it is impossible to find a schedule with an initiation interval of 3. This is a case where the bound is not achievable because it ignores the effect of packing instruction templates of different shapes into a reservation table.

Table 4.2: Reservation table for code segment in figure 4.3

| | Cycle 0 | Cycle 1 | Cycle 2 | Cycle 3 | Cycle 4 |
|---|---|---|---|---|---|
| Issue | 0 | 1 | | | 2 |
| FPU Mult. | 0 | 1 | | | |
| FPU Add | | | 0 | 1 | 2 |

Note here that the functional units can be fully utilized if multiply-add triad and add instructions could be issued alternately, since that does allow the templates to be packed together tightly. For this example, it is possible to issue the *fmad* instructions in cycles 0 and 2 and the *fadd* in cycle 3, thereby permitting a new iteration to begin every 4 cycles. However, there are 15 combinable multiply-add instructions and 6 non-combinable add/subtract instructions in each iteration of loop 8, making it impossible to fully utilize any resource, in particular the Issue and the FPU Add resources.

Loop 9 has 7 floating point multiply-add triads, 2 floating point adds and 1 floating point multiply for each iteration of the loop. The fact that the scheduler faces the same template packing problem, should be masked there because the bottleneck is on the CEU/XIU side of the issue. Loop 9 still achieves only 92.96% of the MAC bound. We believe that is due to the fact that OCO uses list scheduling without backtracking in its heuristic approach to cyclic scheduling and the order in which the instructions are inserted in the MRT happens to result in a larger than expected initiation interval. Based on the k=1 MAC bound for loop 9, $t_l$ is 15 but the initiation interval for the cyclic-scheduled code is 16. For the same reason the initiation interval for loop 8 is 36, although it should be possible to find a schedule that deals with the template packing problem described above and has an initiation interval of 30.

This problem of packing together complex and asymmetric instruction templates also exists for machines like the Astronautics ZS-1 [6]. Other machines avoid getting into this situation by using similar templates for all combinable and non-combinable floating point instructions, *e.g. in* the IBM RS/6000 all non-combinable adds and multiplies go through the add-multiply pipeline and have the same latency as a triad.

## 4.4 Shortcomings of List Scheduling for the KSR1

The code segment presented in figure 4.3 illustrates a case where the bounds model does not produce an achievable bound because the instruction templates have incompatible shapes that cannot be packed tightly. There might be other instances when there exists a schedule that meets the bound, but the order in which list scheduling inserts instructions in the reservation table leads to sub-optimal schedules.

Consider the problem of scheduling two floating point multiply-add triads and two floating point adds. As before, we assume that there are no data dependencies between these instructions and that the CEU/XIU side of the instruction pairs contains only *nops*. The 4 instruction pairs are shown in figure 4.4.

```
(1)         fmad8.tr    %f2, %f7, %f4     ;   cxnop

(2)         fmad8.tr    %f8, %f5, %f10    ;   cxnop

(3)         fadd8.tr    %f6, %f13, %f12   ;   cxnop
```

Figure 4.4: Another example code segment

The bound for these instructions is an initiation interval of 4. This bound is achievable in practice if the two different instruction types are interleaved when they are issued. The reservation table for such a stream of instructions where the issue order is 1, 3, 2, 4 is shown in table 4.3. Note that a new iteration can begin in cycle 4 and the Issue and the FPU Add resources are fully utilized.

Table 4.3: Reservation table for issue order 1, 3, 2, 4

| | Cycle 0 | Cycle 1 | Cycle 2 | Cycle 3 | Cycle 4 |
|---|---|---|---|---|---|
| Issue | 1 | 3 | 2 | 4 | |
| FPU Mult | 1 | | 2 | | |
| FPU Add | | 3 | 1 | 4 | 2 |

The list scheduling approach used in OCO does not backtrack and the schedule it generates is affected by the order in which the instructions are inserted in the reservation table. If the two triads are inserted first followed by the two add operations, a schedule with an initiation interval of 6 results, as shown in table 4.4. This demonstrates that the order in which operations are inserted in the reservation table is extremely important for list scheduling. Chapter V considers the problem of optimal scheduling of essential operations.

Table 4.4: Reservation table for issue order 1, 2, 3, 4

| | Cycle 0 | Cycle 1 | Cycle 2 | Cycle 3 | Cycle 4 | Cycle 5 |
|---|---|---|---|---|---|---|
| Issue | 1 | 2 | | | 3 | 4 |
| FPU Mult | 1 | 2 | | | | |
| FPU Add | | | 1 | 2 | 3 | 4 |

# CHAPTER V
# APPROACHING THE MA BOUND

## 5.1 Scheduling a Loop Body

The objective of using OCO to generate software pipelined schedules was to demonstrate how well the KSR1 compiler-generated code can be restructured automatically, *i.e.* to display the effectiveness of cyclic-scheduling for the KSR1 by showing where the improved performance lies relative to the compiled code on the one hand and the MAC bound on the other. Cyclic scheduling is known to be a very useful approach for hiding latency. It is therefore expected to yield the greatest benefits in an architecture with deep pipelines. The KSR1, on the other hand does not have deeply pipelined functional units. The floating point add and multiply units are only 2 stages each, and the subcache has a 2 clock access latency. Despite the small latencies, cyclic-scheduled code produced by OCO was observed to be within 97% of the k=1 MAC bound for 8 of the 12 loops, with loops 2, 6, 8 and 9 being the exception due to template packing problems, outer loop overhead, and the heuristic approach to cyclic scheduling used by OCO. OCO achieves a significant speedup over the -O1 option, k=1 compiled code, but does not approach the -O2, large k, compiled code in performance.

