Characterizing and Improving the Performance of Bioinformatics Workloads on the POWER5 Architecture

Vipin Sachdeva, Evan Speight, Mark Stephenson
Novel Systems Architectures
IBM Austin Research Lab
Austin, TX 78758
Email: vsachde,speight,mstephen@us.ibm.com

Lei Chen
IBM Systems and Technology Group
Austin, TX 78758
Email: chenl@us.ibm.com

***Abstract*— This paper examines several mechanisms to improve the performance of life science applications on high-performance computer architectures typically designed for more traditional supercomputing tasks. In particular, we look at the detailed performance characteristics of some of the most popular sequence alignment and homology applications on the POWER5 architecture offering from IBM. Through detailed analysis of performance counter information collected from the hardware, we identify the main performance bottleneck in the current POWER5 architecture for these applications is the high branch misprediction penalty of the most time-consuming kernels of these codes. Utilizing our PowerPC full system simulation environment, we show the performance improvement afforded by adding conditional assignments to the PowerPC ISA. We also show the impact of changing the number of functional units to a more appropriate mix for the characteristics of bioinformatics applications. Finally, we examine the benefit of removing the two-cycle penalty currently in the POWER5 architecture for taken branches due to the lack of a branch target buffer. Addressing these three performance-limiting aspects provides an average 64% improvement in application performance.**

I. Computational Biology and High-Performance Computing

With the discovery of the structure of DNA and the development of new techniques for sequencing the entire genome of organisms, biology is rapidly moving towards a data-intensive, computational science. Biologists search for bio-molecular sequence data to compare with other known genomes in order to determine functions and improve understanding of biochemical pathways. This understanding can lead to disease prevention and cure, and an understanding of the mechanisms of life itself. Computational biology has been aided by recent advances in both technology and algorithms, such as the ability to sequence short contiguous strings of DNA and from these to reconstruct the whole genome [1], [2], [3]; and the proliferation of high-speed micro array, gene, and protein chips [4] for the study of gene expression and function determination. These high-throughput techniques have led to an exponential growth of available genomic data. Many new challenges in Bioinformatics require high-performance computing due to either their massive parallelism or optimization difficulties that are often combinatoric and NP-complete.

Previous work presented the BioPerf benchmark suite [5], [6], which represents a wide variety of applications selected from computational biology and Bioinformatics. In this paper, we focus on the performance of sequence alignment and homology applications on the POWER5 architecture. Most sequence alignment applications are integer-based, contain irregular data structures, exhibit a higher percentage of load/store instructions, and utilize conditional branches that are typically difficult to predict. Thus, special effort needs to be made to include the performance profiles of these applications into designing future systems. Due to the fact that current high-end architectures have not been primarily designed with this class of applications in mind, and given its rising importance in the arena of high performance computing, we are interested in uncovering limitations in current architectures and finding architectural trade-offs that can benefit computational biology without degrading performance of traditional supercomputing workloads.

Through detailed analysis and simulation results, we find that we achieve a performance improvement of more than 60% through the use of new instructions and predicated branches in specific functions where the branch misprediction penalty is high. We also explore other issues for enhancing performance, including an increased number of fixed-point units to better suit the instruction profiles of bioinformatics workloads, and reducing the latency of taken branches through the inclusion of a tiny Branch Target Address Cache (BTAC). Finally, we show the performance improvement for each of the applications from each of these steps individually, and when applied together. Our results demonstrate the importance of architectural research for Bioinformatics workloads considering the difference in their performance profiles from past supercomputing applications.

II. Application of Sequence Analysis

We have analyzed the performance of four popular Bioinformatics applications – *Blast*, *Clustalw*, *Fasta* and *Hmmer*. These four applications encompass the sub-field of sequence analysis

in computational biology. As one of the most commonly performed tasks in Bioinformatics, sequence analysis is used to solve the problem of finding similar or different sequences of nucleotides or amino acids. Pairwise alignment finds its application in similarity searching where uncharacterized but sequenced "query"genes are scored against vast databases of characterized sequences.

The *blastp* executable, part of the Blast package, is a widely-used tool for searching protein databases for sequence similarities. The new gapped alignment algorithm implemented in a recent version of Blast uses dynamic programming to extend a central pair of aligned residues in both directions [7]. *Ssearch34_t*, part of Fasta, performs a pairwise sequence comparison using dynamic programming with the Smith-Waterman [8] algorithm. It identifies disjointed regions in the two sequences that are similar to each other. This step takes up almost 99% of the total runtime. Clustalw is a progressive multiple sequence alignment application that takes three steps to complete an alignment. In the first step, all sequences are compared pairwise using the global Smith-Waterman algorithm. This step performs a total of $\frac{n(n-1)}{2}$ alignments for n sequences. A cluster analysis is then performed on each of the scores for the pairwise alignment to generate a hierarchy for alignment (guide tree). The alignment is built by adding one sequence at a time, based on the guide tree. Finally, the *Hmmpfam* binary, part of Hmmer, aligns a sequence with a database of hidden Markov models constructed previously from biological sequence families. Each of these alignments are performed using either the Viterbi algorithm [9] or the forward algorithm. For further details on Blast, Clustalw, Fasta, and Hmmer, refer to [7], [10], [11], and [12] respectively.

