Hierarchical Performance Modeling with MACS: 
A Case Study of the Convex C-240

Eric L. Boyd and Edward S. Davidson
Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan

Abstract

The MACS performance model introduced here can be applied to a Machine and Application of interest, the Compiler-generated workload, and the Scheduling of the workload by the compiler. The MA, MAC, and MACS bounds each fix the named subset of M, A, C, and S while freeing the bound from the constraints imposed by the others. A/X performance measurement is used to measure access-only and execute-only code performance. Such hierarchical performance modeling exposes the gaps between the various bounds, the A/X measurements, and the actual performance, thereby focusing performance optimization at the appropriate levels in a systematic and goal-directed manner. A simple, but detailed, case study of the Convex C-240 vector mini-supercomputer illustrates the method.

1. Introduction

The MACS performance modeling techniques described in this paper facilitate critiquing an architecture, its implementation, and its compiler, as well as providing suggestions for their improvement. The model is based on a Machine and high–level Application code of interest, the Compiler-generated workload, and the Scheduling of the workload by the compiler. We define a hierarchy of bounds equations (MA, MAC, and MACS), each of which fixes the named subset of M, A, C, and S while allowing the others to be idealized. The MAC bound thus poses a limit on what the best possible schedule could achieve on the compiled code while the MA bound may be approchable by the best compiler/scheduler for the specified Machine-Application pair. In contrast, the peak floating point performance claimed for a machine’s “best possible” performance on an application by successively adding constraints on its performance. Hierarchical bounds-based performance modeling thus exposes and explains the specific gap for each pair of bounds equations and measurements. With this understanding, performance optimization can proceed in a systematic and goal-directed manner to minimize the gap between delivered and deliverable performance.

The MACS models are improvements of models originally developed for the Cray vector supercomputer systems [2] [3], the Decoupled Access-Execute (DAE) architecture Astronautics ZS-1 [1] [4], and the superscalar IBM RS/6000 [4] [5]. We have significantly expanded the functionality of this general performance modeling approach, and illustrate its utility with a detailed case study of the Convex C-240 vector mini-supercomputer [6] [7].

As with all vector machines, the delivered performance of the Convex C-240 on a well–vectorized scientific code is primarily related to the efficiency of implementation of inner loops, the architectural flexibility, and the bandwidths and latencies of the machine implementation. Although parallel processes can be executed on the Convex C-240, the throughput penalty of running parallel jobs on a heavily loaded machine makes single-threaded vectorized processes the most common mode of operation.

Our performance study of the Convex C-240 focuses on modeling the steady-state performance of the vector processor on vectorized inner loops, which is the first portion of the code to optimize for running a well-suited application on a vector machine. Ten of the Lawrence Livermore Fortran Kernels (LFKs) [8] are used as workloads in this case study. This benchmark set contains a variety of inner loops, each small enough to be considered in detail, yet representative of many vectorizable scientific codes.
2. Overview of the Convex C-240 Architecture

The Convex C-240 has four CPUs sharing the same physical memory and I/O subsystems. The effective system clock period is 40 ns. Each CPU is self-allocating, taking a task for itself when ready. Processes can be run in scalar mode, vector mode, or parallel vector mode. CPUs are single-issue; instructions are issued in order with hardware interlocks. Each CPU contains an Address/Scalar Unit (ASU) and a Vector Processor (VP).

The ASU contains the scalar function units, scalar registers, and cache. It controls all machine instruction execution and executes all scalar instructions, addressing functions for scalar operations, and most addressing functions for vector operations.

The VP concentrates solely on the execution of all vector operations. The VP data path includes eight vector registers of 128 elements each, a memory interface, the scalar processor interface, the vector merge register, and the three pipelined function units (load/store, add, multiply) used to perform all vector operations. The load/store function pipe is its only interface to memory. The add function pipe handles all types of additions, population counts, shifts, logical functions, and data type conversions. The multiply function pipe executes multiplications, divisions, square root operations, and vector edits. The three function pipes may execute different instructions concurrently. Operand chaining permits the output of one pipeline to be fed directly into the input of another pipeline.\(^1\) The VP accesses memory directly, bypassing the scalar unit data cache. The Convex C-240 is a load/store architecture with eight vector registers.

The memory subsystem has five ports: one for I/O operations, and one for each of the four independent CPUs. Memory contention is typically minimal assuming a reasonable pattern of memory accesses among different processes. The memory system consists of 32 banks in the standard system configuration. Each memory word is eight bytes and the bank cycle time is eight clock cycles. Under an ideal no-conflict situation, the four processors can request and the 32 memory banks can satisfy one memory access per processor per cycle.

3. Hierarchy of Performance Models and Measurements

We analyze the performance of a target architecture using the hierarchy of performance bound models and measurements shown in Figure 1. The performance models are constructed using selected parameters of the machine and static analysis of the high-level application code (A) of interest, the compiler-generated workload (C), and the scheduling (S) of the workload by the compiler. In ascending through the bounds hierarchy from \(t_{Ma}\), the model becomes increasingly constrained as it moves in several steps from potentially deliverable toward actually delivered performance, \(t_a\). Each step quantifies a performance gap associated with a particular cause. The model is constrained to fit a specific compiler-generated workload in \(t_{MAC}\), and then to fit a specific schedule for that workload in \(t_{MACS}\).

