2008

A Comparative Xeon and CBE Performance Analysis

Randy Fort
San Jose State University

A Comparative Xeon and CBE Performance Analysis

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Randy Fort

Spring 2008

ABSTRACT

The Cell Broadband Engine is a high performance multicore processor with superb performance on certain types of problems. However, it does not perform as well running other algorithms, particularly those with heavy branching. The Intel Xeon processor is a high performance superscalar processor. It utilizes a high clock speed and deep pipelines to help it achieve superior performance. But deep pipelines can perform poorly with frequent memory accesses. This paper is a study and attempt at quantifying the types of programmatic structures that are more suitable to a particular architecture. It focuses on the issues of pipelines, memory access and branching on these two microprocessor architectures.

ACKNOWLEDGEMENT

During my research and writing, I came across numerous delays, due to a heavy work schedule. I have had previous experience in CS247 of working with Dr. Chun, and observing how he goes to great lengths to accommodate students who work for a living while working on their Masters Degree. During this process Dr. Chun has been incredibly accommodating and flexible with the numerous delays I have encountered. It is highly likely that I would not have finished without his help. Likewise Dr. Kim and Dr. Stamp have been more than kind in granting an extension, and being incredibly flexible. I deeply appreciate how hard the committee works in accommodating working students.

Table of Contents

Table of Figures

1 Introduction

1.1 Overview and Abstract

The CBE (Cell Broadband Engine) is a high performance multi-core processor. It was designed by IBM, Sony, and Toshiba [9]. These companies recognized the need to build power-efficient high-performance microprocessors not only for gaming, but also for a wide variety of scientific and consumer applications. The CBE consists of nine processing cores on a single chip. The main processor is called the PPE (PowerPC Processing Element). As the name implies, it is a 64-bit Power PC based processor. It is based on the PPC 970 with vector/SIMD (Single Instruction Multiple Data) extensions [2]. The main processing core works with eight 32-bit SPE (Synergistic Processing Element) processors [2].

The CBE (as a whole) employs SIMD architecture, rather than the more common pipeline and superscalar designs used in the Intel Pentium family, Sparc designs, and many other processors. The CBE can provide impressive performance increases for certain classes of problems. For example, FFTs (Fast Fourier Transforms) run on a CBE can exhibit performance increases up to 30 times faster than a comparable 64-bit Intel Xeon processor [3].

This paper will compare and contrast the SIMD architecture of the CBE with the Xeon architecture. The intent of this research is to illustrate which structures in code would be better suited to a CBE SIMD approach, or a deeply

pipelined superscalar Xeon architecture. Specifically, this paper will examine memory access performance and branching. Two areas are discussed. First, the memory access performance of the Xeon vs. the DMA approach of the CBE will be compared. Then, branch performance and penalties of the CBE will be examined and compared with the results of the speculative branching performance of the Xeon. The minimal support provided by the CBE SPEs for branch prediction will be compared to the approach and performance of the Xeon, which has hardware support for speculative branching, and performs much better on branch-laden control-intensive code.

1.2 Selection of the Best Processor for the Job

What is the best approach for a system engineer to evaluate candidate processors when characteristics of the problem are known? There are a byzantine number of processors with varying pipeline lengths, caches, ILP and SIMD approaches, including x86, Motorola 68000 family, PPC and MIPS. These choices are further complicated by the ulterior motives of companies in providing deceptive (or at best misleading) benchmarks. In this environment, making an informed decision is difficult at best. These choices are rapidly becoming even more complex with multicore to multicore comparisons. For example, consider a 4 core Xeon might need to be compared with a 16 core MIPS64 chip such as those made by Cavium [20].

It is well documented in [3] that many algorithms such as matrix multiplications and FFTs benefit greatly from the architecture of the CBE. The

authors note performance gains of up to 30x over the Itanium [3]. Likewise, many problems are well suited to the high speed, deep pipeline of the Xeon. The best solution, if cost were not a factor, is for engineers to perform their own comparisons with their particular problem.

But cost is a factor, and a huge one. And one of the main problems a system engineer faces is that the CBE is especially difficult to evaluate. You can't simply recompile your test and run it. Running on the CBE can involve a time consuming port of code. The algorithm must be suited to parallel processing. In addition to porting the logic and algorithm, there is a steep learning curve to learn the set of intrinsics, programming models, DMA and signaling mechanisms.

For example, consider just a couple of the complications that have arisen in the course of my study. First, there are two distinctly different cores on the chip, 1 64 bit PPE, and 8 32 bit SPEs, both of which have separate compilers and word sizes. Utilizing the vector intrinsics on the CBE to achieve maximum performance in the SIMD architecture requires a significant investment in time to surmount the learning curve. Also, different word sizes on the PPE and the SPEs needs to be taken into account. The SPEs are 32 bit processors and the PPE is a 64 bit processor. This can cause many headaches if the programmer is not careful 32 bit integers unexpectedly roll over their maximum value. Also, the single precision floating point support of the SPE may be insufficient for many computational needs [9]. Furthermore, if code is not well suited to parallel

processing, the 8 SPEs are unlikely to help. Thus, many times, the cost of a port to the CBE just to evaluate feasibility would be prohibitively expensive.

This study is a first step toward developing characteristic code which clearly demonstrates the known characteristic strong and weak points of the processors. Hopefully this set of example code will grow over time by further work in the field. These are not “pure” problems, but problems which provide code structures likely to be found in real applications. That is, they do not illustrate the strength of a particular architecture and exclude its weaknesses for the purpose of obtaining the best possible performance. This allows a researcher to look for a structure that most closely matches their own problem space, as opposed to the “pure” problems (often touted by marketing departments) that are extremely suitable for one processor or the other.

For example, consider a hypothetical problem which appears to be well suited to the CBE. That is, a small streaming algorithm with high data throughput, and low memory latency requirements. If candidate code was found to have more branching and poorer steady state branch performance than the branching code I set forth, the CBE may well be eliminated as a candidate processor without expensive testing. This code could illuminate and quantify the heavy branch misprediction penalties for a researcher. This is information not highly emphasized IBM.

The selection of the correct architecture for “pure” problems is straightforward. If a problem domain has branch intensive code, and the

algorithm's "steady state prediction behavior [1]" is relatively consistent, then a pipelined super scalar is the best choice. If you have an algorithm that streams continuous amounts of data through a relatively small amount of code, possibly coupled with low memory latency requirements and minimal (or predictable) branching, the CBE, with its high bandwidth EIB (element interconnect bus) data ring and low latency DMA would be a good choice. Real world problems are seldom that straightforward. Users often need to evaluate problems with both characteristics.

1.3 The Roadmap of this Study

In order to examine and compare the impacts of memory access and branching frequencies on pipelined and SIMD architectures, I will briefly mention several areas of study which are necessary to illustrate performance results of these architectures. I will briefly review the classic 5 stage pipeline, and its characteristics as expounded by Hennessey and Patterson. Although, I do not have access to Itanium hardware for comparison, it is a useful study to compare it to other pipelined processors. The issues of static vs. dynamic issue give them different advantages. Then I will mention AMDs and Intel's different approaches to increasing pipelined performance. I will also consider cache performance to document how they can affect performance.

This groundwork is necessary in order to fully expound comparisons between the CBE and pipelined processors. For example, much of the efficiency of the CBE is due to what it does not do. The SPEs have a very short simple

pipeline. This is significant when compared to the Xeon, which brings all of the complications that Hennessey and Patterson illustrate on their 5 stage pipeline; amplified by the fact the Xeon has a 31 stage pipeline [4].

I will discuss the CBE architecture in much greater detail than the Xeon. This is necessary since it is probably more unfamiliar to computer architects and students than the more well known pipelined superscalar architectures. This will include a study of how the CBE utilizes separate optimization of the control and data planes. This hybrid aspect of the CBE architecture sets it apart from SIMD architectures of the past, and is in large part responsible for its success. The EIB (Element Interconnect Bus) and MFC (Memory Flow Controller) will be explained in order to show their influence on programming approaches. Then the design considerations of the DMA test cases on the CBE should be clear.

I will then explain the vector intrinsics in the CBE to provide an understanding of the superior performance of the SIMD approach on best case and worst-case problems. I will present code examples and an analysis of the performance results. Finally, I will discuss the hard lessons I learned. That is, the surprising things I learned which were not intended to be a part of my research topic.

2 Xeon Architecture

2.1 Power

Power has always been a consideration in microprocessor design. With processors having longer pipelines and higher clock speeds necessary to support them, it is becoming even more critical. The power consideration exists beyond the thermal considerations of the chip itself. Increasing demands on data center power are also compounded by the fact that once the data center is powered, all that dissipated heat must be cooled by larger cooling systems.

In examining the Xeon, and other processors which provide significant support for operations in silicon, it is important to remember that increasing silicon complexity directly results in increased power consumption [1]. The performance increases afforded by direct support may well be worth the cost, both in dollars and MIPS per watt. But it is good to be mindful of the significant complexities in hardware support many of these techniques require. This is especially true of processors with deep pipelines.

2.2 Pipelining

The Intel Xeon uses pipelining and super scalar techniques to achieve high performance. This allows the Xeon, and other similar processors to execute multiple instructions at multiple pipelined stages. As users demand greater performance from microprocessors, two schools of thought have emerged regarding pipeline length. One emphasizes a shorter pipeline with higher

efficiency, and contains more execution units running at a lower clock rate [4]. The other emphasizes higher clock speed supported by a longer pipeline. AMD has placed substantially more emphasis on the former, while Intel has historically favored the latter.

The AMD Opteron is notable for its different pipeline approach. It utilizes a shorter pipeline of only 12 stages [4]. In many cases, the Opteron can achieve equivalent or superior performance than the Xeon at much lower clock rates [4]. As always, the specific performance depends on the program under test. With the shorter pipeline, the AMD can provide memory latency improvements over the Xeon in the 10-40% range [4]. It is highly likely that the shorter Opteron pipeline, which has lower memory latency, could have very different results than a Xeon for a given test, and may be an alternative choice if the CBE is not suitable.

For example, if the CBEs low memory latency was highly desired for a particular application, but excessive branching eliminated it from contention, the AMD could provide a better solution than the Xeon. The lower memory latency of the Opteron could meet requirements, but still have the necessary hardware support for speculative branching and thus provide superior performance.

As Intel continued to push clock speeds higher, they required increasingly deeper pipelines to keep the processor busy. Intel decided to make the trade-off and sacrifice a more efficient pipeline for speed. The single core Xeon is one of the most extreme examples of this approach, running a 31-stage pipeline at up to

3.8 GHz [4]. Higher clock speeds can also cause other design complications. One of the complications of this approach is that power consumption increases more than linearly with clock speed. Thus, heat dissipation can become the limiting performance factor [9].

Increased clock speeds, and the deeper pipelines needed to support them have had significant impact on memory latency. These long pipelines have latencies approaching 1000 cycles [9]. Applications which have frequent memory accesses can perform poorly on these architectures. Applications of this sort could benefit from shorter pipelines, but could possibly reap much larger performance benefits form the CBEs SIMD architecture and DMA approach [9].

2.2.1 Clockspeed and Marketing

As could be expected, Intel and AMD endlessly squabble over which benchmark is better to illustrate chip performance. Intel invariably shows benchmarks which derive maximum benefit from clock speed, and AMD chooses tests which highlight its more efficient pipeline. Up until recently, Intel had apparently won the battle of the marketing message. So successful has been their campaign that the average user equates clock speed with performance. The money involved in this market virtually ensures that the facts will be distorted by creative marketing. I devoted a section to these issues in the Performance Analysis section.

It is interesting to note that the clock speed mantra may be coming back to haunt Intel. Now that the clock speed seems to have hit a practical wall in the 3.8 GHz range, multicore processors running at lower speeds become much more attractive both in terms of computational speed and power consumption. Thus Intel must back track on its long standing marketing message. Old timers in academia and industry find it highly ironic when Intel puts out a paper entitled "Don't Judge a CPU only by its GHz [21]".

2.3 Multiple Issue

A processor which can issue multiple instructions in a clock cycle is called superscalar [1]. Multiple-issue processors have two basic forms, superscalar and VLIW (very long instruction word) [1]. Superscalar processors are either statically scheduled (using in order execution) or dynamically scheduled (which can use out of order execution). Out of order execution is constrained by data hazards which I will briefly discuss later. VLIW designs are always static, the order of the instructions are determined at compile time. Thus, the quality of the compiler in analyzing the code for performance is of paramount importance in VLIW designs [1].

> "Fallacy: There is a simple approach to multiple-issue processors that yields high performance with out a significant investment in silicon area or design complexity" [1].