If OCO were given a set of operations that contained only the essential operations, it could apply cylic scheduling techniques to generate further improved schedules that might approach the MA bound. Eliminating redundant operations introduced by the KSR1 compiler, would require a preprocessing step to identify and eliminate these operations, leaving behind only the essential operations. The current implementation of OCO does not have any mechanism to eliminate redundant operations. In this situation, one option would be to do this preprocessing step by hand and then use OCO to reschedule the essential set of operations. This is not such a clean fix, since removing redundant operations from the code itself requires some degree of code restructuring. Since the effort required to do this is not much less than the effort for completely rescheduling the code by hand to produce an optimal schedule, we decided to hand code the first 12 LFK loops in an attempt to produce schedules that reach the MA bound.

Our approach to hand coding consists of removing all redundant operations from the compiler-generated code and rescheduling the essential operations. Since this approach is based on modifying the compiled code, the compiler-selected degree of unrolling, k, is not changed. Most of the redundant operations generated fall into two categories: redundant memory references and unnecessary register moves. It is difficult to isolate the effect of these factors and measure the performance gain achieved by eliminating each of them individually. This is due to the fact that the KSR1 is statically scheduled, and does not have any hardware interlocks. Since elimination of redundant operations therefore requires some degree of code rescheduling, the performance gain is influenced not only by the removal of these operations, but also by the new schedule. We discuss our treatment of these two categories of redundant operations separately in the following sections.

## 5.2 Redundant Memory References

All twelve loops examined, except loops 9 and 10, had at least one redundant load operation. Loops 2, 6 and 8 were the ones with the most extra memory references and are discussed separately.

### 5.2.1 Loop 2

This loop had redundant operations because the data flow analysis in the compiler failed to identify reuse of an operand across iterations, even when the loop was unrolled. Two code segments are presented. The first, in figure 3.1 is the loop code generated by the compiler. Note that there are multiple instances of two loads from the same memory location, *e.g.* 32 (%c9), into registers %f12 and %f14 (marked with * in the code). It is possible to eliminate the second load and substitute %f12 for %f14 as the second source in the triad instruction. This extra load is a recurring problem and shows up in each iteration. It is also not necessary to wait for the result of the triad instruction to be stored before initiating the operand loads for the next iteration. Delaying these loads would only be necessary if there were an insufficient number of registers available and the store operation would free up a register which could then

```
.L 293:
            itsteq8     8, %i11            ;  ld8     -8 (%c9), %f14
            sub8.ntr    %i11, 8, %i11      ;  ld8     -16024 (%c9), %f13
            add8.ntr    81, %i31, %i31     ;  ld8     -16 (%c9), %f15
            finop                          ;  ld8     -16016 (%c9), %f0
            finop                          ;  ld8     0 (%c9), %f1
            fmas8.tr    %f13, %f15, %f14   ;  ld8     -16008 (%c9), %f5
            finop                          ;  ld8     -16000 (%c9), %f7
            fmas8.tr    %f0, %f1, %f14     ;  ld8     -15992 (%c9), %f9
            finop                          ;  ld8     -15984 (%c9), %f11
            finop                          ;  ld8     -15976 (%c9), %f13
            finop                          ;  ld8     -15968 (%c9), %f16
            finop                          ;  ld8     -15960 (%c9), %f17
            finop                          ;  ld8     -15952 (%c9), %f18
            finop                          ;  ld8     -15944 (%c9), %f19
            finop                          ;  ld8     -15936 (%c9), %f20
            finop                          ;  ld8     -15928 (%c9), %f21
            finop                          ;  ld8     -15920 (%c9), %f22
            finop                          ;  ld8     -15912 (%c9), %f23
            finop                          ;  ld8     -15904 (%c9), %f24
            finop                          ;  st8     %f14, 0 (%c10)
            finop                          ;  ld8     0 (%c9), %f6
            finop                          ;  ld8     8 (%c9), %f15
            finop                          ;  ld8     16 (%c9), %f10
            finop                          ;  ld8     24 (%c9), %f0
            finop                          ;  ld8     32(%c9), %f12 *
            fmas8.tr    %f9, %f10, %f0     ;  cxnop
            finop                          ;  cxnop
            fmas8.tr    %f11, %f12, %f0    ;  cxnop
            finop                          ;  cxnop
            finop                          ;  cxnop
            finop                          ;  st8     %f0, -48 (%c10)
            finop                          ;  ld8     32 (%c9), %f14 *
finop                            ;  ld8          40 (%c9), %f1
finop                            ;  ld8          48 (%c9), %f15
fmas8.tr     %f13, %f14, %f1     ;  cxnop
finop                            ;  cxnop
fmas8.tr     %f16, %f15, %f1     ;  cxnop
finop                            ;  cxnop
finop                            ;  cxnop
finop                            ;  st8          f1, -40 (%c10)
finop                            ;  ld8          48 (%c9), %f0
finop                            ;  ld8          56 (%c9), %f5
finop                            ;  ld8          64 (%c9), %f1
finop                            ;  cxnop
fmas8.tr     %f18, %f1, %f5      ;  cxnop
finop                            ;  cxnop
finop                            ;  cxnop
finop                            ;  st8          %f5, -32 (%c10)
finop                            ;  ld8          64 (%c9), %f5
finop                            ;  ld8          72 (%c9), %f6
finop                            ;  ld8          80 (%c9), %f7
fmas8.tr     %f19, %f5, %f6      ;  cxnop
finop                            ;  cxnop
fmas8.tr     %f20, %f7, %f6      ;  cxnop
finop                            ;  cxnop
finop                            ;  cxnop
finop                            ;  st8          %f6, -24 (%c10)
finop                            ;  ld8          80 (%c9), %f8
finop                            ;  ld8          96 (%c9), %f9
fmas8.tr     %f21, %f8, %f7      ;  cxnop
finop                            ;  cxnop
fmas8.tr     %f22, %f9, %f7      ;  cxnop
finop                            ;  cxnop
finop                            ;  cxnop
finop                            ;  st8          %f7, -16 (%c10)
finop                            ;  ld8          96 (%c9), %f10
finop                            ;  ld8          104 (%c9), %f8
finop                            ;  ld8          112 (%c9), %f11
fmas8.tr     %f23, %f10, %f8     ;  sadd8.ntr    0, %c9, 128, %c9
finop                            ;  cxnop
fmas8.tr     %f24, %f11, %f8     ;  cxnop
finop                            ;  bcc.qn       @citst, .L293
finop                            ;  cxnop
finop                            ;  st8          %f8, -8 (%c10)
```

