Chapter-1
INTRODUCTION

The increasing importance of multimedia, game applications, and other numerically intensive workloads has generated an upsurge in novel computer architectures tailored for such applications. Such applications include highly parallel codes, such as image processing or game physics, which have high computation and memory requirements. They also include scalar codes, such as networking or game artificial intelligence, for which fast response time and a full-featured programming environment are paramount. Developed with such applications in mind, the CELL processor provides both flexibility and high performance. This Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page.

The first generation CELL processor includes a 64-bit multithreaded Power Processor Element (PPE) with two levels of globally-coherent cache. It supports multiple operating systems including Linux. For additional performance, a CELL processor includes eight Synergistic Processor Elements (SPEs). Each SPE consists of a new processor designed for streaming workloads, a local memory, and a globally-coherent DMA engine. Computations are performed by 128-bit wide Single Instruction Multiple Data (SIMD) functional units. An integrated high bandwidth bus glues together the nine processors and their ports to external memory and IO. In this report, we present the compiler approach to support the heterogeneous parallelism found in the CELL architecture, which includes multiple, heterogeneous processor elements and SIMD units on all processing elements. The proposed approach is implemented as a research prototype in IBM's XL product compiler code base and currently supports the C and Fortran languages. First contribution is a set of compiler techniques that provide high levels of performance for the SPE processors. To achieve high rates of computation at moderate costs in power and area, functionality that is traditionally handled in hardware has been partially offloaded to the compiler, such as memory realignment and branch prediction. We provide techniques to address these new demands in the compiler.

The second contribution is to automatically generate SIMD codes that fully utilize the functional units of the SPEs as well as the VMX unit found on the PPE. The proposed approach minimizes the overhead due to misaligned data streams and is tailored to handle many of the code structures found in multimedia and gaming applications.

There is an average speed up factor of 1.3 for the proposed SPE optimization techniques. When they are combined with SIMD, we achieve average speedup factors of, 9.9 and, on suitable benchmarks.

Chapter-2
CELL ARCHITECTURE

The implementation of a first-generation CELL processor includes a Power Architecture processor and 8 attached processor elements connected by an internal, high bandwidth Element Interconnect Bus (EIB). Figure 2 shows the organization of the CELL elements and the key bandwidths between them. The Power Processor Element (PPE) consists of a 64-bit, Multi - threaded Power Architecture processor with two levels of on-chip cache. The cache preserves global coherence across the system. The processor also supports IBM's Vector Multimedia eXtensions (VMX) [1] to accelerate multimedia applications using its VMX SIMD units.

A major source of compute power is provided by the eight on-chip Synergistic Processor Elements (SPEs) [2, 3]. An SPE consists of a new processor, designed to accelerate media and streaming workloads, its local non-coherent memory, and its globally-coherent DMA engine. Key units and bandwidths are shown in Figure 2b. Nearly all instructions provided by the SPE operate in a SIMD fashion on 128 bits of data representing either 2 64-bit double floats or long integers, 4 32-bit single float or integers, 8 16-bit shorts, or 16 8-bit bytes. Instructions may source up to three 128-bit operands and produce one 128-bit result. The unified register file has 128 registers and supports 6 read and 2 write per cycle. The memory instructions also access 128 bits of data, with the additional constraint that the accessed data must reside at addresses that are multiples of 16 bytes. Namely, the lower 4 bits of the load/store byte addresses are simply ignored. To facilitate the loading/storing of individual values, such as a byte or word, there is additional support to extract/ merge an individual value from/into a 128-bit register.