Although the superscalar design can have huge performance benefit, in the case of dynamic processors it requires significant hardware support. As I mentioned in section 2.1, this also plays a role in power consumption. Furthermore, the number of instructions issued at once can also have significant impacts on the complexities of pipeline hazards. I will address this further in the branching and hazards section. It also makes speculative branch prediction even more important, since mispredicted branches in a long pipeline can carry a very heavy penalty, as I will discuss further in the next section [1].

2.4 Branching

2.4.1 Frequency

> "For typical MIPS programs the average dynamic branching frequency is often between 15% and 25%, meaning that between four and seven instruction execute between a pair of branches" [1].

The number of branches in a program becomes extremely important when evaluating a processor with speculative hardware support (such as the Xeon) vs. one that has very little (such as the SPEs on a CBE). With hardware support, if the steady state behavior of a branch is relatively stable, the branch prediction will ensure the correct instructions are fetched with little branch penalty.

But on a CBE, a mispredicted branch will incur an 18 cycle penalty [9]. So while the Xeon can suffer little or no penalty with good steady state algorithm behavior regardless of which branch is taken, the CBE can not. This single fact

alone can completely remove the CBE from contention in a processor evaluation. The CBE does have a branch hint mechanism, but it is static. Branch hints can not change during program execution.

This 18 cycle penalty can potentially result in significant performance degradation considering the nature of how a SIMD processor is used. Algorithms are often implemented as tight loops when processing streaming data, and when mispredicted branches exist in these loops, they are executed many times. So the number of mispredicted branches encountered can be quite large, each incurring an 18 cycle clock penalty. The impact poses enough of a concern such that one of the preferred techniques in programming the CBE is to actually eliminate a branch, execute both clauses, and return all results to the PPE [2].

"Fallacy: There is such a thing as a typical program" [1].

Although Hennessey and Patterson stated the above quote in regards to ISAs (Instruction Set Architectures), it does state a good general principle. Namely, it is very difficult to devise code that represents a typical program, let alone a large set of applications. In [1], they provide an excellent discussion of the complications a branch can impose on a pipeline.

> "In the examples we have considered so far it has been possible to resolve a branch before having to speculate on another. Three different situations can benefit from speculation on multiple branches simultaneously: a very high branch frequency, significant clustering of branches and long delays in functional units" [1].

Consider a superscalar architecture that can issue multiple instructions. In theory, the more instructions issued at once, the higher the performance. But the more instructions issued at once also increases the probability that that one or more will be a branch. And the more branches that exist in the pipelines, the greater the probability that pipeline hazards will stall the pipeline [1]. Thus, the hardware support required to resolve hazards becomes even more important. But that support is not without complexity in silicon and hence power consumption. Hennessey and Patterson repeatedly emphasize the silicon investment required for multiple issue processors [1].

2.4.2 Speculation

> "As we try to exploit more instruction-level parallelism, maintaining control dependencies becomes an increasing burden. Branch prediction reduces the direct stalls attributable to branches, but for a processor executing multiple instructions per clock, just predicting branches accurately may not be sufficient to generate the desired amount of instruction-level parallelism. A wide issue processor may need to execute a branch every clock cycle to maintain maximum performance" [1].

The Xeon provides significant support for speculative branch prediction which has proven to be very effective [12]. In control and branch intensive code such as operating systems, this can provide a significant increase in throughput. Without highly effective branch prediction, it is likely that the hazards of pipelining

and multiple issue would impose a performance penalty instead of a gain. This is especially true with a 31 stage pipe line like the Xeon.

The best way of keeping a pipeline from stalling is to predict how the branch will behave by keeping a history based on what it has done previously in execution. This is known as speculative execution [12]. If this speculation is wrong, then the pipeline may stall or have to be flushed. But if correct, the execution can proceed without penalty, resulting in excellent pipeline efficiency. The simple branch predictor shown below, when implemented in hardware, can negate branch penalties if the algorithm exhibits a steady state behavior. That is, if it takes the same branch more often than random.

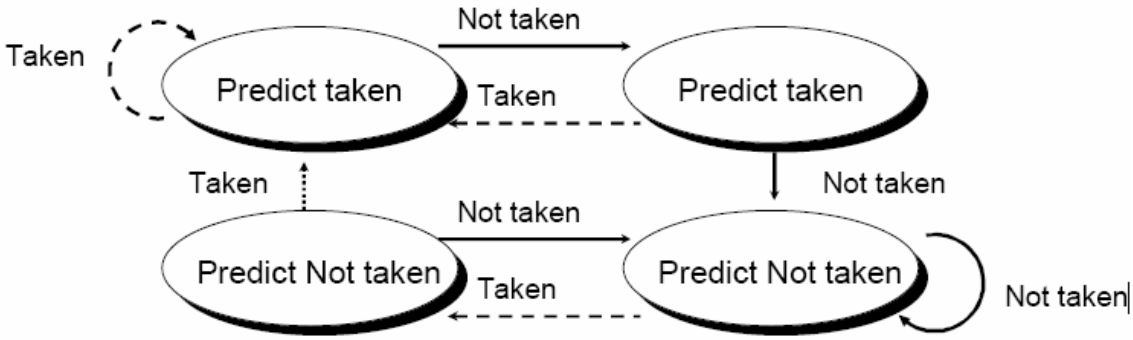


Figure 2-1 2 Bit Prediction Scheme [1]

The actual prediction scheme used in later processors of the Pentium family is called “two level branch prediction [12]”. It is more effective, and slightly

more complicated. But the principle is the same: provide hardware support for better branching prediction.

2.5 Hazards

As microprocessor designers increase performance below one CPI (cycles per instruction), they must resort to new and more sophisticated techniques. In order to overlap instructions and execute more than one instruction per clock cycle [1], the complications of data hazards must be minimized. Also, according to [1], there are only 4-7 instructions between branches in a typical block of MIPs code. Although this refers only to MIPS code, it is fairly safe to assume that other architectures are not significantly different. So for effective ILP (instruction level parallelism), hazard resolution often needs to extend across multiple blocks [1].

Although [1] explains in detail the necessary principles for understanding pipeline hazards, a short review is in order. There are three basic pipeline hazards: structural, data and control. We have been and will be looking at control (branching) in the CBE section. Data hazards, RAW (Read after Write), WAW (Write after Write), and (Write after Write) can be a consequence of out-of-order execution.

> "Structural hazards arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped exaction" [3].

This will cause a stall in the pipeline, and the pipeline can not proceed at its ideal throughput of 1 CPI (Cycle per Instruction) in that case of non-superscalar processors. Super scalar processors capable of greater than 1 CPI will not reach their full potential if these hazards stall the pipeline.

2.5.1 Itanium ILP – a Brief Word

There are two different ways of implementing ILP. One depends on hardware to look for ILP parallelism, and the other relies on software [1]. These are called and dynamic and static respectively. In the static approach, utilized by VLIW processors such as the Itanium, the compiler is responsible for resolving the hazards, and determining what instructions are issued. In the dynamic approach, there must be hardware support for data hazards. This requires a significant investment in silicon [1]. The Itanium is a processor that has so far not lived up to its expectations, but still has great promise for the future. Current versions of the Itanium have shown superior performance on floating point operations [4].

2.6 Consideration of Cache Capacity

Often, many of the drawbacks of a particular architecture can be mitigated with some clever ideas. We will see many of these in the CBE. In the Xeon, the drawbacks of the very long 31 stage pipeline need to be minimized. In many cases, the problems of pipeline inefficiencies can be mitigated with caches. The Intel Xeon I employed in my testing has a 2MB cache and a 31 stage pipeline.

The latest versions on the Itanium have large 24 MB cache and an 8 stage pipeline. The Opteron has a 12 stage pipelines and the family has numerous cache sizes available [4].

Random memory access, which can cause pipeline bubbles which require a pipeline to be flushed, can have a huge impact on performance. When comparing problems of this type to an Opteron, with its shorter 12-stage pipeline, the Xeon may not be the best choice for a particular problem [4]. To show this, one of my tests is crafted to ensure that that Xeon cache can not mitigate the memory access latencies. I will document test results showing different memory access block sizes. This will illustrate types of algorithms which benefit from the caches and those that do not.

3 A Description of the CBE Architecture

"SIMD Computers: Several Attempts, No Lasting Successes [1]."

3.1 CBE – A New Application of the SIMD Approach

As Hennessey and Patterson note in [1], there are no real success stories for the SIMD architecture. However, the story of the CBE is more complicated than that for a new iteration of a SIMD design. The SIMD concept has existed essentially since 1958 [1]. What the new CBE design entailed was a multicore hybrid architecture with 9 processing cores. The goal was to have a conventional processor optimize the control plane, and the SIMD processors process data, thus combining the best aspects of each. The CBE architecture is a radical departure from traditional processor designs. The CBE implements many good old ideas, such as pipelining (on the PPE) and SIMD processors. It then combines them in a new way on a multicore processor. Then it adds some very clever new ideas, such as the super high bandwidth Element Interconnect Bus (EIB), 8 processing cores, and an asynchronous Memory Flow Controller (MFC) with some ingenious new programming models. This results in what is arguably the first successful SIMD implementation.

In contrast to the Xeon which exploits parallelism in the instruction stream, the CBE, as a whole, seeks to exploit parallelism in the data stream. This is done by using 8 SPE SIMD processors that can divide the work on the data stream. These are coupled with a rich set of vector intrinsics that allow each of

the 8 SPE processors to operate on multiple words at once. High speed DMA support is provided via the EIB to allow the SPEs to DMA memory between themselves and main memory [9].

3.2 SIMD Vectors

The heart of the SIMD concept is to operate on multiple data elements at one time. These are called vectors. Vector and SIMD extensions are supported by both the PPE and the SPEs [2, 9].

> "A vector is an instruction operand containing a set of data elements packed into a one-dimensional array. The elements can be fixed-point or floating-point values. Most Vector/SIMD Multimedia Extension and SPU instructions operate on vector operands. Vectors are also called Single-Instruction, Multiple-Data (SIMD) operands, or packed operands" [9].

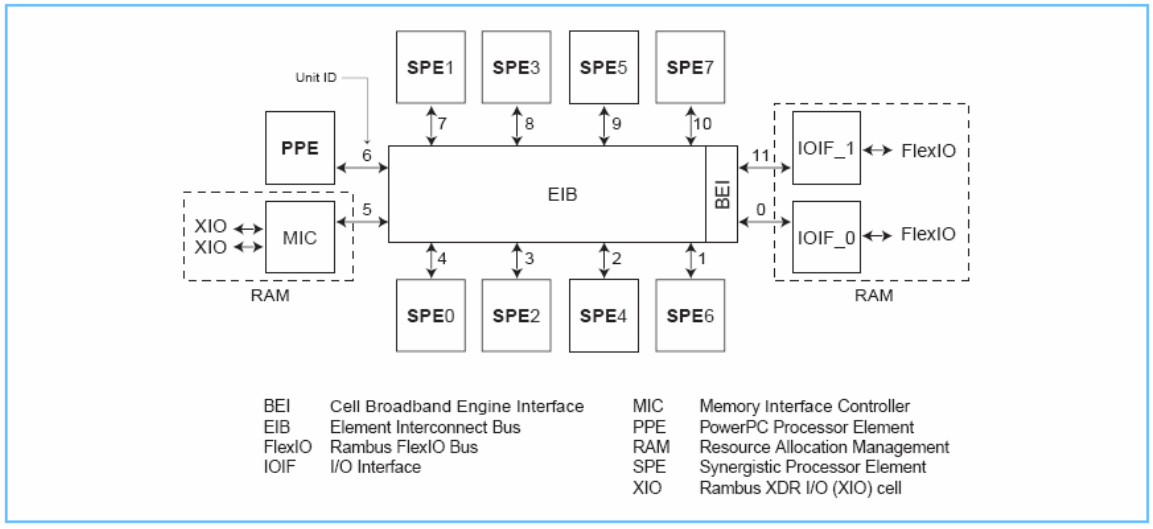


Figure 3-1 CBE Diagram [2]

3.3 Power, Memory and Frequency

The designers of the CBE saw three basic problems that they wanted to solve: power, memory and frequency [9]. “Power dissipation has become the limiting factor of processor performance [9]”. In IBMs view, heat dissipation is the limiting factor to increasing microprocessor performance. IBM contends in [9] that additional hardware resources in silicon could not bring proportional gains in performance unless power efficiency was improved at the same rate. The elegant solution in solving the power problem was to separately optimize the control and data planes by having a multicore processor with two types of processors: a single conventional pipelined processor for the control and 8 SIMD processing elements for the heavy computational data tasks.