Figure 5.1: Compiler-generated code for loop 2

be used as the target of a load, or if the stored data was being reloaded for use in the next iteration. In this case, there are sufficient registers, but the compiler does not do sufficient address index analysis to determine that the loads do not depend on the store, nor is there a suitable directive (as in OCO) to inform it of this independence. The rescheduled code, shown in figure 5.2, reuses the value loaded into %f12. It also schedules loads early enough to avoid delaying execute operations that use the loaded values. Similarly stores are deferred.

By these techniques the number of loads per iteration is reduced from 3 to 2 and the *cxnop* instructions are eliminated. Thus the MA bound is achieved.

```
.L293:
            finop                            ;   ld8      -16 (%c9), %f15
.L294:
            itsteq8     8, %i11              ;   ld8      -8 (%c9), %f14
            sub8.ntr    %i11, 8, %i11        ;   ld8      -16024 (%c9), %f13
            finop                            ;   ld8      -16016 (%c9), %f0
            finop                            ;   ld8      0 (%c9), %f1
            fmas8.tr    %f13, %f15, %f14     ;   ld8      -16008 (%c9), %f5
            finop                            ;   ld8      0 (%c9), %f6
            fmas8.tr    %f0, %f1, %f14       ;   ld8      8 (%c9), %f15
            finop                            ;   ld8      -16000 (%c9), %f7
            finop                            ;   ld8      16 (%c9), %f8
            fmas8.tr    %f5, %f6, %f15       ;   ld8      -15992 (%c9), %f9
            finop                            ;   ld8      -15984 (%c9), %f11
            fmas8.tr    %f7, %f8, %f15       ;   ld8      32 (%c9), %f12
            finop                            ;   ld8      -15976 (%c9), %f13
            fmas8.tr    %f9, %f8, %f0        ;   st8      %f14, 0 (%c10)
            finop                            ;   ld8      40 (%c9), %f1
            fmas8.tr    %f11, %f12, %f0      ;   ld8      -15968 (%c9), %f16
            finop                            ;   ld8      48 (%c9), %f15
            fmas8.tr    %f13, %f12, %f1      ;   ld8      64 (%c9), %f17
            finop                            ;   ld8      56 (%c9), %f5
            fmas8.tr    %f16, %f15, %f1      ;   st8      %f15, 8 (%c10)
            finop                            ;   ld8      -15952 (%c9), %f18
            fmas8.tr    %f17, %f15, %f5      ;   ld8      -15944 (%c9), %f19
            finop                            ;   ld8      72 (%c9), %f6
            fmas8.tr    %f18, %f17, %f5      ;   ld8      80 (%c9), %f7
            finop                            ;   ld8      -15936 (%c9), %f20
            fmas8.tr    %f19, %f17, %f6      ;   ld8      -15928 (%c9), %f21
            finop                            ;   ld8      88 (%c9), %f8
            fmas8.tr    %f20, %f7, %f6       ;   ld8      96 (%c9), %f9
            finop                            ;   ld8      -15920 (%c9), %f22
            fmas8.tr    %f21, %f7, %f8       ;   ld8      -15912 (%c9), %f23
            finop                            ;   ld8      104 (%c9), %f26
            fmas8.tr    %f22, %f9, %f8       ;   ld8      -15904 (%c9), %f24
            finop                            ;   ld8      112 (%c9), %f15
```

```
fmas8.tr    %f23, %f9, %f26     ;   st8         %f0, 16 (%c10)
finop                           ;   sadd8.ntr   0, %c10, 64, %c10
fmas8.tr    %f24, %f15, %f26    ;   st8         %f1, -40 (%c10)
finop                           ;   sadd8.ntr   0, %c9, 128, %c9
finop                           ;   st8         %f5, -32 (%c10)
finop                           ;   st8         %f6, -24 (%c10)
finop                           ;   bcc.qn      @citst, .L294
finop                           ;   st8         %f8, -16 (%c10)
finop                           ;   st8         %f26, -8 (%c10)
```

