Optimizing Compiler for a CELL Processor

Alexandre E. Eichenberger[†], Kathryn O'Brien[†], Kevin O'Brien[†], Peng Wu[†], Tong Chen[†], Peter H. Oden[†], Daniel A. Prener[†], Janice C. Shepherd[†], Byoungro So[†], Zehra Sura[†], Amy Wang[‡], Tao Zhang[⋆], Peng Zhao[‡], and Michael Gschwind[†]

[†]IBM T.J. Watson Research Center Yorktown Heights, New York, USA.
[‡]IBM Toronto Laboratory Markham, Ontario, Canada.
[⋆]College of Computing Georgia Tech, USA.

ABSTRACT

Developed for multimedia and game applications, as well as other numerically intensive workloads, the CELL processor provides support both for highly parallel codes, which have high computation and memory requirements, and for scalar codes, which require fast response time and a full-featured programming environment. This first generation CELL processor implements on a single chip a Power Architecture processor with two levels of cache, and eight attached streaming processors with their own local memories and globally coherent DMA engines. In addition to processor-level parallelism, each processing element has a Single Instruction Multiple Data (SIMD) unit that can process from 2 double precision floating points up to 16 bytes per instruction.

This paper describes, in the context of a research prototype, several compiler techniques that aim at automatically generating high quality codes over a wide range of heterogeneous parallelism available on the CELL processor. Techniques include compiler-supported branch prediction, compiler-assisted instruction fetch, generation of scalar codes on SIMD units, automatic generation of SIMD codes, and data and code partitioning across the multiple processor elements in the system. Results indicate that significant speedup can be achieved with a high level of support from the compiler.

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 first generation CELL processor includes a 64-bit multi-threaded 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 paper, we present our 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.

Our 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.

Our 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.

Our third contribution is to enhance the programmability of the CELL processor by parallelizing a single source program across the PPE and 8 SPEs. Key to this is our approach of presenting the user with a single shared memory image, effected through compiler mediated partitioning of code and data and the automatic orchestration of any data movement implied by this partitioning.

We report an average speedup factor of 1.3 for the proposed SPE optimization techniques. When they are combined with SIMD and parallelization compiler techniques, we achieve average speedup factors of, respectively, 9.9 and 7.1, on suitable benchmarks. Although not integrated yet, the latter two techniques will be cumulative as they address distinct sources of parallelism.

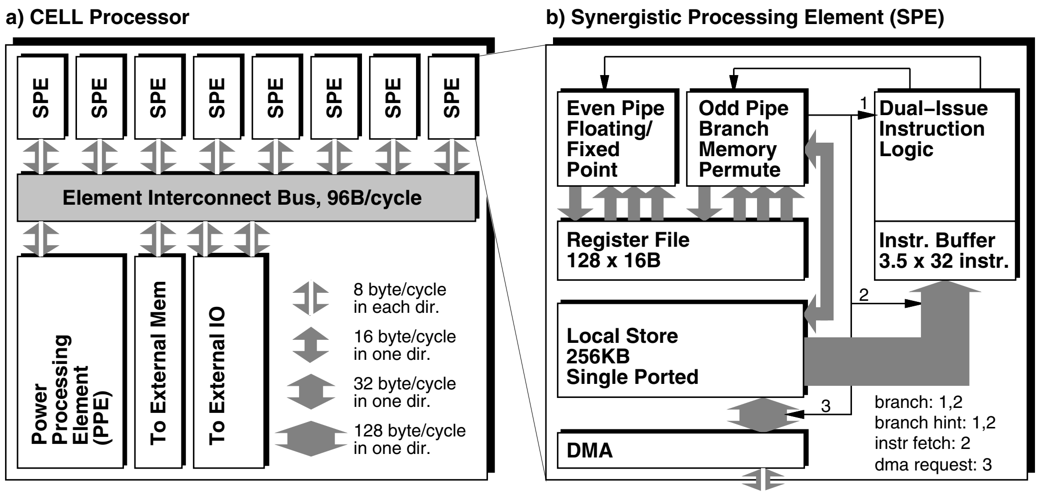
a) CELL Processor
SPE
SPE
SPE
SPE
SPE
SPE
SPE
SPE
Element Interconnect Bus, 96B/cycle
Power Processing Element (PPE)
To External Mem
To External IO
8 byte/cycle in each dir.
16 byte/cycle in one dir.
32 byte/cycle in one dir.
128 byte/cycle in one dir.
b) Synergistic Processing Element (SPE)
Even Pipe 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 1: The implementation of a first-generation CELL Processor