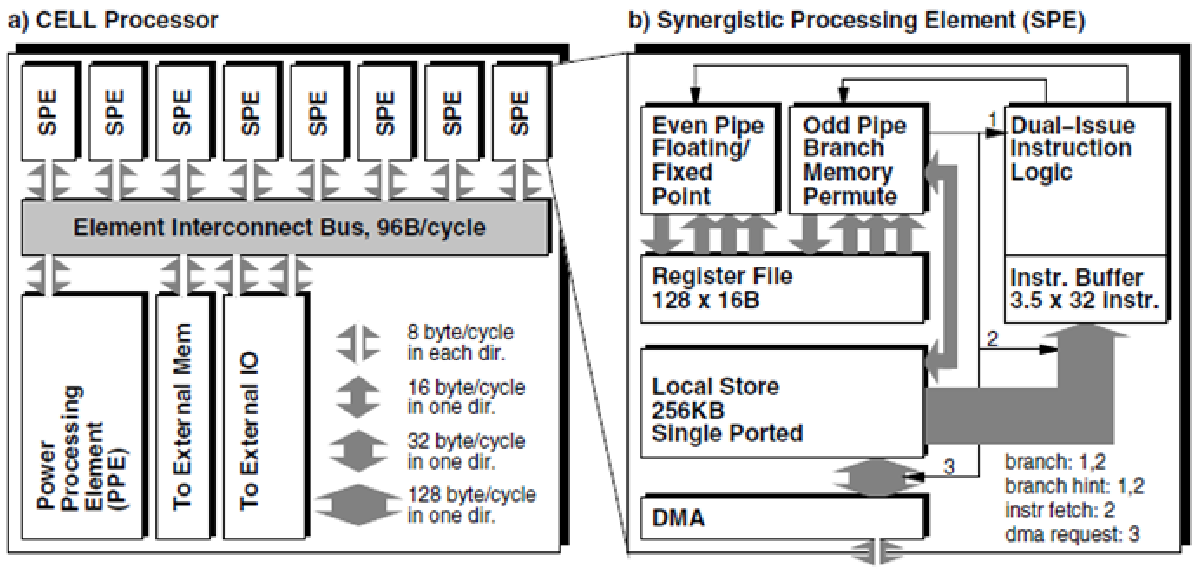
a) CELL Processor
SPE
Element Interconnect Bus, 96B/cycle
Power Processing Element (PPE)
To External Mem
To External IO
8 byte/cycle in each dir.
16 byte/cycle in one dir.
32 byte/cycle in one dir.
128 byte/cycle in one dir.
b) Synergistic Processing Element (SPE)
Even Pipe Floating/ Fixed Point
Odd Pipe Branch Memory Permute
Dual-Issue Instruction Logic
Register File 128 x 16B
Instr. Buffer 3.5 x 32 Instr.
Local Store 256KB Single Ported
DMA
1
2
3
branch: 1,2
branch hint: 1,2
instr fetch: 2
dma request: 3

Figure 2: The Implementation of CELL Processor

An SPE can dispatch up to two instructions per cycle to seven execution units that are organized into even and odd instruction pipes. Instructions are issued in order and routed to their corresponding even/odd pipe. Independent instructions are detected by the issue logic hardware and are dual-issued provided they satisfy the following code-layout conditions: the first instruction must come from an even word address and use the even pipe, and the second instruction must come from an odd word address and use the odd pipe. When this condition is not satisfied, the two instructions are simply executed sequentially. The SPE’s 256K-byte local memory supports fully pipelined 16-byte accesses for memory instructions and 128- byte accesses for instruction fetch and DMA transfers. Because the memory is single ported, instruction fetches, DMA, and memory instructions compete for the same port. Instruction fetches occur during idle memory cycles, and up to 3.5 fetches may be buffered in the 32-instruction fetch buffers to better tolerate burst peak memory usages. To further avoid instruction starvation, an explicit instruction can be used to initiate an inline instruction fetch. The branches are assumed non-taken by the SPE hardware but the architecture allows for a branch hint instruction to override the default branch prediction policy. In addition, the branch hint instruction pre-fetches up to 32 instructions .starting from the branch target, so that a correctly hinted taken branch incurs no penalty. One of the instruction fetch buffers is reserved for the branch hint mechanism. In addition, there is extended support for eliminating short branches using bit-wise select instructions.

Data is transferred between the local memory and the DMA engine in chunks of 128 bytes. The DMA engine can support up to 16 requests of up to 16K bytes. Both the PPE and SPEs can initiate DMA requests from/to each other’s local-memory as well as from/to global memory. The DMA engine is part of the globally coherent memory address space; addresses of local DMA requests are translated by an MMU before being sent on the bus. Bandwidth between the DMA and the EIB bus is 8 bytes per cycle in each direction. Programs interface with the DMA unit via a channel interface and may initiate blocking as well as non-blocking requests.

