

# Hardware for Exploiting Data Level Parallelism

## Written Preliminary Examination II

Andrew D. Hilton

### Abstract

Current trends in performance improvement favor increasing parallelism over increasing the clock rate for a variety of reasons. Although many forms of parallelism exist, Data Level Parallelism (DLP) is one form which scales well and is particularly useful in scientific as well as media applications. This paper examines four modern architectures for exploiting DLP in media applications: VIRAM, Scale, Imagine, and Cell, with focus on the architectures' approaches to dealing with the problems of increasing memory latency, capabilities to handle code which is partially data parallel, programmability and applicability to other domains.

## 1 Introduction

Current trends in performance improvement favor increasing parallelism over increasing the clock rate. Whereas increasing the clock frequency has been popular in the past, it suffers from three major problems. First, power consumption increases quadratically with increasing clock frequency. This problem makes clock rate increases particularly unattractive for laptops and other devices in which heat dissipation or battery life are important considerations. Secondly, since the speed of main memory is not affected by the CPU clock, increases to the CPU clock frequency make accesses to main memory appear slower to the CPU. This "memory wall" means that clock rate increases strongly experience diminishing marginal returns for actual performance gained. Third, fast clock rate and completing a large number of instructions per cycle (IPC) are generally opposed goals. Increasing the pipeline depth is an easy way to increase clock rate, but it introduces more hazards, decreasing IPC. Likewise, achieving high ILP generally requires large, complex structures, and therefore a slow clock rate.

Improving performance by exploiting more parallelism, however, does not suffer from these problems. Generally, power consumption scales linearly when increasing the number of copies of some hardware resource, making parallelism a much more power efficient alternative. Likewise, memory latency typically stays constant while increasing the number of instructions being executed in parallel. Although there could be some slowdown due to contention, some forms of parallelism significantly help in this area by allowing useful work to be done while waiting for memory.

Parallelism comes in several different forms. The most fine-grained form is Instruction Level Parallelism (ILP). ILP is the form of parallelism exploited by superscalar and VLIW architectures, in which multiple independent operations are executed in parallel. Whereas this form of parallelism is very general, it is hard to obtain high degrees of parallelism. The widest attempted superscalar was the cancelled Alpha 21464, which was 8 wide, with more typical widths being approximately 4. This difficulty with width results from several factors, such as the  $N^2$  nature of bypassing and dependency checking, the difficulties of fetching past taken branches, and the number of ports needed on the register file. Additionally, finding many independent operations to execute simultaneously— whether dynamically in a superscalar, or statically in a VLIW— requires a very large scheduling window, which can be a difficult task, not only to build, but also to fill with useful instructions.

Another form of parallelism is Thread Level Parallelism (TLP). TLP is a much coarser form of parallelism than ILP. Instead of trying to execute multiple independent instructions simultaneously, TLP allows for the execution of multiple independent tasks simultaneously. A single processor can support multiple threads, and can switch between them in a variety of fashions. Running multiple threads on one processor can provide the processor with useful work to do on one thread while another handles a lengthy cache miss. Although this will not increase the performance of any one thread (and will likely decrease it), it can improve overall throughput. Another way in which TLP can be

exploited is with multiple processors, each of which can run one or more threads. The downside to TLP is that the software must be explicitly written with multiple threads; there is presently no way to take a single threaded program and have the hardware automatically multi-thread it. Additionally, whereas techniques for ILP can be used to exploit TLP- for example, a superscalar can do Simultaneous Multithreading, in which it executes instructions from a variety of threads at once- it is much harder to improve single thread performance using hardware that supports TLP.

The remainder of this discussion focuses on Data Level Parallelism (DLP). DLP is parallelism in which the same operations are being performed on different pieces of data at the same time. Because the operations are on different data, they are known to be independent, which means that, unlike ILP,  $N^2$  dependence checking is not needed. Furthermore, bypassing complexity and register file size scale linearly for DLP whereas they scale quadratically for ILP. The bypassing complexity is simpler because the output of a functional unit only needs to be bypassed to itself for DLP- as a subsequent operation can work on the same element. The register file does not require additional ports (which require additional bitlines and wordlines), it only requires increasing the width of the existing ports (which requires only additional bitlines). However, hardware cannot easily find DLP to exploit it in a program as it runs, so either the compiler, programmer, or both must perform extra work to specify when data parallelism exists.