The several \(t_a\) and \(t_r\) parameters bound the minimum time for executing only the associated vector floating point operations and memory load/store operations, respectively. We extend the model analysis by measuring actual performance of the execute-only portion of the code, \(t_x\), the access-only portion, \(t_z\), and the entire code \(t_a\). Note that \(t_{MAC}\) and \(t_{Ma}\) are simply the maximum of their \(f\) and \(m\) components, while \(t_{MACS}\) and \(t_a\) are influenced by more complex interactions of the floating point and memory units, as discussed in Sections 3.4 and 3.6.

\[
\begin{align*}
\text{MEASURED TIMES} & \quad t_x \quad t_z \quad \text{MERGE} \Rightarrow t_a \\
\text{CALCULATED BOUNDS} & \quad t_f \quad t_w \quad \text{MERGE} \Rightarrow t_{MACS} \\
& \quad t_f \quad t_w \quad \text{MAX} \Rightarrow t_{MAC} \\
& \quad t_f \quad t_w \quad \text{MAX} \Rightarrow t_{Ma}
\end{align*}
\]

\textbf{Figure 1: Hierarchy of Performance Models and Measurements}

3.1. MA and MAC Performance Models

Performance analysis can be simplified by focusing on bottleneck functional units. In scalar machines such as the Astronautics ZS-1 and the IBM RS/6000, common bottleneck units include the instruction issue unit, the memory interface (load/store) unit, the floating point arithmetic unit (additions/multiplications), and a dependence pseudo-unit to model loop-carried dependence. \([4]\) For vector machines such as the Cray-1, the instruction issue unit is not a bottleneck and can be ignored. \([2]\) \([3]\) No true loop-carried dependence cycle appears in the ten LFKs of the case study workload and most memory accesses are unit stride. We thus focus our model of the Convex C-240 exclusively on bottlenecks caused by operations in the vector load/store, add, and multiply function pipes. The multiply and add pipes each have a peak rate of 1 floating point operation per

---

\(^1\) Superscalar machines such as the IBM RS/6000 mimic chaining in single-port vector machines by employing “combined multiply-add” instructions which can be issued simultaneously with one memory load or store (with address update), and test and branch. \([4]\) \([5]\)
clock cycle. The single memory port per CPU has a peak rate of 1 memory operation per clock cycle. The peak memory rate could be reduced for nonunit stride accesses by defining a fifth degree of freedom, D, after M, A, C and S to bind the allocation (decomposition) of the data structures in memory.

The MA bound, \( t_{\text{MA}} \), is a lower bound on Cycles Per inner Loop iteration, CPL, which models the application and the target machine. The application is defined by a particular high level code. Modifications to the high level code that may alter the essential workload are considered as creating a new application. The machine model characterizes the bottlenecks of interest in the target architecture. The MA bound for the Convex C-240 is calculated by counting the number of additions \( f_s \) and multiplications \( f_m \) that appear in the loop body of the high level code, and the necessary number of loads \( l_s \), and stores \( s_s \) of floating point data assuming perfect index analysis that eliminates reaccessing memory when data is reused among the iterations. The MA bound ignores additional operations introduced by the compiler and schedule-specific effects including dependence within and between iterations, enforcing Fortran precedence, register spilling, and bank conflicts due to nonunit stride memory accesses. The MA machine model assumes the effectiveness of compiler and schedule-specific effects including dependence within and between iterations, enforcing Fortran precedence, register spilling, and bank conflicts due to nonunit stride memory accesses. The MA machine model assumes that the \( f_s, f_m, l_s, \) and \( s_s \) operations can each be executed at one per clock. Hence \( t_m = l_m + s_m \), and \( t_{\text{MA}} = \max(t_m, t_m') \) in units of CPL.

The MAC bound, \( t_{\text{MAC}} \), also models the compiler (not including the scheduler) by counting all the operations of the classes of interest \( f_s', f_m', l_s', \) and \( s_s' \) in the compiler-generated assembly code rather than in the high-level source code. The MAC bound thus includes the effects of added operations, particularly from poor data reuse in the vector registers and spilling. Hence \( t_m' = \max(f_m, f_m'), t_m' = l_m' + s_m' \), and \( t_{\text{MAC}} = \max(t_m', t_m') \) in units of CPL.

Bounds and measurements in units of CPL can be converted to Cycles Per Floating point operation (CPF) by dividing by \( f_s + f_m \), the number of floating point arithmetic operations in the loop body of the high level code:

\[
t_{\text{MA}}(\text{CPF}):= \max(t_m / (f_s + f_m)) \quad (2)
\]

\[
t_{\text{MAC}}(\text{CPF}):= \max(t_m' / (f_s + f_m)) \quad (3)
\]

The average CPF over a set of applications yields their harmonic mean performance in megaflops as:

\[
\text{HMEAN(MFLOPS)} = \frac{\text{clockrate(MHz)}}{\text{averageCPF}} \quad (4)
\]

### 3.2. Pipeline and Memory Effects

The MA and MAC bounds model a single vector instruction as taking VL clock cycles (1 clock cycle per vector element) to complete. Unmodeled effects may, however, significantly reduce delivered performance. These include pipeline start-up, memory access conflicts, cache miss and page fault effects, and the time that a dynamic memory needs periodically to refresh its contents. These bounds thus produce performance goals to strive for, but better explanations of delivered performance require assessing schedule-specific phenomena, and a more detailed machine model.