Chapter-3
OPTIMIZED SPE CODE GENERATION

Here we discuss the current compiler optimization techniques that address key architectural features of the SPE. During initial development of the SPE instruction set, a preliminary scalar compiler was prototyped as proof-of concept. This allowed us to quickly investigate the performance impact of a SIMD-only instruction set with a unified scalar/vector register file as well as with a memory interface based on loading a 128-bit vector, extracting, and appropriately formatting the desired scalar value

3.1 Scalar Code on SIMD Units

Nearly all SPE instructions operate on 128-bit wide SIMD data fields, including all memory instructions. One notable exception is the conditional branch instructions which branch on nonzero values from the primary slot of a 128-bit register. The other notable exception is the memory address fields which are also expected to reside in the primary slot by the memory instructions. When generating scalar codes on an SPE, we must ensure that the SIMD nature of the processor does not get in the way of program correctness. Mainly, we must ensure that the alignment of the data within SIMD registers is properly taken into consideration.

To illustrate the impact of SIMD units on scalar code, consider the following a= b+c scalar computation. Because the SPE memory subsystem processes only 16-byte aligned memory requests, loading b from memory yields the 32-bit b value2 plus 96 bits of unrelated data. The actual location of the b value within the register is determined by the 16-byte alignment of b in memory. Once the input data is in registers, the compiler must continue to keep track the alignment of its data since data can safely be operated on in SPE SIMD units only when they reside at the same register slots. For example, the 128- bit registers containing b and c can only be added if the b and c values reside at the same byte offset within their respective registers. When relatively misaligned, we permute the register content to enforce matching alignment.

Since scalar computations in highly optimized multimedia codes are mostly about address and branch-condition computations, the default policy is to move any misaligned scalar data into the primary slot of their registers. The storing of a scalar value also requires some care on the SPEs since stores are also 128-bit wide instructions. For scalars, the compiler must first load the original 128-bit data in which the result resides, then splice in the new value, and store the resulting 128-bit data to the memory. We take several steps to avoid such overhead. First, we allocate all temporary and global scalar variables in the primary slot of their own, private 128-bit chunk in memory. The padding overhead is insignificant compared to the code size increase incurred by additional instructions that would be otherwise needed to realign the data at runtime. Second, we perform aggressive register allocation for all local computations, such as address and loop index variables, to make good use of the 128-entry register file. As a result, most local variables reside exclusively in the primary slot of their respective registers, and thus memory storage and associated load/store instructions are not needed. Finally, there is an attempt to automatically simdize code to reduce the amount of scalar codes in an application.

3.2 Branch Optimizations

The SPE has a high branch-misprediction penalty of 18 cycles. Because branches are only detected late in the pipeline at a time where there are already multiple fall through instructions in flight, the SPE simply assumes all branches to be non-taken. Because taken branches are so much more expensive than non-taken branches, the compiler first attempts to eliminate taken branches. One well-known approach for eliminating short if-then-else constructs is if-conversion via the use of select instructions provided by the SPE. Another well-known approach is to determine the likely outcome of branches in a program, either by means of compiler analysis or via user directives, and perform code reorganization techniques to move cold paths out of the fall-through path.

To boost the performance of the remaining taken branches, such as function calls, function returns, loop closing branches, and some unconditional branches, the SPE provides a branch hint instruction. This instruction, referred to as Hint for Branch or hbr, specifies the location of a branch and its likely target address as shown in Figure 3.2. Instructions from the hinted branch target are pre-fetched from memory in a dedicated hint instruction buffer and the buffered instructions are then inserted into the instruction stream immediately after the hinted branch. When the hint is correct and scheduled at least 11 cycles prior to its branch, the branch latency is essentially one cycle; otherwise, normal branch penalties apply.