III. Current Performance of Bioinformatics Kernels on POWER5

We examined the performance of four applications from the BioPerf suite when executing on a 1.65 GHz IBM eServer OpenPower 720 [13] with the POWER5 processor and 16GB of system memory. The applications were compiled with the GCC (version 4.1.1) compiler using the -O3 optimization level. We choose the metric "instructions per cycle", or IPC, as our measure of performance. This indicates the average number of instructions the superscalar processor was able to commit in a single cycle. Obviously, for a fixed number of instructions, a higher IPC indicates better performance. The POWER5 is an eight-way issue processor, but other pipeline stages limit the commit throughput to five instructions per cycle at most. Typically, an IPC of above 2 is considered quite good for this architecture.

Previous data on sequence alignment applications have shown that they have negligible floating point operations but a higher percentage of loads/stores and branch instructions compared with more traditional high performance computing benchmarks such as SPEC [14]. Most of the branch instructions are conditional branches with a high branch misprediction rate [15]. Branch misprediction is the primary reason for the low IPC numbers for these applications (See Table I).

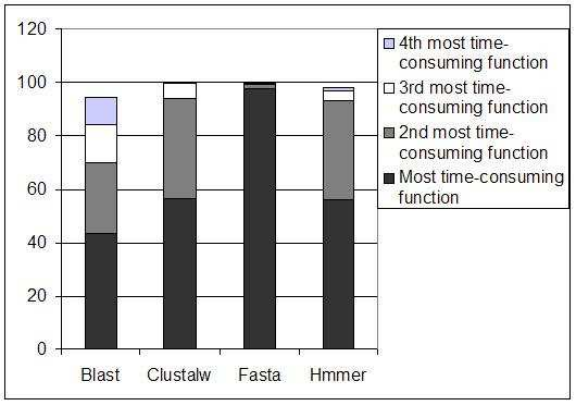

Fig. 1. Function-wise breakout of Blast, Clustalw, Fasta, and Hmmer

To begin our analysis, we used the *gprof* tool to determine the four most time-consuming functions for each application. As shown in Figure 1, all the applications except Blast spend more than half of the execution time in a single function: *P7Viterbi* for Hmmer, *forward_pass* for Clustalw, and *dropgsw* for Fasta. Blast spends more than 40% of the execution time in the function *SEMI_G_ALIGN_EX*. More importantly, **dynamic programming** is employed in all of these functions. For all results shown in this paper, we use the largest *class-C* inputs included in the BioPerf [5] suite.

Among these four applications, Blast, Clustalw, and Hmmer use much shorter alignment data than Fasta. However, they execute the dynamic programming kernel multiple times. As a result, the dynamic programming functions take up most of the execution time, and small performance degradations in the dynamic programming kernel will lead to a large increase in execution time. Other research [16], [17], [18] has shown that dynamic programming is not only common in sequence analysis and comparison, but is also fundamental to many solutions in phylogeny and secondary structure protein prediction. These areas comprise a large part of the research in computational biology. Therefore, any effort to optimize the dynamic programming kernel will improve the performance of many other computational biology applications.

Because dynamic programming kernels are common and time-consuming for many sequence analysis packages and computational biology as a whole, we sought to optimize the performance of this kernel for maximum performance improvement for each application. We analyzed the dynamic programming approaches in each of these kernels at the architecture level by collecting hardware performance counters data. We concentrate on the data for Clustalw, although the results are generally applicable to the other three packages as well.

POWER5 processors have 140 different performance counter groups, each of which provides six counters to monitor six different events dynamically at certain intervals. Table I shows the level 1 data (L1D) cache miss rate, the percentage of branch mispredictions due to incorrect direction prediction, and the completion stalls represented as a percentage of total execution cycles. Note in particular that although these appli-

Application	IPC	L1D Miss Rate	% Mispredicted Branches Due to Incorrect Direction	Stalls due FXU instructions
Blast	.9	3.9%	99.98 %	14.9%
Clustalw	1.1	0.1%	99.8%	25.3%
Fasta	.8	1.3%	99.8%	14.3%
Hmmer	1.0	1.5%	96.8%	5.7%

TABLE I
HARDWARE COUNTER DATA FOR BLAST, CLUSTALW, FASTA, AND HMMER

cations have an extremely low L1D miss rate, the IPC is not particularly high, confirming our supposition that something other than cache performance is limiting overall application performance.

Figure 2 shows how the IPC and branch misprediction rate vary with time for the Clustalw application. This application exhibits a very high L1D cache hit rate (over 99.5%). As evidenced by Figure 2, the IPC tracks the variations in the branch prediction rates for this application. Anywhere from one-in-five to one-in-eight branches is mispredicted in Clustalw, depending on the phase of the application. There are two components to branch prediction: predicting branch direction, and predicting the branch target address. Table I indicates that nearly all branch mispredictions shown in Figure 2 are due to incorrectly predicting branch direction. In other words, it is very difficult to predict whether a branch in this code will be taken or not taken. Contrast this with loop-based codes, in which every branch ending a loop is generally correctly predicted as taken except for the last iteration of the loop.