Specially constructed calibration loops were used to verify specific aspects of machine performance and to confirm and clarify the start-up overheads claimed in [7]. Calibration loops are simple test loops constructed specifically for evaluating such parameters for a particular machine when detailed knowledge of the machine implementation or its minimum specifications is unavailable or needs to be confirmed. A similar approach is used in [9][10] to characterize memory hierarchy performance. The time to execute a single independent vector instruction can be expressed as:

\[
X + Y + Z * VL \quad (5)
\]

where for any vector instruction:

\[
X = \# \text{of clock cycles for initial overhead}. \quad (6)
\]

\[
Y = \# \text{of additional clock cycles until the first element result is available}. \quad (7)
\]

\[
Z = \# \text{of additional clock cycles per vector element to complete the instruction}. \quad (8)
\]

\[
VL = \text{Vector Length} \quad (9)
\]

The Convex-specified \( X, Y \), and \( Z \) parameters (listed in Table 1) were compared to the results of running calibration loops for several common vector instructions. To achieve a

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector load</td>
<td>ld.(a5),v0</td>
<td>2</td>
<td>10</td>
<td>1.00</td>
<td>2</td>
</tr>
<tr>
<td>vector store</td>
<td>st.(v0,(a5))</td>
<td>2</td>
<td>10</td>
<td>1.00</td>
<td>4</td>
</tr>
<tr>
<td>vector add</td>
<td>add.(v0,v1,v2)</td>
<td>2</td>
<td>10</td>
<td>1.00</td>
<td>1</td>
</tr>
<tr>
<td>vector multiply</td>
<td>mul.(v0,v1,v2)</td>
<td>2</td>
<td>12</td>
<td>1.00</td>
<td>1</td>
</tr>
<tr>
<td>vector subtract</td>
<td>sub.(v0,v1,v2)</td>
<td>2</td>
<td>10</td>
<td>1.00</td>
<td>1</td>
</tr>
<tr>
<td>vector divide(^a)</td>
<td>div.(v0,v1,v2)</td>
<td>2</td>
<td>72</td>
<td>4.00</td>
<td>21</td>
</tr>
<tr>
<td>vector reduction(^b)</td>
<td>sum.(v0)</td>
<td>2</td>
<td>10</td>
<td>1.35</td>
<td>0</td>
</tr>
<tr>
<td>vector negation(^b)</td>
<td>neg.(v0)</td>
<td>2</td>
<td>10</td>
<td>1.00</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table 1:** Vector Instruction Execution Times (VL = 128)

\(^a\) The extended number of cycles for a vector divide instruction may be masked by other instructions if no resource conflict exists.
\(^b\) It is claimed in [7] that for vector reduction, \( Z = 1.0 \). According to Patrick McGehearty at Convex Computer Corporation, \( Z = 1.5 \) for vector reduction instructions. Calibration loops determined that for vector reduction \( Z \) ranged between 1.39 and 1.43. For purposes of the MACS bounds equations, \( Z \) was set conservatively at 1.35. B is set to zero due to the uncertainty in \( Z \). Equivalently, we could set \( Z = 1, B = 45 \).
fit with measured calibration loop performance, we introduce an additional parameter, a bubble of B cycles, as described in Section 3.3.

Periodically the Convex C-240 memory must refresh itself. A memory refresh occurs every 16 µs (400 cycles) and lasts 8 cycles, a possible 2% performance penalty. If four vector memory accesses appear at the memory in succession, taking over 500 cycles total to complete, a memory refresh will force the VP to stall for eight cycles. If the vector memory accesses are not successive, the memory refresh might be masked during a time when no memory access operation is accessing memory. Memory refreshes that might be masked are never included in the bound. Some of the parameters given in Table 1 vary slightly when VL < 128, and in particular run time no longer improves when VL drops below some operation-specific threshold.

3.3. Vector Chime Effects

Vector computers such as the Convex C-240 have multiple function pipes and allow vector instructions with non-conflicting resource requirements to be issued in quick succession and executed concurrently. We refer to such a group of concurrent vector instructions as a chime. (Chime commonly refers to the unit of time taken by such a group of concurrent vector instructions.) Most vector computers, with the notable exception of the Cray-2, are designed to let multiple dependent vector instructions chain together. Chaining occurs when the output of one vector pipeline feeds directly into the pipeline of another vector function unit, so the two dependent vector instructions are executed concurrently in the same chime. For example, consider the following three instructions:

\[
\begin{align*}
\text{ld.l} & \ 0(a5), v0; & & V0 := \text{values starting at } a5 \quad (10) \\
\text{add.d} & \ v0,v1,v2; & & V2 := V0 + V1 \quad (11) \\
\text{mul.d} & \ v2,v3,v5; & & V5 := V2 \times V3 \quad (12)
\end{align*}
\]

Assume V1 and V3 already contain valid vectors and the vector length VL = 128. If the vector instructions did not chain, the first and second instructions would each take 2+10+VL cycles, and the third would take 2+12+VL cycles for a total of 422 cycles. With chaining, vector processor performance improves dramatically and the same three chained instructions take 162 cycles to complete, as shown in Figure 2.

The interaction of successive chimes yields an even larger improvement over the non-chaining approach. Assume that the chime in the above example is followed by an identical copy of the same three vector instructions in the same order (with a5 appropriately incremented). The first instruction of the second chain would begin at time 6, complete its initial overhead at time 8 and then immediately block, since the load/store pipe is busy. At time 130, the first instruction of the first chime begins to empty the load/store pipe. The first instruction of the second chime can now enter the load/store pipe (beginning its Y time). Its first element will be available at time 140, and that instruction will finish at time 140+VL = 268. Thus its X and Y parameters are essentially masked. The second instruction of the second chime will finish at time 150+VL = 278, and the third at time 290. The total time for the second chime is 290 - 162 = 128 = VL. This process can repeat many times, and even wrap around loop iterations. Most scalar instructions will be masked, and thus the average chime will asymptotically approach VL = 128 cycles to complete.