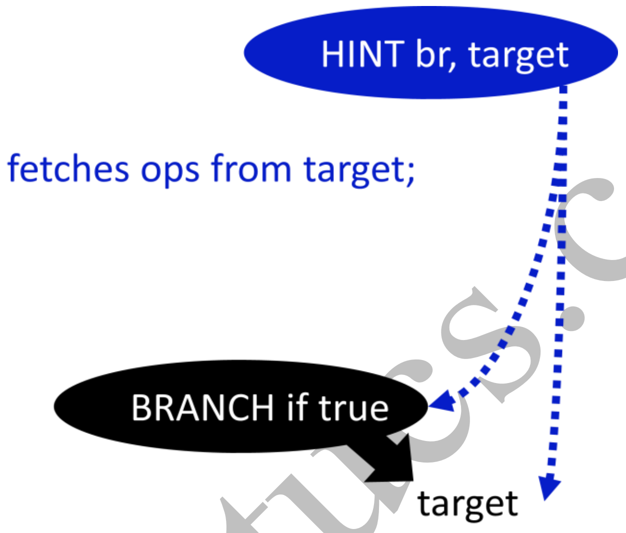

Figure 3.2: Hint For Branch

Presently, the SPE supports only one active hint at a time. Likely branch outcomes can be measured via branch profiling, estimated statically, or provided by the user via expect builtins or exec freq pragmas. Here the latter two techniques. Then insert a branch hint for branches with taken probability higher than a given threshold. For loop-closing branches, an attempt is to move the hbrs outside the loop to avoid repetitive execution of hint instructions. This optimization is possible because a hint remains effective until replaced by another one. Unconditional branches are also excellent targets for branch hint instructions. The indirect form of the hbr instruction is used for hinting function returns, function calls via pointers, and all other situations that give rise to indirect branches.

3.3 Instruction Scheduling

The scheduling process consists of two closely interacting subtasks: scheduling and bundling. The scheduler's main objective is to schedule operations that are on the critical path with the highest priority and schedule the other less critical operations in the remaining slack. While typical schedulers deal with resources and latencies, the SPEs also have constraints that are expressed in numbers of instructions. For example, the hbr branch hint instruction cannot be more than 256 instructions away from its target branch and should be no closer than 8 instructions. Constraints expressed in terms of instruction counts are further complicated by the fact that the precise number of instructions in a scheduling unit is known only after the bundling subtask has completed.

The bundler's main role is to ensure that each pair of instructions that are expected to be dual-issued satisfies the SPE's instruction issue constraints. The processor will dual-issue independent instructions only when the first instruction uses the even pipe and resides in an even word address, and the second instruction uses the odd pipe and resides in an odd word address. Once the instruction ordering is set by the scheduling subtask, the bundler can only impact the even/odd word address of a given instruction by judiciously inserting nop instructions into the instruction stream.

Another important task of the bundler is to prevent instruction-fetch starvation. A single local memory port is shared by the instruction-fetch mechanism and the processor's memory instructions. As a result, a large number of consecutive memory instructions can stall instruction fetching. With a 2.5 instruction-fetch buffers reserved for the fall-through path, the SPE can run out of instructions in as few as 40 dual-issued cycles. When a buffer is empty, there may be as few as 9 cycles for issuing an instruction-fetch request to still hide its full 15-cycle latency. Since the refill window is so small, the bundling process must keep track of the status of each buffer and insert explicit ifetch instructions when a starvation situation is detected.

The refill window is even smaller after a correctly hinted taken branch since there is only 1 valid buffer after a branch as opposed to 2.5 buffers for the fall-through path. In this case, instruction starvation is prevented only when all instructions in the buffer are useful. Namely, the branch target must point to an address that is a multiple of 16 instructions, which is the alignment constraint of the instruction-fetch unit. This constraint is enforced by introducing additional nop instructions before a branch target to push it to the next multiple of 16 instructions.. A final concern of the bundling process is to make sure that there is a sufficient number of instructions between a branch hint and its branch instruction. This constraint is due to the fact that a hint is only fully successful if its target branch address is computed prior to that branch entering the instruction decode pipeline. The bundler will add extra nop instructions when the scheduler did not succeed in interleaving a sufficient number of independent instructions between a hint and its branch.

For best performance, this approach uses a unified scheduling and bundling phase. Here,we generally preserve the cycle scheduling approach where each non-decreasing cycle of the schedule is filled in turns, except that here they may retroactively insert nop or ifetch instructions, as required by the bundling process. When getting ready to schedule the next cycle in the scheduling unit, first investigate if an ifetch instruction is required. When this is the case, forcibly insert an ifetch instruction in that cycle and update the scheduling resource model accordingly. We then proceed with the normal scheduling process for that cycle. When no additional instruction can be placed in the current cycle, then investigate if nop instructions must be inserted in prior cycles to enable dual-issuing. Once this task is completed, proceed to the next cycle.

Chapter-4

AUTOMATIC SPE/VMX SIMDIZATION

In this section, we present the simdization framework which targets both the SPEs and the PPE's VMX SIMD units. While their SIMD units have a distinct instruction set architecture, both units share a set of common SIMD architectural characteristics. For instance, both units support 128-bit packed SIMD registers; and both memory subsystems allow loads and stores from 16 byte aligned addresses only. The framework capitalizes on these commonalities by parameterizing the simdization algorithm and generating platform-specific code in the later phases of the simdization process. The simdization framework is part of the high-level optimization component of the XL compiler. We can thus leverage a rich set of analysis tools and optimizations, such as interprocedural analysis and loop transformations, in a mature optimizing compiler. The focus here on the three steps introduced for simdization purposes, *i.e.*, SIMD-parallelism extraction, alignment handling, and SIMD code-generation.