Another limiting factor that IBM states is that a large amount of time is spent moving data from memory. Long pipelines in high speed processors have latencies approaching 1000 cycles [9]. This is true even in processors with integrated memory controllers, such as AMD [4, 9]. The CBE mitigates this overhead by using a three level memory model consisting of main storage (on the PPE), and local storage along with large register files (which reside on the SPEs). Movement of data from main memory is supported by high speed DMA. This DMA allows the CBE to eliminate the long memory latency that deep pipelines cause [2, 9]. Since the SPEs can access memory directly and asynchronously as explained later in this section, these latencies are drastically reduced.

Since the PPE performs operating system tasks and acts as a top level thread and coordinating resource for the SPEs, the SPEs are free to focus on the computational tasks. The SPEs were designed with simplicity and performance in mind. Since the SPEs do not have a long pipeline, and have direct access to main memory via DMA, they are free to operate at higher frequency.

3.4 Separate Optimization – The PPE

The PowerPC core has a traditional pipelined architecture. Like the Pentium family, it also supports two simultaneous threads of execution. Its primary duty is to run the operating system. It is also intended to act as a management processor for the computational task. For example, it may calculate and divide up SIMD tasks between the SPEs, and perform synchronization issues when needed, among other things [9]. Although the PPE comes from the 970 family, it has exhibited surprisingly low performance in one of the test cases reported in this paper.

The PPE is a general purpose RISC (Reduced Instruction Set Computer) processor. It has a conventional pipeline of 23 stages, and handles control intensive branch laden code, such as the operating system [13]. That is a task that pipelined architectures can do well. It is effectively the controller for the CBE. The PPE can run 32 and 64 bit code. Since the PPE is based on the PowerPC 970, almost all PowerPC 970 code will run on it without modification [9]. The PPE can use DMA, mailboxes, and signal notification registers to move and synchronize data with the SPEs.

3.5 Separate Optimization – The SPE

The 8 SPEs represent the SIMD portion of the CBE architecture. Each SPE is a completely independent 32 bit processor with 256 KB of storage for code and data. The SPEs are not required to do any system management tasks since the PPE handles all system management. Consequently, they do not need to be context switched at all. The programmer retains control over how long they run. This also has an interesting and useful result. That is, as long as they are supplied with data (i.e. they are not stalled), their execution is deterministic. Each SPE has its own asynchronous memory controller so that it does not have to manage the DMA tasks once the DMA request has been issued [9]. The SPE has a very simple pipeline. It varies from 2 to 7 stages as shown below. It does not support out-of-order execution or register renaming that give the Xeon its performance advantage in control intensive code [10]. The SPEs utilize a rich set of vector intrinsics to perform the same operation on multiple data elements at one time.

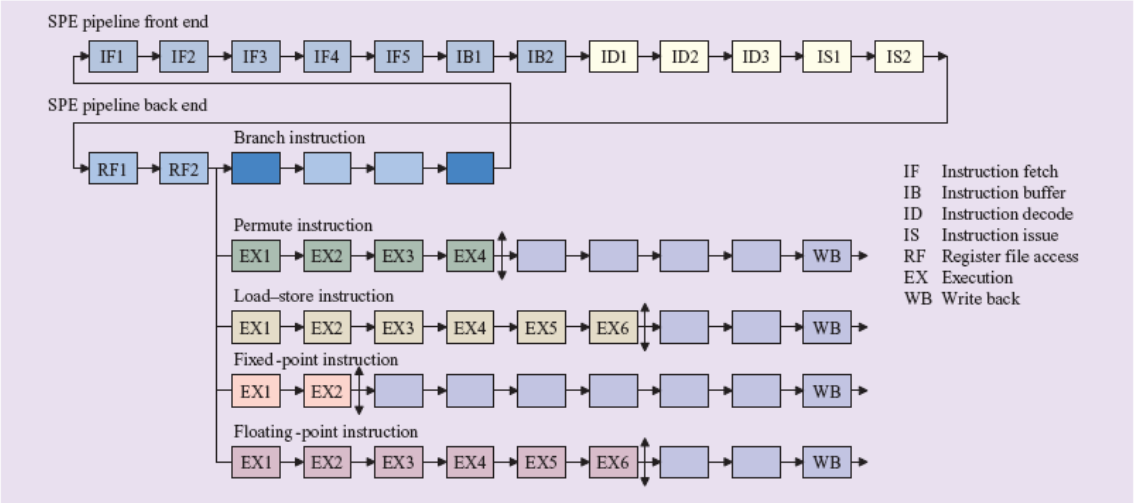


Figure 3-2 Pipeline Timing Diagram [10]

Each of the eight SPEs is an identical and completely independent processor. They have their own local memory, program counter and registers. Each SPE is a 32-bit RISC processor optimized for intensive and demanding applications [2]. Although they commonly are used to run the same code concurrently and divide the data processing, they may each run an entirely separate application.

Each of the 32 bit processors has 256K of local storage. This is termed “unified local store”, as it holds both instructions and data [9]. All code and data must fit within the 256K local store. This is a serious constraint which must be carefully considered. The initial target application of the CBE was for the intense gaming demands of the Sony Play Station 3. The tasks of scientific computing can be similar in nature and handled on the CBE as well. The CBE is particularly well suited to small algorithms which process huge amounts of data. Good

examples are compression or encoding algorithms, FFTs, matrix multiplications, and graphics algorithms [9].

The impressive DMA performance, when coupled with a “double buffer” technique (expounded upon in the MFC section), can keep all 8 SPEs on the CBE processing data with no memory latency. The architecture of the SPEs encourages programmers to initiate DMAs from the SPE to “pull” data to the local store. The PPE can “push” data, but only four DMAs can be in flight at once [9]. However, each of the SPEs can have up to 16 in flight DMAs at one time. So “pulling” DMAs from the SPEs allows a much greater number of in flight DMAs at once, and is the preferred model of operation for this hardware architecture.

3.6 The MFC

The MFC mediates communication with the EIB. The MFC is the SPE’s interface to the EIB and main memory [2, 9]. In addition to managing the DMA transfers, it also handles synchronization with main storage. This is done with mailboxes and signal notification events. The MFCs communication with memory via the EIB is completely asynchronous. Once a DMA has been requested, the SPE may continue to process, while the MFC manages all the data transfer. The SPE can then check when the DMA is complete and process the data. The extremely high bandwidth of the EIB makes it very unlikely that a given problem will be I/O bound. The observation made here is that most algorithms on the CBE are compute bound.

Since the MFC operates asynchronously, the programmer can employ a data model in which the SPEs operate on one block of memory while the MFC loads another block. Computation on the SPE is therefore completely overlapped with I/O without any memory latency from memory. Another buffer of data is always ready for the task when the other buffer is exhausted. This is a highly encouraged programming paradigm called double buffering. This technique is often called ping-pong buffering in many applications. Properly structured problems can eliminate memory access penalties since the SPEs can operate on one buffer while the MFC fills the other. This effectively reduces memory latency to zero once the first buffer is filled and ready for processing. This is a big advantage over the Xeon in memory intensive applications as test cases in this paper will show.

Double buffering is not the only model of computation used on the CBE, but it is by far the most common. This is the model employed on the CBE tests reported in this paper. Other models used in the CBE are the “Function-Offload Model, the Device-Extension Model, the Computation-Acceleration Model, the Streaming Model, the Shared-Memory Multiprocessor Model, the Asymmetric-Thread Runtime Model, and the User-Mode Thread Model” [2]. These are not considered in this study, but the reader is encouraged to refer to references [2] and [9] for discussions of these models.

3.7 The Registers

The SPEs have a large 128 by 128 bit register structure. They store all data types: integers, single, double floats, vectors, scalars and bytes. This aids in loop unrolling, and also helps compensate for the lack of a cache [2, 9]. A programmer can utilize this by unrolling constant loops (that is, loops with a constant index) with iteration counts < 128. The vector intrinsics can operate on a single 128 bit vector, and can operate on multiple elements (e.g. four 32 bit integers) at once [2,9].

The large register file will often provide better performance on “straight line” code than a loop will. In fact, any usage of the register file (function inlining, predication, unrolling etc) that eliminates branches is usually good since the SPE has no speculative branch support other than programmer supplied branch hints. But loop unrolling increases the size of the code. So there is a caveat to this general rule. The programmer must make sure that both code and data still fit in the very limited 256 KB local store [9].

3.8 The Vector Intrinsics

The CBE performance on the eight SPEs is facilitated by the special SIMD instruction set. The ISA (Instruction Set Architecture) operates mostly on vector operands. Although they look like C function calls, they are actually assembly language sequences, which give the programmer the efficiency of natively supported assembly vector operations, with the syntactic simplicity of C

functions. These make it much easier for a programmer to leverage the full power of the SIMD capabilities without the complexity of assembly. “A vector is an instruction operand containing a set of data elements packed into a one-dimensional array” [2]. Multiple data elements can be contained in the vector. For example, four words could be in a single 128 bit vector. The SIMD instructions can also operate on multiple data elements at once. To illustrate this, there is a code fragment in Figure 4-4 showing a vector add operation on all 4 integers at once [2].

3.9 The EIB

The EIB (Element Interconnect Bus) is a ring bus that provides communication with the SPEs. It is more than just a simple bus. It is an important part of the architecture to get superior performance out of the SPEs. In order to keep the SPEs supplied with massive amounts of data, the EIB employs four 128-byte data rings. “Each processor element has on ramp and one off ramp. Processor elements can drive and receive data simultaneously” [9]. Since the EIB ring runs at half the clock speed, that equates to 204.8 GB/s at 3.2 GHZ [9].

4 Performance Comparison

4.1 Methadologies

There are 3 generally accepted performance comparison methodologies: measurement, simulation and analytical modeling. The best resource of measurement and performance related topics is the website of the Standard Performance Evaluation Corporation, at http://spec.org. Since I have access to actual hardware I will use direct measurement. But a brief mention of the other two is in order.

4.1.1 Simulation

> "A simulation is the imitation of the operation of a real-world process or system over time. Whether done by hand or on a computer, simulation involves the generation of an artificial history of a system … to draw inferences concerning the operation characteristics of the real system" [23].

Although we will not use simulation in determining performance, the IBM FSS (Full System Simulator) will be used to validate the correctness of the CBE test cases, and to present the results. The FSS is a powerful full featured tool which can provide "cycle-accurate" and functional simulation of the CBE [2]. For determining exact behavior of the SPEs, which are deterministic, it is a powerful tool. Although I will present results with the Full System simulator, I will not be employing any simulation methodology and will use it merely as a presentation tool.

4.1.2 Analytical modeling

“What is an analytical model? By pure definition and in terms of being applied to computer systems, it is a set of equations describing the performance of a computer system” [22].

4.1.3 Measurement

Since Xeon and CBE hardware were available for this research, direct measurement was the logical choice. Since this research involves comparison of two different processors, the measurement will of necessity be an apples-to-oranges comparison. But that is the emerging nature of performance testing as multicore and SoC (System on a Chip) designs become more common.

4.2 Considerations of Performance Comparison on Different Code

One of the considerations of the Xeon is ease of programming. On a pipelined processor, the memory accesses and caching are transparent to the programmer. Out-of-order execution, multiple instruction issue, and many other exceedingly complex problems are handled transparently for the programmer, as well as structural, control and data hazards. This makes an enormous amount of complexity completely transparent.

However, on the CBE, memory transfers (whether by DMA or the mailbox mechanisms), synchronization between the processors, and coordination of tasks between the 8 SPEs and many other complexities must be explicitly managed by the programmer. IBM has an API for these operations called

“intrinsics”. These are assembly language calls that look and act like regular ANSI C function calls. The coordination of data movement on and off of the SPEs is the responsibility of the programmer.