Tailgating occurs in a pipelined function unit as the first vector instruction begins to drain the pipeline (its last element leaves the first pipeline stage) and the first element of the next vector instruction using that pipeline follows it right in. The tailgating assumed in the above example does not actually work quite as smoothly as described. Results of our calibration experiments indicate that a bubble of one or more cycles in length occurs between successive instructions. We define B to be the number of cycles in a bubble. B is an empirical parameter not mentioned in [6][7], but ob-

![Figure 2: Chaining with Perfect Tailgating in the Function Unit Pipelines](attachment:image2.png)
served in the calibration loops. The experimentally derived values of B are defined in Table 1 for each type of instruction and are summed over all the instructions in a loop iteration. We conjecture that B may represent a handshaking restart penalty for a stalled instruction. Hence for one chime of instructions, Equation (5) is incorrect; the corrected formula to calculate the number of clock cycles to execute a chime (preceded by at least one chime) is given as:

\[
(Z) (VL) + \left( \sum_i B_i \right)
\]

(13)

where B is summed over the instructions in the chime. In the previous example, the B values add 4 cycles to each chime, resulting in 166 cycles for the first chime and 132 cycles per successive chime.

Chaining on the Convex C-240 has some inherent limitations, but it appears to be much more flexible than the Cray-1 or even the Cray–XMP. [3] A chime can include at most one vector operation on each of the three pipelines; regardless of dependence, chimes are also limited by which vector registers are utilized. The Convex C-240 permits at most two reads and one write to each vector register pair during a single chime of two or three instructions. The vector registers \( \{v0, v4\} \) are a vector register pair, as are \( \{v1, v5\} \), \( \{v2, v6\} \), and \( \{v3, v7\} \). The following example sequence of operations will not execute in the same chime since there are more than two read references to the vector pair \( \{v2, v6\} \). [7]

\[
\text{add.d v2,v6,v6; } V6 := V2 + V6
\]

(14)

\[
\text{mul.d v6,v1,v4; } V4 := V6 \ast V1
\]

(15)

Likewise, the following sequence of operations will not execute in the same chime since there is more than one write reference to the vector pair \( \{v2, v6\} \). [7]

\[
\text{add.d v1,v0,v2; } V2 := V1 + V0
\]

(16)

\[
\text{mul.d v2,v1,v6; } V6 := V2 \ast V1
\]

(17)

If scalar instructions are interspersed among vector instructions, chimes of vector instructions will often mask the scalar instructions as there is no resource contention between the ASU and the VP. However, since there is only one port between the CPU and memory, both vector loads and stores and scalar loads and stores must compete for that resource. As a result, a chime including a vector memory access cannot span a scalar memory access instruction. The chime will be terminated just before the scalar or vector memory reference instruction, whichever comes later. A new chime will begin with the next vector instruction. Although it is unclear from [6] [7], some vector instructions utilizing a scalar register as an operand may exhibit resource contention with purely scalar instructions. This effect was not observed in the loops we studied.

### 3.4. MACS Performance Model

The MACS performance model attempts to accurately incorporate the effects of the application, the compiler-generated assembly instruction counts, and the compiler-generated instruction schedule (specifically including the effects of the architectural details discussed in Sections 3.2 and 3.3) to bound steady-state loop performance on a machine. By definition, the MACS bound is specific to a particular code schedule. Reordering the sequence of instructions or reallocating the registers may change the MACS bound, but it will not change the MA or MAC bounds.

The general approach taken in developing the MACS model is to add the number of cycles it takes to execute each chime and divide by VL to achieve a new bound, \( t_{MACS'} \) in units of CPL. In contrast, the MA and MAC bounds equations assume that it takes one clock cycle to operate on each element of a vector instruction and that all possible parallelism between functional pipes is exploited. Thus in the MA and MAC bound models, one iteration of a vectorized loop is assigned a number of cycles equal to VL times the maximum of the number of vector additions, vector multiplications, and vector memory access operations counted.

To calculate the MACS bound, the vectorized inner loop of the compiled code is first partitioned into chimes, according to the rules described in Section 3.3. In most applications that fully exploit functional unit parallelism, the number of chimes will equal the maximum of the number of vector additions, multiplications, and memory access operations. The first vector instruction of a chime contributes B + VL cycles to the total number of cycles for the chime. The second and third instructions of the chime, if they exist, each add B cycles to the time for the chime. The interaction between chimes is effectively modeled by including the bubble B associated with the last instruction in each chime. The interaction of the last chime in the loop followed by the first chime in the loop must also be considered.

The number of cycles for all chimes in one iteration of the inner loop is then totaled. For each group of four or more successive chimes that include one memory operation per chime, the total number of cycles for the group is multiplied by 1.02 to account for the memory refresh penalty. This total is then divided by VL to determine \( t_{MACS'} \) in units of CPL. To obtain the CPF-bound, \( t_{MACS'} \) is divided by \( f_{uf} + f_{eq} \).

Additional insight may be obtained by applying the MACS bound calculation algorithm to a reduced instruction list. In particular, \( t_{MACS'} \) is calculated by applying the MACS algorithm to the compiled application code with all the vector memory access operations deleted. Likewise, \( t_{MACS''} \) is calculated by applying it to the compiled application code with all the vector floating point operations deleted. \( t_{MACS'} \) is not simply the maximum of \( t_{MACS''} \) and \( t_{MACS''} \) due to resource conflicts and bursts of f-only and m-only operations that prohibit a per-
fect merging of the $f$ and $m$ instructions into chimes for the total code.

Calculation of MACS bounds for chimes involving reductions and/or divisions is somewhat more complicated, involving numerous special cases, and is not discussed.