The first two transformations address issues common to SPE and VMX, whereas the last deals with specific target instruction sets.

4.1 Extracting SIMD Parallelism

The use generic vector data types and operations to represent extracted SIMD parallelism. In the framework, a vector initially has arbitrary length and alignment properties. As shown below, this relaxed length constraint is key to extracting SIMD parallelism from both within and across loop Iterations.

4.1.1 Loop-Level Simdization

Loop-level simdization follows an approach similar to vectorization for innermost loops. A simdizable loop must first satisfy the same dependence conditions as a traditionally vectorizable loop. Specifically, a simdizable loop has either no loop-carried dependence or has forward dependences; if it has other dependences, either the dependence distances are greater than the blocking factor of the simdized loop; or it has simple dependence cycles that can be recognized as certain vectorizable idioms, such as parallel reductions. In addition, since SIMD vectors are packed vectors and are loaded from memory only as 16-byte contiguous chunks, strided accesses are considered not simdizable.

Currently we simdize the following 5 types of accesses:
Stride -one memory accesses, loop invariant, loop private, induction, and reduction.

For loops that contain non-simdizable computation, a cost model is employed to determine whether to distribute the loop. Avoiding excessive loop distribution is particularly important for simdization because the SIMD parallelism being exploited is fairly narrow, *e.g.*, 4 way for integer and float. Thus, the overhead of loop distribution, such as reduced instruction level parallelism, register, and cache reuse, may override the benefit of SIMD execution. For example, one of the heuristics is to not distribute a loop if the simdizable portion involves only private variables, many of which are introduced by the compiler during common sub expression elimination. Once deemed simdizable, the loop is blocked, and scalar operations are transformed into generic vector operations on vector data types. The blocking factor is determined such that the byte length of each vector is a multiple of 16 bytes.

4.1.2 Basic-block Level Simdization

Short vectors enable us to extract SIMD parallelism within a basic block. Such parallelism is often found in unrolled loops, either manually by programmers or automatically by the compiler. It is also common in graphic applications that manipulate adjacent fields of aggregates, such as the subdivision computation shown in Figure 4.1.2.

```
for (i=0; i<n; i++) {
1: q = quads[i];
2: v[i].x=w0*q.p[0].x+w1*q.p[1].x+w2*q.p[2].x;
3: v[i].y=w0*q.p[0].y+w1*q.p[1].y+w2*q.p[2].y;
4: v[i].z=w0*q.p[0].z+w1*q.p[1].z+w2*q.p[2].z;
5: v[i].w=w0*q.p[0].w+w1*q.p[1].w+w2*q.p[2].w;
}
```

Figure 4.1.2: An example of basic-block level packing.

During basic-block level simdization, isomorphic computations on adjacent memory are packed into vector operations, using an algorithm similar to Superword Level Parallelism.

In Figure 3, for instance, Statements 2 to 5 can be packed into one vector statement because they are both isomorphic and operate on adjacent fields (*x*, *y*, *z*, and *w*) in memory. Note that Statement 1 in the same loop is an aggregate copy that can also be simdized into a vector copy.

In the framework, basic-block level simdization is performed before loop-level simdization. Thanks to arbitrary length vectors, a loop simdized at the basic-block level may be further transformed by loop-level simdization. For example, if fields *x*, *y*, *z*, and *w* in Figure 3 are 16-bit integers, both basic-block and loop-level simdization are needed to extract enough SIMD parallelism for a 16-byte vector.

4.2 Alignment Handling

The memory subsystems of the SPE and VMX can only access 16-byte contiguous memory from 16-byte aligned addresses. When a simdizable loop contains misaligned accesses, special handling may be required to *re-align* data on the fly to ensure the correctness of the simdized codes.

For example, consider the loop in Figure 4.2.1, where for conciseness the bases of arrays *a*, *b*, and *c* are 16-byte aligned.

```
for (i=0; i<100; i++) {
a[i+2] = b[i+1] + c[i+3];
}
```

Figure 4.2.1: A loop with misaligned accesses.