Figure 5.2: Rescheduled code for loop 2

### 5.2.2 Loop 8

The loop 8 workload has 9 essential memory loads per iteration of the inner loop. The compiler generates 15 loads for each iteration of the loop. A closer look reveals that it is not trivial to eliminate these redundant loads because of the three address format KSR1 instruction set. Consider one of the instruction pairs from loop 8:

```
fmad8.tr   %f19, %f11, %f14  ;   cxnop
```

If it is possible to keep the value that is loaded in %f14 alive until the next iteration, it can be reused in another instruction, and a later reload can be eliminated. In the format shown above it is not possible to do so because %f14 is also the destination register and the value loaded in from memory gets overwritten by the result of this instruction. It is possible to save the value in %f14 if the one of the following two instructions is used, instead of the one shown above:

```
fmda8.tr   %f19, %f14, %f11  ;   cxnop
fmda8.tr   %f11, %f14, %f19  ;   cxnop
```

Here, the result produced is the same and the result register is now %f11 or %f19. Unfortunately, this approach does not work because the old value in %f11 is also needed in later instructions, and %f19 stores one of the constants which are loaded once outside the inner loop and used in every iteration. Due to such problems, we were able to eliminate only one of the six extra memory references from this loop.

Loop 8 also has a few move operations which are completely unnecessary, since in all cases the source register of the move can easily be kept alive until the destination register of the move is used. Removing these redundant moves requires changing later uses of the destination register to the source register. If by that time the source register has been used for another value, this new value can be assigned to a different register that is free--something which is easily accomplished in the KSR1 due to its ample number of registers.

It turns out that it is not necessary to eliminate more than one redundant load. There are 21 essential $t_f$ instructions and a 2 clock cycle branch overhead on the FPU/IPU side. The CEU/XIU side has a 3 clock cycle branch overhead and by eliminating one load we are left with 20 $t_m$ instructions. This means that the left and right side of the instruction pairs are now evenly balanced at 23 instructions and there is no need to eliminate any more loads. However, since there are 15 multiply-add(subtract) and 6 add(subtract) instructions in the 21 essential floating point instructions, this workload leads to the template packing problem discussed in example 2.3 and section 4.3. The best schedule we could find for this workload reached 82.93% of the MA bound.

One approach to solving this template packing problem is to replace some of the multiply-add triad instructions by separate multiply and add instructions. This might be useful because it could divide the workload evenly between triads and diads (multiply, add, and subtract) instructions. Unfortunately, this is not a workable solution here since it introduces situations where in an attempt to reach the bound, 2 floating point instructions and a load would complete in the same cycle and compete for use of the two floating point register file write ports. This is the problem discussed in example 2.2.

### 5.2.3 Loop 6

Loop 6 has only one extra load in the inner loop for a loop body that is unrolled 8 times. However, there are 8 stores, one for each iteration, none of which is necessary since all of them write to the same location in memory. These are trivially eliminated by removing the store operation and doing it only once upon exit from the loop. There are also a few move operations in this loop. Once again these appear for no good reason. A section of the code produced by the compiler is shown in figure 5.3.

```
fmad8.tr   %f14, %f15, %f9  ;  cxnop
finop                       ;  cxnop
finop                       ;  cxnop
finop                       ;  st8        %f9, 16024 (%c10)
mov8_8     %f9, %f10        ;  ld8        15984 (%c8), %f1
finop                       ;  cxnop
finop                       ;  cxnop
fmad8.tr   %f0, %f1, %f10   ;  cxnop
```

Figure 5.3: A section of compiler-generated code for loop 6

We can observe a certain similarity between this code segment and the one presented for loop 2. Once again, it is unnecessary to wait for the result of the earlier triad to be stored before initiating a load which is needed for the triad in the following iteration. That load can be easily migrated above the store. Even with the presence of the redundant store, the mov8_8 instruction is not required. The register %f9 can be used in place of %f10 in the following execute instruction. A better schedule for the above instruction mix would be the one shown in figure 5.4.

```
finop                       ;  ld8        15984 (%c8), %f5
fmad8.tr   %f14, %f15, %f9  ;  ld8        2560 (%c7), %f16
finop                       ;  ld8        15976 (%c8), %f6
fmad8.tr   %f0, %f5, %f9    ;  ld8        3072 (%c7), %f17
```

Figure 5.4: A section of rescheduled code for loop 6

The store operation has been moved out of the loop. It is possible to pack instructions together in this way to produce a schedule that keeps the most heavily used resource, the CEU/XIU issue, busy during all cycles. Even though the resulting complete schedule is optimal it only reaches 63.64% of the MA bound. This performance gap is discussed in section 5.5.