The performance of this workload on POWER5 is hindered by the large number of branch instructions. Adding to the inherent unpredictability of the branches in this type of application, each taken branch, whether predicted or not, incurs a 2-cycle delay (3-cycle if SMT is enabled) during which the next value of the program counter is calculated. This pipeline "bubble" can add significant time to an application's execution if the percentage of taken branches is high. In Section VI, we experiment with eliminating this penalty and measure the resulting performance improvement.

A further analysis of the source code of the Clustalw package reveals that the function *forward_pass* takes up more than 99% of the run cycles for the *pairalign* function. The *forward_pass* function corresponds to the Needleman-Wunsch alignment algorithm, an extension to the Smith-Waterman algorithm. The pseudo code below describes the Smith-Waterman approach.

The **max** statements in the code example are implemented in the source code as short conditional statements ($if(a < b) \;\; a = b$). In the assembly, these conditional statements are translated into compare and conditional branch instructions. For example, the *forward_pass* function in Clustalw includes five such conditional statements.

The prediction accuracy for the conditional branches during the dynamic programming portion of the algorithm was severely limited by the fact that the branches from the **max** statements are highly value dependent. This leads to a high misprediction rate, resulting in frequent pipeline flushes and performance degradation. These results point to the problem of the branch mispredictions as being the most important factor for the performance of these applications. Due to the characteristics of the workload, improving the accuracy of the branch predictor would be difficult and may not be applicable to other workloads. Therefore we look for solutions other than improving the branch predictor to address this problem.

Data : (1) Two sequence S_1 and S_2 of length m and n respectively
(2) Gap initiation penalty of W_g and a gap extension penalty of W_s
(3) Four two-dimensional matrices *V, E, F, G* for storing the intermediate values
(3.1) $V(i,0) = E(i,0) = -W_g - iW_s$
(3.2) $V(0,j) = F(0,j) = -W_g - jW_s$
(4) W_{ij} which is the score of aligning the i^{th} character of the S_1 sequence with the j^{th} character of the S_2 sequence.

Result: The alignment score $V(i,j)$ obtained by aligning the S_1 sequence with the S_2 sequence.

begin
  while $i++ \leq m$ **do**
    while *(*$j++ \leq n$*)* **do**
      (1) $G(i,j) = V(i-1,j-1) + W_{ij}$
      (2) $E(i,j) = \mathbf{max}[E(i,j-1), V(i,j-1) - W_g] - W_s$
      (3) $F(i,j) = \mathbf{max}[F(i-1,j), V(i,j-1) - W_g] - W_s$
      (4) $V(i,j) = \mathbf{max}[E(i,j), F(i,j), G(i,j), 0]$
end

IV. SOLUTIONS TO IMPROVE PERFORMANCE

Guided by the analysis provided thus far, this section suggests additional hardware and software mechanisms to improve the performance of these important workloads on the current POWER5 architecture.

A. Predicated Instructions

To improve the performance of these applications when the branch prediction accuracy is low, we consider the addition of

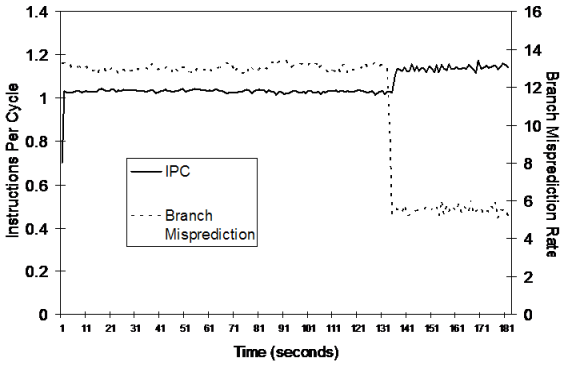

Fig. 2. Clustalw IPC and Branch Misprediction Rate

two instructions to the POWER ISA: integer select (*isel*) and maximum (*max*).

- Since the short conditional statements correspond to finding the maximum of two operands and storing it in one of the operands, we decided to implement a hypothetical *max* instruction. This instruction moves the larger value from two source registers to the target register in a single cycle. We selected an unused PowerPC primary and extended opcode combination as the new *max* instruction. The instruction format and execution resource requirement follow those of an *add* instruction: it has two sources and one target and is executed in the fixed point unit. Similar instructions are common in DSP processors, and are also included in the Altivec [19] velocity engine for vector data types.
- As an alternative to the *max* instruction, we also searched the current range of POWER processors for a branch predication instruction that could serve the same purpose. Note that no instruction equivalent to the x86 *cmov* instruction exists in PowerPC, except for the *isel* instruction included in POWER embedded cores. The *isel* instruction sets the target register to either of the two source register values based on a condition register bit value. The *isel* instruction needs a *cmp* instruction preceding it to set the associated condition register bit, and thus this form of predicated execution requires one more instruction than *max*. However, *isel* is a more general solution that may be applied in more situations than *max*. We implemented the *isel* instruction in our simulation framework to study the performance improvement it would provide.

