RC 21035 (94182) 11/19/1997
Computer Sciences/Mathematics

IBM Research Report

Bounds–Based Loop Performance Characterization: Application to Post–Silicon Analysis and Tuning

Pradip Bose*, Sunil Kim, Francis P. O'Connell** and William A. Ciarfella****

*IBM T. J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

**High–End Processor Development, IBM Austin.

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filed only by reprints or legally obtained copies of the article (e.g. payment of royalties).

Research Division
Almaden ● Austin ● China ● Haifa ● Tokyo ● T.J.Watson ● Zurich

Bounds–Based Loop Performance Characterization: Application to Post–Silicon Analysis and Tuning

Pradip Bose*, Sunil Kim+, Francis P. O'Connell+ and William A. Ciarfella+

IBM Corporation

*T. J. Watson Research Center, Yorktown Heights, NY

+High–End Processor Development, Austin, TX

ABSTRACT

We consider the floating point microarchitecture support in high–end RISC superscalar processors. We propose a simple, yet effective bounds model to deduce the "best–case" loop performance limits for these processors. We compare these bounds to simulation–based (and where available, hardware–based) performance measurements for actual compiler–generated code sequences. From this study, we identify loop tuning opportunities to bridge the gap between "best–case" and "actual" performance in a post–silicon setting. Some of the results of such analysis point to fundamental *hardware performance bugs* which may be removed through relatively minor microarchitectural changes. More frequently, the analysis is useful for suggesting compiler enhancements. The analysis methods described have been used in actual high–end processor development projects within our company. We report our experimental results in the context of a set of application–based loop test cases, designed to stress various resource limits in the core (infinite cache) microarchitecture.

Corresponding author:
Pradip Bose
IBM T. J. Watson Research Center
P.O. Box 218 (Route 134)
Yorktown Heights, NY 10598.
Tel: 914–945–3478
Fax: 914–945–2141
Email: bose@watson.ibm.com

1. Introduction

The major focus of post–silicon tuning of processor performance is machine–specific *compiler optimization.* This is especially true for the processor *core logic:* the on–chip processor microarchitecture, which determines the *infinite cache* performance of the processor. This is because: (a) perturbing the core logic usually implies a re–verification effort, which is costly; (b) increasingly, for high–performance (esp. high MHz) designs, the *hardware* enhancement opportunities are largest in the *cache/memory* subsystem design. Thus, in considering performance tuning options after "first silicon", the attention is primarily limited to: (i) compiler enhancements, (ii) technology–specific circuit tuning and (iii) enhancements to the memory hierarchy organization. Changes in the last category are often limited to high–level features, like cache geometry parameters (e.g. size, associativity and linesize). Although compiler optimizations geared to exploit memory hierarchy features are also addressed during the post–silicon phase, the machine–specific *core* optimizations tend to get emphasized and seem to require more resources. One of the reasons for this may be the need to publish performance figures for processor benchmarks (which are often cache–contained); another reason may be that detailed understanding of a newer microarchitecture is relatively more difficult to acquire and use in refining the existing optimization algorithms.

In this paper, we consider the problem of loop transformation and instruction scheduling for performance tuning of high–end superscalar, RISC machines. In particular, we focus on the core floating point microarchitectures of recent, high–end processors (e.g. [4–6]) used in the RS/6000™ family of technical workstations and servers. The original POWER1 processor [1] has a single floating point unit, supported by a single integer–cum–load/store unit (along with other functional units). The prior high–end PowerPC™ processors (e.g. [2, 3]) have a single load–store unit (LSU) and a single floating point unit (FPU), with several separate integer units. The microarchitecture of the POWER2™ [4] and its follow–on single chip version (P2SC) [5] has enhanced support for technical computing: it has an additional floating point unit, supported by an additional LSU. The POWER3™, a more recently completed processor[1] [6], also has the dual–LSU/FPU feature, in addition to other enhancements. In this paper we consider two basic classes of floating point microarchitectures: (1 LSU, 1 FPU)–super scalar machines [1–3] and (2 LSU, 2 FPU)–machines [4–6]. Note that for the purposes of this paper, we do not need to distinguish a decoupled LSU from one which combines the function of an LSU with that of an integer arithmetic unit.

---

[1]This processor product will be announced at *Microprocessor Forum*, in October, 1997. It is targeted for high–end technical *and* commercial workstations and servers. Unlike the POWER2 family, the POWER3 is PowerPC™ [13] compatible. It has additional strengths incorporated via features like: data–side prefetch, superior integer and branch handling, and support for symmetric multiprocessing.

A key problem in determining the tuning opportunity for a given loop or other benchmark is to understand the fundamental limits of achievable performance. For a given instruction set architecture and machine organization, it is important to be able to compute a set of achievable bounds for the loop kernels of interest. These bounds may range from the "best–case" (idealized) to those which are more realistically achievable, in the context of given machine parameters. We propose a simple model for computing such bounds for loops with a defined structure. This model is based on simple bandwidth arguments and is valid for fully pipelined, super scalar execution models. Initially, we use this model to compute the "best–case" bounds for representative (1 LSU, 1 FPU)– and (2 LSU, 2 FPU)–processors. Later, we derive a corresponding set of "realistic" bounds, by factoring in resource and data–dependencies. We compare these bounds to actual (measured) performance to assess tuning opportunities.

The observed performance gaps lead us to investigate the causes and suggest enhancements. The bounding techniques used are, in principle, robust enough to handle arbitrarily complex loops which have sequential (i.e. branch–free) bodies. However, with complex test loops exhibiting intra– and inter–iteration dependencies, analytical formulation can sometimes become unwieldy and prone to errors. In such cases, it *may* become necessary to use a detailed, cycle–accurate simulator (or "timer") [7], to validate the analytical expectations. In any case, assume that the micro–architecture is modelled to the accuracy needed and the "expected" or "realistic" loop performance bounds are known and understood precisely, Once this is achieved, the measured performance may match the expectation fairly well. But even so, the original idealized bounds (if significantly different from the measured values) are extremely useful in guiding us to post–silicon tuning opportunities. Most of the *software* opportunities are related to *loop unrolling*, with attendant *scheduling* intelligence. Where applicable, *software pipelining* schedules may be improved to reduce the performance gap. We show how the bounds model allows us to deduce the optimal unrolling depths and the need for scheduling improvements in a straightforward manner. In fact, the methods used can in principle be incorporated within the compiler to produce code which results in improved loop performance.

2. A Parameterized Floating Point Microarchitecture Model

Figure 1 shows the assumed high–level machine organization of a generic processor, for which (parameterized) performance models are considered. This machine model can process simple floating point loops only. That is, an input program is restricted to be a single loop which has a sequential loop body consisting of floating point loads, stores and arithmetic operations only. The loop body is terminated by a single conditional branch instruction, which causes the control to branch back to the beginning of the loop if there are more iterations to execute. The conditional branch instruction used in such loops is assumed to be one whose outcome can be resolved exactly by simply decrementing

and testing a COUNT register [13]. This register is assumed to be pre–loaded by the iteration count. The branch target address is assumed to be specified as an immediate offset (relative to the program counter) in the branch instruction itself. Thus, in this machine, the branch unit is very simple, with facilities for branch target computation and branch resolution only. No history–based prediction mechanisms (for branch direction) are required. Branch instruction execution is single–cycle, and perfectly overlapped with other computation. However, depending on the lookahead mechansim and whether or not fetch–prediction logic is present (e.g. in the form of a branch target address cache or BTAC [2,3]), up to a single–cycle pipeline stall may be visible during the instruction fetch or dispatch process at the beginning of each loop iteration. Also, if the loop body straddles a cache line boundary, some processors (e.g. [2,3,6]) may exhibit an additional stall condition in the fetch process, depending on the size of the loop body. These perturbations to the fetch semantics will not be explicitly addressed in this paper. As explained later, we handle the branch processing variations by a single parameter, which determines the *effective* dispatch stall under steady–state loop processing.

Every cycle, a number of instructions (determined by the fetch bandwidth parameter) may be fetched from the instruction cache into the instruction buffer. The fetch address is provided by the "fetch unit" and is either the next sequential address (determined by the last fetch address and the number of instructions fetched during the last cycle) or the target address of the taken, loop–ending branch. The maximum number of instructions which can be decoded, renamed and dispatched per cycle is determined by the dispatch bandwidth, which is another parameter. For each of the instruction classes modelled (i.e., branch, load–store and floating point arithmetic) there is a reservation station, which for the purposes of this paper is a (bypassable) FIFO queue. Thus, out–of–order issue of instructions from these queues (BRQ, LSQ and FPQ) into the corresponding execution units is not allowed. The number of instructions which may be issued (per cycle) from a given queue into the corresponding execution units is equal to the number of distinct execution pipes within each class. Each pipe within the branch unit (BRU) and load–store–unit (LSU) is a single stage (i.e. 1–cycle execution). Each pipe within the floating point unit (FPU) is multi–stage, the latency being specified by a parameter. Bypassing effects within each FPU pipe is modelled by parameters which specify the "dependence bubbles" (or stall cycles) for back–to–back dependent operations. Register renaming is present for all modelled instructions. The number of floating point rename buffers is a model parameter. The number of fixed point rename buffers is effectively assumed to be infinite, without any effect on our results, because we are limited in scope to simple floating point loops only.

Instruction dispatch is controlled by rules which may prevent the attainment of the maximum dispatch bandwidth (or rate) on a given cycle. Most of these constraints have to do with finite sizes of resources like the completion (reorder) buffer, reservation stations, rename buffers, etc. Thus, under infinite queue/buffer assumptions, for branch–free sequential code, the peak dispatch rate is always attained. *Dispatch* is obviously dependent on *fetch*, so the effect of fetch stalls (due to loop–ending

branches) can manifest itself as a dispatch stall. The *issue* process (from the reservation stations to the units) involves dependence analysis to determine the number of instructions (of a given class) which may be issued for execution on a given cycle. Effectively, in loops which exhibit loop–carried dependencies, the analysis to determine the steady–state issue bound may require consideration of multiple iterations. Under idealized conditions of inter–instruction independence, a particular issue rate is bound by the number of execution pipes of the class under consideration.