### 3.5. Example: Calculating MA, MAC, and MACS Bounds

As an example, LFK1 is analyzed using the MACS performance model. The code for LFK1 is relatively straightforward and easily vectorizable. The Fortran listing is as follows:

```fortran
1001 DO 1 k = 1,n
1 X(k)=Q+Y(k)*(R * ZX(k+10)+ T*ZX(k+11))
```

The resulting assembly code for the inner loop is:

<table>
<thead>
<tr>
<th>Chime</th>
<th>Line</th>
<th>L7:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01:</td>
<td>mov s0,VL ; #145</td>
</tr>
<tr>
<td>2</td>
<td>02:</td>
<td>ld1 space1+40120(a5),v0 ; #146, ZX</td>
</tr>
<tr>
<td>2</td>
<td>03:</td>
<td>mul.d v0,s1,v1 ; #146</td>
</tr>
<tr>
<td>2</td>
<td>04:</td>
<td>ld1 space1+40128(a5),v2 ; #146, ZX</td>
</tr>
<tr>
<td>2</td>
<td>05:</td>
<td>mul.d v2,s3,v0 ; #146</td>
</tr>
<tr>
<td>2</td>
<td>06:</td>
<td>add.d v1,v0,v3 ; #146</td>
</tr>
<tr>
<td>3</td>
<td>07:</td>
<td>ld1 space1+32032(a5),v1 ; #146, Y</td>
</tr>
<tr>
<td>3</td>
<td>08:</td>
<td>mul.d v1,v3,v2 ; #146</td>
</tr>
<tr>
<td>3</td>
<td>09:</td>
<td>add.d v2,s7,v0 ; #146</td>
</tr>
<tr>
<td>4</td>
<td>10:</td>
<td>st.l v0,space1+24024(a5) ; #146, X</td>
</tr>
<tr>
<td>11:</td>
<td></td>
<td>#1024,a5 ; #146, 0x400</td>
</tr>
<tr>
<td>12:</td>
<td></td>
<td>#128,s0 ; #146, 0xffffffff</td>
</tr>
<tr>
<td>13:</td>
<td></td>
<td>lt.w #0,s0 ; #146, 0x0</td>
</tr>
<tr>
<td>14:</td>
<td></td>
<td>jbrs.t L7 ; #146</td>
</tr>
</tbody>
</table>
```

A vector instruction is taken to be any instruction that accesses at least one of the eight vector registers \{v0, . . . , v7\}. The workload parameters for LFK1 are given in Table 2, including MAC counts only when they differ from MA counts. Thus $t_{MA} = 3$ CPL = 0.6 CPF and $t_{MAC} = 4$ CPL = 0.8 CPF. To evaluate the MACS bound, we partition the loop into chimes as shown above.

- **Chime 1** contains two vector instructions (load followed by multiply). Chaining will occur since there are one read and one write to the \{v0, v4\} array, and zero reads and one write to the \{v1, v5\} array. The vector load is assigned 24+128 cycles and the vector multiply adds an additional cycle for a total of 131 cycles. A calibration loop duplicating chime 1 yields a time of 131.93 cycles.
- **Chime 2** contains three instructions that chain together. VL + the sum of the B values for this chain is 128 + 4 = 132 cycles. A calibration loop duplicating chime 2 yields 133.33 cycles.
- **Chime 3** is also assessed 132 cycles. Its calibration loop also yields 133.33 cycles.
- **Chime 4** contains a single store instruction with B = 4, resulting in 132 cycles. Its calibration loop yields 132.35 cycles.

The sum of the MACS time bounds for all four chimes is 527 cycles. The sum of the individual experimental timings for all four chimes is 530.94 cycles. When the full LFK1 code is run (all four chimes in sequential order repeatedly), the measured time is 545.28 cycles. Each of the four chimes contains one memory access operation resulting in an additional 2% performance penalty. The $t_{MACS}$ bound is thus 537.54 cycles for one iteration of vectorized code. Thus $t_{MACS} = 537.54 / 128 = 4.200$ CPL = 0.840 CPF for LFK1. The actual measured time is 0.852 CPF.

### 3.6. A/X Performance Measurements

To achieve the goal of decreasing the actual run time toward $t_{MAC}$ (or less aggressively $t_{MACS}$) or to tighten the bounds toward actually achievable performance, a simple approach to locating problematic sections of code is required. Additional information on the potential performance of an application may be gleaned by measuring the actual performance of modified versions of the application.

A logical division of the code into memory access operations and floating point operations suggests itself naturally. [1] This view springs from the Decoupled Access–Execute (DAE) view of computing as two distinct concurrent processes. The access process (A) is responsible for accessing memory. The execute process (X), is responsible for executing functional operations on program data.

In essence, the performance of the A-process, $t_a$, is determined by actually running the application code with all of the vector floating point operations removed. Since scalars are unchanged, control flow is unaffected, but the incremental effect on run time caused by the floating point vector functional units over the time required by A-process instructions alone is removed. Likewise, the performance of the X-process, $t_x$, is determined by running the application code with all of the vector memory access instructions removed. Again control flow is unaffected, but the incremental effect on run time caused by servicing memory requests over the time required by the X-process alone is removed.

The A/X performance measurements are not performance bounds per se, but rather they measure actual machine performance with certain potential bottlenecks entirely eliminated. Normally we would expect:

$$
\text{MAX} \left( t_x, t_a \right) \leq t_p \leq \left( t_x + t_a \right)
$$

\[ (18) \]

---

2. Control flow is preserved in the kernels under observation since vectorization does not occur over operations on data that affect conditional control flow. The loop counter is not modified by vector operations and no floating point data dependent conditional branches occur in these loops.