## 2 DLP Architectures

Originally, Data Level Parallelism was exploited by the use of vectors, primarily in scientific super computing. A typical vector would consist of several (on the order of 32) floating point numbers grouped together. The processor would have a separate vector register file to hold these larger data items, and would have special instructions to load, store, and operate on vector data. Whereas 32 parallel floating point operations would yield very high performance, it would also be very resource intensive (i.e. space and power). Scientific vector processors would typically provide fewer copies of each functional unit, called lanes, than the number of elements they supported. The hardware would then pipeline the vector operations, sending an element down each lane each cycle until the operation completed. This pipelining was also combined with a technique known as “chaining”, in which back to back operations that utilized different types of functional units would overlap- while the first operation worked on the  $(N+1)^{th}$  batch of elements, the second operation would work on the  $N^{th}$  batch of elements. For example, a 4 lane processor might have a load produce 4 vector elements on 1 cycle, which would then be sent to the vector adder on the next cycle while 4 more elements would be loaded. These techniques allowed for fewer actual resources than the vector length, while still gaining significant performance improvements.

More recently, a new application for DLP has emerged- short vectors for multimedia. Multimedia applications commonly use short integer vectors, such as pixels which consist of 4 separate but contiguous integers, each usually one byte long. This application of DLP has led to architectures which support short vectors, or subword parallelism, in which one large integer can be treated as several smaller integers on which parallel operations are performed. Subword parallelism is attractive from a hardware simplicity standpoint as the parallel operations can be carried out on a typical ALU with appropriate carries selectively disabled. In some cases, the datapaths for subword parallelism are wider than the typical datapaths in the machine, to allow more elements to be processed in parallel. A similar technique can be used for short floating point vectors.

VIRAM, Scale, Imagine and Cell are four architectures designed to exploit data level parallelism for multimedia applications. The remainder of this section provides an overview of each of these architectures.

### 2.1 VIRAM

VIRAM [4] is a vector MIPS co-processor, which has two significant design aspects. First, it combines the technique of having functional units in parallel lanes operate on each element, traditionally used by scientific vectors, with the technique of using subword parallelism more recently introduced for multimedia. VIRAM has 4, 64-bit wide

vector lanes, and a vector register file with 32 entries, each of which is 256 bits long. These aspects of VIRAM make it seem like a very short vector length implementation of scientific vectors. However, VIRAM's 64-bit wide lanes are also designed to apply subword parallelism. They can be used for 4 16-bit, 2 32-bit, or 1 64 bit operation. VIRAM's implementation of subword parallelism differs slightly from the common implementations (i.e., MMX) in that VIRAM's operation width is set via a control register, whereas most implementations specify the operation width in the instruction opcode.

The second interesting major design choice in VIRAM is its use of large, on chip DRAM instead of traditional caches. VIRAM has 13MB of software managed DRAM on chip and does not use any caches. Because the memory is on chip, it provides much higher bandwidth than a traditional memory hierarchy, which is beneficial for vector loads and stores, as they move large amounts of data.

Software for VIRAM is compiled using a modified version of the PDGCS compiler for Cray supercomputers which compiles C, C++, and Fortran. In order to automatically vectorize C and C++, the programmer must insert extra annotations to tell the compiler when pointers are guaranteed not to alias each other, so that independence can be determined [3]. In addition to standard vector operations, VIRAM provides some "permutation" operations to aid in reduction operations. The simplest of these copies the later half of one vector register into the early half of another, which is useful for summing elements of a vector among other things. The VIRAM compiler is automatically able to detect cases where it can use this permutation instruction for several common operations. The other permutation operations are more complicated, and are useful specifically in Fast Fourier Transforms. These more complex permutations are not directly exploited by the compiler, but are used in hand coded assembly FFT libraries.