Instruction completion is governed by rules which enforce the in–order retirement of instructions: a feature implemented in most modern processors (e.g. [2, 3, 6]) using a reorder buffer [9] mechanism. This enables the support for precise interrupts in such processors. During dispatch, each instruction is tagged by an instruction identifier (iid), logically associated by the particular slot in the reorder buffer which holds its "in–flight" attributes. One of these attributes is a "finish" bit, which is set to "true" when that instruction "finishes" execution. For loads and arithmetic operations, the "finish" cycle is the one in which the result is written into a rename buffer. For stores, "finish" may be defined as the cycle in which the data to be stored is written into the pending store queue and is successfully paired up with the corresponding store address. Completion is synonymous with the actual retirement of the iid from the processor state. It is often implemented to occur simultaneously with the transfer of data from the target rename buffer to the real architected register. (In the case of stores, this would correspond to the actual writing of data from the pending store queue to the cache arrays). However, this is not absolutely necessary: in some processors, (e.g. [3,6]), due to cycle–time or complexity criteria, completion (retirement) and architectural writeback may be separated by a cycle or more. Every cycle, the completion logic examines the next group of eligible iid's in the completion (reorder) buffer, as given by the completion bandwidth parameter. Of these, the consecutive (i.e. in program order) iid's which have their "finish" bits turned on are actually retired. For our purposes, we need to consider only the completion bandwidth: the maximum number of iid's which can be retired per cycle. This parameter is usually at least as large as the dispatch bandwidth to ensure that the size of the completion (reorder) buffer does not become the primary performance bottleneck.

For the class of real machines referred to in Section 1, the number of LSU's is equal to the number of FPU's as well as to the number of data cache ports, to provide a "balanced, bandwidth–matched" design. However, in our simulation model, arbitrary combinations of these (and other) parameters may be applied. In (1 LSU, 1 FPU)–mode or in (2 LSU, 2 FPU)–mode with proper parameter settings, the model can be used to (accurately) represent the *floating point microarchitecture* of a given high–end super scalar machine among the ones referred to in Section 1.

Let us refer to this generic machine model (Figure 1) as the **LS–FP processor core.** The corresponding architectural simulation model is coded to handle only the following instructions: lfd, lfdu, stfd, stfdu, lfq, lfqu, stfq, stfqu, fadd, fsub, fmul, fdiv, fmadd, fnmadd and bc (conditional branch). The assembly language syntax and register transfer level semantics of the floating point instructions

are given below:

lfd, lfdu: load floating point double without and with update:

Syntax:

lfd(u)   FRT,D(RA)

Notational Semantics:

if RA = 0 then b <— 0

else   b <— (RA)

EA <— b + EXTS(D)

FRT <— MEM(EA, 8)

For lfdu only: RA <— EA

Explanation in English:

Let the effective address (EA) be the sum of the contents of fixed point register RA and the sign–extended displacement D. The doubleword (8 bytes) in storage addressed by EA is placed into the double–precision floating point register FRT. For lfdu, EA is placed into register RA (which must be non–zero) as the final micro–operation.

stfd and stfdu: store floating point register without and with update:

Syntax:

stfd(u)   FRS, D(RA)

Notational Semantics:

if RA = 0 then b <— 0

else   b <— (RA)

EA <— b + EXTS(D)

MEM(EA, 8) <— (FRS)

For stfdu only: RA <— EA

Explanation in English:

The EA is computed as before. The contents of the double–precision floating point register FRS are stored into the doubleword in memory addressed by EA. For stfdu, the EA is placed into register RA, which must be non–zero.

The *lfq(u)* and *stfq(u)* instructions have similar syntax and sematics except that *two* double precision floating point registers (16 bytes) are loaded from or stored to memory. Thus, two floating point registers are specified in these instructions. (These load and store *quadword* instructions are available only on the POWER2 and P2SC [4,5] processors).

fadd, fsub, fmul, fdiv: floating point arithmetic operations:

Syntax:

fop   FRT, FRA, FRB (where fop can be fadd, fsub, fmul or fdiv).

Notational Semantics:

FRT <— FRA (+, –, * or /) FRB

Explanation in English:

The floating point operand in register FRA is operated upon (i.e. add/subtract/multiply/divide) by the floating point operand in register FRB; the result is placed in register FRT.

fmadd and fnmadd: floating point multiply–add and negative multiply–add operations.

Syntax:

fmadd FRT, FRA, FRC, FRB

Notational Semantics:

FRT <— [(FRA)x(FRC)] + (FRB)

Explanation in English:

The floating point operand in register FRA is multiplied by the floating point operand in register FRC. The floating point point operand in register FRB is added to this intermediate result, which is placed in register FRT.

For the *fnmadd* instruction, the *negated* result of the above computation is placed in the target register FRT.

3. Loop Performance Bounds Model

Mangione–Smith et al. [11] have proposed a procedure for establishing loop performance bounds for super scalar machines like the orginal POWER1 [1]. This is based on the analysis of *source–code* level description of application loop kernels. In this section, we propose a generalized bounds model, based on the *compiled* code for such kernels. This has been implemented as part of an early stage loop performance specification and bounding tool, called ***eliot*** (see Figure 2).

The initial (idealized) machine assumptions and parameterization underlying the bounds model are stated below. A performance bound obtained under these assumptions may be referred to variously as the *"best–case"*, *"idealized"* or *"infinite–queue"* bound (or in short, as **I–Bound**) in this paper. (As we shall see later in Section 4.0, a more realistic **R–Bound** can be computed, once the main hardware performance inhibitor(s) are identified).

Model assumptions for I–Bound *computation:*

1. All buffer or queue resources are effectively infinite in size. (This includes the caches as well). Examples are: size of the reorder buffer, number of rename buffers, size of the pending store queue, size of the instruction buffer, sizes of reservation stations, etc.

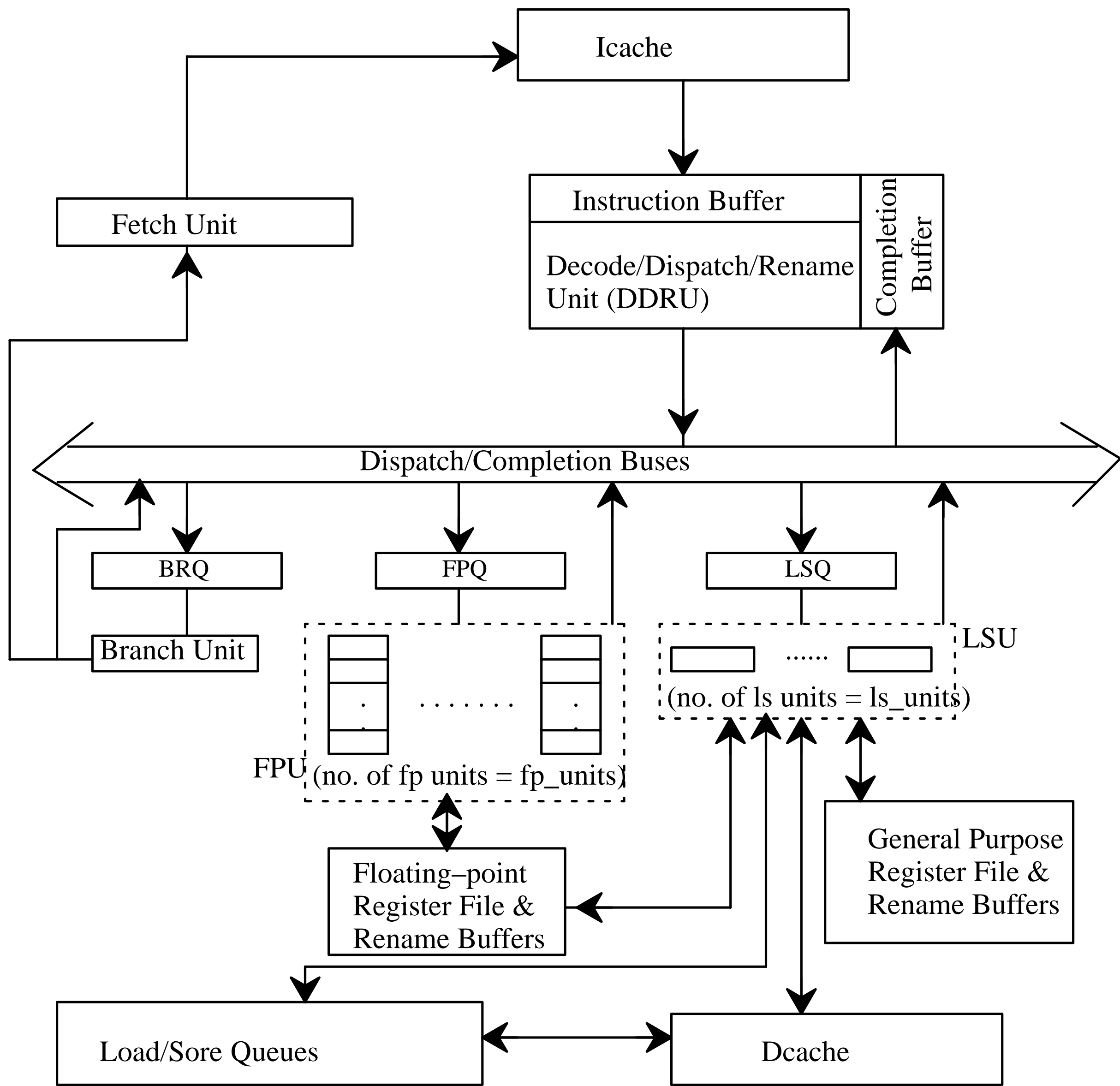

Figure 1. Example LS–FP super scalar machine organization

2. The general processing paradigm assumed is: *in–order fetch* (under control of the next fetch address supplied by the Fetch Unit) from the perfect (infinite) icache to the instruction buffer; *in–order dispatch* from the instruction buffer to the unit reservation stations; *in–order issue* within each instruction class; *out–of–order execution* between different instruction classes; and *in–order completion,* implemented using a standard reorder buffer mechanism. Also, full register renaming support is assumed, with effectively infinite number of rename buffers. (For some processors, e.g. [1], target renaming is available for load instructions only.