where \( t_p \) is the actual total application performance. The numerical program outputs obtained when timing either the A-process or the X-process alone are nonsense, but the timing results indicate the degree of access-execute overlap according to where \( t_p \) lies in the range shown above. Overlap is poor when \( t_x \) and \( t_a \) are both far from \( t_m \). Performance is bottlenecked on X or A when \( t_p \) is close to \( \text{MAX}(t_x, t_a) \) and \( t_x \) and \( t_a \) differ significantly. The performance evaluation hierarchy in Figure 1 also relates to \( t_m' \) and \( t_m'' \). Gaps between them can indicate unmodeled system effects that may need attention within the X or A-processes themselves.

We have developed tools that automatically generate both A-process and X-process executable codes given the assembly code listing of the original compiled code. Although generating the A-process testing code is relatively straightforward, the process of generating the X-process testing code must prime all registers with large, relatively prime, nonzero, floating-point numbers in order to avoid floating point exceptions induced by performing illegal calculations on nonsense data.

### 4. Timing and Model Evaluation

The MA, MAC, and MACS performance models and the A/X performance measurements have been evaluated for the Convex C-240 using ten of the first twelve kernels of the LFKs as a case study. Insights on each kernel's performance are developed by examining the gaps between the various bounds and measurements within the hierarchy.

#### 4.1. Derivation of the MA, MAC, and MACS Bounds

In Table 2, the MA workload for each kernel as well as the MAC workload generated by the C-240 \( fc \) Fortran compiler V6.1 are presented. The MAC values, shown in columns \( f_m' \) and \( l_m' \), show the MAC count of vector adds and loads only where they differ from the MA counts. A dashed line represents no change from the MA counts. No difference was observed between \( f_m' \) and \( f_a' \) or between \( s_m' \) and \( s_a' \) in the kernels examined. These MA and MAC values serve as parameters for the model given in Section 3.1 which yields the MA and MAC performance bounds shown in Table 3.

The actual number of memory access operations per cycle, \( t_m'' \), dominates the MAC bound in all 10 LFKs, and the “excess” memory operations inserted by the compiler cause some significant increases in the MAC bound over the MA bound. The MA bound is itself memory limited except for LFKs 7 and 8.

The MACS performance bounds are also shown in Table 3. A bound for each kernel is calculated by a direct examination of the compiler-generated assembly language source code, as shown for LFK1 in Section 3.5. Fractional values are a result of fixed B overheads for each vector instruction (which are later divided by VL) and from memory refresh. The MACS bounds vary from moderately to significantly greater than the MA and MAC bounds.

#### Table 2: LFK Work Load\(^a\)

<table>
<thead>
<tr>
<th>LFK</th>
<th>( f_m' )</th>
<th>( f_a' )</th>
<th>( f_m'' )</th>
<th>( f_a'' )</th>
<th>( t_m' )</th>
<th>( t_m'' )</th>
<th>( t_a' )</th>
<th>( t_a'' )</th>
<th>( t_m'' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
<td>4.20</td>
<td>-</td>
<td>3</td>
<td>-</td>
<td>3.04</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>6</td>
<td>6.26</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>2.03</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>-</td>
<td>2.08</td>
<td>1</td>
<td>-</td>
<td>1.37</td>
<td>2</td>
<td>-</td>
<td>2.07</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>-</td>
<td>2.45</td>
<td>1</td>
<td>2</td>
<td>2.37</td>
<td>2</td>
<td>-</td>
<td>2.07</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>-</td>
<td>2.46</td>
<td>1</td>
<td>-</td>
<td>1.37</td>
<td>2</td>
<td>-</td>
<td>2.07</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>10</td>
<td>10.50</td>
<td>8</td>
<td>-</td>
<td>9.13</td>
<td>4</td>
<td>10</td>
<td>10.37</td>
</tr>
<tr>
<td>8</td>
<td>21</td>
<td>-</td>
<td>30.15</td>
<td>21</td>
<td>-</td>
<td>21.28</td>
<td>15</td>
<td>21</td>
<td>21.85</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>-</td>
<td>11.55</td>
<td>9</td>
<td>-</td>
<td>9.13</td>
<td>11</td>
<td>-</td>
<td>11.41</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>-</td>
<td>20.95</td>
<td>9</td>
<td>-</td>
<td>9.07</td>
<td>20</td>
<td>-</td>
<td>20.88</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>3</td>
<td>3.13</td>
<td>1</td>
<td>1</td>
<td>1.01</td>
<td>2</td>
<td>3</td>
<td>3.12</td>
</tr>
</tbody>
</table>

#### Table 3: Performance Bounds (boldfaced entries dominate the equations for \( t_{M A} \), \( t_{M A C} \), and \( t_{M A C S} \))

As shown in Table 3, \( (t_m'' - t_m') > 1 \) in LFK7, implying that \( f_m \) and \( f_a \) are not perfectly overlapped in the chimes. This would degrade performance, however, only after \( t_m' \) is reduced closer to \( t_m'' \). Also of interest in Table 3 is the fact that \( t_{M A C S} >> t_m'' \) in LFK8. This is caused by scalar loads splitting potential chimes (as discussed in Section 3.3), resulting in poor overlap of vector instructions.