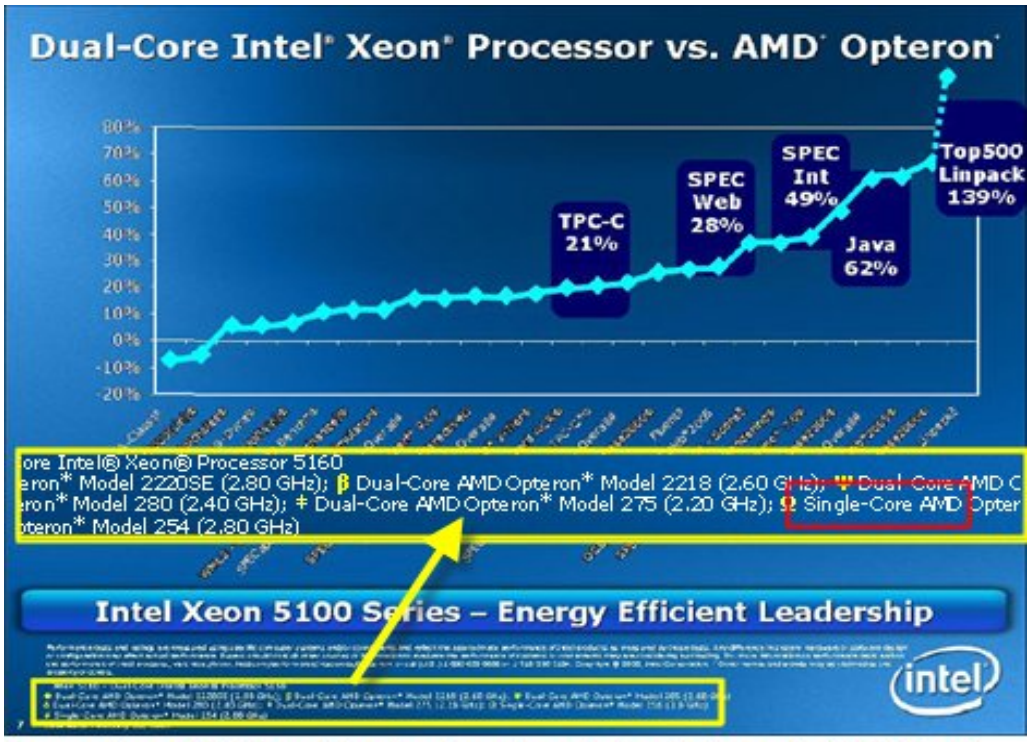
This considerably increases the difficulty of programming the CBE. Furthermore, it considerably complicates the difficult task of comparing 2 different processors since it is not a simple issue of recompiling and re-running the test. Although we cannot run identical test code on both the Xeon and the CBE, we can keep the structure relatively close for the purpose of comparison.

4.3 Lies, Damn Lies, and Statistics [5]

No result presenting relative test comparisons of 2 different processors would be complete without mentioning a long history of benchmark abuse in the computer industry. It would be very naive to consider comparisons between anything in the computer industry without realizing that the stakes for manufacturers both big and small are huge. A prudent researcher should keep a skeptical view towards vendor claims, since they are prone to excess.

Historically, microprocessor vendors could hardly be accused of unbiased objectivity in their benchmarks. But generally speaking, they had not approached the supremely spectacular level of benchmark abuse often seen in the database community. But in March of 2007, Intel released a set of benchmarks where each point on a graph (which showed incredible superiority of Intel chips), was compared to a different AMD Chip! If a current comparison could not be found,

they used the most recent AMD chip for which data was available. This was labeled as “feloniously misleading” by ZDNet staff writer David Berlind [6]. Intel also used benchmarks “that had been officially retired by their authors” [6].



In a series of highly misleading benchmark charts shown to Wall Street analysts to prove the company's chip supremacy in Feb '07, Intel demonstrated its latest greatest dual-core and quad-core server chips thoroughly stomping AMD's Opteron in terms of power-efficiency and performance. However, upon further examination of the fine print (magnified above), Intel used an apples to oranges method that would horrify most professional statisticians. For more than half of the Intel-favorable test results, Intel's latest greatest chip (in the above case, the "Woodcrest" 5160) was not compared to AMD's latest greatest Opteron, but rather, on a test-by-test case, different and older generation Opterons, and in some cases, single-core chips. Many of the comparisons shown to analysts also involved poorly-suited single-threaded tests and/or benchmarks tests that have been officially retired by their authors.

Figure 4-1 Intel Benchmark Data [6].

Although AMD is hardly a saint in this arena, it had not stooped to such depths in their benchmark results. Shortly after the Intel benchmark though, AMD showed its ability to resort to incredibly misleading benchmarks. AMD resorted to different tactics than Intel, but no less dishonest [14]. AMD, instead of cherry picking, omitted results from 2 tests which Intel was significantly superior on, thus biasing the outcome in AMDs favor. The underlying point is that objective benchmarks are difficult to design, execute and compare even when the marketing department does not embellish the engineering department's results. Great care is required in reading and evaluating industry benchmarks to ensure fairness and accuracy. Designing and evaluating useful and meaningful benchmarks is difficult even when the author is free of all bias. It would appear that in industry, with the huge sums of money at stake it is much more difficult.

Published scores on SPEC.org			
	AMD 2222 SE	Intel 5355	Intel 5160
SPECompMbase2001	**13275**	11822	10689
SPECint_rate2006 Peak	56.6	**84.8**	55.2
SPECfp_rate2006 Peak	52.1	**60.2**	45.1

Using dual-core 5160 as baseline performance			
	AMD 2222 SE	Intel 5355	Intel 5160
SPECompMbase2001	**24.2%**	10.6%	0%
SPECint_rate2006 Peak	2.54%	**53.6%**	0%
SPECfp_rate2006 Peak	15.5%	**33.5%**	0%

Figure 4-2 AMD benchmark data with omitted Intel scores in blue [14].

> “Processor and server vendors often point to several well-known benchmark tests when they want to measure processor performance in certain types of situations, such as the various TPC (Transaction Processing Performance Council) benchmarks for online transaction processing or Web serving, and the SPEC (Standards Performance Evaluation Corporation) tests for measuring integer and floating-point performance. But vendors spend millions tweaking their systems to produce favorable results on those tests, which means most customers insist on running test systems in their own environments before making a decision” [7].

4.4 Categories of Performance Evaluation

> “A number of popular measures have been adopted in the quest for an easily understood, universal measure of computer performance, with the result that a few innocent terms have been abducted from their well-defined environment and forced into a service for which they were never intended. Our position is that the only consistent and reliable measure of performance is the execution time of real programs, and that all proposed alternatives to time as the metric or to real programs at the items measured have eventually led to misleading claims or even mistakes in computer design” [1].

As with everything in performance analysis, there are several nuances. Hennessey and Patterson describe 5 separate levels, in order of decreasing accuracy in [1]. I will briefly touch on these, and explain why I settled on the kernel.

4.4.1 Real Applications

The first class described by Hennessy and Patterson are “Real Applications”. These are the “real applications” that user run, such as compilers, office suites, graphics programs etc [1]. The biggest problem in using these as performance evaluation criteria is that they are often modified for portability. This

means that the native abilities of a given architecture may not be utilized in the interest of cross platform support [1].

It is difficult to use real programs to measure the CBE because so few exist for it. In the current test bed used for this paper, even a compiler test can not be profiled since current release of software used for this research for the CBE must be cross compiled from the Xeon. Unlike other tests, where code is highly portable, porting code to the CBE involves structuring and decomposing the problems so that it may be effectively run on the SPEs, and manipulating data such that the vector intrinsics will be effective. Also, memory access which is largely transparent in the Xeon must be explicitly managed in the CBE.

4.4.2 Modified (scripted) applications

Secondly, modified applications are “real-world” applications, which have been modified to make them more suitable for performance evaluation. An example of this type would be an application which has had I/O removed in order to minimize the long latency of disk access. Such applications could then be more suitable for CPU intensive benchmarks [1]. Scripts can be added to simulate user interaction. This category suffers from the same problem of the category above in that there is a performance vs. portability trade-off.

4.4.3 Kernels

Third, kernels extract small critical regions from programs to evaluate performance. They are not “real programs” in any sense, they are useless to

users. They are performance evaluation tools only. “Kernels are best used to isolate performance of individual features of a machine to explain the reasons for differences in performance of real programs [1]”. The tests used in this paper fall into this category. These tests endeavor to determine performance penalties of code which could impact the performance of the Xeons deep pipeline. This research will compare the branching performance of code which has significant branching and other code structures which will differentiate the 2 processors performance.

4.4.4 Toy Benchmarks

The fourth category is toy benchmarks. These are small programs which produce known results. “Programs like the Sieve of Eratosthenes, Puzzle, and Quick sort are popular … The best use of such programs is beginning programming assignments” [1].

4.4.5 Synthetic Benchmarks

Finally, we look at synthetic benchmarks. As Hennessey and Patterson mentioned in [1], synthetic benchmarks and kernels share a similar philosophy. Sometimes it is a little difficult to distinguish the two, and there is more than a little room for semantic hairsplitting. The test cases presented here are considered kernels. They are distinguished from synthetic benchmarks based on two key differences.

“Kernel code is extracted from real programs, while synthetic code is created artificially to match an average execution profile [1].” Although my test cases were not extracted from real code per se, they isolate performance of individual features of a machine to explain the reasons for differences in performance of real programs [1]”. The second distinction drawn is that they were not designed to match any execution profile. Based on those criteria from [1], these tests are considered to be kernels.

4.4.6 Performance Comparison Considerations

In order to understand performance comparisons, the CBE architecture was discussed the in some detail. Many of the architectural items of the CBE suggest require code structure that is different from a Xeon. That is, you can't compile the same code on both and simply compare the results. Consequently, researchers must agree that two sets of code, although possibly implemented differently, constitute a fair comparison between two different architectures. An example of this would be a simple loop which adds up a result on a conventional processor, and SPE code which does the same thing.

```
int a[4], int b[4], c[4];
... // assign variables
for (int i = 0, i < 0; i++){
     a[i] = b[i] + c[i];
}
```

Figure 4-3 A Xeon example of vector addition [2]

```
int a[4], int b[4], c[4];
a = vec_add(b + c);
```

Figure 4-4 A CBE example of vector addition [2]

Here is an example of two different implementations of the same idea; adding 2 vectors. Although the implementation is different, there is little doubt that the comparison is fair since the code is doing the same thing. The 2nd implementation must be different in order to utilize the vector intrinsics to leverage the power of the CBE. Although this example is somewhat trivial, in more difficult tests the work required to show that the comparison is fair is sometimes more difficult.

4.5 Summary

The CBE architecture is a radical departure from traditional processor designs. The SIMD concept has been around since 1958 [1]. The CBE has lots of good old ideas, such as pipelining and SIMD processors. It then combines them in a new way on a multicore processor. Then it adds some very clever new ideas, such as the high bandwidth EIB, 8 processing cores, and an asynchronous MFC with some clever new programming models to come up with a successful new SIMD implementation. The hybrid design of the CBE, dual optimizations, asynchronous memory access, and high speed bus have proven to be first successful commercial implementation of SIMD architecture.

5 Test Cases and Results

5.1 Comments on Timing Measurement

The simple UNIX time command was used to perform the timing measurements reported. Although the system clock could have provided more accurate timing, high resolution was not considered necessary here. Since the performance advantages on the CBE are in the range of orders of magnitude, timing differences of less than 3% were considered relatively insignificant. Also, the experiments were designed to ensure that the PPE was doing no other tasks while test cases were being run, so that there would be no other load on it other than the normal O/S tasks. These O/S tasks typically take up less than 1% of the CPU, and thus, are also considered insignificant.

5.2 Comments on Optimization

As I have previously mentioned and cited from [1], optimization can cause some unexpected results. That is especially true in the first test. Here, if we compile with optimization, the compiler will see that the results of the loop are never accessed until the end of the test. Consequently, it will simply execute the entire loop at compile time and store the result. This is fairly easy to diagnose on the Xeon, since the optimized test executes instantaneously, instead of 10 seconds that it takes when it does the calculations at runtime.

But with the PPE, which exhibits sluggishness highly uncharacteristic of a PPC 970, the optimized test runs in about 10 seconds. An unoptimized test

reveals the true performance problem on the PPE, taking over 100 seconds. Consequently, it the PPE test was optimized, and the Xeon was not, they would be roughly equivalent, leading to an erroneous conclusion. This is an easy mistake to make, since the CBE makefile in the IBM SDK is over 1400 lines long, and the optimization flag is deeply buried.

5.3 The Sum of the First n Intergers – Xeon and CBE

This section will illustrate one of the first surprises encountered on the CBE during this project. Namely, that although the PPE is based upon the PowerPC 970 family, it should not be considered a fully exploitable processor for the purpose of number crunching on the CBE. The PPE should remain a supervisory processor coordinating SPE tasks. As the results reported here show, the PPE does not have the computational horsepower that one would expect of a normal single-core PPC 970. This becomes especially evident when its performance is compared to the Xeon.