## 2.2 Scale

Scale [5, 6] is a "Vector-Thread" architecture, an architecture which exploits DLP, but in a manner that looks similar to TLP. Vector-Thread architectures have a control processor as well as several "virtual processors" (VPs). The control processor executes conventional code, but unlike the normal processors in the other architectures, a Vector-Thread coprocessor also executes vector loads and stores. As far as the software is concerned, each VP corresponds to a vector element in a conventional vector processor. The number of virtual processors is the maximum vector length of the machine. The control processor sets up state for each of the virtual processors, then issues a *vector-fetch* command, which tells them to start executing data parallel code. The virtual processors execute "atomic instruction blocks" (AIBs), which are basically basic blocks, although if there is no explicit jump to a next block at the end, the vector-thread ends. Once the virtual processors have started processing a set of vector-threads, their execution becomes relatively independent. This setup allows much more flexibility than conventional vector processing. In addition to the "normal" things done in a vector system, virtual processors can independently execute conditional branches, or even loops. One implication of this layout is that the virtual processors could be used to run truly separate threads if the application has more TLP than DLP. Scale is a specific instantiation of this Vector-Thread paradigm.

Although the VPs give the illusion of a very long vector length to the software, in actuality, there are fewer physical vector lanes than there are virtual processors. The virtual processors are striped across the physical vector lanes, and have their execution interleaved at AIB granularity. Although done at a larger granularity, this approach resembles pipelining more elements than there are lanes on traditional scientific vector systems. Both approaches allow for longer vector lengths than there are parallel resources. For Scale, this has the advantage that it allows vector related operations executed by the control processor (such as vector loads, vector fetches, etc) to perform more work per instruction. Each physical lane is comprised of 4 heterogeneous execution "clusters". Each cluster contains a register file, ALU, and hardware to communicate between clusters, as well as with the next and previous lanes. One cluster per lane has a load/store unit, another supports branches, while a third has an integer multiplier/divider. Each assembly instruction specifies the cluster on which it is to execute, and must read its inputs from the registers on the cluster it is executing on. It may, however, send its results to another cluster. Execution of code on any given cluster is in order,

however, the only synchronization across clusters occurs when one cluster needs to use data it is waiting to receive from some other cluster. This design gives Scale an unusual form of ILP in addition to its DLP.

Another interesting feature of this design is that, unlike conventional vector processors, Scale is able to parallelize loops with cross iteration dependences. In code with loops containing cross iteration dependences, the producer iteration executes an instruction which sends the value to the next virtual processor. Likewise, the consuming iteration executes an instruction to receive a value from the previous virtual processor. The hardware has queues between adjacent lanes to handle the transfer of these values. If the virtual processors were to execute truly separate threads, this design could potentially allow for efficient cross thread communication mechanisms, although the unidirectional ring topology would place restrictions on the communication patterns.

## 2.3 Imagine

Image [2] is a co-processor architecture designed around the concept of streams and kernels. A kernel takes a stream as input, performs some operations on it, and produces another stream as output. In a sense, a stream is a just very long vector: it is the data which the application is processing in a data parallel fashion, but is much longer than vectors in other DLP settings, for example, Imagine supports streams up to 32K words long. These streams are very applicable to media related applications, in which large images, videos, or audio clips need to be processed in a data parallel fashion. The application is organized into a set of kernels between which the data flows as streams. Although a single Imagine processor would exploit DLP within a kernel, but serialize one kernel after another, multiple Imagine processors could be used to set up a kernel pipeline, effectively exploiting TLP as well.

In order to support such long “vectors”, Imagine introduces a structure called the “Stream Register File” (SRF). The SRF is an SRAM which holds 32K words, organized into 32 word blocks. A block is the granularity at which the space is allocated to individual stream. The SRF is used to hold the stream data so that it is available at a higher bandwidth than if it were in main memory. To process the data, Imagine has 8 identical arithmetic clusters, each of which have 8 functional units and a local register file (LRF). The 8 clusters execute operations in a data parallel fashion with VLIW style instructions. The SRF has a single 32 word wide port which is connected to 22 “stream buffers”, each of which hold 2 blocks. When data needs to be moved from memory into the SRF, it is first loaded into the stream buffer. The data is only moved into the SRF when an entire block is available in the stream buffer. Likewise, when data needs to be moved from the SRF to the LRFs, an entire block is moved into a stream buffer, and the data is then copied into the LRFs from there. This organization gives the speed and area benefits of a single ported SRAM, while providing the illusion of multiple simultaneous accesses.