4.2. Using the MA, MAC, and MACS Bounds as Indicators of Performance

Comparing the MA, MAC, and MACS bounds and the actual CPF performance $t_c$ in Table 4 and Figure 3 demonstrates first the effect of extra workload inserted by the compiler over the ideal, then the schedule-specific added run time effects as modeled by $t_{MACS}$ and finally the incremental effect of all unmodeled run time seen in $t_c$. The simple, highly idealized MA bound explains 80% or more of actual measured run time only in LFKs 3, 9, and 10. The MAC bound does likewise for all but LFKs 2, 4, 6, and 8. The MACS bound is at least 90% of $t_c$ for all but LFKs 2, 4, and 6. Explanations for the various gaps are discussed in Section 4.4.

The single process run times were measured by running

<table>
<thead>
<tr>
<th>LFK</th>
<th>$t_{MA}$ (CPF)</th>
<th>$t_{MAC}$ (CPF)</th>
<th>$t_{MACS}$ (CPF)</th>
<th>$t_c$ (CPF)</th>
<th>% of MA Bnd($t_{MA}/t_c$)</th>
<th>% of MAC Bnd($t_{MAC}/t_c$)</th>
<th>% of MACS Bnd($t_{MACS}/t_c$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.600</td>
<td>0.800</td>
<td>0.840</td>
<td>0.852</td>
<td>70.4%</td>
<td>93.9%</td>
<td>98.6%</td>
</tr>
<tr>
<td>2</td>
<td>1.250</td>
<td>1.500</td>
<td>1.566</td>
<td>3.773</td>
<td>33.1%</td>
<td>39.8%</td>
<td>41.5%</td>
</tr>
<tr>
<td>3</td>
<td>1.000</td>
<td>1.000</td>
<td>1.044</td>
<td>1.128</td>
<td>88.7%</td>
<td>88.7%</td>
<td>92.6%</td>
</tr>
<tr>
<td>4</td>
<td>1.000</td>
<td>1.000</td>
<td>1.226</td>
<td>1.863</td>
<td>53.7%</td>
<td>53.7%</td>
<td>65.8%</td>
</tr>
<tr>
<td>6</td>
<td>1.000</td>
<td>1.000</td>
<td>1.226</td>
<td>2.632</td>
<td>38.0%</td>
<td>38.0%</td>
<td>46.4%</td>
</tr>
<tr>
<td>7</td>
<td>0.500</td>
<td>0.625</td>
<td>0.656</td>
<td>0.681</td>
<td>73.4%</td>
<td>91.8%</td>
<td>96.4%</td>
</tr>
<tr>
<td>8</td>
<td>0.583</td>
<td>0.583</td>
<td>0.824</td>
<td>0.858</td>
<td>67.9%</td>
<td>67.9%</td>
<td>97.7%</td>
</tr>
<tr>
<td>9</td>
<td>0.647</td>
<td>0.647</td>
<td>0.679</td>
<td>0.749</td>
<td>86.4%</td>
<td>86.4%</td>
<td>90.7%</td>
</tr>
<tr>
<td>10</td>
<td>2.222</td>
<td>2.222</td>
<td>2.328</td>
<td>2.442</td>
<td>91.0%</td>
<td>91.0%</td>
<td>95.3%</td>
</tr>
<tr>
<td>12</td>
<td>2.000</td>
<td>3.000</td>
<td>3.132</td>
<td>3.182</td>
<td>62.9%</td>
<td>94.3%</td>
<td>98.4%</td>
</tr>
<tr>
<td>AVG</td>
<td>1.080</td>
<td>1.238</td>
<td>1.352</td>
<td>1.900</td>
<td>66.6%</td>
<td>74.6%</td>
<td>82.3%</td>
</tr>
</tbody>
</table>

MFLOPS 23.15 20.19 17.79 13.16

Table 4: Comparison of Bounds with Measured Performance
on a single processor while making the other three processors idle. Figure 3 also shows multiple process run times which were calculated by measuring the run time for the kernel of interest on one processor while an uncontrolled normal workload (from other users) was running on the other processors. The load average at this time was 5.1, which is greater than the number of processors. In all cases the effects of memory contention when multiple processes are running simultaneously leads to noticeable performance degradation.

A rough rule of thumb is that if all four processors are running different programs simultaneously, memory contention will typically cause a 20% performance degradation. If all four processors are executing different processes for the same executable, the programs will tend to fall into lockstep, typically causing only a 5% to 10% performance degradation. Although memory access instructions can theoretically achieve a peak performance of one access per 40 ns cycle, in practice typical memory contention reduces the performance to one access every 56 to 64 ns. Performance will not, however, degrade proportionately because some of the degradation in memory access performance is masked by other operations. More of this degradation will be exposed, however, as performance is improved toward the bound. Timing experiments designed to reduce contention do allow performance to approach the 40 ns peak. However this is not a realistic set of working conditions for running standard applications.

### 4.3. A/X Performance Measurements

Measurements of the A-process and the X-process alone for the ten LFK kernels are presented in Table 5. Except for LFKs 2, 4, and 6 the calculated $t_{MACS}$ and $t''$ closely model the measured results of $t_a$ and $t_x$, respectively. The $t''$ bound explains at least 95% of the measured $t_x$, except for LFKs 2, 4, and 6. The measured $t_x$ is larger than $t_a$ except for LFKs 4, 6, and 8. The gap between $t_a$ and $t_x$ is always large, although this is of little concern if $t_a \gg t_x$ as that indicates that the chief performance bottleneck is within the A-process. In LFKs 2, 4, 6, and 8, $t_p > \text{MAX} \{t_a, t_x \}$, and thus the existing code does not overlap the A-process and the X-process well. In this workload $t_x$ is usually larger than $t_a$, and when it is not both are significantly smaller than $t_x$. As a result, the Convex C-240 performance on these loops is dominated either by memory accesses or by bad coupling between the access and execute processes.