## 5.3 Redundant Register Move Operations

Although redundant register move operations are also present in loops 6 and 8, their effect is by far the most noticeable in loop 7. Loop 7 also presents a situation where reuse of operands already residing in registers is not possible because some of these are overwritten when the register holding them ends up being the destination register of an instruction. The compiler recognizes this opportunity to use operands loaded from memory more than once, and saves them by copying them into other registers before their use. Unfortunately, that produces 11 move instructions for k = 4 (*i.e.*, 2.75 per iteration), leading to further congestion in the FPU/IPU issue unit which is already a bottleneck. About half of these moves are unnecessary since the value being moved also remains live in the source register at least until the first use of the destination register of the move. These operations are trivially removed by replacing the destination register of the move instruction by the source in all instructions following the move that use the moved value. If the source register of the move instructions is being used for another value after it has been copied, this new value can be assigned to a different register that is free.

The number of essential memory references in loop 7 is one-half the number of essential execute instructions. This fraction is even lower if the move instructions are considered. This means that some redundancy can profitably be introduced into the memory references in exchange for reducing the load on the FPU/IPU issue unit. Thus instead of saving operands for later use by move instructions, these are reloaded from memory. For k=4, this results in 32 $t_f$ instructions, all of which are essential, and 27 $t_m$ instructions of which 11 are redundant.

## 5.4 Performance of the Hand-Coded Loops

After removing redundant operations and doing some fine grain code scheduling, a speedup of 1.3 times is observed over the compiled code. The rescheduled code, with the same degree of unrolling, k, as the compiled code, achieved 87.58% of the MA bound, up from 67.34% for the compiled code. The degree of improvement for individual loops varied. The largest performance gain, a 2.18 speedup, was observed for loop 6, even though its performance is still far from the MA bound. At the other end of the spectrum, only a 1.04 speedup was achieved for loop 10 which was already close to the bound. Eight of the loops now achieve greater than 90% of the MA bound performance.

A cyclically scheduled loop body, in which the steady-state kernel had eight iterations of the loop, was generated for loop 1. The schedule suffers noticeably from the effect of a long prologue and epilogue and is therefore relatively far away from the bound at 91.56%. In the absence of a prologue the performance would be much closer to the MA bound since the MRT cell corresponding to the busiest functional unit is being fully utilized.

Loop 8 achieves only 78.05% of the bound for the reasons discussed in section 5.2.2.

In general, removal of redundant operations coupled with effective scheduling of the instructions that remain is the essence of our hand-coding approach.The resulting performance improvements indicate the gains that could be achieved by using better dataflow analysis techniques in the KSR compiler. In addition to loop 8, loops 2, 4 and 6 also exhibit performance that does not come close to the MA bound. This is counter to expectations since the steady-state inner-loop schedules for each of these loops is optimal, which in this case is the same as the bound. The reason for this seemingly poor performance is the outer-loop overhead, as shown in the next section.