In order to study the effect that the addition of the two instructions makes on these applications, we need a mechanism for inserting the instructions when appropriate into the code. In the code example shown in Algorithm III, the points to insert these are obvious, i.e., the *max* macro. Other sequences of code can also be easily identified by inspection, such as the following from ClustalW:

$if((hh = hh - g - h) > (f = f - h))$
$f = hh;$
which, in the case of the *max* instruction, becomes
$hh = hh - g - h;$
$f = f - h;$
$f = max(hh, f);$

We searched the four applications under study for such opportunities for either *max* or *isel* and manually inserted these assembly instructions. After analyzing the results (presented in Section VI), we were convinced there were other opportunities to take advantage of predicated instructions that could be recognized by implementing them in the framework of a compiler.

B. Compiler Support

We modified gcc's (version 4.1.1) code generator and the GNU binutils to emit *max* and *isel* instructions for the POWER architecture. Before gcc performs basic loop optimizations and register allocation, it runs an if-conversion pass that attempts to convert control flow hammocks — regions corresponding to if-then and if-then-else statements — into branchless regions. The if-conversion transformation simply identifies common code patterns that correspond to functions such as min, max, absolute value, the ternary operator, etc. Through pattern matching, the compiler identifies suitable regions and inserts complex instructions appropriately.

However, because the compiler must make conservative assumptions, many hammocks cannot be if-converted. Consider the following C example:

```
c = (a > b) ? A[i] : B[i];
```

Because the accesses to `A[i]` or `B[i]` could fault, the compiler cannot safely generate an *isel* instruction (which requires that both `A[i]` *and* `B[i]` be loaded regardless of the outcome of the conditional). To preserve the semantics of the programmer's statement in this case, the compiler must either be able to prove that the memory accesses will not fault, or it must leave the control flow hammock intact.

We improved the aggressiveness of gcc's basic if-conversion functionality by providing a utility that, in many cases, can determine whether a memory access is safe (at a given location

in the program). However, as we show in Section VI, gcc still misses several crucial opportunities that a programmer would likely catch. There are some cases where it is impossible for the compiler to statically prove that a hammock can be safely if converted:

```
if (x[i-1] > C)
    c = x[i];
```

In the above example, without knowing the range of `i` and the bounds of the array `x`, the compiler cannot assert that the *then* part of the statement can always safely execute. In addition, there are many cases where prior compiler passes sufficiently mangle the code such that legally if-convertible regions are obscured. Finally, memory aliasing can preclude generating *max* instructions. The compiler must ensure that memory operands used in *max* computations are not aliased — a task that is relatively easy for programmers, but often extremely difficult for a compiler.

C. Hardware Support

As previously mentioned, the *isel* instruction is merely a multiplexor that selects one of two operands based on the result of an accompanying compare instruction. The *max* instruction however, bundles both the comparison and the selection process into a single, one-cycle operation. Fundamentally, a *compare* instruction is an arithmetic subtract operation and the comparison result resides in the *carry out* bit. Thus, the *max* instruction can use a functional unit's existing carry generator unit to produce the control signal that selects the input operand with the maximum value. A single-cycle *max* instruction can be added to a processor's instruction set without increasing the critical path. Both the *isel* and *max* instruction are handled by one of the the processor's fixed point units.

D. Branch Target Address Cache

The two-cycle branch delay in the POWER5 is a performance bottleneck for Bioinformatics applications, which tend to be branch-intensive. This paper also investigates the ramifications of adding a small Branch Target Address Cache (BTAC) [20], which can eliminate branch delay in many instances.

A small BTAC (eight entries) suffices for the Bioinformatics workloads we consider. Each entry in the BTAC contains a tag (*tag*), a target address (*nia*), and a score field (*score*). The *tag* contains a subset of bits of the instruction fetch address. The *nia* is the predicted next instruction address. Since the target of a branch is usually not far away from the branch, we do not need to save the full 64-bit instruction address in this field. The *score* field is a saturating counter indicating the liklihood that *nia* prediction would be correct according to past prediction performance for the associated branch. Hard-to-predict branches will have low scores; the BTAC will forgo prediction for such branches because the penalty of misprediction is greater than the two-cycle branch delay.

When the BTAC receives an instruction fetch address (at the same time as it is sent to the L1 tag array), it searches for a matching entry. For each entrance to the block (first instruction to be executed within the block), it has an exit which is either the end of the block or a taken branch. If a program has a repeatable pattern, it is likely the same instruction sequence will be executed: if it enters a block at a given entrance, it is likely that it will exit the block at the same place as previous occurrences did. The result of the BTAC table lookup is known one cycle later. If the result is a tag match and the score of the matching entry is high enough, the *nia* of the matching entry will be used as the address for the next instruction fetch.

The BTAC update proceeds as follows. If the BTAC prediction is correct, the *score* of the matching entries in the BTAC table will be incremented. If the BTAC prediction is incorrect, the associated *score* value will be decremented. If there was no matching entry the BTAC, the table will allocate an entry and fill in the fetch address, the target address, and the initial score value (zero in the default configuration). The BTAC uses a score-based replacement policy.