3. The numbers of LSUs and FPUs are given by the parameters *ls_units* and *fp_units* respectively. Normally, ls_units = fp_units. There is a single, simple branch unit, which is able to predict the loop–ending branches perfectly (in the steady–state sense). The number of penalty (stall) cycles, p (≥ 0) in processing such a branch is a simple parameter

in the model. Thus, if p=0, the branch is perfectly overlapped and the correct (taken) branch path can be fetched *and* dispatched without any pipeline stall; p=1 implies a dispatch pipeline stall of 1 cycle, and so on. The value of p used for a given hardware model, *may* depend on the number of instructions per iteration of the input loop, and other factors, such as cache line crossings. In this paper, for simplicity, we shall assume a constant value of p = 1 (in close conformity with current generation processors [3,5,6]). Even though the POWER2 or P2SC processor does not have a BTAC, it has a superior branch lookahead and "folding" hardware, which enables it to enjoy the same value of p (= 1). Since p is constant, it will not appear explicitly in the bounds formulation equations.

4. The maximum number of instructions which can be fetched per cycle (from the instruction cache to the instruction buffer) is governed by the parameter, *fetch_bw.*
5. The numbers of load and store ports to the data cache are parameters (*l_ports* and *s_ports*). The cache is normally assumed to be perfect, with no interleaf conflicts.
6. The instruction dispatch bandwidth is a parameter, *disp_bw.* It specifies the maximum number of instructions which can be dispatched, per cycle, from the instruction buffer to the unit reservation stations. The instruction completion bandwidth is a parameter, *compl_bw.* It specifies the maximum number of instructions which can be completed (from the reorder buffer) per cycle.
7. The LSU and FPU unit issue bandwidths are given by parameters *lsu_issue_bw* and *fpu_issue_bw* respectively. They specify the maximum number of instructions (of the load–store or floating op kind, respectively) per cycle that can be issued from the corresponding reservation station to the underlying functional unit pipe(s). Normally, (i.e. for one of the real machines referred to in Section 1) lsu_issue_bw = fpu_issue_bw = ls_units = fp_units.
8. Instructions waiting to be issued into execution from a given reservation station (e.g. LSQ or FPQ) are data–independent; i.e., an instruction can be issued as soon as an execution pipe is available.
9. The number of read and write ports of the (rename and architectural) register arrays are more than the peak requirements; i.e. they are effectively infinite.
10. Once an instruction is issued to an execution pipe, its latency is fixed, determined by the number of stages of the pipeline. All operations are assumed to be fully pipelined.
11. There are no result bus contentions between multiple pipes. There is no hard limit on the number of instructions which may "finish" per cycle, except as constrained by the total number of execution pipes.

The basic performance metric which we shall compute using the bounds model, is steady–state ***cycles–per–iteration,*** denoted by cpI. From this number, the steady–state cycles–per–instruction (cpi) performance is simply computed using the equation:

cpi = cpI/N .... (3.1)

where, N is the number of instructions per loop iteration. Recall that the instructions within each iteration consist of floating point load/store instructions, floating point arithmetic operations and a single, loop–ending conditional branch instruction (bc). Since floating point divide instructions are typically implemented as long–latency, non–pipelined operations, in order to abide by item number 10 in the list of assumptions above, *let us limit our discussion to loops which do not have floating point divide instructions.*

Let us assume that, on a per iteration basis,

(a) the number of load instructions = N_L

(b) the number of store instructions = N_S

(c) the number of floating point arithmetic instructions (excluding compound operations like fmadd/fnmadd and non–pipelined operations like fdiv): add (fadd), subtract (fsub), and multiply (fmul) instructions = N_F

(d) the number of floating point multiply–add instructions (fmadd or fnmadd) = N_{MA}

Thus, clearly, $N = N_L + N_S + N_F + N_{MA} + 1$, where the final '1' counts the loop–ending branch instruction. Note also that an fmadd *instruction* counts as *two* floating point *operations,* so that the number of floating point operations (flops) F, per loop iteration is given by: $F = N_F + 2*N_{MA}$.

Following well–established understanding of steady–state pipeline flow, the steady–state cpI performance can be easily seen to be given by:

$$cpI = \max(cpI_{fetch-bound}, cpI_{agen-bound}, cpI_{load-port-bound}, cpI_{store-port-bound}, cpI_{dispatch-bound}, cpI_{lsu_issue_bound}, cpI_{fpu-issue-bound}, cpI_{compl-bound}) \quad .... \quad (3.2)$$

where, each of the individual hardware resource bounds are derived as follows:

$$cpI_{fetch-bound} = \lceil N/fetch_bw \rceil \quad .... \quad (3.3)$$

$$cpI_{load-port-bound} = \lceil N_L/l_ports \rceil \quad .... \quad (3.4)$$

$$cpI_{store-port-bound} = \lceil N_S/s_ports \rceil \quad .... \quad (3.5)$$

$$cpI_{dispatch-bound} = \lceil N/disp_bw \rceil \quad .... \quad (3.6)$$

$$cpI_{compl-bound} = \lceil N/compl_bw \rceil \quad .... \quad (3.7)$$

$$cpI_{agen-bound} = \lceil (N_L + N_S)/ls_units \rceil \quad .... \quad (3.8)$$

$$cpI_{lsu-issue-bound} = \lceil (N_L + N_S)/lsu_issue_bw \rceil \quad .. \quad (3.9)$$

$$cpI_{fpu-issue-bound} = \lceil N_F / fpu_issue_bw \rceil \quad .... \quad (3.10)$$

The corresponding bound for *cycles–per–floating–point–operation* (cpf) are obtained by dividing the cpI bounds by the number of flops per iteration, F.

The above equations reflect the fundamental limits of loop performance, under the simplifying assumptions stated. For example, the address generation related bound, $cpI_{agen-bound}$ is determined by the maximum number of address generations (agens) possible, per loop iteration. The maximum number of agens per cycle must be equal to the number of load–store units (LSUs), as given by the parameter: ls_units. This is because each LSU is equipped with an address generation adder, and up to *ls_units* load or store instructions can be issued from the LSQ (reservation station) per cycle. Since the number of load/store instructions per iteration is $(N_L + N_S)$, equation (3.8) becomes clear. The ceiling function is required to take care of the cases where $(N_L + N_S)$ is odd: for example, 3 or 4 load/store instructions would both take 2 cycles for agen, if there are 2 LSUs.

Similarly, the other bounds equations can be seen to hold true. The overall equation (3.2) is merely a statement of the fact that steady–state loop performance is determined by the narrowest

bottleneck, i.e. by the hardware constraint which results in the largest, individual cpI bound. Note that under the stated assumptions, individual unit pipeline latencies do not need to be considered in these (idealized) I–Bound formulations. Thus, the latency of the FPU execution pipes, or the data cache access latency do not figure in these equations. The key underlying assumptions are: (a) all cache access and instruction processing paths are fully pipelined at the machine cycle granularity; (b) all queue, buffer and cache *size* resources are effectively infinite; and (c) there are no direct data dependencies between instructions eligible for issue in either reservation station (LSQ and FPQ). We shall see examples later (see sub–section 4.3) where pipe latency parameters are used in computing a realistic R–Bound, in which finite resources and data dependencies are considered.

It should be noted that in practice, it may be useful to assign more than one value to a given bandwidth parameter, depending on the particular context. For example, the dispatch bandwidth, *disp_bw* may depend on the instruction mix. For the original POWER1 processor [1], a maximum of 4 instructions can be dispatched per cycle: a branch, a logic–on–condition–register (LCR) instruction, a fixed point instruction (which may be an integer operation or a load/store instruction) and a floating point instruction. However, for our specific context of simple floating point loops (as defined), we can think of the POWER1 as a 3–dispatch machine (since LCR instructions are absent). Furthermore, when considering the execution of the branch–free *loop body,* the POWER1 is a 2–dispatch processor since branches are not encountered. Thus, disp_bw can have at least two distinct values (2 and 3) during the course of a simple loop execution in POWER1.

As indicated earlier on, the I–Bounds formulation can be incrementally augmented to factor in additional effects. These may relate to the input loop characteristics; or, they may concern realistic limits on resources and bus arbitration capabilities. For example, the assumption listed in item 8 above may prove to be too restrictive for a given loop. If there are intra–loop or loop–carried dependencies, the $cpI_{fpu-issue-bound}$ may need to be computed on a loop–by–loop basis; a general formulation becomes cumbersome, with the need to introduce parameters to describe the dependence relations (graph). Special cases may be more easily dealt with. For example, let the N_F floating ops (say fadds) have a pair–wise, successive dependence chain, with no loop–carried dependence. If the N_F fadds are part of the same issue group, then, a simple augmentation is possible, as follows:

$$cpI_{fpu-issue-bound} = 1 + (N_F - 1) * fpu_dep_delay \quad \text{.................................(3.10b)}$$

where fpu_dep_delay is the pipeline stall between successive initiations of a pair of dependent floating point operations into the FPU execution pipes.

Thus, for example, consider a sequence of 4 fadd instructions that could be dispatched in a single cycle of a 4–issue superscalar machine. If they are "back–to–back", pairwise data dependent and fpu_dep_delay is 1 cycle, then it takes ((1+4–1) = 4) cycles to issue the 4 fadds. If fpu_dep_delay is 2

cycles, the issue sequence would be the same string of fadds, but with an intervening null–issue cycle between every pair; hence the number of issue cycles would be ((1 + (4–1)*2) = 7) cycles.