In this example, the data involved in any given loop iteration always reside at different byte offsets (*i.e.*, slots) of their respective vector registers after being loaded from memory.
For example, data involved in the i = 0 iteration, namely b[1], c[3], and a[2] as shown in, respectively, Figures 5a, 5c, and 5f, reside in different slots within their respective vector registers. This is a problem since each arithmetic SIMD instruction (such as the add instruction in this example) operates in parallel over the data residing in the same slot of its respective input vector register. To ensure the correctness of simdized computation, part of the simdization process deals with "re-aligning" data in registers, so that the same data involved in the original computation reside in the same slots in their respective registers after simdization. Figure 4.2.2 illustrates one such scheme. In Figure 4.2.2b, first shift right3 by one integer slot the stream of data generated by b[i+1] for i=0 to 99. In Figure 4.2.2d, shift left by one integer slot the stream of data generated by c[i+3] for i=0 to 99. At this stage, both the b and c register streams start in the third integer slot. The SIMD add instruction is then applied to the shifted streams and produces the expected results, b[1]+c[3],. . . ,b[100]+c[102].

The algorithm deals with streams of contiguous data in memory that are represented as stride-one accesses in a loop.

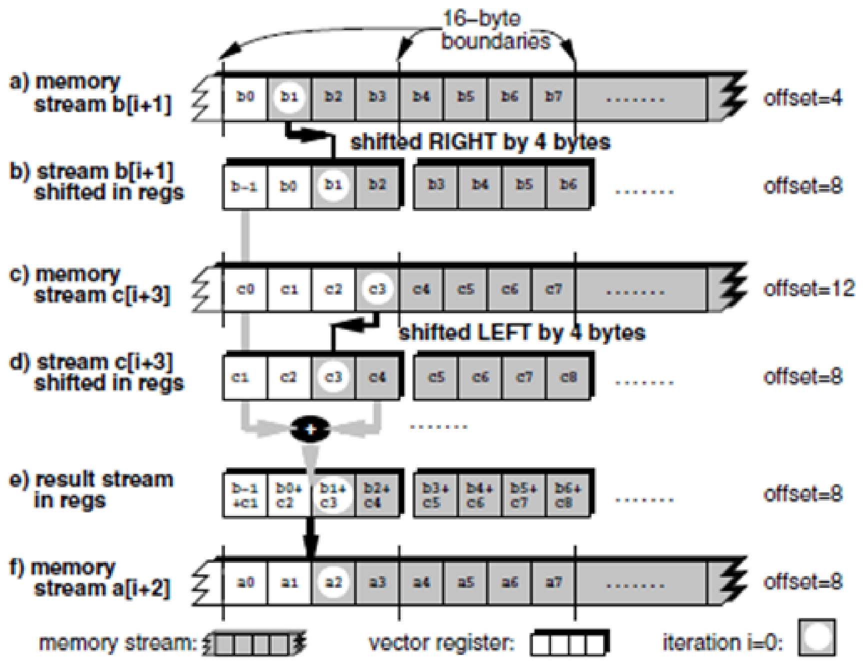

Figure 4.2.2: Simdization of a[i+2]=b[i+1]+c[i+3] with minimum data reorganizations.

For example, the data touched by access a[i+2] for i = 0 to 99 are considered as a stream. If a computation involves misaligned accesses, the algorithm will shift in registers entire streams associated one or more misaligned accesses so that data involved in the original computation reside in the same vector register slot before SIMD arithmetic operations are performed.

For a given statement, there are typically several ways to shift misaligned streams to satisfy alignment constraints. Then refer to an algorithm that generates one such way as a stream-shift placement policy. For example, the scheme illustrated in Figure 4.2.2 uses the *eager-shift* policy, where load streams are *eagerly* shifted to the alignment of the store stream.

Another commonly used policy is called *lazy-shift* policy. Consider a different a[i+3]=b[i+1]+c[i+1] loop. Instead of shifting each misaligned input to the alignment of the store, this policy will *lazily* shift the result stream of b[i+1]+c[i+1], since both streams participating in the addition are relatively aligned with each other's.

To understand the applicability of this scheme, it is critical to realize that “shifting left” and “shifting right” are data reorganizations that operate on a stream of consecutive registers, not the traditional logical or arithmetical shift operation.