The test is simple: sum up the first 2^{32} integers. This solution can be found by Gauss' formula N (N+1)/2, which equals 9,223,372,039,002,259,456. Since the goal of this test was to obtain the approximate relative running times for each processor, using the UNIX time command to measure real time worked satisfactorily. The test results were 9.738 seconds for an average of 3 runs on the Xeon, and 102.188 seconds running on the PPE, a factor of about 10. This illustrates the PPE is not a good number crunching processor like its full PPC 970 brethren. If it is used in that fashion, it might be taxed beyond its ability to

oversee the SPEs with data traffic management, and thus result in data starvation for the SPEs further worsening the overall throughput of the chip. Ironically, the ten fold performance degradation could have been the result of an ill-fated attempt at seeking extra speed from the CBE. This result shows that the PPE should be used only for O/S duties and SPE management - it should not be doing computation.

5.4 Memory Access - Xeon

In this test, the goal was to compare the performance of memory access. This is a strong point of the CBE, since it does not have a pipeline to induce any latency. The Xeon tries to mitigate pipeline latency by the use of a large cache. In this test, a constant number of increments are performed to a block of memory on both the Xeon and the CBE. On the Xeon however, different sizes of memory blocks are used to illustrate the difference that a cache has on memory access. A convenient number, 2^{34} is used, since it results in test cases ranging from 13 to 45 seconds of run time, which is long enough for accurate measurement, but short enough to allow for multiple runs during testing. On the Xeon, 2^{34} increments of an array of 4 byte integers were performed. On each subsequent test, the size of the memory block being incremented is increased by a power of 2 while being iterated over by a factor of 2 less, thereby always performing a constant number of 2^{34} increments.

For example, in the first test run, the program receives arguments of 3 and 31. This means an array of $2^3 = 8$ integers will be iterated over and incremented

2^{31} times, for a total of 2^{34} increment operations. The next run will have arguments of 4 and 30, meaning an array of 2^4 = 16 integers will be iterated over and incremented 2^{30} times, for a total of 2^{34} increment operations, etc.

The test starts with arguments of (3, 31) and ends at (27, 7) for practical limitations. Error checking code is used at the end of the test to verify that increments have indeed taken place by adding up all the integers in the array. It was not possible to start the test with a smaller array size with arguments such as (2, 32), (that is an array of 2 integers iterated over 2^{32} times), because the error checking code and the integers in the array would overflow a 32 bit integer. On the other end of the scale, the first argument is upper-bounded by the amount of memory available to create the array. On the particular cell blade (a Mercury Computing DCBB) used in these experiments, 512 MB can be allocated to the running process. This constrains the first argument to 27, for 2^{27} (134,217,728) integer array, of 4 bytes each, totaling 512 MB (536,870,912 Bytes) of memory. Thus the entire sequence of arguments for this test runs from (3, 31) to (27, 7).

5.5 Memory Access Results - Xeon

Test Argument 1	Test Argument 2	Array Size (Integers)	Array Iterations	Test Run Time, #1	Test Run Time, #2	Test Run Time, #3	Average
3	31	8	2147483648	19.924	20.118	19.961	20.001
4	30	16	1073741824	19.832	20.214	19.851	19.966
5	29	32	536870912	20.020	20.014	20.037	20.024
6	28	64	268435456	19.874	20.126	19.883	19.961
7	27	128	134217728	14.359	14.264	14.336	14.320
8	26	256	67108864	13.788	13.989	13.789	13.855
9	25	512	33554432	13.840	13.708	13.846	13.798
10	24	1024	16777216	13.695	13.870	13.693	13.753
11	23	2048	8388608	13.902	13.755	13.905	13.854
12	22	4096	4194304	13.795	13.962	13.794	13.850
13	21	8192	2097152	13.361	13.898	13.363	13.541
14	20	16384	1048576	13.470	13.396	13.470	13.445
15	19	32768	524288	13.792	13.473	13.794	13.686
16	18	65536	262144	13.740	13.798	13.738	13.759
17	17	131072	131072	13.976	13.762	13.797	13.845
18	16	262144	65536	13.904	14.015	13.922	13.947
19	15	524288	32768	14.350	14.178	14.350	14.293
20	14	1048576	16384	19.399	19.410	20.233	19.681
21	13	2097152	8192	38.253	38.381	37.981	38.205
22	12	4194304	4096	42.633	41.465	42.332	42.143
23	11	8388608	2048	41.966	42.907	42.207	42.360
24	10	16777216	1024	42.988	42.437	42.908	42.778
25	9	33554432	512	42.571	43.053	42.618	42.747
26	8	67108864	256	43.275	42.819	43.241	43.112
27	7	134217728	128	43.203	43.599	43.217	43.340

Figure 5-1 Xeon Memory Access Results

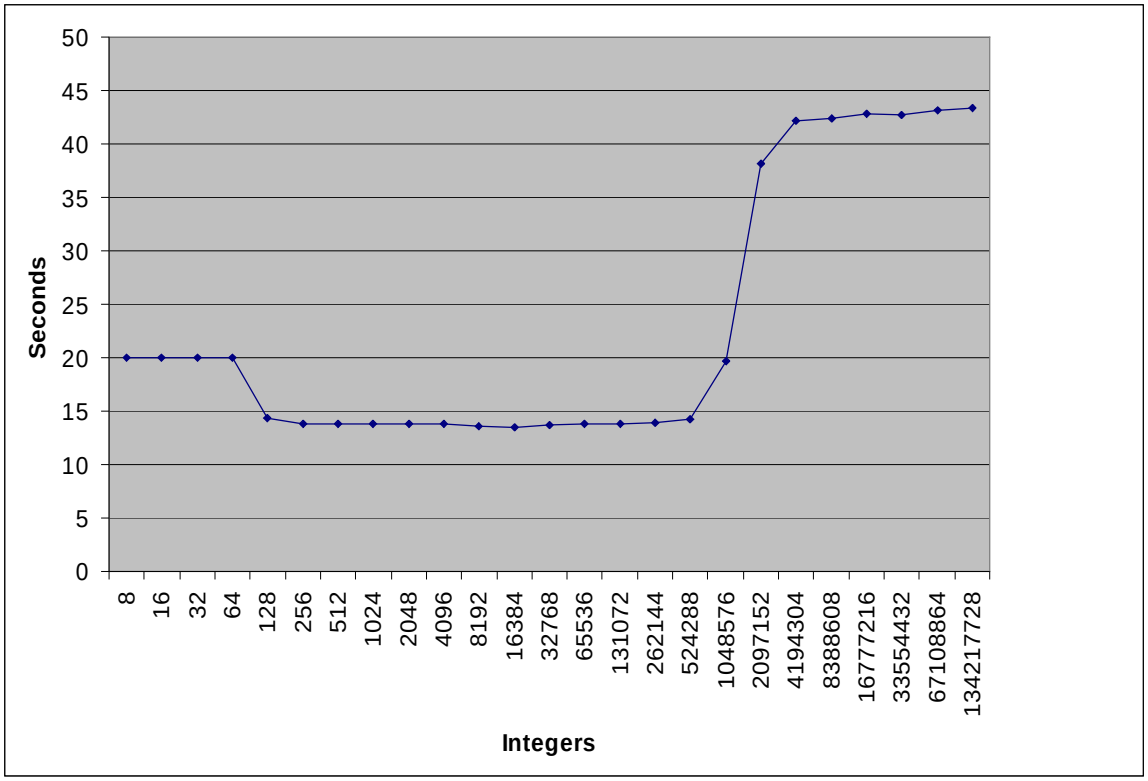


Figure 5-2 Xeon Memory Access Results

5.6 Memory Access Analysis - Xeon

Since the Xeon used has a 2 MB cache, one would expect that the array being iterated upon to be stored there. However, at some point, the size of the array will be larger than the cache capacity. At that point, the memory access time should increase significantly since all accesses must then run through the Xeon's long 31 stage pipeline. Indeed this is exactly what happens, and execution times start to increase. When the array is 2^{19} integers at 4 bytes each for a total memory usage of 2 MB, the performance is still consistent with previous results in the 14 second range. However, when the array is 2^{20} integers

at 4 bytes each for a total memory usage of 4 MB the execution time increases by 37% to almost 19.7 seconds.

The next test with 2^{21} iterations shows that the array is now 8 MB and the cache is no longer effective. Execution time doubles to over 38 seconds. Performance continues to degrade, but the change is much smaller finishing at 43.34 seconds when the full memory block of 512 MB is used. This is a degradation of over 300% which is solely attributable to the Xeon's inability to use its cache to mitigate the effects of its long pipeline. It is interesting to note that array sizes of less than 64 integers perform poorly before dropping down to the most optimal performance in the 13.7 second range. No explanation can be found for this behavior.

5.7 Memory Access – CBE

There was no need to try different sizes of memory blocks in the CBE, since the memory is not hindered by a pipeline. The direct access of a nearly constant DMA time is one of the CBE strengths. The SPEs will access the memory in 16 KB blocks, and use DMA to pull the memory to their local store. The memory will then be incremented and pushed back to XDR. The exact same number of increments (2^{34}) were performed.

There were 3 results, each of which was the average of 3 runs. The "verified" run was the result of 2^{34} increments being performed, and a subsequent verification by the PPE. That verification involved the PPE summing

up each integer in the memory block under test. The problem with this is that the PPE exhibits poor performance. So once that that test is working properly the verification code was #ifdef'ed out the to get the "unverified" result. Since the result is always the result of 2^{34} increments being performed, this does not affect the validity of the test, and ensures the measurement of only the memory access.

Verified	Unrolled	Run 1	Run 2	Run 3	Average
Yes	No	5.688	5.703	5.712	5.701
No	No	4.392	4.449	4.456	4.432
Yes	Yes	6.29	6.232	6.134	6.219

Figure 5-3 CBE Memory Access Results

5.8 Memory Access – CBE Analysis

In the CBE results, the memory access performance was impressive, measuring in at 4.432 seconds. That is 3 times faster than the best time recorded by the Xeon, and the Xeon only achieves that time if the memory access blocks are of optimal size for it. On larger blocks of memory the CBE is truly an order of magnitude faster. An anomaly with the data reported in Table 4-1 is the unrolled loop performance, which is usually faster. One possible explanation that can be offered is the following. Since the branching algorithm used by the cell is by default "assume taken", a loop (representing the "rolled" case) will not have any branching penalty until the very last iteration, boosting this test case's performance; however, the larger code size presented by an

unrolled test case might be somehow responsible for a degradation in performance. This would certainly be an interesting area for future investigation.

5.9 Branch Penalties – CBE

The SPEs use a very simple scheme for branch prediction - they predict the branch will be taken, unless a branch hint is used. Since common Xeon and CBE code with known performance numbers already exists at this point, branches will simply be inserted into that code.

The branch penalty test attempts to determine the amount of branching that would reduce the performance of the CBE to the level of the Xeon when the Xeon was accessing optimal sized memory blocks. Since the goal was to determine what happens when the branch is not taken, some random code using if-then statements was inserted into the “if” clause. The path of execution was then forced always go through the “else” clause. Since the “if” clause will never be executed during the test, it will have no impact on performance other than code size. In this manner, the performance penalty of the mispredicted branch can be measured.

5.10 Playing Cat and Mouse with the Compiler

This is a test that we want to compile with optimization for the best possible performance. However, for the branches, we want code in which the mispredicted branch is always taken. Since we are not using branch hints, this

will always be the “else” clause. For this test, since the CBE has a simple “assume taken” algorithm, we will put a two meaningless instruction in the else clause so performance impact will be minimized. But the “if” porting of the branch has to be rigged a little bit.

The sole reason for the cryptic code in the branches is to make sure that the code will be generated by the compiler and not optimized out. This will cause the branch penalty to always be incurred. If the compiler can determine that the “if” clause is never executed, it could optimize it out. Consequently, the code in the “if” clause must be sufficiently complex so the compiler can not recognize that it is never taken, and the branch penalty to the “else” clause is always incurred. Significant experimentation time was invested to ensure the branches in the test are always false. Consequently the compiler can not optimize them out. As the branches were inserted, the results showed that only 4 mispredicted branches caused the CBE to drop below the Xeons best performance.