As we shall see, the I–Bound formulation, given by equations (3.2) through (3.10), is very useful in diagnosing the primary causes of performance shortfall. For example, let us assume that the measured performance is much worse compared to the idealized bounds. On examining the code sequence, if it is found that there are a lot of dependent floating point operations, we can suspect that the $cpI_{fpu-issue-bound}$ formulation is optimistic. Consequently, this may point to an opportunity for improvement of the register allocation and/or scheduling.

An example loop and its performance potential on a (1 LSU, 1 FPU)–model:

Let us consider the well known *daxpy* application kernel, which is the key loop within the floating point benchmark called Linpack. The FORTRAN specification of daxpy is:

```
do i = 1, n
   x(i) = x(i)  +  s* y(i)
enddo
```

where, the 1–dimensional arrays x, y and the scalar s are declared to be double precision floating point variables. The corresponding compiled code, per inner loop iteration, is as follows:

```
A:   lfd      1,  0x8(6)   /* load flt reg 1; 0x8: hex displacement; base address register: #6 */
B:   fmadd  1,   0,  2,  1   /* flt mpy–add */
C:   lfdu     2,  0x8(5)     /* load flt reg 2, with update */
D:   stfdu   1,   0x8(6)    /* store flt reg 1, with update */
E:    bc   <addr specifier>
```

(Note that there is an initial load to floating point register 2, before the main loop is entered. This is not shown, but is assumed implicitly. The instructions above are labelled with alphabets A, B, C, D and E for ease of referral in the discussion below. The compiler is used in PowerPC architecture mode. The exact sequence of instructions may vary with the compiler version).

Let's try to apply our bounds model to this loop for a (1 LSU, 1 FPU) machine (such as the original POWER1 [1]). We consider the loop processing under *steady–state* operation of the super–scalar pipeline, i.e. after many iterations have already been completed. For this processor, the parameters of interest are:

fetch_bw = 4;

disp_bw = 3, 2, 1 or 0 depending on context as follows:

> On a given dispatch cycle (machine cycle n), let us refer to the next 3 instructions available for dispatch in the instruction buffer as the *dispatch–group*. If the dispatch–group consists of a load or store, a floating point arithmetic operation and a terminating branch (e.g. the group C–D–E above) then disp_bw = 3. However, in this case, since the branch was resolved to be taken, instructions from the taken path will be available for dispatch in the instruction buffer only in machine cycle n+2. Any conditionally dispatched instructions in cycle n+1 (from the not–taken or sequential path) will be cancelled. So, effectively, disp_bw = 0 in cycle n+1. If the dispatch–group consists of non–branch instructions only, then disp_bw = 2. On occasion, the number of instructions eligible for dispatch in the instruction buffer may be a number d, with $1 \leq d < 3$. For example, assume that D–E or E alone is the only correct instruction(s) available for dispatch on a given cycle. The sequential instructions beyond the branch, although available for conditional dispatch, are later cancelled after the branch is resolved taken. Hence, in such situations, we assume the effective disp_bw to be d (< 3) = number of eligible instructions in the dispatch buffer. As in the disp_bw = 3 case, however, the following cycle will necessarily be a 0–dispatch cycle.

l_ports = s_ports = ls_units = lsu_issue_bw = 1;

fp_units = fp_issue_bw = 1

Thus, for the POWER1 processor, and the daxpy loop, we get the following bounds:

$cpI_{fetch-bound} = \lceil 5/4 \rceil = 2;$

$cpI_{dispatch-bound} = \lceil 2/2 \rceil + \lceil 3/2 \rceil = 3;$

$cpI_{agen-bound} = cpI_{lsu-issue-bound} = \lceil 3/1 \rceil = 3;$

$cpI_{load-port-bound} = \lceil 2/1 \rceil = 2;$

$cpI_{store-port-bound} = \lceil 1/1 \rceil = 1;$

$cpI_{fpu_issue_bound} = \lceil 1/1 \rceil = 1;$

The $cpI_{compl-bound}$ is not applicable for this processor, since it does *not* have a completion mechanism (as described for our general model, Figure 1) using a reorder buffer. (In–order completion for precise interrupt support is implemented using a checkpointing scheme).

Clearly, this loop is load–store bound, in that equation (3.8) will always be the gating factor. Thus, the overall bound in this case is 3 cycles per iteration, which implies a cpi (cycles–per–instruction) of 3/5 = 0.6. ***This agrees exactly with actual measurement on an existing POWER1 workstation.*** In other words, the hardware–compiler pair works effectively in attaining the peak infinite cache daxpy performance on the POWER1 for this loop; there is no room for further post–hardware tuning in this particular case.

On the other hand, due to the presence of dispatch restriction rules and other constraints, some prior processors do not attain their idealized daxpy bounds (see [15]). The description in [6] shows how the POWER3 design attains its daxpy performance bound, by careful consideration of the latency, bandwidth and resource parameters. In a post–silicon framework, when the hardware cannot be

altered, compiler–based loop tuning methods (like unrolling with scheduling) can be employed to achieve or approach the best–case cpI values for certain loops. Such application of bounds–based characterization is precisely what we shall attempt to illustrate in the next section for two of the *most recent* processor products developed at IBM for the RS/6000 server family.

4. Loop Bounds, Measurement and Tuning Opportunities

In this section, we first present a set of basic loops, taken from our full test case repository. We then show the measured performance numbers and compare them with the “best–case” and “realistic” bounds. This procedure sets the stage for identifying the “root causes” behind the performance gaps. Where post–silicon hardware fixes are deemed to be infeasible, we experiment with software loop tuning. These include enhancements to loop unrolling, software pipelining, register allocation and instruction scheduling.

Experimental Set–Up:

Figure 2 shows the overall tools–based methodology used in our post–silicon performance validation and tuning experiments. A given source test case is compiled into an executable (xcoff) file. This can be traced and run on a cycle–accurate processor simulator (or “timer”) [7] such as the H–timer tool used in the initial phase of the POWER3 development project. It can also be converted into a format suitable for running on the full RTL simulator (DSL/Texsim) which essentially captures the full function and timing behavior of the processor. The compiled test cases may also be run on existing hardware, with similar microarchitectures in order to do a comparative analysis. The methods used for hardware measurement of loop performance are as described in [8]. The cycle–by–cycle pipeline flow behaviors, visualized using the timer and the Texsim models, can be compared to cross–validate the detailed timings predicted by the two models. A special–purpose, loop timer called eliot is used to compute loop performance bounds (I–Bound and R–Bound). This is developed during the early–stage design definition stage [6] in order to set performance targets for key test cases. It is also used in post–silicon validation and tuning, as discussed later in this section. Eliot–based bounds computation can also be used to predict optimal loop unrolling depths, as discussed in sub–section 4.4.

4.1 Loop Test Cases

Table 1 lists a set of elementary Fortran DO LOOP test cases. These were compiled and (a) run on the actual hardware or the Texsim model; and (b) traced and run on the eliot tool for projection of “best–case” and “realistic” bounds. Each loop was unrolled to a depth indicated by the value of u in column 1. These loops, selected from a wide range of real applications, form the very basic set of

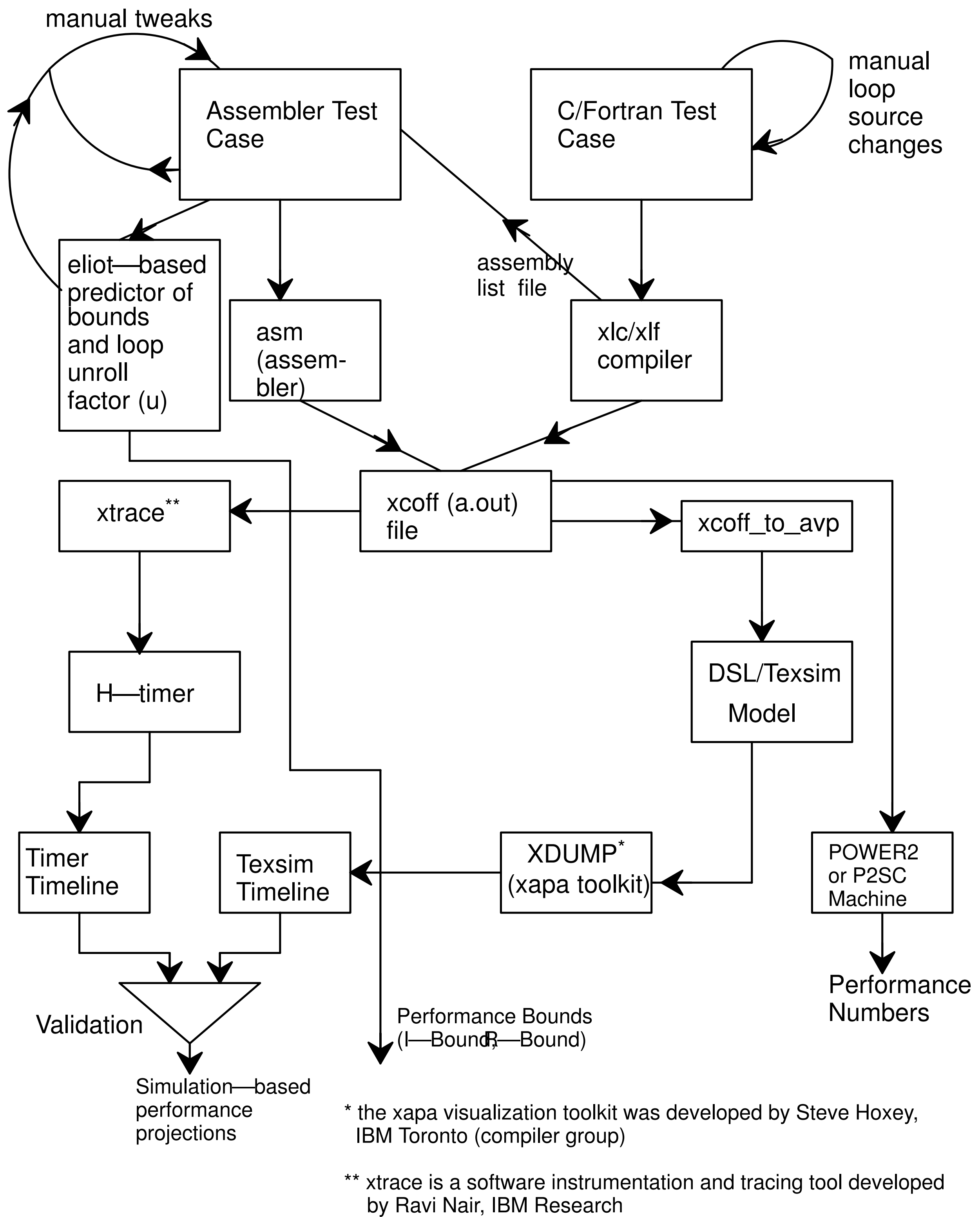

Figure2. Post—RTL Performance Validation and Tuning Methodology

application–based test cases used for assessment of infinite cache (floating point) performance tuning opportunities. (The actual list of test kernels used in practice is much larger, of course).