V. Experimental Methodology

Our simulation framework is based on the SystemSim [21] PowerPC full system simulator. We configured our simulator to model a POWER5 system based on the configuration of the in-lab machine we presented statistics for in the previous sections. We only simulate one core of the POWER5's dual-core chip, and we do not make use of SMT for these experiments. We boot the Linux kernel (version 2.6.7), and then use a uniform sampling methodology to capture performance similar to the SMARTS system [22]. SystemSim provides an extremely fast "turbo" mode to skip to particular areas of interest, as well as fully cycle-accurate and warm-up modes. We make use of all of these in our studies.

In order to measure the performance improvement of potential extensions of predicated branches, we extended SystemSim's POWER5 model with the *max* and *isel* instructions, added a BTAC model, and varied the number of fixed-point units.

By replacing the conditional branches in the critical kernels for Clustalw, Fasta, Hmmer, and Blast with the inline assembly *max* and the *isel* instructions, we can gauge the performance impact of adding predicated instructions such as these to the POWER5 architecture. The *isel* instruction requires a preceding *cmp* instruction to set the appropriate condition register bit. The 32-bit PowerPC condition register is divided into eight fields, with four bits each. Three of these four bits can be set by a *cmp* instruction depending on the comparison result (less than, greater than, or equal to). The following *isel* instruction then reads the *greater than* bit value, and moves one of the two source register values to the target register, effectively implementing a comparison of two source registers and select of the larger value.

Compared with the inline assembly *max* instruction, the *isel* and the *cmp* instruction immediately preceding it could incur larger problems in scheduling other instructions out of

order, without proper compiler support. This is especially the case in the kernels of these applications, where the number of such conditional statements is very high compared with other arithmetic instructions. For example, in the *forward_pass* kernel of the Clustalw package, there are five such conditional statements of which three are consecutive. Due to this scheduling problem, we expected the increase in performance with the *isel*/*cmp* instruction to be lower than that with the *max* instruction.

We also experimented with eliminating the two-cycle taken branch delay present in POWER5. In the Section IV-D, we have explained the design of a branch target buffer that eliminates the two-cycle penalty for most taken branches. Variations in the performance of this structure due to differing design decisions are beyond the scope of this paper.

Finally, because of the large number of integer operations in these applications, we experimented with varying the numbers of fixed-point units in the core to quantify the performance impact. These additional fixed-point units share the same issue queue logic as those already present in the POWER5 core. The instructions we are proposing, *max* and *isel*, use the fixed point pipeline, and thus their inclusion will put additional pressure on these resources. Our experiments with additional fixed point units are designed to explore the impact of this pressure, and find a better number of fixed point units for use with Bioinformatics applications.

VI. Results

In this section, we examine the performance benefits on four Bioinformatics workloads of adding the *isel* and *max* predicated instructions, utilizing a small BTAC to remove the taken branch penalty, and increasing the number of fixed-point units of the current POWER5 architecture.

A. Predicated Instructions

Figure 3 shows the performance improvement observed when the *max* and *isel* instructions are used to ameliorate the effects of mispredicted branches on POWER5. We show results for both hand-inserted *isel* and *max* instructions, as well as for compiler-generated code.

Looking first at the hand-inserted method, we see that for all four applications, the *max* instruction performs better than the *isel* implementation. The *isel* instruction improves performance by an average of 29.8% over the baseline. The *max* instruction performs an average 34.8% better. As explained in Section III, the *isel* instruction requires a compare instruction prior to its use to set the condition register, a requirement not present with the use of the *max* instruction. In Clustalw, 9.3% of all instructions executed were either *isel* or *max*, but the number of *cmp* instructions jumped from 5.1% in the *max* case to over 14% in the *isel* case, increasing the path length of the code (although both are shorter than the original).

Clustalw and Fasta include the same kernels for pairwise sequence alignment, and we expect similar improvement. The input to Fasta, however is more than twice the length of the Clustalw input, and therefore the kernel takes up a larger

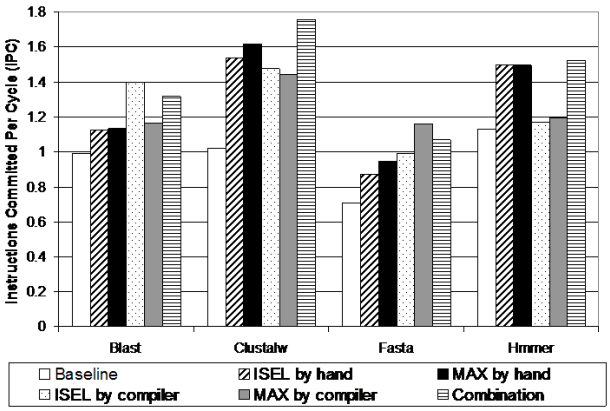

Fig. 3. IPC with *max* and *isel* Instructions

percentage of time of the overall Clustalw application. We get improvements of 50.7% with the *isel* instruction, and 58% with the *max* instruction for Clustalw. The results for Fasta are similar (23.1% and 34.2% for *isel* and *max*, respectively). Blast shows a smaller improvement due to the increased complexity of the code, resulting in less obvious places for hand-inserted instructions to make a difference in performance. Hmmer, sees an identical 32% improvement with both *isel* and *max*.