5.11 Branching Results - CBE

Branches	Run 1	Run 2	Run 3	Average
0	4.392	4.449	4.456	4.432
1	8.903	8.992	9.152	9.016
2	10.793	10.587	10.598	10.659
3	11.558	11.657	11.682	11.632
4	15.601	15.82	15.902	15.774

Figure 5-4 CBE Branching Results

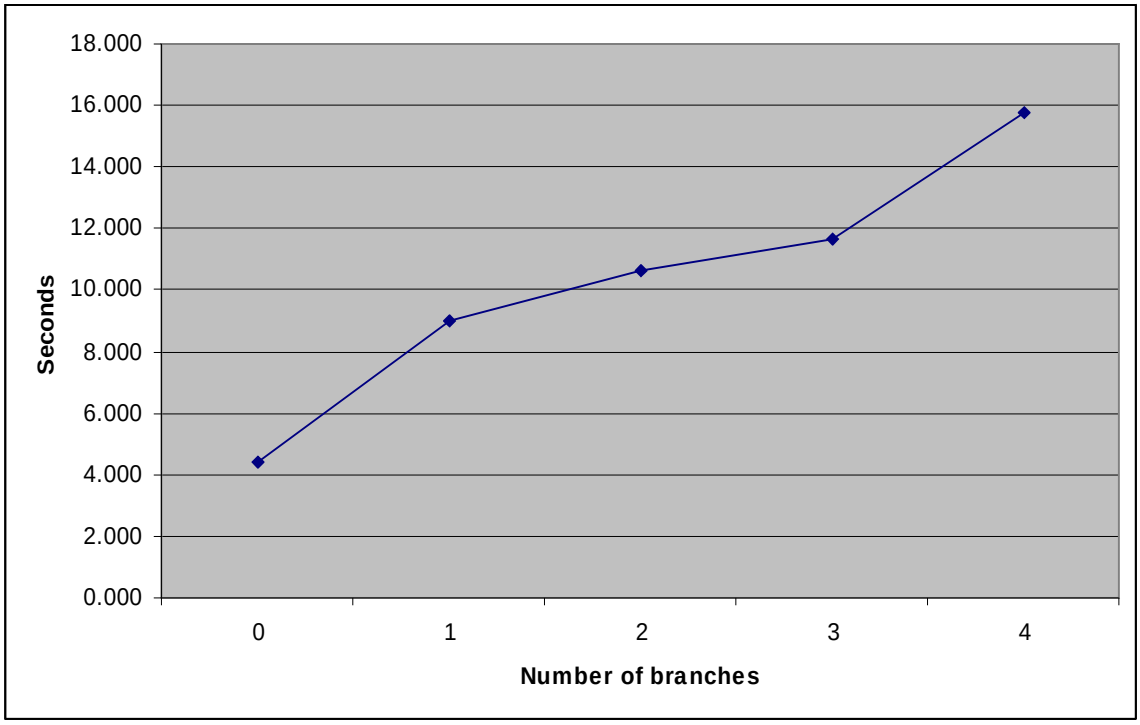


Figure 5-5 CBE Graphical Branching Results

5.12 Branching Analysis - CBE

This test may look a little surprising in that only four branches were required to degrade performance below the Xeon; however, it should be noted that this is an absolute worst case scenario. The test code forced 4,294,967,296 iterations of a loop (these iterations were distributed across the 8 SPEs) through 17,179,869,184 mispredicted branches. Even a perfectly random distribution of branches would double performance over the result obtained. Also, the code being executed in the "else" clause is very minimal. So almost no computation is being accomplished. Consequently, the SPEs were forced to spend most of their

time incurring the 18 cycle branch penalty. This happened every time the SPE tried to execute a vector add operation.

This result underscores why IBM puts so much emphasis on branching. Applications should be carefully coded to use branch hints if the probability is greater than random. In addition, if at all possible, programmers should consider removing them entirely. If the “if” or “else” clause is small, it may be better to simply execute both speculatively, rather than risk an 18 cycle clock hit in a frequently executed loop if the prediction is wrong.

6 Conclusion and Future Work

6.1 Hard Lesson Learned, the “Unknown Unknowns”

> “Reports that say that something hasn't happened are always interesting to me, because as we know, there are "known knowns"; there are things we know we know. We also know there are "known unknowns"; that is to say we know there are some things we do not know. But there are also "unknown unknowns" — the ones we don't know we don't know” [17].

As I concluded my testing a started writing up this report, I was struck by the fact that most of what I wanted to communicate about my experience had little to do with my research topic or my test results. What I continually pondered and discussed with like minded engineers were the problems I didn’t even know I had.

For example, I knew from the IBM literature that branching penalties existed. I also knew that there could be problems in long pipeline latency from taking CS247 (Advance Computer Architecture) with Dr. Chun, where we study pipelines and readings from Hennessey and Patterson, one of the definitive works in the field. The main focus of my research topic was to quantify them and provide guidelines that would aid in the selection of the best solution for a given task.

Likewise in the CBE I felt I only needed to quantify the performance impact of branching penalties. But what surprised me most, and what I would like others to learn from my experience, is that exposing the “unknown

unknowns” in research provided far more learning and is far more applicable to my future research than the “known unknowns” that I originally thought to be the main focus of my efforts.

This change in my thinking really occurred in the last 6 weeks before the submission of my paper, when I attended a training class at Mercury Computing. Mercury manufactures the CBE hardware we use, and provides consulting on CBE programming hardware. They are a great source of CBE knowledge and I had the luxury of asking some very knowledgeable engineer many questions. It slowly began to dawn on me that the most important things that I have learned had to do with the “unknown unknowns”. This section is important to me personally, since it represents what I think are the most important things I learned.

6.2 Beware of the Compiler

In most research projects involving system code and other performance oriented code, the GNU C/C++ compiler is the compiler of choice. The GNU compiler is an excellent compiler, but it is used far more often on x86 architectures. Far more effort is expended optimizing for x86 than for MIPS or PPC cores. Consequently, performance differences may be more attributable to more robust optimization than chip architecture.

This is not a new problem to CBE, MIPS, PPC or any other type of core. The fact that compiler optimization can skew test results is well known. As a

matter of fact, it is a concern raised by Hennessy and Patterson in [1]. They mention that a good optimizing compiler can recognize and discard over 25% of Dhrystone code [1]. The fact that smart compiler technology can lead to results that are misleading on the good side is well documented. But what I had not considered is that a compiler which does not optimize well can lead to erroneous poor results.

A good example of this is to consider how well the compiler can do loop unrolling. The CBE architecture with its 128 x 128 bit wide register structure is able to perform multiple operations in a single clock cycle if loops are unrolled [9]. However, you have no guarantee that the optimizing compiler (using the GNU –O3 flag) properly saw and unrolled the loop. Loop unrolling is a common feature of compilers and has almost come to be expected by programmers. But apparently, according to [18], performance has been inconsistent. Thus it is common for the IBM literature to recommend doing this manually in your C code.

6.3 Consider WYSIWYG Assembly

One thing I found very significant was that Mercury Computing writes its MCF (Multi-Core Framework) for the CBE in assembly. The drawbacks of assembly are many. The code is very cryptic, productivity is low, competent programmers are hard to find, and they tend to burn out quickly [18]. But despite this, when performance is critical, you may have to resort to assembly.

Mercury has some of the most accomplished assembly language coders for the CBE. Extensive libraries of scientific algorithms for government customers require the highest possible performance. Compiler problems they have presented in their training classes that led them to embrace assembly despite all its drawbacks include the following problems in [18]. The three following subsections are mentioned in [18] as reasons Mercury uses assembly in their core libraries.

6.3.1 Compiler Performance is a Black Art

Getting optimal code out of compilers is a black art [18]. In order to understand what it is doing you have to look at the generated assembly code. Consequently, it is often more productive to start with assembly than to try and determine the efficiency of the compiler generated code. Also, what code the GNU compiler may generate with a high degree of efficiently on one processor like the x86, may be much less efficient on PPC or MIPS architectures. Achieving proper usage of the register in a SPE so that the vector intrinsics can be utilized most effectively is of paramount importance. Such important performance issues may be better handled with assembly rather than guessing if the compiler will generate efficient code.

6.3.2 C Code is not Necessarily More Readable or Portable

Assembly is cryptic, but optimized C code is not necessarily that much easier. Once pragmas, optimized statements, processor specific directives have

all been inserted, optimized C code can be very ugly indeed. Also, in order to deal with many different types of optimization and levels of debugging and logging, C can vary quickly become littered with #ifdefs. One section of code I have personal experience with had 5 inter-nested #ifdefs leading to 2^5 or 32 separate code paths.

In many instances, C can be just as difficult to read, and you may still get inefficient code from the compiler on less popular processors like MIPS and the PPC since less time is devoted to their performance optimization. At least assembly is WYSIWYG, what you see is what you get [18]. For all its drawbacks, assembly code does not hide anything, it is exactly what will be executed.

6.3.3 Optimized C Code can be Very Inconsistant

Even if performance optimization in C code is productive, it still tends to be non-linear. Small differences in code structure or complexity (which is the case when developing high performance algorithms) can lead to drastic changes in performance [16,18].

One final example from my code indicates a significant difference between 2 different processors. In early experiments I was conducting (but did not use in this report), a loop index was labeled as volatile along with other variables to see what impact it had. It did not have an effect on performance on the Xeon. I moved the code to the CBE to conduct some similar experiments, but mistakenly

did not remove the volatile qualifier on the loop index. When I discovered it and removed it, execution time per iteration dropped to 5.8 seconds from 7.2. A 20% reduction in execution time. I would like to research the difference in generated code in future work.

6.4 Reasonable Assumptions May Not Be…

Anyone who has studied computer architecture is no doubt familiar with IEEE 754 floats, and rounding modes which are employed. The 3 most common are truncation, simple rounding, and round to nearest even [8]. Truncation is very undesirable because of the strong negative bias it causes[8]. The superiority and prevalence of round to nearest even is best since it cancels out bias. But even ordinary rounding with its slight bias is probably acceptable for most applications. If you are experienced engineer could reasonably assume that one of the better rounding schemes would be employed in the SPEs and your rounding error in floating point operations would be minimized. You would also be wrong.

In what a CBE engineer at Mercury Computing described as "***criminally negligent engineering***" the SPE only supports round toward zero (aka truncation)[18]. Whether or not this was a good design choice on IBM's part is a matter of debate. But it does illustrate a very good point: Sometimes the most reasonable assumption may not be.

6.5 Get Your Mind Off the Desktop (Think Like an Embedded Engineer)

In the same way the GNU C compiler is highly optimized and very effective for the x86 architecture, modern desktop mother boards are highly optimized. As a general rule, the higher volume of hardware shipped, the more a manufacturer can amortize expensive hardware as well as software driver development over a larger number of units. However, embedded hardware is much more prone to surprises if your primary programming experience is on desktop hardware. It is easy to become somewhat removed from the complications of programming lower level hardware. The PCI drivers on your hardware (for example video cards) are likely to come from high volume hardware manufacturers. Thus, a considerable amount of time and money has probably been expended on developing high quality drivers. But low volume embedded computers and software drivers often represent a work in progress (both with regard to hardware and software, and business issues). The following are couple of examples I have run into while working on CBE hardware.

6.5.1 You May Have to Manage More Details on an Embedded Device

For an example, consider the block diagram of a CBE device that I have worked with extensively, the Mercury CAB board. The following diagram is a block diagram of a CBE implemented on a PCI card manufactured by Mercury Computing. It is called a CAB (Cell Accelerator Board) and is one of their most popular CBE products.

This diagram shows the XDR DRAM which the CBE can access. The I/O bandwidth of the CBE is very impressive. In a desktop or server system running Linux you do not have to be concerned about which bank of memory you are accessing. But on a CBE you do. This implementation has 16 banks of XDR memory, each 128 bytes wide [9]. Memory is striped across the XDR [16]. When all SPEs access the same memory bank at the same time, read latencies can go up significantly, and your I/O bandwidth will go down significantly. Thus a CBE programmer may need to take the time to consider how memory access is distributed across XDR banks.

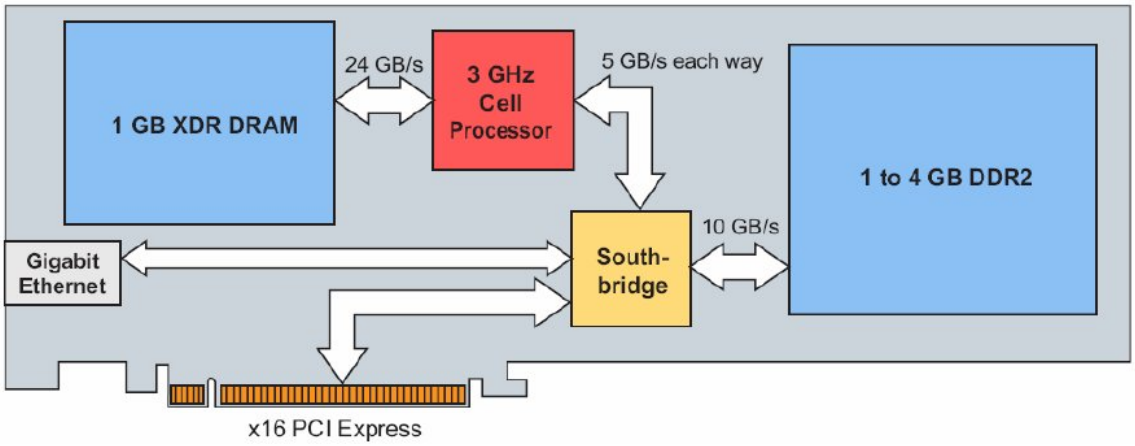


Figure 6-1 A Mercury PCI CAB Board Block Diagram

6.5.2 Manufacturors are Not Eager to Tell You What Doesn't Work

In my research, I was often working with beta quality hardware. Problems with chipsets and drivers are common. However, the ugly side of business realities often intrude into the development process. As I showed before, with

the multi-billion dollar microprocessor companies, there is a temptation to stretch the truth regarding performance numbers. With smaller companies, which often have a significant sale on the line, there is a great temptation to conceal problems the customer has not asked about yet.