Table 1. Description of Selected Fortran Loop Test Cases

TRACE (manual source unroll, u=2, is used prior to auto—unroll)	Instrs/original iteration	Loop body (original, u=1, PowerPC, assembly), compiler xlf ver. 3.2.5.x (Pre—630)	Instrs/unrolled iteration (—O3 optimization to invoke auto—unroll)	Loop body (original, source)
loop01.tr (u=16)	2	{stfdu, bc}	17	x(i) = s
loop02.tr (u=8)	3	{lfdu, stfdu, bc}	17	x(i) = y(i)
loop03.tr (u=4)	5	{fadd, lfdu, lfdu, stfdu, bc}	17	x(i) = a(i) + b(i)
loop04.tr (u=4)	5	{fadd, lfdu, lfd, stfdu, bc}	17	x(i)=x(i) + b(i)
loop05.tr (u=4)	4	{fadd, lfdu, stfdu, bc}	13	x(i)=u+a(i)
loop06.tr (u=4)	6	{fadd, lfdu, fadd, lfdu, stfdu, bc}	21	x(i)=u+a(i)+b(i)
loop07.tr (u=2)	8	{lfdu, fadd, lfdu, fadd, stfdu, lfdu, fadd, bc}	15	x(i)=u+a(i)+b(i) + c(i)
loop08.tr (u=2)	9	{lfdu, fadd, lfdu, lfdu, fadd, lfdu, fadd, stfdu, bc}	17	x(i)=a(i)+b(i)+c(i)+ d(i)
loop09.tr (u=4)	6	{lfdu, lfdu, lfdu, fma, stfdu, bc}	21	x(i) = c(i)+b(i)*a(i)
loop10.tr (u=4)	6	{fmadd, lfd, lfdu, lfdu, stfdu, bc}	21	x(i)=x(i)+b(i)*a(i)
loop25.tr (u=4)	9	{fmadd, fmadd, lfdu, fmadd, lfdu, fmadd, lfdu, lfdu, bc}	33	s1 = s1 + b(i)*a(i) s2 = s2 + b(i)*c(i) s3 = s3 + d(i)*a(i) s4 = s4 + d(i)*c(i)
daxpy.tr (u=4)	5	{fmadd, lfdu, lfd, stfdu, bc}	17	x(i)=x(i)+s*y(i)
ddot.tr (u=4)	4	{fmadd, lfdu, lfdu, bc}	13	s=s+x(i)*y(i)

4.2 POWER3 Loop Performance

In this section, we compare the actual loop performance of the POWER3[1] processor and compare it with the (idealized) I–Bound and (realistic) R–Bound predictions as well as pre–silicon full model timer runs. As indicated in Section 1, this processor is a (2 LSU, 2 FPU) processor, with many new features to provide enhanced performance [6]. The data cache has two load ports and a

single store port. However, the cache array itself is 4–way interleaved, and bank conflicts prevent the effect of a true dual–port cache.

Table 2 shows a comparison of the "best–case" performance specification bounds predicted by eliot and the real performance of the ("taped–out") POWER3 processor[1]. *All results assume or simulate a perfect (infinite) cache model.* (Recall: Texsim is an RTL–level simulator of the integrated chip logic model, coded using an IBM internal hardware description language called DSL. Texsim simulation speed is about 50–100 target machine cycles per second, which is too slow for performance studies. Trace–driven timer model speeds are at least two orders of magnitude faster: hence their use in pre–silicon architecture analysis studies). The comparison data shown in Table 2 was collected shortly after the first tape–out for fabrication.

Columns (A) and (B) show "best–case" bounds predicted by the early–stage specification timer, eliot (see Figure 2 and related description). Column (A) shows the most optimistic (idealized) bounds, based on a true dual–load–ported, perfect data cache (without any cache interleave conflicts) and with no execution cycle stalls for data dependencies in the floating point unit. Column (B) reflects a more realistic set of bounds, with stalls caused by factoring in L1 interleaved data cache conflicts and a 2–cycle bubble caused by back–to–back data dependencies in the floating point execution unit. Column (C) shows data obtained by the detailed, trace–driven POWER3 timer, which was developed for pre–silicon analysis. Column (E) shows the corresponding performance numbers obtained from the Texsim (RTL) model runs. Column (D) shows the timer run data obtained by turning off interleaved data cache conflict checking. We see from Table 2 that the POWER3 timer and the Texsim models agree (exactly) for all the test cases considered, (cf. columns (C) and (E)) except for loop03. (The discrepancy for loop03 was found to be due to a modelling error in the timer). In comparing the eliot–specified bounds with actual RTL model performance, we find significant performance gaps in many of these basic cases, except for loop01 and loop02, where the expectation was met precisely. These performance gaps pointed us immediately to post–silicon tuning opportunities.

Table 3 summarizes the diagnosis of the performance bugs and the initial set of recommendations, based on the results presented. These are under current consideration by the design team, in league with our compiler experts. The main deficiencies identified are: (a) interleaved data cache conflict–related stalls of the dual load–store unit; (b) floating point operation issue stalls due to large values of the fpu_dep_delay (see equation 3.10b) in the currently implemented design; (c) inability-to fetch across an I–cache line boundary in a given instruction fetch cycle; and, (d) an identified performance bug in the initial "tape–out" RTL model, which causes stores to finish a cycle later than they should.

[1]Measured numbers for POWER3 are based on Texsim/RTL simulation of the first tapeout model.

Table 2. Eliot generated bounds vs. timer vs. Texsim model results (POWER3)					
Loop–Trace (as in Table 1, with pre–POWER3 compiler)	"Best–case" (eliot) bounds (I–Bound) (A)	"Realistic" achievable bounds (R–Bound) (B)	POWER3 CPU timer (C)	POWER3 timer w/o interl. confl. (D)	Texsim (full RTL model) (E)
	cpi_{ss}	cpi_{ss}	cpi_{ss}	cpi_{ss}	cpi_{ss}
loop01 (u=16)	0.941	0.941	0.941	0.941	0.941
loop02 (u=8)	0.471	0.471	0.471	0.471	0.471
loop03 (u=4)	0.353	0.412	0.412	0.381	0.426
loop04 (u=4)	0.353	0.373	0.382	0.373	0.382
loop05 (u=4)	0.307	0.307	0.333	0.333	0.333
loop06 (u=4)	0.286	0.333	0.381	0.381	0.381
loop07 (u=2)	0.267	0.333	0.422	0.422	0.422
loop08 (u=2)	0.294	0.353	0.451	0.431	0.451
loop09 (u=4)	0.381	0.476	0.476	0.381	0.476
loop10 (u=4)	0.381	0.428	0.476	0.429	0.476
loop25 (u=4)	0.273	0.303	0.333	0.303	0.333
ddot (u=4)	0.307	0.615	0.923	0.923	0.923
daxpy (u=4)	0.353	0.353	0.382	0.382	0.382

Eliot bounds for column (A) assume absence of L1 bank conflicts and 0–cycle bubble for back–to–back flt op dependence

Eliot bounds for column (B) assume L1 cache bank conflicts, and 2–cycle bubble for back–to–back flt op dependence, per current design specs.

cpi_{ss} (steady–state cycles per instruction) is calculated from the *steady–state* part of the corresponding cycle–by–cycle listing for these loop traces.

Loop unrolling and instruction scheduling; measured impact:

Most, if not all of the hardware change recommendations in Table 3, for example, have a possible compiler solution. However, in some cases, a general hardware fix, if feasible, is the preferred solution. In other cases, a hybrid hardware/software compromise solution may be sought. For example a localized re–ordering technique, using a crossbar to sort even/odd references and an "address bellow" register has been used in a prior processor [10] (developed by Silicon Graphics Computer Systems) as a hardware aid to resolve conflicts in the interleaved data cache. This mechanism eases the local misalignment problem for the processor referred to; but, as stated in [10], the compiler is still expected to be responsible for solving the global even/odd address mix problem.

Another example of software tuning is compiler *loop transformation* [12], which offers a known potential for floating point performance enhancement. Loop unrolling, in particular, is an important transformation for enhancing floating point performance in dual–fpu superscalar processors like the

POWER2, POWER3 or the TFP processor [10]. Sufficient unrolling allows additional instruction scheduling opportunities to the compiler within the loop body, and reduces the number of loop–ending branches to be predicted. Excessive unrolling, on the other hand, can cause overhead "spill code" to be generated, or may cause instruction dispatch stall cycles due to resource limits (e.g. the finite number of register rename buffers). In order to feed back POWER3–specific loop unrolling and scheduling heuristics to the compiler group, we experimented with various loop tuning alternatives.