Most vector processors support loads and stores of non-contiguous vectors, typically in a “strided” or “indexed” fashion. In vector processors with a conventional cache design, moving the data from the L2 cache to the vector register files poses a significant challenge in these cases, as the striding or indexing might cause bank conflicts. Imagine supports these modes, as well as “bit-reversed record addressing”. Imagine’s memory hierarchy’s design simplifies these problems, as the data for the whole stream can be loaded into a stream buffer, where it is laid out into a contiguous format before being put into the SRF.

One observation that Imagine’s designers make is that media applications written with streams use approximately an order of magnitude more bandwidth at each level of the memory hierarchy, and therefore the Image design exploits this fact. The majority of data movement during kernel execution is between the LRF and the functional units, which provides the highest bandwidth (544 GB/s). The next level of the memory hierarchy— between the LRF and the SRF— is only needed when the clusters finish one set of stream elements and need to work on the next, so this level provides an order of magnitude lower bandwidth (32 GB/s). Finally, accesses to main memory are exceedingly rare, as they are needed only to load the initial input, to store the final, or to spill streams which cannot fit in the 128KB SRF, so this provides yet another order of magnitude lower bandwidth (2.67 GB/s).

## 2.4 Cell

IBM's Cell processor [1] provides many forms of parallelism. A Cell chip has one "Power processor element" (PPE), and 8 "synergistic processor elements" (SPEs). The ISA for the PPE is a typical 64-bit Power Architecture with SIMD extensions. For the most part, the microarchitecture of the PPE is relatively normal. The PPE is a 2 threaded, dual issue, in-order 23 stage pipeline, with a branch predictor and two levels of caches. Most of the interesting design aspects of the PPE are focused on obtaining a fast clock (11FO4) in a relatively power efficient manner, which is not relevant to this analysis.

The SPEs are the aspect of the chip which makes the Cell processor interesting from a parallelism standpoint. In some ways, the fact that each of the SPEs is a complete and independent processor makes the Cell resemble a 9 core CMP, however, the SPEs are very different from conventional processors. One major difference is that the SPEs do not have caches, and therefore no cache coherence. Instead the SPEs each have a 256KB SRAM, whose contents are controlled by explicit DMA commands from the software. Although the SPEs do not have coherent caches, they are able to communicate with each other, and with the PPE. The SPEs are also unconventional in their exceedingly large register file (128 registers, each 128 bits), complete lack of branch predictor (18 cycle penalty, but hint instructions are available), and the fact that every instruction uses subword parallelism (from 2x 64-bit to 128x 1 bit).

## 3 Comparison of the Architectures

Each of these architectures attempts to exploit data level parallelism to obtain performance in a power efficient way. Power efficiency is especially important for the three architectures which are intended specifically for embedded systems (VIRAM, Scale, and Imagine), and is also a top level design goal of Cell. For the most part, all of the architectures typically target, or include in the set of applications they target, multimedia processing. With these similar goals in mind, the architectures exhibit surprisingly different designs. These comparisons are summarized in Table 1.

	VIRAM	Scale	Imagine	Cell
<b>Approach to Memory</b>				
<b>Memory Hierarchy</b>	13MB Software managed DRAM	Caches	SRF	256KB Software managed SRAM
<b>Strategy</b>	Low latency, on chip memory	Amortize latency over many elements	Work in streams instead of words	Low latency, on chip memory
<b>Handling difficult to vectorize code</b>				
<b>Cross Iteration Dependences</b>	Permutations	FIFO queues	Communication unit	EIB
<b>Conditionals</b>	Mask bits	Branches	???	Predication and branches
<b>Outer loop</b>	No	Yes	No	Yes
<b>Programmability</b>				
<b>Compilation</b>	Compiler hints	???	Streaming languages	???
<b>Other Problems</b>	Software managed memory Reductions may change values			Software managed memory

Table 1: Summary of Comparisons

### 3.1 Approach to memory

One of the most significant impediments to performance is the “memory wall”, the increasing relative latency of memory accesses to the speed of the processor. Of the four architectures discussed, only Scale uses a conventional cache hierarchy. Scale’s main approach to dealing with the “memory wall” is to use vector load and vector stores in order to obtain large quantities of data at once, allowing it to exploit locality and to amortize long latencies. This approach to memory makes Scale’s ability to deal with loops which have cross iteration dependencies especially important, as it allows for vector load and store commands to still be used even when the computation is not fully parallel.