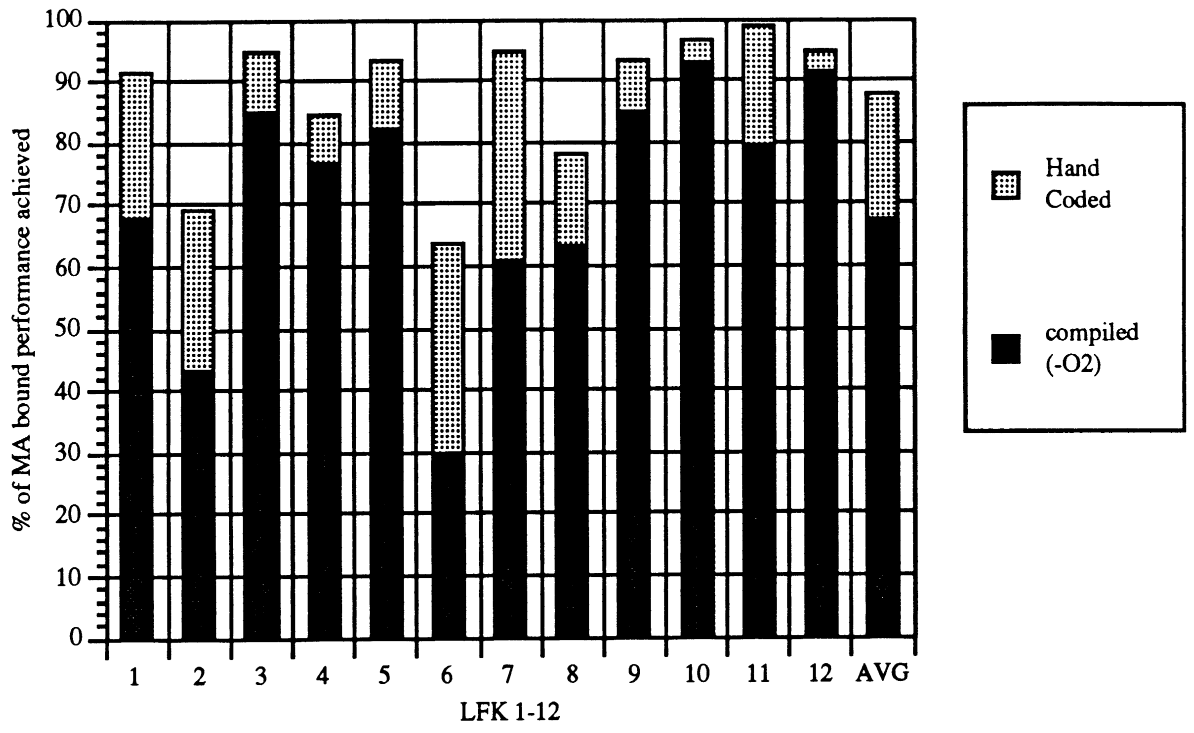

Figure 5.5: Performance results of the modified LFK 1-12

Table 5.1: Comparison of hand-coded vs. compiled LFK 1-12

| LFK | MA Bound (CPF) | Compiled (CPF) | Resched (CPF) | Speedup | % of MA bound achieved |
|---|---|---|---|---|---|
| 1 | 0.65 | 0.96 | 0.71 | 1.35 | 91.56 |
| 2 | 1.34 | 3.09 | 1.93 | 1.60 | 69.43 |
| 3 | 1.12 | 1.32 | 1.18 | 1.12 | 94.92 |
| 4 | 1.19 | 1.55 | 1.41 | 1.10 | 84.4 |
| 5 | 2.0 | 2.43 | 2.14 | 1.14 | 93.46 |
| 6 | 1.19 | 4.07 | 1.87 | 2.18 | 63.64 |
| 7 | 0.56 | 0.92 | 0.59 | 1.56 | 94.92 |
| 8 | 0.64 | 1.01 | 0.82 | 1.23 | 78.05 |
| 9 | 0.68 | 0.80 | 0.73 | 1.10 | 93.15 |
| 10 | 2.28 | 2.46 | 2.36 | 1.04 | 96.61 |
| 11 | 2.25 | 2.83 | 2.28 | 1.24 | 98.68 |
| 12 | 2.25 | 2.46 | 2.37 | 1.04 | 94.94 |
| Average | 1.34 | 1.99 | 1.53 | 1.30 | 87.58% |

## 5.5 Steady-State Inner-Loop Performance

The performance of loops 2, 4 and 6 remains noticeably less than the bound, even though their hand scheduled codes are optimal. Although we have attempted to include significant performance-limiting effects in the model, such as branch and iteration count overhead, there are some effects that are ignored in the bounds model. Our performance model ignores the effect of cache misses, although that is not an issue for the LFK data set. Cache misses do occur for the first iteration of a loop, but each code section is repeated a large number of times due to the presence of an outer timing loop, which makes the effect of the initial misses negligible. Stalls occur only when the processor misses in the subcache and these also do not occur frequently enough to have any effect.

For some of the loops, there might be some operations that need to be performed before iterating over the inner loop. For example, these may be related to setting up of the inner loop limit, as in loop 2. The bounds models ignore the effect of these instructions since this overhead is amortized over all of the inner loop iterations. However, if the inner loop limit is not large enough to mask this overhead, the performance measurements will include a significant term due to outer-loop overhead. In other words

Measured Performance = Actual Steady-State Inner-Loop Performance + Overhead (3.1)

Since the bound is for steady-state inner-loop performance it should be independent of the number of inner or outer loop iterations. It has been shown that by curve fitting, the measured CPF can be approximated by $k \times n^{-h} + c$, where n is program parameter found in each loop which determines the number of inner loop iterations per outer loop iteration [4] [5]. Curve-fitting is done by measuring CPF over a range of values of n, while keeping n small enough to avoid cache misses. The constant term c, is the achieved steady-state CPF for the inner loop, while $k \times n^{-h}$ accounts for the overhead incurred as a function of n. Table 5.2 presents the values of parameters k and h and the constant c, which were obtained from curve fitting for the rescheduled codes.

Table 5.2: Performance of LFK 2, 4 and 6 after eliminating outer-loop overhead

| LFK | k | h | c | MA bound | % achieved of MA |
|---|---|---|---|---|---|
| 2 | 127.94 | 1.19 | 1.37 | 1.34 | 97.81 |
| 4 | 4503.18 | 1.65 | 1.27 | 1.19 | 93.7 |
| 6 | 50.57 | 1.06 | 1.21 | 1.19 | 98.35 |

After eliminating outer loop overhead in loops 2, 4 and 6, all but loop 8 achieve at least 91.56% of the bound and the average CPF for LFK 1-12 is 1.41 which corresponds to 95.04% of the MA bound performance. This result shows that our simple performance model accurately characterizes the achievable steady-state inner-loop performance for these scientific codes, except for loop 8 which now runs at approximately 29.5 clocks per iteration, but suffers from a template-packing problem that the MA bound of 23 clocks does not model.