Looking at the performance of the compiler-inserted *isel* and *max* instructions, we see several interesting trends. For Blast and Fasta, the compiler-generated code for either *max* or *isel* outperforms both versions of the hand-inserted code, which is what we expect. This indicates there are opportunities in these codes for predicated instructions to help performance beyond those we were able to identify by inspection. However, for Clustalw and Hmmer, the hand-inserted code actually performs better than the compiler-generated code. While many of the if-statements are amenable to predication in these two applications, the heavy use of memory array references causes two main problems. First, the compiler attempts to hoist many of the load instructions, thereby obfuscating available *max* opportunities (*i.e.*, confuses the pattern matcher). Second, the compiler must be able to prove that there are no intervening aliased stores between a hoisted load and its use in the if-then-else statement. We are currently working to improve the compiler's performance for these two applications.

The bars labeled "Combination" in the graphs show the effects of combining the hand-inserted *max* instructions while using the modified gcc to additionally emit *isel* instructions where appropriate. We chose this combination because, on average, hand-identification of *max* instruction insertion points outperforms that of the compiler, but the compiler does a better job with the more flexible *isel* insertion. As shown in the graph, this combination performs best on Clustalw and Hmmer. In the case of Blast, the best performing option is using the compiler to insert *isel* instructions (indicating that there are other predicated opportunities than *max* functionality

but it requires a compiler to identify them). Again for Hmmer, the compiler is severely limited by the abundant array memory references.

Table II gives statistics related to branches for the four applications studied. Looking first at the percentage of instructions that are branches, we see that for all applications, the use of these two new instructions reduces the overall percentage of branches. Most application see a relatively uniform reduction, except in the case of compiler-inserted Hmmer because of the large number of memory array references in this code. The percentage of branches in Clustalw reduces by nearly half, as does the hand-inserted versions of Hmmer. For Fasta, the compiler is able to remove more branches from the code than we accomplished by hand-insertion.

The next column shows the change in the branch misprediction rate. In general, the branch misprediction rate goes down or remains unchanged (Blast, Fasta, and the hand-inserted versions of Hmmer). For the compiler version of Clustalw, however, the branch misprediction rate goes up. This simply indicates that while we were successful in reducing the overall number of branches, those that are left are difficult to predict. We plan on investigating how to better predict the remaining branches as part of our future work.

B. BTAC

In order to examine the effect the POWER5's two-cycle taken branch penalty has on application performance, we added a very small (8-entry) BTAC as described in Section IV. Figure 4 shows the performance improvement the BTAC makes when added to the baseline POWER5 architecture, and when added to the architecture enhanced with hand-inserted *max* instructions and compiler-inserted *isel* instructions. Table II shows the percentage of branches that are taken for all different options considered. The performance improvement is higher for the original POWER5 design (1.8% to 7.9% gain), with Fasta showing the largest performance improvement. The table in Figure 4 shows that the misprediction rate of the BTAC is quite low (1.4% to 2.5%), indicating that the choice of a BTAC with a small number of entries is acceptable for these applications.

C. Additional Fixed-Point Units

Figure 5 depicts the performance of the Bioinformatics workloads when the number of fixed-point units (FXU) is varied. The first two bars show the performance gain when two additional FXUs are added to the POWER5 architecture. Fasta and Blast show modest gains in performance, but probably not worth the cost of the additional functional units. Clustalw shows slightly more improvement, but Hmmer benefits greatly from these additional fixed point units. The next three sets of bars show the performance benefit to the "Combination" configuration of Figure 3 when a total of three and four FXUs are used. Again, Fasta shows little improvement and Hmmer shows a large improvement, although moving from three to four does not benefit either application significantly. Blast's performance improves nearly linearly with additional

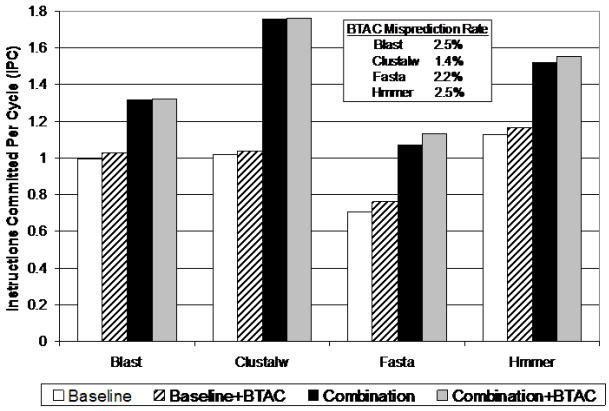

Fig. 4. Effect of Adding an Eight-Entry BTAC

functional units. Finally, Clustalw can make good use of an additional fixed point unit, but the improvement from three to four is minimal.