4.3 SIMD Code Generation

Since extraction of SIMD parallelism and alignment handling both operate on generic vector operations, they are common to VMX and SPE simdization. In the SIMD code generation phase, generic vector operations are translated into SIMD intrinsic for specific platforms, which will be further optimized by the back-end compiler. This phase also handles arbitrary length vectors. Recall that during loop-level simdization, the loop is often blocked to ensure all vectors be of multiple of 16 bytes. In this phase, operations on vectors that are multiple of 16 bytes are mapped to multiple operations on vectors of 16 bytes. The others are reverted back into multiple scalar operations.

The simidization process can be summarized as follows:

As shown below, successful simdization involves several tasks that closely interact with each other. First Single Instruction Multiple Data (SIMD) parallelism must be extracted from an application, which may involve extraction from a basic-block, a loop, or a combination of both . Once that parallelism is extracted, various additional SIMD hardware constraints must be satisfied, such as memory alignment requirements, physical vector length, and hardware instruction set. The approach uses a virtual vector representation as an initial model to successfully extract SIMD parallelism from multiple sources in the code. Virtual registers have no alignment constraint and have arbitrary lengths. The virtual vectors are then progressively lowered in level to the physical SIMD registers of the target machine in successive de-virtualization steps.

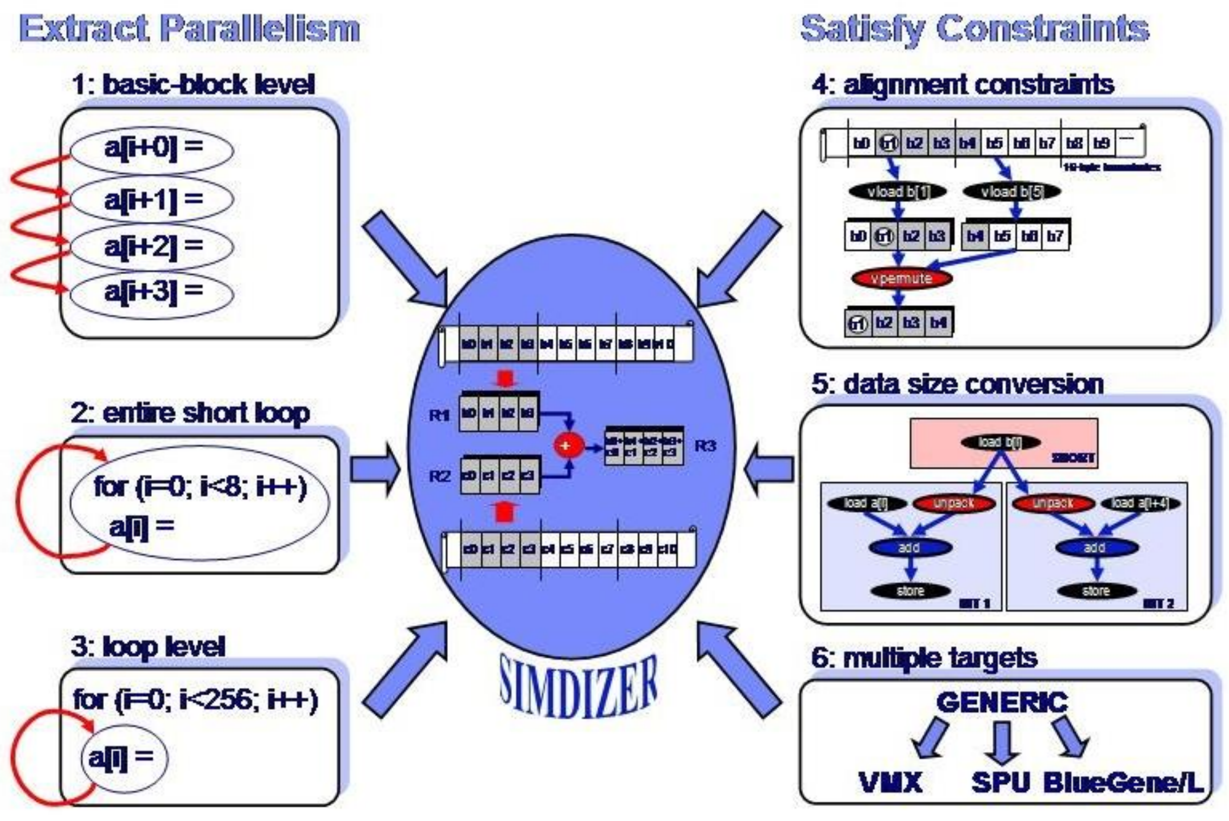

Figure 4: Simdization

Embedded in the Toronto Portable Optimizer (TPO) component of the XL compiler, the simdization optimization leverages a rich set of high-level optimizations such as inter procedure analysis and loop transformations. Here there is focus on describing the new components that are introduced specifically for simdization purposes. There are 6 new phases. The first 3 phases are mainly concerned with extracting SIMD parallelism from various code structures, e.g., basic blocks or loops.