This paper is organized as follows. We describe the CELL processor in Section 2 and our programming model in Section 3. We present our SPE optimization techniques in Section 4 and our automatic SIMD code generation techniques in Section 5. We discuss our techniques for parallelization and partitioning of single source programs among the PPE and its 8 SPEs in Section 6. We report performance results in Section 7 and conclude in Section 9.

2. CELL ARCHITECTURE

The implementation of a first-generation CELL processor [1] includes a Power Architecture processor and 8 attached processor elements connected by an internal, high-bandwidth Element Interconnect Bus (EIB). Figure 1a 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) [2] 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) [3, 4]. 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 1b.

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.

Instructions	Pipe	Lat.
arithmetic, logical, compare, select	even	2
shift, rotate, byte sum/diff/avg	even	4
float	even	6
16-bit integer multiply-accumulate	even	7
128-bit shift/rotate, shuffle, estimate	odd	4
load, store, channel	odd	6
branch	odd	1-18

Figure 2: Latencies and pipe assignment for SPE.

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 inorder 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 instruction latencies and their pipe assignments are shown in Table 2.

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 bursty 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 prefetches 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 transfered 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.

3. PROGRAMMING MODELS

Programmers of the CELL architecture are presented with a wide range of approaches to extracting maximum performance. At one end of the spectrum, a dedicated developer has full control over the machine resources. At the other end, an application writer can enjoy a single-source program targeting a shared-memory multi-processor abstraction.

By programming the SPE in assembler, a dedicated programmer can select which registers to use, choose scalar or SIMD instructions, and schedule the code manually. This provides a high degree of flexibility and, when used judiciously, will yield the highest performance. Within this realm there are a number of different approaches to masking the system complexity, including the use of libraries, the function offload model, and task pipelining.

Prototypes developed using this model have demonstrated significant performance potential. Notwithstanding, this approach can be extremely labor intensive, and may represent an approach adopted by a small fraction of CELL programmers. Frequently, even dedicated application developers want to deploy high level programming languages, for example, for less performance-critical sections of an application, to reach higher productivity and programmability.

At the next level of support, we provide distinct compilers highly tuned to the respective ISAs. The programmer still has to deal with separate SPE and PPE programs, their respective address spaces and explicit interprocessor communication. However, most of the machine dependent optimization is performed automatically by the highly tuned compilers.

For critical sections of an application, a programmer can use a rich set of SPE and VMX intrinsics to precisely control the SIMD instruction selection and data layout, while leaving scheduling and register allocation decisions to the compiler. An intrinsic provides to the programmer a function that mimics the behavior of the underlying instructions in the target architecture. An intrinsic generally translates to a single instruction in the final output of the compiler, and we provide intrinsics for nearly all SPE/VMX instructions, including a full complement to support DMA operations. However, even programming with intrinsics and addressing data-layout issues manually can incur a significant investment of time. Moreover, the resulting source code will not be portable.

At the next level of support, our compiler provides automatic simdization targeting the SPE/PPE SIMD units. As will be shown in Section 5, we have developed techniques to extract SIMD parallelism from multiple language scopes such as across loop iterations or within a basic block. We also provide a systematic solution to addressing alignment constraints imposed by SPE/PPE memory subsystems. At this level, the application developer must still manually partition the program in consideration of the differing characteristics of the underlying ISAs and the constraints of the underlying memory subsystem. Code and data transfers must be positioned to ensure maximum overlap of communication and computation. The application must be hand parallelized to run on all the CELL processing elements to ensure the maximum performance benefit.