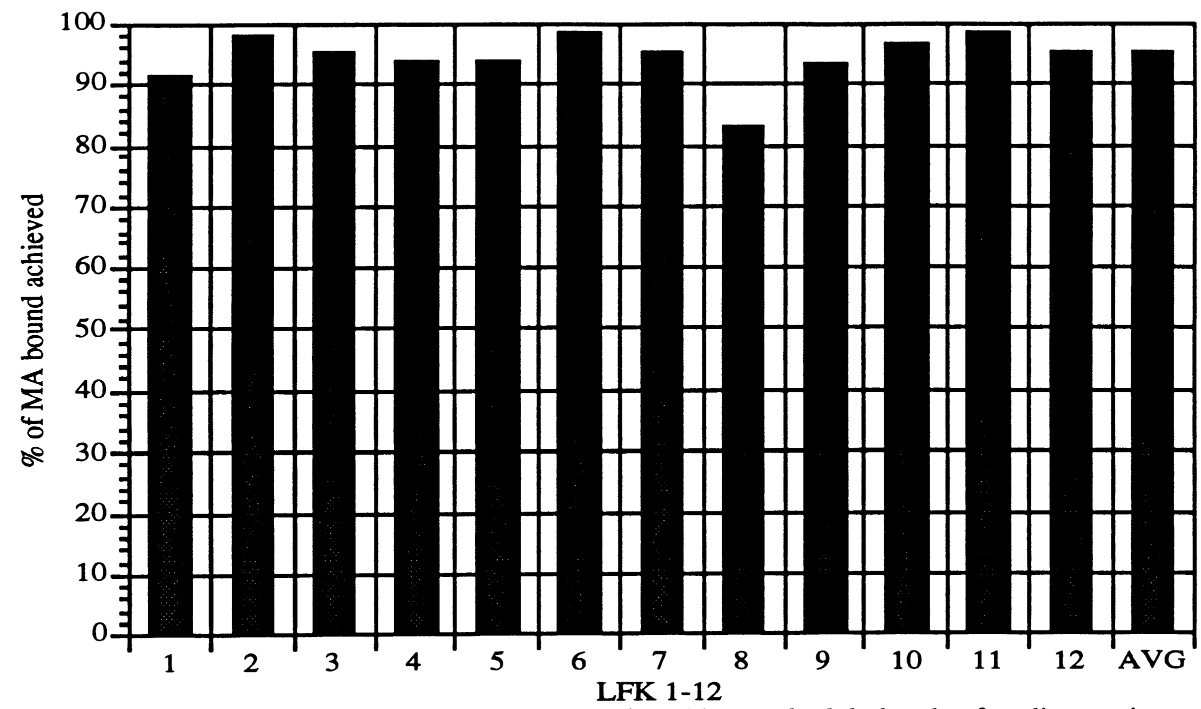

Figure 5.6: % of MA bound performance achieved by rescheduled code after discounting outer loop overhead in LFK 2, 4 and 6

# CHAPTER VI
# CONCLUSIONS

Application-specific bound models, such as the one developed in this study for the KSR1, permit effective performance evaluation of compilers and architectures. They are useful not only for improving the design of future compilers and machine implementations, but also highlight areas that require the attention of application programmers. Discovering generally applicable code optimization techniques for future compilers is one incentive to investing time and effort into hand coding such optimizations.

We make use of a hierarchical succession of bounds models that evaluate the potential of the machine implementation for a particular high-level application code (MA), the compiler-generated workload (MAC), and the actual compiler-generated schedule for that workload (MACS). This hierarchy of bounds highlights aspects of each kernel that need code optimizations and facilitates making effective decisions regarding which optimization techniques to apply in each case. These bounds provide metrics that can be used by compiler writers, system designers and application developers. The bounds model also helps in deciding when to give up optimization efforts aimed toward unrealizable performance improvements.

The first twelve LFK loops were compiled with the -O2 option of version 1.1.3 of the KSR1 Fortran compiler. The compiled code achieves only 67.34% of the MA bound and 69.34% of the MAC bound, but does achieve 92.5% of the MACS bound. This indicates that the largest performance degradation is due to inefficient scheduling of the compiler-generated workload. The overall effect of the difference between the number of operations in the compiler-generated workload and the essential set is not very large, although redundant memory operations do account for a large part of this difference. However, these redundant operations also introduce new dependence cycles within loop iterations making effective code scheduling a more complicated, or even an impossible task.

We hand-coded the first twelve loops in an attempt to produce schedules that meet the MA bound. Since the bound is calculated for the degree of loop unrolling introduced by the compiler, we did not attempt to alter the compiler-selected degree of unrolling when hand-coding. The main focus of our approach to hand-coding was elimination of redundant operations and fine-grain code scheduling of the remaining set of operations. We were able to closely pack together templates for these remaining instructions without major restructuring for all loops, except loop 1, where we had to resort to a cyclic schedule, and loop 8. Template packing problems arising from the different shapes of reservation templates among the various instruction classes are an impediment to closely packing the instruction templates in loop 8. It is for this reason that the hand-coded schedule for loop 8 achieves only 78.05% of the bound.