This happened on several occasions. I will discuss 2 of these regarding a CAB and Cell Blades I worked with. On the first occasion, we were testing network throughput, and found a gigabit Ethernet interface had a chip-set problem and was only performing at 400 Mb. Once I called technical support and informed them of the problem, I was told a fix was in progress. The second was that in the above diagram, the Cell Southbridge chip was not performing at the 5 GB bandwidth which the Manufacturer was illustrating. In actuality, 3.1 was the best it could do.

Despite that fact that we would like all information on what does not work as expected, small companies are often under enormous competitive pressure. Many times with smaller companies, the future of the company is riding on the line. Purchasing decisions are often made based on the advertised performance of the product. It could be detrimental for a company in a competitive proposal to be straightforward with current product limitations and flaws. There is an absolutely huge temptation to keep them concealed until a customer inquires about them.

The fact that these details and problems must be managed is not a huge problem. That is, ***once you know*** you have the problems. The problem with

embedded computers is that you do not go looking for problems that you do not know that you have. “Unknown unknowns” can take very significant amounts of time to ferret out. You may not even know you have a rounding problem until your test results come out a little bit further off than you expected. You may not know that you have a memory bank access problem until your performance is slow for no known reason. You may not know you have a performance problem with the chip set until the manufacturer tells you. With embedded hardware and software, you are much more likely to be in the position of solving problems you do not know you have, the “Unknown unknowns”.

6.6 Future Work

The test results presented have quantified results in memory access code on the Xeon, and branching penalties on the CBE. They have also shown the excellent DMA performance of the CBE. This will provide a first step for researchers evaluating Xeon and CBE processors by providing code with known performance characteristics.

The confidence in the CBE test cases is not complete. The GNU gcc compiler is known to have inconsistent optimization on architectures less common than the x86. Future work in the field will involve writing the CBE test cases in assembly to ensure that the CBE test cases are not performing poorly due to poor compiler optimization.

7 Source Code

7.1 Sum of the First n integers

```
#include <stdio.h>

int
main(void)
{
    long long unsigned sum = 0;
    long long unsigned it;
    long long unsigned i = 0;

    it = (1LL << 32)+1;

    printf("Compiled on %s %s\n",__DATE__,__TIME__);
    for(i = 0; i < it; i++){
        sum += i;
    }

    printf("sum is %llu, it is %llu\n",sum,it);
    return 0;
}
```

7.2 Memory Access, Branching, Loop Unrolling

7.2.1 make file

```
#!/bin/bash
# Build optimized (O) and Debug (G)
PROGS = simplexO simplexG
all     : $(PROGS)

simplexO: simplex.c
      gcc -O3 -Wall -o $@ simplex.c -lrt

simplexG: simplex.c
      gcc -g -Wall -o $@ simplex.c -lrt

clean :
      rm simplex *.o core*  $(PROGS) > /dev/null 2>&1
```

7.2.2 Source code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/times.h>
#include <errno.h>
#include <time.h>
#include <math.h>
#include <unistd.h>

#undef ERROR_TEST

void memoryLoop(unsigned long a,unsigned long b);

int
main(int argc, char **argv)
{
    unsigned a=0,b=0;
    if(argc != 3){
        printf("USAGE: $ %s OUTERLOOP INNERLOOP.\n",argv[0]);
        exit(-1);
    }

        a = strtoul(argv[1],NULL,0);
        b = strtoul(argv[2],NULL,0);

        fprintf(stdout, "a = %u : b = %u \n",a,b);

        if(a+b != 34){
            fprintf(stdout, "interation error, a+b != 34\n");
            exit(-1);
        }

        memoryLoop(a,b);
        return (0);
}

void memoryLoop(unsigned long a,unsigned long b){
    unsigned long x,y,z;
    volatile unsigned long memory;
    unsigned long sum = 0;

    x = a;
    y = b;

    a = 1L << a;
    b = 1L << b;

    memory = a * sizeof(unsigned);
    printf("a = %lu ; b = %lu ; memory = %lu \n",a,b,memory);
```

```
    unsigned int *ar;  // ar always points to the base address
                       //of the allocated memory

    unsigned int *ptr; // ptr advance across the array
                       // on each iteration.


    ptr=ar=(unsigned int*)malloc(memory);

    if(ar==NULL){
        printf("malloc() failure %s:%d\n",__FILE__,__LINE__);
        exit(-1);
    }

    memset(ar,0,memory);

    for(x = 0; x < b; x++){
        ptr = ar;
        for(y = 0; y < a; y++){
            (*ptr) ++;
            ptr++;
        }
    }

    sum = 0;
    for(z = 0; z < a; z++){
        sum += ar[z];
    }

    printf("check sum = %lu, ar[0] == %u\n",sum,ar[0]);
    free(ar);
}
```

7.3 CBE Branching

7.3.1 Header

```
//  The IBM roadmap had 32 SPEs due in 2009, and
//  64 tentatively in 2020, so it would be helpful
//  to #define this.

#define SPE_COUNT  8
#define CACHE_LINE 128
#define MEMORY_BLOCK 134217728;

//  For debugging and status
#define  VERBOSE

#define  VERIFY_SUM
#define BRANCHING
#undef  UNROLLED
```

```
typedef struct TASK{
  unsigned int  spe_ea_block;
  unsigned int  baseAddress;
  unsigned char pad[120];
} task_t;
```

7.3.2 PPE

```
#include <stdlib.h>
#include <string.h>
#include <sched.h>
#include <libspe.h>
#include <stdio.h>
#include <errno.h>
#include <time.h>
#include <sys/times.h>
#include <unistd.h>
#include "../header.h"


//  This is the program_instance structure, which is use
//  to consolodate system information.
typedef struct PROGRAM_INSTANCE{
    spe_gid_t group;
    task_t task[SPE_COUNT] __attribute__ ((aligned (128)));
    speid_t sid[SPE_COUNT];  // SPE ID
    int status[SPE_COUNT];   // exit status
} program_instance_t;

program_instance_t program;

extern spe_program_handle_t spe;

//  The effective address (That is the PPEs address in XDR)
int *ea;

int
main(void) {
  unsigned i; // Loop index

  //  This is the main block of XDR memory we will operate on.
  //  We are limited to 512MB on the system I have available
  //  I named this XDR_Memory_Block to differentiate this from
  //  DDR memory, where I may store blocks in future work.
  unsigned XDR_Memory_Block;

  //  Each SPE will use a specific block.  The ea (effective address)
  //  of the block to use will be calculated and used as an offset
  unsigned spe_ea_block;
```

```
  XDR_Memory_Block = MEMORY_BLOCK;

  //  divide up amongst SPE_COUNT  SPEs
  spe_ea_block = XDR_Memory_Block / SPE_COUNT;

  //  Malloc the memory into the effective address in XDR
  ea = (int *) malloc((CACHE_LINE-1) + XDR_Memory_Block*sizeof(int));

  //  This is a common idiom in the IBM SDK for alligning on a cache.
  //  If any of the 7 lowest bits are set (from 1 to 127) then the
  //  pointer is not a multiple of 128, so we increment it.  Once no
  //  bits are set, the number is an even multiple of 128.  The worst
  //  case is that we will a pointer which is CACHE_LINE + 1,
  //  and we will have to increment 127 times.  This is the reasone for
  // the additional CACHE_LINE + 1 memory in the malloc above [11]
  while (((int) ea) & 0x7f)
      ea++;

  memset(ea,0, XDR_Memory_Block*sizeof(int));

  // define the group and scheduling policy
  // SCHED_RR and SCHED_FIFO are not tested
  program.group = spe_create_group (SCHED_OTHER, 0, 1);
  if (program.group == NULL) {
    perror("spe_create_group()\n");
    return -1;
  }

  if (spe_group_max (program.group) < SPE_COUNT) {
    perror("spe_group_max()\n");
    return -1;
  }

#ifdef VERBOSE
  printf("%s compiled on %s at %s\n",__FILE__,__DATE__,__TIME__);
  printf("ea = %p  spe_ea_block = %p\n",(void*)ea, (void*)
spe_ea_block);
#endif

  //  Load the structures with the block data of the memory they will
  //  operate on
  for (i = 0; i < SPE_COUNT; i++) {
    memset(&(program.task[i]),0,sizeof(program.task[i]));
    program.task[i].spe_ea_block = spe_ea_block ;
    program.task[i].baseAddress = (unsigned int)(ea + spe_ea_block*i);
  }
```

```
  // Launch the SPEs
  for (i = 0; i < SPE_COUNT; i++) {
    program.sid[i] = spe_create_thread (program.group, &spe,
        (unsigned long long *) &(program.task[i]), NULL, -1, 0);
    if (program.sid[i] == NULL) {
      perror("spe_create_thread()\n");
      exit (-1);
    }
  }

  //  Now wait for all SPEs to complete and get the return status.
  //  We do not use the status in the current implementation, but
  //  I will in future work.
  for (i=0; i<SPE_COUNT; i++){
      spe_wait(program.sid[i], &(program.status[i]), 0);
  }

#ifdef VERIFY_SUM
  unsigned long long sum = 0;
  for (i=0; i<XDR_Memory_Block; i++) {
    sum += ea[i];
  }
  printf("sum = %llu\n", sum);
#endif

  return 0;
}
```

7.3.3 SPE

```
#include <cbe_mfc.h>
#include <spu_mfcio.h>
#include <stdio.h>
#include "../header.h"

// 32 single bit tags are available to designate the buffer to hold for
// DMA completion.  We will arbitrarily choose 8 for the task blocks,
// And 1 and 2 for the ping pong buffers

#define PING 0x01
#define PONG 0x02
#define TASK_BLOCK_TAG 0x08
#define DMA_BLOCK_SIZE 16384

int pingpong[DMA_BLOCK_SIZE/2] __attribute__ ((aligned (128)));

int *pingpong_pointer[2];

int iterations = 0;
unsigned loopCount  = 0;
unsigned j  = 0;
unsigned k  = 0;
```

```
task_t task __attribute__ ((aligned (128)));

void dma(int *dest) {
  unsigned i;
  vector unsigned int *buff;
  vector unsigned int increment = (vector unsigned int) {1, 1, 1, 1};
buff = (vector unsigned int *) dest;

#ifdef UNROLLED
    loopCount=8;
#else
    loopCount=1024;
#endif

j = (int)dest;

  for (i=0; i<loopCount; i++) {

// Use this to remove branching when not testing that.
#ifdef BRANCHING

#if 0
#endif
    if(j==489292UL){
        printf("IF");
        j = 1231;
        j = j*i;
        j += 21;
        j = j*71;
        k += j*i+32421;
    }
    else{
        j= j>>2;
        j = j ^ (int)dest;
    }
    if(j==847294UL){
        printf("IF");
        j = 1232;
        j = j*i;
        j += 8222;
        j = j*728;
        k += j*i+32422;
    }
    else{
        j= j<<3;
        j = j ^ (int)dest;
    }
```

```
    if(j==7877695UL){
        printf("IF");
        j = 1283;
        j = j*i;
        j += 28;
        j = j*733;
        k += j*i+32423;
    }
    else{
        j = i + j + 8774;
        j = j^(int)dest;
    }


    if(j==82293UL){
        printf("IF");
        j = 123343;
        j = j*i;
        j += 2888;
        j = j*73847;
        k += j*i+328423;
    }
    else{
        j= j<<3;
        j = j ^ (int)dest;
    }
#if 0
#endif
#endif