At the opposite extreme, VIRAM proposes to simply put all of the memory that the application will need <sup>1</sup> on chip so that it can be accessed quickly. With VIRAM’s slow clock and a lengthy pipeline for vector load/store instructions (15 stages), all memory accesses occur relatively quickly. Additionally, memory accesses have uniform latency, whereas with a conventional cache hierarchy, the latency can vary depending on what level of the memory system has it. The lack of variability in memory accesses may simplify the hardware, as it does not need to detect and stall for cache misses. The downside to such a design is that if an application cannot fit everything into the on chip memory, extra software complexity is required to ensure that the data is transferred in a timely fashion.

Cell’s approach is similar to VIRAM’s- each SPE has its own local RAM, which is software managed. The SPEs’ local RAMs hold not only the data, but also the code for the SPE to execute, and are much smaller than the on chip RAM in VIRAM. Although the smaller RAM allows for single cycle access, it means that applications are more likely to need to perform significant management of the space.

Like Scale, Imagine’s solution to the memory problem is primarily to combine several memory accesses into one instruction in order to allow high bandwidth to amortize long latencies. Unlike Scale, Imagine uses special purpose hardware (the SRF) instead of a conventional cache to hold the data between memory and the register files. Both Cell and Imagine share the “order of magnitude” bandwidth scaling between each level of memory. Cell’s off chip memory bandwidth is 12.8 GB/s, compared to the 25.6 GB/s between processing units on the chip, and 32 B/cycle into each of the 8 SPEs’ register files. This similarity makes sense, as it means that Cell’s bandwidth is available in similar proportions to Imagine’s if Cell were to be used for streaming applications.

### 3.2 Handling difficult to vectorize code

Sometimes the body of a loop will be partially data parallel, such as when there are some cross iteration dependences, but most of the operations are data parallel. An architecture that can exploit the parallelism that does exist, while correctly handling the non parallel aspects can have a significant performance advantage over an architecture which is prevented from gaining any parallelism due to the problematic code. VIRAM provides a few special permutation operations to help with some common cases of cross iteration dependences. Although this approach allows for simple hardware and covers many common cases, it could also result in many lost chances for parallelism. Scale provides a more flexible solution- it allows a virtual processor to send values to the next VP-allowing values to be passed from one loop iteration to the next. This approach increases the hardware complexity slightly, as it requires FIFO queues between each of the vector lanes. This complexity increase seems relatively small compared to the ability to exploit more parallelism. Cell and Imagine both allow any execution unit (SPE or cluster) to communicate with any other- via DMA transfers in Cell, or via the communication functional unit in each execution cluster in Imagine. This solution provides the most flexibility, but with the most hardware complexity. Most of this complexity is already used for other purposes in Cell (since it uses the DMA hardware), but Imagine dedicates an entire functional unit in each cluster specifically to this task.

---

<sup>1</sup>VIRAM allows explicit software data transfer to and from other off-chip memory, however, it looks like all of their performance evaluation was done on applications which fit in the on chip DRAM

Another difficulty that can arise in exploiting data parallelism is conditional branches inside the body of a loop. VIRAM provides the ability to execute vector operations under a bit mask specifying which elements should actually be operated on. This approach has the same benefits and disadvantages as general forms of predication: for conditionals with very small control dependent regions, predication is beneficial, however, for conditionals with large control dependent regions, much time and power will be wasted fetching and executing the instructions whose predicates are false. The power disadvantage is exacerbated by the fact that VIRAM executes the operations for all elements, but does not write back the values for those whose corresponding conditional mask bits are false. Scale's design allows it great flexibility in executing conditional branches. Since each VP can actually execute conditional jumps, Scale is able to execute exactly the code that needs to be executed. Scale does not need to predict these branches, as the operations are not executing on a conventional pipeline- instead, when the virtual processor reaches the jump (ending the AIB), the next virtual processor which is on the same physical lane will be given a chance to run an AIB. This organization means that even if the target AIB is not in the AIB cache, the execution of other processors on the same lane will occur while the miss is handled. Cell's SPEs are full fledged processors, so they allow conditional jump instructions, however, there is no branch prediction, and the penalty for not correctly predicting a branch is 18 cycles. Instead of the hardware having a branch predictor, the software is able to provide hint instructions that tell the SPE where an upcoming branch is going to go. The Cell SPE's ISA also provides a "three source bitwise select" operation, to reduce the need for branches. The combination of true conditionals and conditional move instructions allows for the software to choose what form of conditional is most appropriate to its needs. Both Scale and Cell can handle not only interior branches, but also loops, allowing them to exploit outer loop level parallelism. The Imagine paper does not give details on how it deals with conditional branches, although it states that the communication unit is involved in conditional streams.