Phase 1: Basic-Block Level Aggregation. During this phase, there is aggregate isomorphic statements that address consecutive memory locations into virtual vectors. This phase is particularly successful at extracting SIMD parallelism in unrolled loops and consecutive elements of structs, such as 3-dimensional coordinates found in multimedia applications. It also handles semi-isomorphic statements, for example statements that are nearly identical but for an add versus a subtract operation.

Phase 2: Short-Loop Aggregation. During this phase, aggregate entire loops with small trip counts into virtual vectors. This approach is successful in eliminating short innermost loop and may thus enable further SIMD aggregation within the second innermost loop. Typical examples are FIR based algorithms.

Phase 3: Loop-Level Aggregation. During this phase, aggregate instances of a statement in consecutive iterations of a loop. This phase is particularly successful at extracting SIMD parallelism in loops and recognizing special patterns such reductions and linear recurrences. The blocking factor is selected so that the length of each virtual vector is a multiple of the physical vector length: 16 bytes in the VMX/Cell/BGL architectures.

These first 3 phases proceed regardless of the alignments and lengths of the vectors, as if targeting an idealized SIMD machine. This simplifies the extraction phases greatly, especially in the presence of mixed data sizes. Each phase extracts SIMD parallelism, treating input scalar or virtual vectors (generated by a prior phase) in a similar fashion.

The next 3 phases deal with specific constraints of SIMD units, such alignment, physical vector length, and SIMD instruction sets.

Phase 4: Alignment Devirtualization. During this phase, satisfy the alignment constraint which states that the data being logically operated on by a SIMD operation must reside in the same slot of their respective input and output SIMD registers. In other words, when adding *a[i]* and *b[i+1]* in SIMD fashion, must ensure that both *a[0]* and *b[1]* values are in the same relative position in their respective register. Data realignment instructions are inserted only for relatively misaligned memory streams. There is an attempt to minimize the number of data realignment instructions, both in the presence of compile-time and runtime misalignment.

After the completion of this phase, all virtual vector accesses are to aligned data regardless of the compile time/runtime nature of their alignment. A virtual vector may still have a virtual length, i.e. a length that is too long with respect to the physical vector registers provided by the underlying architecture.

Phase 5: Length De-virtualization. During this phase, long vectors are chunked into physical length vector registers, or they may revert back to scalar statements. This phase also handles data size conversion, e.g., generating twice as many instructions for 4-byte integer expressions that use results generated by 2-byte short integer expressions.

After this phase, all virtual vectors are aligned and have the physical vector length, namely they are compatible with the physical vector registers provided by the underlying SIMD architecture.

Phase 6: SIMD Code Generation. Up to this point, most SIMD instructions were generic operators such as "add," "mult," or primitive data reorganization instructions. During this phase, generate SIMD instructions that are specific to the target SIMD units.

After this last phase, the simdization process is complete and the SIMD operations can be safely scheduled and register allocated.

CONCLUSION

Developed for multimedia, game applications, and other numerically intensive workloads, the first generation CELL processor implements on a single chip a Power Processor Element (PPE) and eight attached Synergistic Processor Elements (SPEs). In addition to processor-level parallelism, each processing element has multiple SIMD units that can each process from 2 double-precision floating points up to 16 bytes per instruction. Here, there is a compiler approach to support the heterogeneous parallelism found in the CELL architecture. CELL compiler implements SPE-specific optimizations including support for compiler assisted memory realignment, branch prediction, and instruction fetch. It addresses fine-grained SIMD parallelization.

REFERENCES

[1] IBM Corporation. PowerPC Microprocessor Family: AltiVec Technology Programming Environments Manual, 2004.

[2] Brian Flachs et al. The Microarchitecture of the Streaming Processor for a CELL Processor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, 2005.

[3] Michael Gschwind, Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. A Novel SIMD Architecture for the CELL Heterogeneous Chip-Multiprocessor