The three address register-to-register instruction set architecture of the KSR1 can inhibit register reuse in loops that contain triad instructions. Eight of the twelve LFK loops achieve greater than 91% of the MA bound performance with loops 2, 4, 6 and 8 being the exception. The average CPF of the hand-coded loops for LFK 1-12 is 1.53, which achieves 87.58% of the MA bound performance and a speedup of 1.3 over the -O2 compiled code. After eliminating outer loop overhead from loops 2, 4 and 6 so that delivered steady-state inner-loop performance can be measured, all but loop 8 achieve at least 91.56% of the bound. The average steady-state inner-loop CPF of the hand-coded loops for LFK 1-12 is 1.41, which achieves 95.04% of the MA bound performance.

We have adapted and used the Object Code Optimizer (OCO) which automatically reschedules code using cyclic scheduling. Using OCO, we have demonstrated that it is possible to automatically reschedule code for the KSR1 to produce close to optimal schedules in most cases. The schedules produced for the compiler-generated workload by OCO achieved 93.42% of the (k = 1) MAC bound performance with an average CPF of 2.57. For steady-state inner-loop performance the schedules produced by OCO achieved 96.98% of the (k=1) MAC bound with an average CPF of 2.48.

Our results indicate that future KSR compilers would benefit from a more efficient data flow analysis. The current compiler also does not schedule the compiler-generated code very well. Cyclic scheduling is superior to loop unrolling at the expense of higher code complexity and loop start-up time. A 1.26x speedup over -O1compiled code (k=1) resulted from the use of OCO. OCO would be even more effective for machines with more deeply pipelined functional units, *i.e.* with larger latencies to hide. It would also benefit from a preprocessor to eliminate redundant operations and choose an effective degree of unrolling, and from backtracking to improve template-packing when needed.

# REFERENCES

[1] *KSR1 Principles of Operation*, Kendall Square Research Corporation, 1991.

[2] *KSR1 Technical Summary*, Kendall Square Research Corporation, 1992.

[3] J. Tang, E.S. Davidson and J. Tong, "Polycyclic Vector Scheduling vs. Chaining on 1-Port Vector Supercomputers, *Proc. Supercomputing 88*, CS Press, Los Alamitos, Calif., Order No. FJ882, pp. 122-129, 1988.

[4] W. Mangione-Smith, T. P. Shih, S. G. Abraham and E. S. Davidson, "Approaching a Machine-Application Bound in Delivered Performance on Scientific Code," to appear *IEEE Proceedings (Special Issue on Computer Performance Analysis)*, **August 1993.**

[5] T. P. Shih, *Performance Evaluation of the IBM RS/6000*, Directed Study Report, EECS Department, University of Michigan, Ann Arbor, May 1992.

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

[7] W. Mangione-Smith, S. G. Abraham and E. S. Davidson, "Architectural vs. Delivered Performance of the IBM RS/6000 and the Astronautics ZS-1," *in Proc. Twenty-Fourth Hawaii International Conference on System Sciences*, pp. 77-88, January 1991.

[8] E. L . Boyd and E.S. Davidson, "Hierarchical Performance Modeling with MACS: A Case Study of the Convex-C240," *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 203-212, May 1993.

[9] M. Lam, *A Systolic Array Compiler*, Ph.D. Thesis, Carnegie Mellon University, 1987.

[10] B. R. Rau, C. D. Glaeser and R. L Picard, "Efficient Code Generation for Horizontal Architectures: Compiler Techniques and Architectural Support, " *Proceedings of the 9th Annual Symposium on Computer Architecture*, pp. 131-139, April 1982.

[11] R. F. Touzeau, "A Fortran Compilerfor the FPS-164 Scientific Computer," *in Proc. ACM SIGPLAN '84 Symposium on Compiler Construction*, pp. 48-57, June 1984.

[12] P. Y. Hsu, *Highly Concurrent Scalar Processing*, Ph.D. Thesis, Coordinated Science Laboratory Report #CSG-49, University of Illinois at Urbana-Champaign, 1986.

[13] W. Mangione-Smith, *Performance Bounds and Buffer Space Requirements for Concurrent Processors*, Ph.D. Thesis, EECS Dept., University of Michigan, Ann Arbor, 1992.

[14] 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.

[15] J. H. Patel and E. S. Davidson, "Improving the Throughput of a Pipeline by Insertion of Delays," in *Proc. of International Symposium on Computer Architecture*, pp 159-164, January 1976.

[16] A.V. Aho, R. Sethi, J. D. Ulman, "*Compilers: Principles, Techniques and Tools*," Addison-Wesley Publishing Company, 1986.

[17] C. Eisenbies, W. Jalby and A. Lichnewsky, "Compiler Techniques for Optimizing Memory and Register Usage on the Cray-2," *International Journal on High Speed Computing*, June 1990.