### 3.3 Programmability

Although architectures that exploit data parallelism provide much potential for high performance, either the compiler, programmer, or both must find the parallelism in order for the potential performance to be realized. Although automatic vectorization is generally hard for languages in which aliasing can occur, VIRAM side steps the problem by allowing the programmer to add annotations to help the compiler. This approach, combined with the fact that VIRAM only uses vectors for truly data parallel code, works well, as programmers only need to do a small amount of extra work, and the compiler can do the rest. One downside to this approach is that the programmer does not know, without looking at the generated assembly, where vectorization is being performed and where it is not. If vectorization is not being performed where the programmer expects, it may be difficult for the programmer to figure out how to convince the compiler to do so. If VIRAM were to be used on memory intensive applications which do not fit in its on chip memory, either the compiler or programmer would have to deal with the additional task of inserting instructions to move data back and forth between off chip and on chip memory. A potential surprise for unsuspecting programmers is that use of the permutation instructions reorders the operations being performed in a way that could change the results. The paper acknowledges the reordering, but states it will only affect exception behavior or lower bits. Unfortunately, in some cases, reordering floating point operations can have significant impact on the results:  $3 \times 10^{23} + (-3 \times 10^{23}) + 42 = 42$ , while  $3 \times 10^{23} + 42 + (-3 \times 10^{23}) = 0$ . Changes in the results of a program between vectorized and non-vectorized versions of the code could significantly confuse a programmer.

Imagine utilizes a different programming paradigm- kernels being applied to streams- in order to make the data parallelism explicit. Because this programming paradigm seems well suited to the multimedia applications which Imagine is designed to run, once application programmers have learned to program in the streaming paradigm, programming for Imagine should be relatively simple. Even though Imagine's SRF is software managed, its management fits naturally into the programming paradigm- streams are loaded into from memory as they are needed and stored from it to memory as they are completed, so the compiler's job in performing this memory management is

relatively simple.

One Scale paper claims that “Scale was designed to be compiler-friendly, and a C compiler is under development” [5]. Scale’s unusual design allows it to exploit parallelism in outer loops, in loops with cross iteration dependences, or in loops with large conditionals, however, it still needs to have the same guarantees of independence for code that is being executed in a data parallel fashion as other DLP architectures. Furthermore, the ability to exploit parallelism in unusual ways could complicate the compiler’s decisions for how and when to use each feature, especially since compilers cannot easily automatically exploit TLP at present.

With its broad variety of parallelism, Cell’s design is flexible enough to allow it to exploit parallelism in the same manner as any of the other three designs. By treating Cell’s PPE like the control processor in Scale, and treating the SPEs like Virtual Processors, Cell could exploit data level parallelism in the same fashion as Scale. Because the SPEs are independent of each other just like Scale’s VPs are, Cell can use the same techniques to “vectorize” outer loops, loops with conditional branches, or loops with cross iteration dependences. Thinking of the Cell processor this way suggests that loops with no internal branches could be “vectorized” onto the SPEs in such a way that its lack of a branch predictor would not matter.