At the highest level of support, our compiler provides an abstraction of the underlying architectural complexity, presenting a single shared memory, multi-processor image, and providing parallelization across all the processing elements. In our current implementation, we use an OpenMP programming model, whereby the user specifies the parallel sections of the program. Our approach is equally applicable to automatic parallelization support and other parallel programming paradigms such as UPC.

4. OPTIMIZED SPE CODE GENERATION

In this section, we describe 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. However, in this section we describe our later implementation within IBM's XL compiler code base.

4.1 Scalar Code on SIMD Units

As mentioned in Section 2, 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[1] 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` value[2] 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

[1]The primary slot corresponds to highest order (leftmost) 32 bits of a register. For subword data types, only the least significant bits in the primary slot are considered.

[2]This is technically true only when the data is naturally aligned, e.g. when a 32-bit integer resides in memory at addresses that are multiples of 4 bytes. In this paper, we assume and support only naturally aligned data elements.

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, we attempt to automatically simdize code to reduce the amount of scalar codes in an application.

4.2 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. Instructions from the hinted branch target are prefetched 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. Presently, the SPE supports only one active hint at a time.

Likely branch outcomes can either be measured via branch profiling, estimated statically, or provided by the user via `expect` builtins or `exec_freq` pragmas. We use the latter two techniques. We then insert a branch hint for branches with taken probability higher than a given threshold.

For loop-closing branches, we attempt to move the `hbr`s 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.

4.3 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. As mentioned in Section 2, 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. Recall in Section 2 that 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. Our heuristics are fairly successful at scavenging any idle issue slots so that `nop` instructions may be inserted without performance penalties.

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, our approach uses a unified scheduling and bundling phase. We generally preserve the cycle scheduling approach where each nondecreasing cycle of the schedule is filled in turns, except that we 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, we first investigate if an `ifetch` instruction is required. When this is the case, we forceably 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, we then investigate if `nop` instructions must be inserted in prior cycles to enable dual-issuing. Once this task is completed, we proceed to the next cycle.

5. AUTOMATIC SPE/VMX SIMDIZATION

In this section, we present our 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. Our framework capitalizes on these commonalities by parameterizing the simdization algorithm and generating platform-specific code in the later phases of the simdization process.

Our 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. We 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.

5.1 Extracting SIMD Parallelism

We use generic vector data types and operations to represent extracted SIMD parallelism. In our 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 [5].

5.1.1 Loop-Level Simdization

Loop-level simdization follows an approach similar to vectorization for innermost loops [6, 7]. 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. We currently 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 our 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 subexpression 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.