Table 4 shows data from one such experiment using the same loop traces described earlier. We initially used an older, pre–POWER3 compiler, which supported automatic loop unrolling, but which did not have a POWER3–specific scheduling option. With a "–qtune=pwr2" flag, we could obtain code scheduled with the POWER2 processor organization [4,5,8] in mind; without this option, the compiler generates code assuming a (1 LSU, 1 FPU) organization. Column (A) shows the performance data for codes obtained by *manually* unrolling the given loop once, and then applying compiler auto–unrolling. Column (C) is the same as Column (A), except that the "–qtune=pwr2" option was added. Column (B) also uses "–qtune=pwr2" but skipping the manual unrolling step used for Column (A). Column (D) shows the results obtained with a newer compiler, with some *very preliminary* POWER3–specific optimizations. The *effective* unrolling depth in each case is indicated by the parameter u.

The effect of initial unrolling of the source loop by hand (Columns (A) or (C)) was to reduce the number of "load–floating–point–with–update" or lfdu instructions, in favor of plain "load–floating–point" (lfd) instructions. The semantics of an lfdu instruction in the PowerPC architecture [13] calls for updating the address index register contents by the address computed in the current (lfdu) instruction. With machines like the POWER2 [4], in which a 3–input adder is available as part of the address generation logic, such an instruction can be executed ("finished") in a single cycle. In the POWER3, such an instruction can still execute in a single cycle, but if an immediately following load instruction uses the same index register, it will not be able to execute concurrently with the lfdu. Thus a general heuristic which enables reduction of the number of lfdu's generated will help boost POWER3 loop performance; or, in particular, lfdu–lfd pair generation (with a common index register) should be avoided in a POWER3–specific compiler switch. This is observable from the data in Table 4: column (B) shows degraded performance (compared to column (A) in a majority of the cases. It benefits in cases where the code is compute–intensive, and is further enhanced by improved code scheduling ("–qtune=pwr2").

In comparing columns (C) and (D), we see that the biggest improvement achieved in the newer version of the compiler (with a POWER3–specific optimization switch) is reflected in the performance of the dot product loop test case, ddot. This loop exhibited the worst performance gap from the eliot–predicted early bounds specification (see Tables 2 and 3) when using the older compiler. However, the problem caused by generation of excessive (lfdu–lfd) sequences is still seen in the newer results. Also, the loop unrolling depths used by the compiler is not always optimal, from the point of view of steady–state loop cpi values. Note that with auto–unrolling, the old and new compilers do not always use the same unrolling depths (cf. columns (B) and (D)). In a recent paper [14], we presented methods for predicting the "best" loop unrolling depths for POWER3–like machines, using static bounds–based heuristics. We touch on this issue briefly in sub–section 4.4.

Table 3. Performance Bug Diagnosis Table and Recommended Solutions					
Test Case	Deviation of tape—out model* perf from eliot—based(best—case, ideal-ized) specs	Primary Cause of deviation, or of poor per-formance	Secondary Cause	Possible fix(es), or, possible enhance-ments rmance	Current Rec-ommendation
Loop01	~0 %	1 store port (Dcache)		(Eff.) 2 store ports	None
Loop02	0 %	Same;above		Same;above	None
Loop03	17.8 %	Dcache interl. conflicts	Ifetch/disp restrictions	True2—ported dcache; or, memory re-quest reorder buffer and/or compiler help	Mem. request reorder buffer (ROB) scheme + compiler support
Loop04	16.7 %	Dcache interl. conflicts		Same;above	Same;above
Loop05	9.1 %	Icache line crossing, ifetch/disp restrictions		Ifetch across line boundary in same cycle	Investigate h/w logic add to fix problem
Loop06	31.8 %	flt. pipe de-pendency delay		Tuneout—of—order flt disp. and result for-warding logic; compiler tune	Improve logic, circuit design for dep. check, reor-dering
Loop07	54.8 %	Same;above		Same;above	Same;above
Loop08	29.5 %	Same;above	Dcache interl. conflicts	See above (loops 06, 03)	See above (loops 06, 03)
Loop09	37.4 %	Dcache interl. confl.	Flt. pipe dep. stall	See above (loops 03, 06)	See above (loops 03, 06)
Loop10	25.7 %	Dcache interl. confl.	Flt. pipe dep. stall	See above (loops 03, 06)	See above (loops 03, 06)
Loop25	21.9%	Dcache interl. confl	Ifetch/disp restrictions	See above (loops 03, 05)	See above (loops 03, 05)
ddot	99.7 %	flt pipe dep.		See above (loop06)	See above (loop06)
daxpy	8.21 %	late store fin		H/w fix	H/w fix

Table 4. Compiler Scheduling Sensitivity

Loop Trace	cpi_{SS} values for POWER3—Texsim (RTL) model runs			
	Old (pre—POWER3) Compiler			Newer compiler with *some* (preliminary) POWER3—specific scheduling (xlf ver. 4.1.0.3)
	a) hand–unrolled once b) then auto–unrolled (–O3) (A)	**a)auto unrolled(–O3) b)qtune=pwr2 (B)**	**a)hand–unrolled once b)then auto–unrolled(–O3) c)qtune=pwr2 (C)**	**auto–unrolled (–O3) (D)**
Loop01	0.941 (u=16)	0.889 (u=8)	0.941 (u=16)	0.889 (u=8)
Loop02	0.471 (u=8)	0.471 (u=8)	0.471 (u=8)	0.485 (u=16)
Loop03	0.426 (u=4)	0.412 (u=4)	0.441 (u=4)	0.470 (u=8)
Loop04	0.382 (u=4)	0.412 (u=4)	0.455 (u=4)	0.409 (u=8)
Loop05	0.333 (u=4)	0.385 (u=4)	0.333 (u=4)	0.320 (u=8)
Loop06	0.381 (u=4)	0.321 (u=4)	0.321 (u=4)	0.341 (u=8)
Loop07	0.422 (u=2)	0.333 (u=2)	0.356 (u=2)	0.336 (u=4)
Loop08	0.451 (u=2)	0.467 (u=2)	0.364 (u=2)	0.432 (u=4)
Loop09	0.476 (u=4)	0.619 (u=4)	0.619 (u=4)	0.500 (u=8)
Loop10	0.476 (u=4)	0.537 (u=8)	0.476 (u=4)	0.512 (u=8)
Loop25	0.333 (u=4)	0.471 (u=2)	data not avail.	0.424 (u=4)
daxpy	0.353 (u=4)	0.485 (u=8)	0.394 (u=4)	0.409 (u=8)
ddot	0.923 (u=4)	0.960 (u=8)	0.960 (u=4)	0.385 (u=4)
REMARKS	partial manual unrolling reduces number of lfdu's	too many lfdu's; hurts many loops;	partial manual unrolling reduces number of lfdu's	lfdu problem still present
		qtune=pwr2 helps reduce penalty due to flt pipe dep. delays		POWER3—specific scheduling does an even better job of reducing penalty due to flt pipe dep. delays (esp. ddot)

It should be noted that the *steady state* loop cpi values (cpi_{SS}) shown in Tables 2 and 4 can be slightly misleading when comparing the total execution times for a *finite* number of iterations of the

original (not unrolled) loop. A more meanigful metric to use, in such studies, may be *cycles per logical (not unrolled) iteration,* cpI. The relation between cpi_{ss} and cpI_{ss} is given by:

$$cpI_{ss} = (cpi_{ss} * ipI)/u \qquad (4.3.1)$$

where, ipI denotes the number of instructions per iteration in the unrolled loop and u is the depth of unrolling. (Note: upper case I is used to abbreviate *iteration*; lower case i stands for *instruction*). Thus, for example in loop01, the original (not unrolled) loop body consists of a single floating point store with update (stfdu) followed by the loop–ending branch, so ipI = 2 for u=1. With u=8 (columns (B) or (D)), we have ipI=9 (8 stores plus a branch); with u=16 (columns (A) or (C)), ipI=17 (16 stores plus a branch). With these values, the cpI_{ss} value (using equation 4.3.1) for *any* of the columns (A), (B), (C) or (D) for loop01 would be 1.0. That is, the steady–state cycles per (logical) iteration performance for loop01 is 1.0, irrespective of the compiler options experimented with (Table 4).

In the POWER2/P2SC loop performance tuning studies reported in the next sub–section, we use the cycles–per–(logical)–iteration (cpI) metric as the basis of evaluation.

4.3 POWER2 and P2SC Loop Performance

The POWER2 (and P2SC) infinite cache, floating point microarchitecture is similar to that of the POWER3, in that both are (2 LSU, 2 FPU) machines. There are some important distinctions, though. The POWER2 architecture has the *load/store* floating point *quadword* instructions (see Section 2); POWER3 does not. On the other hand, the POWER3 has a branch target address cache (BTAC) [3] mechanism for fetch prediction, which eliminates fetch stalls for loop–ending branches.

Let us first examine some comparative performance data to understand the characteristics of POWER2. This data illustrates how the performance may vary drastically, even though the basic organization is still a (2 LSU, 2 FPU)–structure and the compiler version is unchanged. Table 5 shows the comparative "cycles per logical iteration" (cpI) performance of POWER2 and POWER3. The POWER3 data shown is of course based on the latest available RTL *simulation* results, with the latest available compiler (column (D) data in Table 4). The POWER2 hardware measurement data are reported for the same compiler, but with two different options of the instruction set architecture. The "–qarch=com" mode generates code in the so–called "common" mode, which implies an "intersection" of POWER™, POWER2™ PowerPC™ architecture opcode domains (see [13]). The –qarch=pwr2 allows the use of additional instruction opcodes which are unique to the POWER2 machine: in particular, the floating point *load and store quadword* instructions (see Section 2).