Cell, can also perform streaming media computations effectively, both in the sense of Imagine streams as well as “spatial streaming”. A single SPE could handle Imagine’s super long vector style of streams by using part of its local SRAM like Imagine’s SRF. However, Cell can also stream data thru many different SPEs to accomplish “spatial streaming”, running one kernel on each SPE and having the data flow between them. This model is the one which the Cell paper refers to when it discusses streaming as a programming model for Cell. Interestingly enough, the Cell paper also points out that rather than moving data between SPEs, the code could be migrated instead, generally resulting in less overall data transfer. Imagine can also accomplish this variety of streaming by connecting several Imagine chips together. These programming model also suggests that stream languages used to program Imagine could be effectively compiled to Cell.

Except for the special reduction operations, VIRAM exploits DLP only on truly vectorizable code. Because the reduction operations are simply rearrangements of data, Cell could implement these with an appropriate combination of SPE communication and its own subword vector permutation operations. Furthermore, if you think of the SPEs as vector lanes for the PPE, then both Cell and VIRAM increase their effective vector length by providing subword parallelism inside a vector lane. This design gives both architectures the flexibility to exploit more parallelism when applications use smaller or less precise datatypes. Because VIRAM is purely a vector processor, treating each SPE as if it were a vector lane allows for the Cell processor to exploit parallelism in the same way as VIRAM, as long as the software can effectively manage the SPE’s local memories. Because Cell has software managed memory like VIRAM, its programmability is also complicated by the issues surrounding it, however, since each SPE’s local memory is much smaller than VIRAM’s memory, these issues are likely to be compounded.

## **4 Applicability to other workloads**

Although all four architectures were designed primarily for multimedia applications, some of them seem like they would handle other types of workloads well. Many of the architectures have the ability to exploit other forms of parallelism besides DLP, or features that could be generally useful. The suitability of each architecture to different workloads is summarized in Table 2.

### **4.1 Scientific Computation**

Scientific computation is where the first exploitation of data level parallelism in the form of vectors appeared. Because scientific computation is well suited for data level parallelism, it would be reasonable to expect that most of these architectures would be reasonably well suited to such tasks.

	<b>VIRAM</b>	<b>Scale</b>	<b>Imagine</b>	<b>Cell</b>
<b>Scientific</b>	+ Vectors prevalent	+ Vectors prevalent + Outerloop parallelism + X-iter deps		+ Vectors prevalent + Outerloop parallelism + X-iter deps - Slow double precision
<b>Server</b>	- No TLP	+ TLP	? Maybe stream DBs	+ TLP
<b>Integer</b>		+ Caches ? Maybe pre-exec threads	+ 8 wide VLIW - Hard to move data thru	? Maybe pre-exec threads
<b>Realtime</b>				+Constant latency ops
<b>All</b>	- Software managed memory		? Streams only	- Software managed memory

Table 2: Suitability to other workloads

VIRAM may experience some complexities as many scientific computations would exceed the on chip memory, and therefore require moving back and forth to off chip memory. If the application's memory access patterns are such that they constantly require data movement on and off chip, performance could be worse than thrashing a typical cache, since many extra instructions would be needed to control the movement. Scientific computations would also likely result from more significant problems due to the reordering of floating point operations discussed previously.

One would expect that Cell would be well suited to physics, as it is an important aspect of games, however, graphics and physics in games typically need only single point precision. Because of this fact, Cell has a much faster implementation of single precision floating point than double precision floating point (6 cycles fully pipelined vs 7 cycles sequentially). This slower implementation of double precision floating point could cause significant performance degradation on scientific applications which need double precision.

If the application could be fit into Imagine's streaming framework, then it could gain performance benefits by running on Imagine. Otherwise, Imagine will not be likely to do much for it. Scale has a lot of potential for scientific applications, since it can not only deal effectively with long vectors, but also exploit more parallelism by providing mechanisms for cross iteration dependences and interior conditional branches. Scale's ability to do outer loop parallelism could also be beneficial, as many scientific applications exploit this form of parallelism with OpenMP or MPI. As Cell can use the same techniques for cross iteration dependences and outer loop parallelism, it would have the benefit in the same way.

## 4.2 Server Workloads

Server workloads are not typically a target for data level parallelism, instead favoring thread level parallelism. VIRAM would therefore, not be at all well suited to a server setting. Imagine would not typically be well suited to a server workload, although one could conceive of expressing database queries as streams, which may have the potential to make a streaming architecture like Imagine well suited to a data base server. Scale and Cell, however, both provide an easy means for exploiting TLP, which could make them useful for running Server workloads.