5.1.2 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 3.

```
   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 3: 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 [8]. 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 our 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.

5.2 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, 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: 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 5 illustrates one such scheme. In Figure 5b, we first shift right[3] by one integer slot the stream of data generated by `b[i+1]` for `i=0` to `99`. In Figure 5d, we 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]`.

In prior work [9, 10], we proposed an alignment handling scheme that automatically re-arranges data to satisfy alignment constraints imposed by hardware like SPE and VMX. Our algorithm deals with streams of contiguous data in memory that are represented as stride-one accesses in a loop. 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, our 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. We refer to an algorithm that generates one such way as a stream-shift placement policy. For example, the scheme illustrated in Figure 5 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 others.

[3]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.

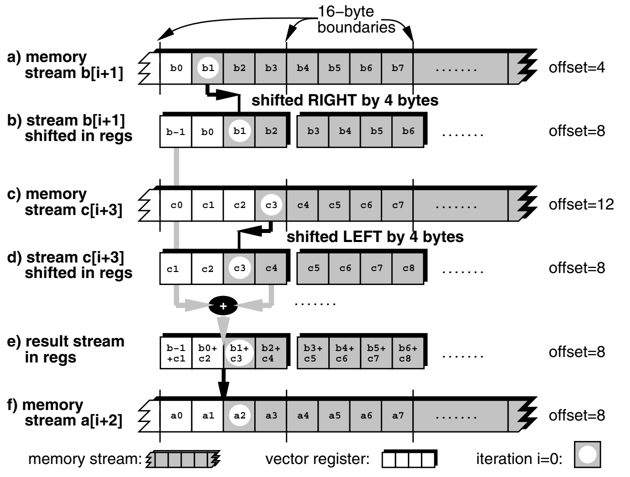
16-byte boundaries
a) memory stream b[i+1]
b0 b1 b2 b3 b4 b5 b6 b7 ....
offset=4
shifted RIGHT by 4 bytes
b) stream b[i+1] shifted in regs
b-1 b0 b1 b2 b3 b4 b5 b6 ....
offset=8
c) memory stream c[i+3]
c0 c1 c2 c3 c4 c5 c6 c7 ....
offset=12
shifted LEFT by 4 bytes
d) stream c[i+3] shifted in regs
c1 c2 c3 c4 c5 c6 c7 c8 ....
offset=8
+
e) result stream in regs
b-1 +c1 b0+ c2 b1+ c3 b2+ c4 b3+ c5 b4+ c6 b5+ c7 b6+ c8 ....
offset=8
f) memory stream a[i+2]
a0 a1 a2 a3 a4 a5 a6 a7 ....
offset=8
memory stream:
vector register:
iteration i=0:

Figure 5: Simdization of `a[i+2]=b[i+1]+c[i+3]` with minimum data reorganizations.

More details on shift placement policies, code generation for stream shifts such as handling runtime stream shifts, SIMD prologue/epilogue code generation for misaligned stores, and alignment handling in the presence of data conversion can be found elsewhere [9, 10].

5.3 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 intrinsics 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.

6. GENERATION OF MIMD CODE

In this section, we describe compiler generation of parallel codes for the CELL processor. Guided by user directives, the compiler distributes loop iterations across the multiple SPE processors of the CELL. In some respect, we are applying and extending existing techniques in a novel environment, but it is our approach to managing the hybrid memory subsystem of the CELL architecture which enables us to do this. As such, a key component of our parallelization approach is the presentation of a Single Shared Memory abstraction.

6.1 User-Directed Parallelization

Programming the SPE processor is significantly enhanced by the availability of an optimizing compiler which supports SIMD intrinsics and automatic simdization. However, programming the CELL Processor, the PPE and its 8 SPEs, is a much more complex task, requiring partitioning of an application to accommodate the limited local memory constraints of the SPE, parallelization across the multiple SPEs, orchestration of the data transfer through insertion of DMA commands, and compiling for two distinct ISAs.

We present an approach wherein the compiler manages this complexity while still enabling the significant performance potential of the machine. We refer to this as *Single Source* compilation to denote the fact that the user need only code a single program or set of programs targetting a CELL processor, without regard for the underlying architectural heterogeneity.

Our parallel implementation currently uses OpenMP [11, 12] pragmas to guide parallelization decisions. Making use of the existing parallelization infrastructure of the underlying IBM XL compiler, we outline[4] parallel code sections, in the first pass of the optimizer, and apply machine independent optimizations to the outlined functions. In the second pass of optimization, these outlined functions and their contained subgraphs are cloned and optimized codes are generated for both the SPE and the PPE processors. The *Master Thread* runs on the PPE processor and partakes in all work sharing constructs. Since there is no operating system (OS) support on the SPE, this thread also handles all OS service requests. Since cloning of the call graph occurs in the second pass of optimization, after interprocedural analysis, the compiler can generate versions of all library codes as appropriate for both the SPE and the PPE.

Since the limited size local memory of the SPE must accommodate both code and data, there is always a possibility that a single SPE object will be too large to fit. With our parallelization approach, since the unit of SPE compilation is a user defined parallel region, most often a loop nest, the extent to which we encounter very large SPE executables is greatly reduced. In a later section, we will describe our technique for handling very large SPE executables. This is fully integrated with our shared memory parallelization, such that we never encounter outlined SPE executables which are too large for a single SPE local memory.

6.2 Single Shared Memory Abstraction

Although a common mode of programming the CELL consists of manually partitioning code and data into PPE and SPE portions as well as explicitly managing the transfers of code and data between system memory and local store, we believe that, in many instances, this imposes too great a burden on the programmer. In particular, it is usual for a programmer to view a computer system as possessing a single addressable memory, and for all the program data to reside in this space.