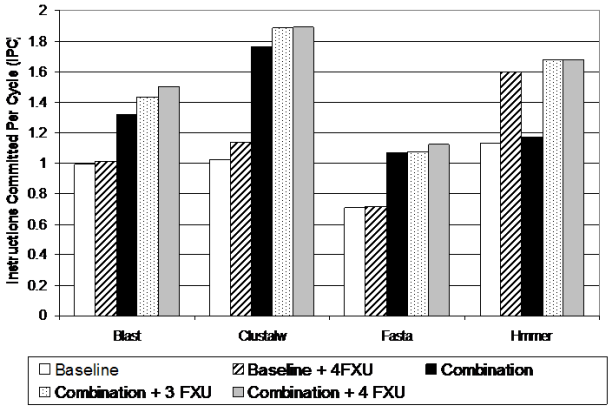

Fig. 5. Effect of Additional Fixed Point Units

D. Combined Gains

Finally, we look at the performance gains when all three enhancements are used. In Figure 6 the cumulative performance previously presented is shown for each individual change. What is shown here is the baseline IPC observed on the POWER5, then changes to the IPC observed for adding predication, adding the BTAC, and adding two additional fixed-point units. The fifth category shows the "residual" performance, which is the difference in the performance we observed by simulating all improvements at once versus adding up the deltas observed for each one individually. Thus, the "residual" category shows how the combination of these improvements actually improves overall performance more than the sum of each individual one due to the close interaction with the methods we are examining. In terms of final performance, Clustalw performs the best. The IPC for this application nearly doubles, from 1.02 to 1.93. Blast and Hmmer show a 53% and

Application			Percent Branches/Instrs	Branch Mispredict Rate	Percent Taken Brs/Branches
Blast	*isel*	hand	15.3%	5.7%	65.7%
		comp.	12.9%	4.2%	52.3%
	max	hand	16.2%	5.9%	65.1%
		comp.	14.4%	5.6%	66.0%
	Original		20.7%	6.1%	67.4%
Clustalw	*isel*	hand	7.4%	2.6%	85.5%
		comp.	7.2%	8.0%	85.2%
	max	hand	8.1%	2.7%	84.5%
		comp.	8.9%	7.0%	82.6%
	Original		14.6%	5.7%	69.6%
Fasta	*isel*	hand	23.2%	7.8%	75.6%
		comp.	19.2%	7.9%	74.2%
	max	hand	22.3%	7.5%	73.6%
		comp.	18.0%	7.4%	76.2%
	Original		25.9%	7.9%	69.0%
Hmmer	*isel*	hand	7.9%	4.4%	62.6%
		comp.	12.0%	6.2%	71.3%
	max	hand	8.3%	4.7%	63.2%
		comp.	11.7%	6.1%	65.2%
	Original		13.8%	5.7%	71.7%

TABLE II

BRANCH PERFORMANCE OF APPLICATIONS WITH PREDICATED INSTRUCTIONS ADDED

51% gain, and Fasta improves by 69%. All applications, with the exception of Fasta, show observably higher residual gains from the combination of these improvements.

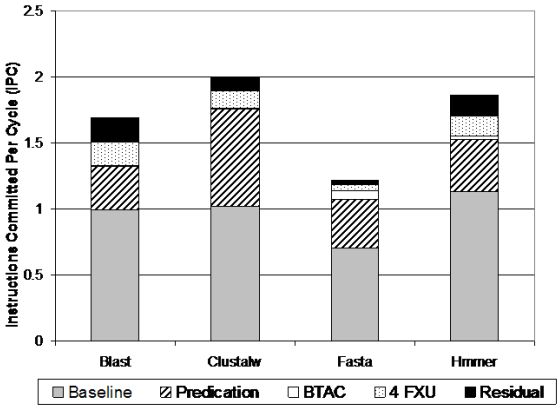

Fig. 6. Effect on IPC of Combining Predicated Instructions, BTAC, and Four Fixed-point Units to POWER5

VII. RELATED WORK

Several papers have discussed the performance analysis and optimization of Bioinformatics applications. Ratanaworabhan and Burtscher analyze the effects of speculative loads on several BioPerf applications in [23]. They show how manual hoisting of loads in the source code can mitigate the effects of branch mispredictions. In contrast, our approach aims to remove hard-to-predict branches altogether.

In [24], the hotspots of Hmmer are analyzed and optimized. Three approaches are presented in the work. The first approach analyzes the source code and removes unnecessary variables, conditions, and loop dependencies. The second approach targets array aliasing and *cmov* instructions. The third approach involves SSE2 extensions and MPI implementation. We also found the *cmov*-like instructions are important in improving the performance of Bioinformatics applications and provided further implementations and analysis.

Vectorizing Hmmer and Blast on the PowerPC 970 servers is explored in [25]. The authors suggest that supporting operation widths of 21 or 25 bits may further improve the Hmmer performance. For Blast, enhanced bandwidth using AltiVec is useful in that fewer instructions and register accesses are incurred.

The last level cache performance of a CMP running parallel Bioinformatics workloads is studied in [26]. Multiple private caches or a single shared cache configurations are compared while running different parallel Bioinformatics applications. Due to the high degree of data sharing for these applications, a single shared last-level cache requires significantly lower bandwidth than private last-level caches.

VIII. CONCLUSIONS AND FUTURE WORK

In this work, we have shown the importance of branch predication for several important sequence alignment workloads in computational biology. We improved the instructions per cycle achieved over that of a conventional POWER5 processor by an average of 64% for four important Bioinformatics workloads. This was accomplished through the use of predicated instructions, increased numbers of fixed point units, and the introduction of a branch target address cache to eliminate the two-cycle taken branch penalty currently found in the POWER5 architecture.