### 4.3 Integer Applications

Single threaded integer applications (i.e. SpecInt) typically favor instruction level parallelism. VIRAM does strictly vectors, so it provides no benefit to ILP, although some applications would benefit from a chip with as much cache as VIRAM has memory. Although Imagine's 8 wide VLIW clusters may sound like a good fit, they can only access data from the SRF. This restriction means that the program would have to be reorganized to explicitly move data thru the SRF. Because the application would not be utilizing memory in the streaming patterns for which the memory system is designed, memory accesses would incur terrible performance penalties.

Cell's PPE is a typical 2 wide in-order chip with a fast clock, so it would likely obtain decent performance on integer applications. With some cleverness, one might be able to leverage the TLP present in Scale and Cell to run pre-execution threads on the VPs/SPEs. Such a technique would likely be easier in Scale, where both the main processor and the VPs share a cache, so the cache misses generated by the pre-execution threads would automatically be made available to the main thread. In Cell, the SPEs' would have to take extra work to communicate the required data to the PPE. Both architectures would suffer from the fact that the instruction sets on the VPs/SPEs differ from that of the main processor, necessitating special compilation of the pre-execution threads.

Additionally, integer applications benefit significantly from traditional cache hierarchies. Although the main chip in each architecture has a normal cache hierarchy, the vector related hardware only has access to the caches in Scale. The software management of memory for the other architectures could be much more difficult in integer applications than in media applications.

### 4.4 Realtime Systems

Cell's architecture also has significant benefits for real time systems. Cache misses can make a system ill suited for real time systems, as a load instruction may take a few cycles, or many hundreds of cycles to complete. The fact that the SPEs do not have caches means that the latency of operations is very well known, which makes them well suited to controlling a device requiring real-time monitoring. Furthermore, the fact that there are many SPEs on the chip would make it viable to have one chip monitor several devices. This setup would allow the PPE to run non-realtime software, such as code to interact with a user, while having the SPEs deal with the time critical tasks.

## 5 Conclusion

Modern processors are designed to exploit parallelism in the programs they run. Although exploitation of instruction level parallelism is the most ubiquitous, it also incurs the most hardware complexity. Thread level parallelism and data level parallelism can be exploited with simpler hardware, but require additional work by the developer and/or compiler to expose the parallelism. Data level parallelism was originally exploited by scientific applications in the form of long vectors. More recently, subword parallelism has emerged as a means of exploiting DLP for multimedia applications.

VIRAM, Scale, Imagine, and Cell are four architectures designed to run multimedia applications, each of with its own approach to exploiting DLP. Some of these architectures also include support for other forms of parallelism. VIRAM strictly exploits DLP by combining the scientific approach to vectors with subword parallelism. Scale exploits DLP in a way that makes it seem like TLP, which allows Scale to exploit DLP in some ways that are not typical. Imagine deals with streams, which are basically very long vectors, and uses streaming languages to expose the parallelism. Cell exploits many forms of parallelism: it has 9 cores on the chip, all of which are capable of subword parallelism, and 2 wide issue. Cell's design is flexible enough that it can exploit parallelism using the same techniques as any of the other three architectures.

## References

- [1] J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Mauerer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, July/September 2005.
- [2] B. Khailany, W. Dally, U. Kapasai, P. Mattson, J. Namkoong, J. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media processing with streams. *IEEE Micro*, March-April 2001.
- [3] C. Kozyrakis, D. Judd, J. Gebis, S. Williams, D. Patterson, and K. Yelick. Hardware/Compiler codevelopment for an embedded media processor. In *Proceedings of the IEEE, Vol. 89, No 11*, November 2001.
- [4] C. Kozyrakis and D. Patterson. Vector Vs. Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks. In *35th International Symposium on Microarchitecture*, Istanbul, Turkey, November 2002.
- [5] R. Krashinsky, C. Batten, S. Gerding, M. Hampton, B. Pharris, J. Casper, and K. Asanović. The Vector-Thread Architecture. In *31st International Symposium on Computer Architecture*, Munich, Germany, June 2004.
- [6] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanović. The Vector-Thread Architecture. *IEEE Micro*, November-December 2004 2004.