We see from Table 5, that in terms of *architectural* (cycles–per–iteration) performance, the POWER2 does better than the initial tape–out version of POWER3 (and compiler) for many of the elementary loop test cases considered in this paper. (Of course, the POWER3's MHz performance is considerably higher than that of the latest POWER2 and P2SC processors. Also, in terms of real, finite cache performance, which is beyond the scope of this paper, other features like data cache prefetch enable the POWER3 to meet pre–silicon targets. The POWER3 design factors in requirements for server products with a broader overall market than its predecessors. It provides for PowerPC architecture compatibility, superior integer/branch performance and support for multiprocessing. As

stated in the introduction, overall system performance and tuning issues are not dealt with in this paper).

Table 5. POWER3 cycles–per–Iteration (cpI) comparison with POWER2 hardware measurements			
	POWER2(–qarch=com)	POWER2 (–qarch=pwr2)	POWER3 RTL simulation (–qarch=com)
loop01	0.84	0.42	1.0
loop02	1.02	0.51	1.0
loop03	1.77	1.01	1.94
loop04	1.52	0.89	1.69
loop05	1.27	1.20	1.0
loop06	1.65	1.52	1.75
loop07	3.53	1.53	2.44
loop08	3.54	2.53	3.56
loop09	2.03	1.57	2.56
loop10	2.03	1.54	2.63
loop25	2.53	2.53	2.5
ddot	1.02	0.77	1.25
daxpy	1.78	0.89	1.69

The amount of benefit exploited by POWER2 using load/store *quadword* instruction support can be seen by comparing data columns 1 and 2 of Table 5. Once the effect of load/store quads is factored out, the POWER3 performance (data column 3) compares favorably with POWER2 performance (data column 1): in a couple of cases (loop05 and loop07) the POWER3 actually does quite a bit better, because of its superior out–of–order instruction scheduling.

Let us now examine the loop performance characteristics of the POWER2/P2SC microarchitecture in detail. In Table 6, we present loop performance data measured on a 160 MHz P2SC processor, using the latest available compilers. We compare this with predicted I– and R–Bounds. We have also listed the *primary* hardware constraints which inhibit attainment of the "best–case" I–Bounds.

As before, the effective unroll factor (u) is shown for each loop in Table 6. For lack of space, it is not possible to discuss each loop in detail. Let us consider a couple of the ones which exhibit a large gap between the I–Bound and either the corresponding R–Bound or the actual measured performance. These loops are: loop01, loop05, loop06 and loop08. Of these, loop05 and loop06 have the same underlying cause behind their performance shortfall. Let us consider loop01 and loop05 in some detail. The compiled code sequence of interest for **loop01** is:

```
stfq  31,  0, 16(3)
stfq  31,  0, 32(3)
stfq  31,  0, 48(3)
```

stfqu 31, 0, 64(3)
bc

Table 6. Actual measured performance (cycles per iteration, cpI): P2SC

LOOP	Current xlf compiler*	Current xlhpf compiler+	I–Bound	R–Bound	Primary h/w bottlenecks	Compiler solution ?
loop01(u=8)	0.43	0.43	0.25	0.417	store–q size	No
loop02(u=8)	0.52	0.52	0.50	0.50	–	–
loop03(u=4)	0.89	0.89	0.75	0.85	store–q, rename buffs	No
loop04(u=4)	0.85	0.85	0.75	0.85	–as above–	–as above–
loop05(u=4)	1.01	1.01	0.50	1.0	store–add anti–dep., renames	unroll deeper, s/w pipeline
loop06(u=4)	1.76	1.76	1.0	1.25	–as above–	–as above–
loop07(u=4)	1.66	1.53	1.5	1.5	–	–
loop08(u=4)	3.02	2.28	1.5	1.75	rename buffs	improved reg alloc
loop09(u=4)	1.15	1.14	1.0	1.15	–as above–	–as above–
loop10(u=4)	1.15	1.14	1.0	1.15	–as above–	–as above–
loop25(u=4)	2.07	2.06	2.0	2.0	comp. bound (2 FPU), reg pressure	better schedule
daxpy(u=8)	0.85	0.85	0.75	0.85	see loop03	see loop03
ddot(u=4)	0.77	0.77	0.50	0.75	load–fma direct dep.	schedule

* xlf version 4.1.0.4, with –O3 –qarch=pwr2
\+ xlhpf version 1.2. with –qnohpf –O3 –qarch=pwr2
(In using xlhpf, a new, improved register allocator is invoked)

Note that store quadword instructions have been used, since they are available on the POWER2 and P2SC machines. Each stfq corresponds to two stfd instructions and hence two iterations of the original, not–unrolled loop (see Table 1). Thus, clearly, the effective unrolling depth invoked by the compiler in this case is u=8. Each stfq instruction stores the pair of floating point registers: #31 and #0. The two doublewords (or one quadword) forming the data, are stored at the address specified by a displacement and a base address register. The latter is specified to be the fixed point register #3 in each case. The difference between two successive displacements (e.g. 16 and 32) is 16 bytes, which is one quadword. Since the number of LSU's is 2, and the number of store–ports is 2, it is easy to see

that the steady state cycles per iteration bound of this loop is 4/2 = 2. (The branch is overlapped with store execution). However, one iteration of this loop corresponds to 8 iterations of the original loop. Hence, the steady–state cycles–per–(logical)–instruction I–Bound is: cpI = 2/8 = 0.25.

The actual measured performance for this loop on the P2SC is cpI = 0.43. Since the code consists essentially of a sequence of stores, the primary suspect in terms of finite hardware resources, is clearly the pending store queue. For this machine, the size of this queue is 6. As verified through exact cycle–by–cycle simulation data, this limitation causes the actual, steady–state completion pattern of stores (stfq's) to be: 2–2–2–0–0–2–2–2–0–0–.......; i.e., three consecutive pairs of completion, followed by 2 stall cycles. This results in a cycle count of 5 cycles for 12 (original, not–unrolled) iterations, which gives: cpI = 5/12 = 0.417. The stalls happen because of the latency mismatch between the address generation path and the path which reads the store data from the source registers and writes it into the store queue. Let us denote these two pipe latencies to be *agen_pipe* and *sdata_pipe*. If these effective latency numbers are factored into the bounds model, the stall pattern above can be predicted. If (sdata_pipe – agen_pipe) > 3, then after every 3 cycles (or 6 stfq agens), the store queue fills up, causing a stall. The number of *stall–cycles* can easily be shown to be given by (sdata_pipe – agen_pipe – 2). For the P2SC, sdata_pipe = 5 and agen_pipe = 1. (Note, these are the *effective* values of these latencies under steady–state processing of the loop shown. The hardware latency of the sdata_pipe is actually only 3. However, an agen–stall occurs a cycle before the store queue fills up. Similarly, there is a cycle loss because a cache–array write from the head of the store queue begins a cycle after the store data is paired with the store address in the queue. This results in an effective sdata_pipe latency of 3+1+1 = 5. The two extra latency cycles are subtracted out during the calculation of the number of stall cycles. This is because, the agen–stall is eliminated immediately after the store queue begins to get drained). Thus, the R–Bound (= 0.417) generated by eliot agrees with simulation–based expectation for the P2SC microarchitecture.

Given the nature of the hardware constraint, which results in an R–Bound of about 0.42, it is quite clear that changes to the loop unrolling depth will not help in reducing the gap between I–Bound and R–Bound in this case. Irrespective of the unrolling depth, the effective code sequence is a string of stores, which results in the stall pattern shown. Of course, the use of store quadword instructions helps; use of the lower bandwidth stfdu instructions would glean a cpI of 0.84 (see Table 5, first data column).

For **loop05,** the compiler–generated loop body is as follows:

```
stfq   0, 1, 16(4)
stfqu  6, 7, 32(4)
fadd   0, 31, 2
fadd   1, 31, 3
```

```
lfq    2, 3, 16(3)
fadd   6, 31, 4
fadd   7, 31, 5
lfqu   4, 5, 32(3)
bc
```

In I–Bound mode, the steady–state issue groups would be: (stfq, stfqu, fadd, fadd) and (lfq, fadd, fadd, lfqu, bc), yielding a rolled–loop cpI of 2. Since the effective unrolling depth is u=4, the real I–Bound is cpI = 2/4 = 0.5. The actual measured performance is 1.01, which is half the speed of the "best–case" expectation. The reason behind the performance gap is not hard to infer: the data dependencies between the stores and the subsequent fadd instructions. In the POWER2 and P2SC implementations, target register renaming is supported only for floating point loads, *not* for functional operations. Thus the anti–dependency between the first stfq and the first fadd (via register #0), causes an issue stall of (sdata_pipe – agen_pipe – 1) = (3 – 1 – 1) = 1 cycle. Note that in this case, the effective value of sdata_pipe is equal to the hardware latency of 3 stages, since there is no agen–stall to be considered. The other 1 cycle saving occurs due to store data forwarding to the FPU pipe. Thus, the steady–state issue groups, in R–Bound mode are: (stfq, stfqu), ( ), (fadd, fadd, lfq), (fadd, fadd, lfqu, bc). This gives a rolled–loop cpI of 4 and the final R–Bound cpI is 4/4 = 1, which matches the hardware measurement.

Anytime there are dependency–caused stalls as above, the question to ask is: can the I–Bound be met or approached through better register allocation and/or scheduling? In such situations, it is often the case that deeper unrolling exposes better opportunities to create a schedule which is free of dependence–stalls. In the above case, by using manual unrolling at the source–level, we were able to coax the compiler into generating a software–pipelined schedule, with a measured cpI of 0.64. The inner loop of interest in this case was as follows:

```
stfq   2, 3, 16(4)
fadd  4, 31, 4
lfq    2, 3, 16(3)
fadd   5, 31, 0
stfq    6, 7, 32(4)
fadd   1, 31, 1
lfq    6, 7, 32(3)
fadd  0, 31, 0
stfq  4, 5, 48(4)
fadd  2, 31, 2
lfq    4, 5, 48(3)
```

```
fadd  3, 31, 3
stfqu  0, 1, 64(4)
fadd  6, 31, 6
lfqu  0, 1, 64(3)
fadd  7, 31, 7
bc
```

This schedule eliminates the dependence–stall and results in a rolled cpI of 4, and an I–Bound cpI of 4/8 = 0.5. The degraded value of the measured cpI (0.64) can now be explained as being due to dispatch stalls resulting from rename buffer pressure. Since the renamed loads run ahead of the computation (due to the longer latency FPU pipes), there are eventually stalls created because of lack of free rename buffers. For the above case, it can be shown that the effective rolled cpI becomes 5, yielding an R–Bound of 5/8 = 0.63, which matches the measured performance.