### 4.4. Evaluation of Performance on Benchmark Set

Examining the performance gaps in the hierarchy between pairs of performance bounds and measurements for the ten LFKs yields insights on where run time is spent and how it might be reduced.

- **LFK1,7,12**: The gap between the MA bound and the MAC bound is caused by the extra $t_a - t_x$ memory references inserted by the compiler. Vector elements reused in the next iteration are shifted by the loop index increment. This misalignment typically forces a vector reload (or theoretically a vector shift). A scalar processor could keep reused array elements in registers. In LFK7, $(t'' - t_a) > 1$ implies that the floating point vector additions and multiplications are not perfectly overlapped thus creating a ninth chime. This has little impact, however, since $t_a = 10$.

- **LFK2**: Large gaps in Table 5 between $t_a$ and $t''$, $t_x$, and $t''$ and $t_{MACS}$ indicate that unmodeled activity dominates the performance of this kernel. The conspicuous $t_{MACS}$ to $t_x$ gap raises an appropriate warning flag for this loop. Indeed, this loop has significant outer loop overhead, nonuniform memory access strides, and difficulty in vectorizing due to its multiple exits. Outer loop overhead and scalar code could be modeled as in [5].

- **LFK3,9,10**: The small gap in Table 4 between $t_{MACS}$ and $t_a$ indicates that 86% to 91% of the ideally deliverable performance has been achieved. $t_{MACS} - t_{MACS}$ explains 1/3 to 1/2 of the $t_x - t_{MACS}$ gap. The remainder is primarily due to access-execute overlap effects, as shown by $t_x - t_a$.

- **LFK4,6**: Since $t_x >> t_a$, we can conclude that the interaction of the vector reduction operator with vector memory accesses is a significant bottleneck. Since $t_x >> t''$, and $t_a >> t''$, we can conclude that

<table>
<thead>
<tr>
<th>LFK</th>
<th>$t_p$</th>
<th>$t_{MACS}$</th>
<th>$t_a$</th>
<th>$t''$</th>
<th>$t_x$</th>
<th>$t_{MACS}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.26</td>
<td>4.20</td>
<td>3.13</td>
<td>3.04</td>
<td>4.20</td>
<td>4.14</td>
</tr>
<tr>
<td>2</td>
<td>15.09</td>
<td>6.26</td>
<td>9.05</td>
<td>2.03</td>
<td>13.39</td>
<td>6.22</td>
</tr>
<tr>
<td>3</td>
<td>2.26</td>
<td>2.09</td>
<td>1.47</td>
<td>1.37</td>
<td>2.07</td>
<td>2.07</td>
</tr>
<tr>
<td>4</td>
<td>3.73</td>
<td>2.45</td>
<td>2.91</td>
<td>2.37</td>
<td>2.44</td>
<td>2.07</td>
</tr>
<tr>
<td>6</td>
<td>5.26</td>
<td>2.44</td>
<td>3.74</td>
<td>1.37</td>
<td>3.29</td>
<td>2.07</td>
</tr>
<tr>
<td>7</td>
<td>10.89</td>
<td>10.50</td>
<td>9.55</td>
<td>9.13</td>
<td>10.35</td>
<td>10.37</td>
</tr>
<tr>
<td>8</td>
<td>30.90</td>
<td>30.15</td>
<td>22.77</td>
<td>21.28</td>
<td>22.53</td>
<td>21.85</td>
</tr>
<tr>
<td>9</td>
<td>12.73</td>
<td>11.55</td>
<td>9.61</td>
<td>9.13</td>
<td>11.62</td>
<td>11.41</td>
</tr>
<tr>
<td>12</td>
<td>3.18</td>
<td>3.13</td>
<td>1.05</td>
<td>1.01</td>
<td>3.15</td>
<td>3.12</td>
</tr>
</tbody>
</table>

**Table 5**: MACS Bounds and Measurements (in CPL) (boldfaced entries indicate that $t_x$ is within 10% of $t_a$)
significant run time is spent on the substantial amount of scalar code which is not removed from either the X or A-process code and which is not modeled by either $t_f$ or $t_m$. In LFK4, the gap between $t_f$ and $t_f'$ is explained by the footnote to Table 2.

- LFK8: Since $t_p > t_x \approx t_a$, we can conclude that the A-process and the X-process are poorly overlapped. This is due to the large number of scalar loads splitting potential chimes. This effect is captured by $t_{MACS}$ but not by $t_f'$ or $t_m'$, since a scalar load will split a potential load-add-multiply chime ($t_{MACS}$) but not an add-multiply chime ($t_{m}'$) or a load chime ($t_p'$). As a result, $t_{MACS}$ explains all but 2.3% of $t_p$.

The ability to pinpoint the causes of performance degradation through the use of the hierarchy of performance bounds and measurements in these ten LFKs demonstrates the viability of the MACS hierarchical approach to performance analysis.

5. Conclusion

The MACS performance modeling and measurement approach provides a coherent framework for analyzing and improving performance. The hierarchy of performance models and measurements allow a user, compiler-writer, or computer architect to pinpoint areas where performance is lost and to identify what improvements might be most effective in the application, compiler, or machine.

A case study of the Convex C-240 using the ten vector loop kernels was used to illustrate the power of this approach. The relative magnitude of the hierarchy of MACS bounds, their component terms, and the A/X and total code performance measurements clearly, and fairly automatically, identified for each kernel the factors that contributed to its run time and prevented it from achieving its ideal performance.

We believe that this approach can be generalized and automated to assess a broad range of machines and scientific applications. Aspects of the MACS bounds hierarchy could be incorporated within a goal-directed optimizing compiler that would efficiently assess where and how best to spend its time.

Acknowledgments

We are grateful for many illuminating discussions of the C-240 generously provided by Patrick McGehearty, Greg Astfalk, Aaron Pottler, and Carl Jackson of the Convex Computer Corporation.

References