#ifdef UNROLLED
    buff[0]   = spu_add(buff[0],  increment);
    buff[1]   = spu_add(buff[1],  increment);
    buff[2]   = spu_add(buff[2],  increment);
    buff[3]   = spu_add(buff[3],  increment);
    buff[4]   = spu_add(buff[4],  increment);
    buff[5]   = spu_add(buff[5],  increment);
    buff[6]   = spu_add(buff[6],  increment);
    buff[7]   = spu_add(buff[7],  increment);
    buff[8]   = spu_add(buff[8],  increment);
    buff[9]   = spu_add(buff[9],  increment);
    buff[10]  = spu_add(buff[10], increment);
    buff[11]  = spu_add(buff[11], increment);
    buff[12]  = spu_add(buff[12], increment);
    buff[13]  = spu_add(buff[13], increment);
    buff[14]  = spu_add(buff[14], increment);
    buff[15]  = spu_add(buff[15], increment);
    buff[16]  = spu_add(buff[16], increment);
    buff[17]  = spu_add(buff[17], increment);
    buff[18]  = spu_add(buff[18], increment);
    buff[19]  = spu_add(buff[19], increment);
    buff[20]  = spu_add(buff[20], increment);
    buff[21]  = spu_add(buff[21], increment);
```

```
buff[22]  = spu_add(buff[22], increment);
buff[23]  = spu_add(buff[23], increment);
buff[24]  = spu_add(buff[24], increment);
buff[25]  = spu_add(buff[25], increment);
buff[26]  = spu_add(buff[26], increment);
buff[27]  = spu_add(buff[27], increment);
buff[28]  = spu_add(buff[28], increment);
buff[29]  = spu_add(buff[29], increment);
buff[30]  = spu_add(buff[30], increment);
buff[31]  = spu_add(buff[31], increment);
buff[32]  = spu_add(buff[32], increment);
buff[33]  = spu_add(buff[33], increment);
buff[34]  = spu_add(buff[34], increment);
buff[35]  = spu_add(buff[35], increment);
buff[36]  = spu_add(buff[36], increment);
buff[37]  = spu_add(buff[37], increment);
buff[38]  = spu_add(buff[38], increment);
buff[39]  = spu_add(buff[39], increment);
buff[40]  = spu_add(buff[40], increment);
buff[41]  = spu_add(buff[41], increment);
buff[42]  = spu_add(buff[42], increment);
buff[43]  = spu_add(buff[43], increment);
buff[44]  = spu_add(buff[44], increment);
buff[45]  = spu_add(buff[45], increment);
buff[46]  = spu_add(buff[46], increment);
buff[47]  = spu_add(buff[47], increment);
buff[48]  = spu_add(buff[48], increment);
buff[49]  = spu_add(buff[49], increment);
buff[50]  = spu_add(buff[50], increment);
buff[51]  = spu_add(buff[51], increment);
buff[52]  = spu_add(buff[52], increment);
buff[53]  = spu_add(buff[53], increment);
buff[54]  = spu_add(buff[54], increment);
buff[55]  = spu_add(buff[55], increment);
buff[56]  = spu_add(buff[56], increment);
buff[57]  = spu_add(buff[57], increment);
buff[58]  = spu_add(buff[58], increment);
buff[59]  = spu_add(buff[59], increment);
buff[60]  = spu_add(buff[60], increment);
buff[61]  = spu_add(buff[61], increment);
buff[62]  = spu_add(buff[62], increment);
buff[63]  = spu_add(buff[63], increment);
buff[64]  = spu_add(buff[64], increment);
buff[65]  = spu_add(buff[65], increment);
buff[66]  = spu_add(buff[66], increment);
buff[67]  = spu_add(buff[67], increment);
buff[68]  = spu_add(buff[68], increment);
buff[69]  = spu_add(buff[69], increment);
buff[70]  = spu_add(buff[70], increment);
buff[71]  = spu_add(buff[71], increment);
buff[72]  = spu_add(buff[72], increment);
buff[73]  = spu_add(buff[73], increment);
buff[74]  = spu_add(buff[74], increment);
buff[75]  = spu_add(buff[75], increment);
```

```
buff[76]  = spu_add(buff[76], increment);
buff[77]  = spu_add(buff[77], increment);
buff[78]  = spu_add(buff[78], increment);
buff[79]  = spu_add(buff[79], increment);
buff[80]  = spu_add(buff[80], increment);
buff[81]  = spu_add(buff[81], increment);
buff[82]  = spu_add(buff[82], increment);
buff[83]  = spu_add(buff[83], increment);
buff[84]  = spu_add(buff[84], increment);
buff[85]  = spu_add(buff[85], increment);
buff[86]  = spu_add(buff[86], increment);
buff[87]  = spu_add(buff[87], increment);
buff[88]  = spu_add(buff[88], increment);
buff[89]  = spu_add(buff[89], increment);
buff[90]  = spu_add(buff[90], increment);
buff[91]  = spu_add(buff[91], increment);
buff[92]  = spu_add(buff[92], increment);
buff[93]  = spu_add(buff[93], increment);
buff[94]  = spu_add(buff[94], increment);
buff[95]  = spu_add(buff[95], increment);
buff[96]  = spu_add(buff[96], increment);
buff[97]  = spu_add(buff[97], increment);
buff[98]  = spu_add(buff[98], increment);
buff[99]  = spu_add(buff[99], increment);
buff[100] = spu_add(buff[100], increment);
buff[101] = spu_add(buff[101], increment);
buff[102] = spu_add(buff[102], increment);
buff[103] = spu_add(buff[103], increment);
buff[104] = spu_add(buff[104], increment);
buff[105] = spu_add(buff[105], increment);
buff[106] = spu_add(buff[106], increment);
buff[107] = spu_add(buff[107], increment);
buff[108] = spu_add(buff[108], increment);
buff[109] = spu_add(buff[109], increment);
buff[110] = spu_add(buff[110], increment);
buff[111] = spu_add(buff[111], increment);
buff[112] = spu_add(buff[112], increment);
buff[113] = spu_add(buff[113], increment);
buff[114] = spu_add(buff[114], increment);
buff[115] = spu_add(buff[115], increment);
buff[116] = spu_add(buff[116], increment);
buff[117] = spu_add(buff[117], increment);
buff[118] = spu_add(buff[118], increment);
buff[119] = spu_add(buff[119], increment);
buff[120] = spu_add(buff[120], increment);
buff[121] = spu_add(buff[121], increment);
buff[122] = spu_add(buff[122], increment);
buff[123] = spu_add(buff[123], increment);
buff[124] = spu_add(buff[124], increment);
buff[125] = spu_add(buff[125], increment);
buff[126] = spu_add(buff[126], increment);
buff[127] = spu_add(buff[127], increment);
```

```
#else
    buff[i] = spu_add(buff[i], increment);
#endif

  }//for
}// dma()

// Now we will DMA the buffers and process them until the task
// blocks are completed.  There are many algoritms for doing this
// this is (in my opinion) the best. It is a minor modification
// from the SDK code.  It may be downloaded from [11]
void dma_pingpong(unsigned int addr) {
  int i;
  mfc_get(pingpong_pointer[0], addr, DMA_BLOCK_SIZE, PING, 0, 0);
  for (i=1; i<iterations; i++) {
    // Set the tag mask for the buffer to wait for DMA completion.
    mfc_write_tag_mask(1<<(PING+(i&1)));
    mfc_read_tag_status_all();
    mfc_get(pingpong_pointer[i&1], addr+DMA_BLOCK_SIZE*i,
           DMA_BLOCK_SIZE, PING+(i&1), 0, 0);
    // Set the tag mask for the buffer to wait for DMA completion.
    mfc_write_tag_mask(1<<(PONG-(i&1)));
    mfc_read_tag_status_all();
    dma(pingpong_pointer[(i-1)&1]);
    mfc_put(pingpong_pointer[(i-1)&1], addr+DMA_BLOCK_SIZE*(i-1),
        DMA_BLOCK_SIZE, PONG-(i&1), 0, 0);
  }
  // Now that we have broken out of the loop, we have to do
  // one last time for the last buffer
  mfc_write_tag_mask(1<<PONG);
  mfc_read_tag_status_all();
  dma(pingpong_pointer[1]);
  mfc_put(pingpong_pointer[1], addr+DMA_BLOCK_SIZE*(iterations-1),
     DMA_BLOCK_SIZE, PONG, 0, 0);
  // Now that DMA is completing, we must wait for both buffers
  mfc_write_tag_mask((1<<PING)|(1<<PONG));
  mfc_read_tag_status_all();
}
```

```
int
main(unsigned long long speid,
     unsigned long long argp, unsigned long long  envp) {

  speid = envp = 0;

  // Get the task blocks.  A SPE can't recieve an argument
  // pthread style, so we must get it manually
  mfc_get(&task, (unsigned)argp, sizeof(task), TASK_BLOCK_TAG, 0, 0);
  mfc_write_tag_mask(1<<TASK_BLOCK_TAG);
  mfc_read_tag_status_all();

  iterations = task.spe_ea_block / (DMA_BLOCK_SIZE / sizeof(int));

  //  Set the pointers,  Once again, an SDK idiom [11]
  pingpong_pointer[0] = &pingpong[0];
  pingpong_pointer[1] = &pingpong[4096];

  int i;
  for(i = 0; i < 128; i++){
    dma_pingpong(task.baseAddress);
  }
  return 0;
}
```

8 Acronyms

CAB Cell Accelerator Board

CBE Cell Broadband Engine

CPI Cycle per Instruction

DMA Direct Memory Access

EIB Element Interconnect Bus

FSS Full System Simulator.

ILP Instruction Level Parallelism

ISA Instruction Set Architecture

MFC Memory Flow Controller

PPE PowerPC Processing Element

SIMD Single Instruction, Multiple Data

SoC System on a Chip

SPE Synergistic Processing Element (Sometimes referred to by IBM as the SIMD Processing Element). The latter is more properly descriptive.

VLIW Very Long Instruction Word

RISC Reduced Instruction Set Computing

WAR Write after Read

WAW Write after Write

WYSIWYG What you see is what you get

XDR Rambus Proprietary Extreme Data Rate DRAM

9 References

[1] D.A. Patterson and J.L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, 3rd ed. edition, 2003.

[2] IBM , “Cell Broadband Engine Programming Tutorial”, (October 2005)

[3] Samuel Williams, John Shalf, Leonid Oliker et al, “The Potential of the Cell Processor for Scientific Computing”, Lawrence Berkeley National Laboratory 2005

[4] HP Technology Brief , “Characterizing x86 processors for industry-standard servers: AMD Opteron and Intel Xeon”, (October 2005)

[5] Generally attributed to Benjamin Disraeli.

[6] D. Berlind , “AMD’s no angel, but Intel’s public usage of benchmark data is feloniously misleading”, March 2005; http://blogs.zdnet.com/Berlind/?p=366

[7] Tom Krazit “Chipmakers admit: Your power may vary”, http://news.com.com/Chipmakers+admit+Your+power+may+vary/2100-1006_3-6082352.html

[8] R. Chun, CS247 Course Reader, Fall 2004

[9] IBM , “Cell Broadband Engine Programming Handbook”, (April 2006)

[10] Thomas Chen et al, “Cell Broadband Engine Architecture and its first implementation” 29 Nov 2005

[11] IBM, http://www.research.ibm.com/cell/

[12] A. Fog, “Branch prediction in the Pentium family”, Dr. Dobbs Microprocessor resources, http://www.x86.org/articles/branch/branchprediction.htm

[13] J. A. Kahle et al, “Introduction to the Cell Multiprocessor”, 2005

[14] G. Ou, “Strike 3 for AMD hypocrisy on benchmarking”, http://blogs.zdnet.com/Ou/?p=463

[16] Mercury Computer Engineering, “CBE Performance Considerations”, Unpublished.

[17] Donald Rumsfeld, News breif, February 2002

[18] Mercury Computer Engineering ,“SPE Assembly Development Kit (SPEAD-K) Training Class”, Internal Training Presentation, 2007

[19] Mercury Computer Engineering ,MCF Training Class, Boston MA, Training Presentation, February 2008

[20] Cavium Networks , “OCTEON Multi-Core Processor Family”, http://www.caviumnetworks.com/OCTEON-Plus_CN58XX.html

[21] Intel Corporation , “Don’t Judge a CPU only by its GHz”, (April 2007)

[22] Caliri ,Gregory , “Introduction to Analytical Modeling”, (May 2000)

[23] Jerry Banks, John S. Carson, Barry L. Nelson, and David M. Nicol. *Discrete-Event System Simulation*. Prentice-Hall, Upper Saddle River, NJ, third edition, 2000.