These results can be extended to gene finding software *Glimmer*, protein prediction software *Predator* and the phy-

logeny reconstruction application *Phylip*, other applications that are part of the BioPerf suite. Furthermore, efforts are underway to improve the compiler support first presented here to identify more opportunities for predicated instruction use. Finally, we are examining remaining bottlenecks in the processor core for these applications as we believe they represent an important class of emerging workloads.

ACKNOWLEDGMENT

This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004.

REFERENCES

[1] E. Anson and E. Myers, "Algorithms for whole genome shotgun sequencing," in *Proceedings of the 3rd Annual International Conference on Computational Molecular Biology (RECOMB99)*. Lyon, France: ACM, Apr. 1999.

[2] J. Venter and *et al.*, "The sequence of the human genome," *Science*, vol. 291, no. 5507, pp. 1304–1351, 2001.

[3] J. Weber and E. Myers, "Human whole-genome shotgun sequencing," *Genome Research*, vol. 7, no. 5, pp. 401–409, 1997.

[4] M. Schena, D. Shalon, R. Davis, and P. Brown, "Quantitative monitoring of gene expression patterns with a complementary DNA microarray," *Science*, vol. 270, no. 5235, pp. 467–470, 1995.

[5] D. A. Bader, Y. Li, T. Li, and V. Sachdeva, "BioPerf: A benchmark suite to evaluate high-performance computer architecture on bioinformatics applications," in *Proceedings of the IEEE International Symposium on Workload Characterization*, Austin, TX, Oct. 2005.

[6] D. Bader and V. Sachdeva, "An open benchmark suite for evaluating computer architecture on bioinformatics and life science applications," in *Proceedings of the SPEC Benchmark Workshop 2006*, Austin, TX, Jan. 2006.

[7] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs," *Nucleic Acids Research*, vol. 25, no. 17, pp. 3389–3402, 1997.

[8] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, pp. 195–197, 1981.

[9] A. J. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE Transactions on Information Theory*, vol. IT-13, pp. 260–269, 1967.

[10] J. D. Thompson, D. G. Higgins, and T. J. Gibson, "CLUSTALW: improving the senstivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice," *Nucleic Acids Research*, vol. 22, pp. 4673–4680, 1994.

[11] W. R. Pearson and D. J. Lipman, "Improved tools for biological sequence comparison," *Proceedings of the National Academy of Sciences USA*, vol. 85, pp. 2444–2448, 1988.

[12] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge, UK: Cambridge University Press, 1998.

[13] "http://www.redbooks.ibm.com/redpapers/pdfs/redp1965.pdf."

[14] SPEC Benchmarks, "The Standard Performance Evaluation Corporation," 1997, www.specbench.org.

[15] Y. Li, T. Li, T. Kahveci, and J. Fortes, "Workload characterization of bioinformatics applications on Pentium 4 architecture," in *Proceedings of the International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Atlanta, GA, 2005.

[16] J. M. Pipas and J. E. McMahon, "Method for predicting RNA secondary structure," *Proceedings of the National Academy of Sciences*, vol. 72, pp. 2017–2021, 1975.

[17] D. Sankoff, R. J. Cedergren, J. B. Kruskal, and S. Mainville, "Fast algorithms to predict DNA secondary structures containing multiple loops," *Time Warps, String Edits and Macromolecles. The Theory and Practice of Sequence Comparison*, pp. 93–120, 1983.

[18] D. Sankoff and P. Rousseau, "Locating the vertices of a steiner tree in an arbitrary metric space," *Mathematical Programming*, vol. 9, pp. 240–246, 1975.

[19] Apple Computer, Inc., *PowerPC G5: White Paper*, Jun 2004.

[20] S. Duvvuru and S. Arya, "Evaluation of a branch target address cache," *HICSS '95: Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)*, 2005.

[21] J. L. Peterson, P. J. Bohrer, L. Chen, E. N. Elnozahy, A. Gheith, R. H. Jewell, M. D. Kistler, T. R. Maeurer, S. A. Malone, D. B. Murrell, N. Needel, K. Rajamani, M. A. Rinaldi, R. O. Simpson, K. Sudeep, and L. Zhang, "Application of full-system simulation in exploratory system design and development," *IBM Journal of Research and Development*, vol. 50, no. 2/3, 2006.

[22] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe, "Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling," *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

[23] P. Ratanaworabhan and M. Burtscher, "Load instruction characterization and acceleration of the BioPerf programs," in *Proceedings of the IEEE International Symposium on Workload Characterization*, San Jose, CA, Oct. 2006.

[24] J. Landman, J. Ray, and J. Walters, "Accelerating hmmer searches on opteron processors with minimally invasive recoding," *Proceedings of the 20th International Conference on Advanced Information Networking and Applications*, vol. 2, pp. 628–636, 2006.

[25] D. Citron, H. Inoue, T. Moriyama, and M. Kawahito, "Exploiting the altivec unit for commercial applications," *Proceedings of the Ninth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-9)*, 2006.

[26] A. Jaleel, M. Mattina, and B. Jacob, "Last level cache (LLC) performance of data mining workloads on a CMP – a case study of parallel bioinformatics workloads," *Proceedings of the 12th International Symposium on High Performance Computer Architecture*, 2006.