4.4 Prediction of Optimal Loop Unrolling Depth

In this section, we briefly illustrate the use of bounds–based characterization in determining a suitable loop unrolling depth in order to get the best performance [14]. This is a topic of ongoing work and the details of the unrolling algorithms will be reported later in a separate report. Here, we discuss the basic comcept only, with the use of a simple example from our suite of loops.

Let us consider **loop03** which is the third loop in our suite. The compiled code for the innermost loop for a certain compiler version (without unrolling, i.e. with **u=1)** was :

```
lfd      0,  0x808(4)
lfd      1,  0x1608(1)
fadd    0,   0,  1
stfdu   0,   0x8(4)
bc
```

The compiled codes for **u =2** and **u=3** are also given below:

For u = 1, and a (1 LSU, 1 FPU)–machine, performance is load–store bound. The steady–state, idealized cycles–per–instruction (cpi) and cycles per–flop (cpf) performance of loop03 can be computed using the simple bounds model as being: (cpI = 3) ==> cpi = 0.6 and cpf (cycles per floating point operation) = 3.0. For u=2, it is still load–store (agen) bound, and the idealized performance is: (cpI = 6) ==> cpi = 0.66, cpf = 3.0. For u =3, we have: (cpI = 9) ==> cpi = 0.69, cpf = 3.0. For u =4, we would have: (cpI = 12) ==> cpi = 0.705, cpf = 3.0. In general, we would have: cpI = 3*u and N = 4*u + 1, and N_F = u. From these, the above values can be computed.

u = 2:

```
lfd    0,  0x808(4)
lfd    1,  0x1608(4)
fadd   2,  0,  1
lfd    0,  0x816(4)
lfd    1,  0x1616(4)
fadd   0,  0,  1
stfd   2,  0x8(4)
stfdu  0,  0x8(4)
bc
```

u = 3:

```
lfd    0,  0x808(4)
lfd    1,  0x1608(4)
fadd   3,  0,  1
lfd    2,  0x1624(4)
lfd    0,  0x816(4)
lfd    1,  0x1616(4)
fadd   1,  0,  1
stfd   3,  0x8(4)
lfd    0,  0x824(4)
fadd   0,  0,  2
stfd   1,  0x16(4)
stfdu  0,  0x24(4)
bc
```

Thus, for a (1LSU, 1FPU)–machine, it would not pay to unroll a loop like that illustrated above to get additional floating point performance. This is clear from simple bounds–based reasoning, but was verified via detailed timer runs.

Let us now look at an enhanced design: a (2 LSU, 2 FPU)–machine, like POWER3, with l_ports = 2, s_ports = 1, and disp_bw = compl_bw = 4. The loop performance (with or without unrolling) is still agen–bound; however, now we can see a benefit in unrolling. For u = 1, we have: (cpI = $\lceil 3/2 \rceil$ = 2) ==> cpi = 0.4 and cpf = 2.0. For u = 2, we get: (cpI = $\lceil 6/2 \rceil$ = 3) ==> cpi = 0.33, cpf = 1.5. For u = 3, we have: (cpI = $\lceil 9/2 \rceil$ = 5) ==> cpi = 0.38, cpf = 1.67. For u=4, we get: (cpI = $\lceil 12/2 \rceil$ = 6) ==> cpi = 0.353, cpf = 1.5. In general, we would have: cpI = $\lceil (3*u)/2 \rceil$, N = (4*u) + 1, and N_F = u. From these, the above values can be computed. Clearly, from an idealized, steady–state cpf (megaflops) performance, u = 2, 4, 6, ... etc would result in the optimal performance of cpf = 1.5. However, in the real case, due to other limits like the number of rename buffers or the size of the reorder buffer, etc., performance would actually go down for higher choices of u. In fact, using a pre–silicon timer (e.g. H–timer; see Figure 2) we verified that for a POWER3, peformance actually degrades for u = 6 and beyond.

5. Conclusion

We have presented the practical use of a simple, bounds–based analysis method in the context of post–silicon loop performance validation and tuning. After the microprocessor chip is back from the silicon fabrication process, the first concern, of course is *functional* validation. Once the key test cases are operational (i.e. they produce correct results), the next concern is *performance*. The post–

silicon analysis and tuning experiments generate knowledge and understanding required for machine–specific compiler tuning. In order to validate performance, one needs to know what the desired specifications (or targets) are. Bounds–based analysis provides a means for fixing a range of such targets: from the "best–case" or idealized, to a more practical, achievable one. Comparison of the idealized and realistic bounds tells us how much degradation is caused by hardware resource constraints and data dependencies. Comparing the bounds with the actual performance data allows us to assess the effectiveness of the (microarchitecture, compiler) pair. In some cases, it is possible to suggest very specific compiler enhancement ideas to bridge the observed performance gap.

Loop performance can be a crucial determinant of overall floating point performance. As reported earlier [15], a simple hardware "fix" to achieve the daxpy performance I–Bound for a *pre–POWER3* design point resulted in a SPECfp92 performance boost of 14 %. (Since this performance bug had to do with store instruction latency and store queue size, the solution led to an unexpected 11 % boost in SPECint92 performance as well!). In this paper, we illustrated the application of bounds–based analysis to post–silicon diagnosis and tuning for two of the most recent super scalar processors developed by IBM. Both of these processors are designed to provide superior floating point performance, and are targeted for high–end RS/6000 workstations and servers. We discussed the basic concepts and their application in the context of a few application–based loop kernels. In practice, a suite of well over 100 loops and other test cases, along with other applications are used in our post–silicon performance validation and tuning exercises.

REFERENCES

1. G. F. Grohoski, "Machine organization of the IBM RISC System/6000 processor," *IBM J. Res. Develop.,* vol. 36, no. 1, pp. 37–58, January 1990.

2. S. Song, M. Denman and J. Chang, "The PowerPC 604 RISC microprocessor," *IEEE Micro,* pp. 8–17, October 1994.

3. D. Levitan, T. Thomas and P. Tu, "The PowerPC 620TM microprocessor: a high performance superscalar RISC microprocessor," *Proc. COMPCON*, pp. 285–291, March 1995.

4. S. W. White and S. Dhawan, "POWER2: next generation of the RISC System/6000 family," in PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000, IBM Corporation, publication no. SA23–2737–00, 1994.

5. L. Gwennap, "IBM crams Power2 on to single chip," *Microprocessor Report*, August 26, 1996.

6. H. Q. Le, P. Bose, D. Schroter and M. Mayfield, "Design point definition of the core microarchitecture of a high end PowerPCTM processor," (under clearance for publication).**

7. P. Bose and S. Surya, "Architectural timing verification of CMOS RISC processors," *IBM Journ. Res. & Develop.*, vol. 39, no. 1/2, pp. 113–129, January/March 1995.

8. E. L. Hannon, F. P. O'Connell and L. J. Shieh, "POWER2 performance on engineering/scientific applications," in PowerPC and POWER2: Technical Aspects of the New IBM RISC/System/6000™, IBM Publication Number SA23–2737–00, 1994; see also, *Proc. ICCD,* pp. 336–339, October 1994.

9. J. E. Smith and A. R. Pleszkun, "Implementation of precise interrupts in pipelined processors," *Proc. Int'l. Symp. on Computer Architecture (ISCA)*, pp. 36–44, 1985.

10. P. Y–T Hsu, "Design of the TFP Microprocessor," *Proc. IEEE Micro,* 1994.

11. W. Mangione–Smith, T.–P. Shieh, S. G. Abraham, and E. S. Davidson, "Approaching a machine application–bound in delivered performance on scientific code, *Proc. IEEE,* vol. 81, pp. 1166–1178, August 1993.

12. D. F. Bacon, S. L. Graham and O. J. Sharp, "Compiler transformations for high–performance computing," *ACM Computing Surveys,* vol. 26, no. 4, pp. 345–420, December 1994.

13. C. May, E. Silha, R. Simpson, H. Warren, ed., The PowerPC Architecture: A Specification for a New Family of RISC Processors, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 2nd. edition, May 1994.

14. P. Bose and J–D Wellman, "Compiler–aided loop tuning opportunities for high–end PowerPC™ Machines," *Proc. Workshop on Interaction between Compilers and Computer Architectures,* (held in conjunction with 3rd. Int'l. Symp. on High–Perf. Computer Arch., HPCA–3), Feb. 1997; available as an IBM Research Report.

15. P. Bose, "Performance analysis and verification of super scalar processors," IBM Research Report RC 20094, June 1995; parts of this were presented as a talk at the ISCA–95 workshop on pre–silicon performance analysis and validation, Santa Margherita, Italy, 1995.

IBM is a registered trademark, and POWER, PowerPC, PowerPC Architecture, PowerPC 604, PowerPC 620, POWER2, POWER3 and RISC System/6000 are trademarks of International Business Machines Corporation. In this document the terms "604" and "620" are used as abbreviations for the phrases "PowerPC 604 microprocessor" and "PowerPC 620 microprocessor" respectively. SPEC is a registered trademark of Systems Performance Evaluation Corporation.

** This is a PowerPC processor, developed by IBM, which will be announced as POWER3.

Copies may be requested from:

IBM Thomas J. Watson Research Center
Publications Office, 16–220
Post Office Box 218
Yorktown Heights, NY 10598

Some reports are available via the
Cyberjournal on the WWW.
http://www.watson.ibm.com:8080