In the CELL processor, the local stores, which alone are directly addressable by their respective SPEs, are memories separate from the vastly larger system memory. Each SPE possesses the capability to transfer data, by means of the DMA engine, between its local store and the system memory. The compiler, under certain conditions, can undertake the task of orchestrating this data transfer between the system memory and the local memory of the SPEs by explicitly inserting DMA commands. Moreover, there are, as we describe later, many optimizations that the compiler can perform to optimize these data transfers, especially when memory references are regular. In our approach, we attempt to abstract the concept of separate memories by allocating SPE program data in system memory and having the compiler automatically manage the movement of this data between its home location and a *temporary* location in the local store. When this movement is done to satisfy the demands of an executing SPE program, and the resulting buffers are organized in such a way as to permit reuse, we refer to the mechanism employed as a Software Cache.

Compiler-Controlled Software Cache

In our initial implementation of the Software Cache, a separate directory is maintained in each SPE. The compiler, using interprocedural analysis, replaces a subset of load and store instructions with instructions that explicitly look up the effective address of the datum in a directory. If the line containing the datum is present in the directory, the address of the requested variable is computed and the load or store continues using this local store location. Otherwise, a subroutine, the miss handler, is invoked and the requested data is transferred from system memory. As is usual in hardware caches, the directory is updated, and typically another line is selected for eviction in order to make room for the new element. Our current implementation provides a 4-way associative cache, and all 4 ways are probed inline, exploiting the SIMD parallelism of the instruction set. Our cache implementation can optionally be used either for a serial program, or a parallel program. There is some additional cost involved in supporting parallel execution, since we must keep track of which bytes in a cache line have been modified, in order to support multiple writers of the same line.

In the current version of our compiler, the software cache has 4 sets of 128 lines, each of which is 128-byte long. The line length was chosen based on consideration of efficiency of DMA transfer. Clearly many other choices could be made, and we intend in the future to evaluate some of the tradeoffs inherent in these choices. Given a particular set of choices, the code required to perform a cache lookup can be determined. In order to effect the lookup, we proceed in the following fashion. Initially, having decided that a particular variable is to be allocated in system memory, the compiler flags that variable as requiring a cache access. The intermediate code for accesses to both cached and non-cached variables initially looks the same, with load and store operations for each reference as appropriate. This allows the optimizer to operate on all accesses equally, thereby eliminating many redundant memory references. Late in the optimization phase of our compiler, the remaining loads and stores to the cached references are expanded by inserting inline the instructions which lookup the cache directory. In keeping with the spirit of caching, the expanded code assumes a hit, so the branch to the misshandler is treated by the rest of the compiler as an atomic operation, rather than a subroutine call. This means that the expected path is not impacted by the need to follow the normal calling conventions. Because of this, each reference requires a separate out-of-line miss “stub” which saves registers and explicitly branches back to the missing load or store upon return from the handler. Our current implementation takes no care to optimize these stubs, thus, in codes with a high miss rate we suffer a higher penalty than needed.

The lookup code itself is fairly straightforward, and the operations it performs are modeled on those of a hardware cache. It is first necessary to compute the address of the datum in system memory. The load or store that we are replacing will usually have the address expressed as a base register plus either a displacement or an index register. These we

[4] Outlining refers to the process of creating a new function for a particular section of code and replacing the original code section with a call to that newly created function.

add to produce the value that is to be looked up (this add would normally be implicitly performed by the load/store unit in hardware). From the result we extract a 7 bit index field, which is used directly to index into the cache directory. The directory entry thus indexed contains a set of 4 tags, followed by 4 corresponding line addresses. The upper 25 bits of the address are compared in SIMD with the 4 tag fields, and if any of the 4 compare equal, the line address in the same slot is combined with the lower 7 bits of the address and used as the byte offset in local memory of the required datum. If no tag value is equal, the miss handler is invoked and, when it returns, the address of the datum will be available as above. In the miss handler, a line is chosen for eviction and DMA operations are initiated to store back the evicted line if required, and to bring in the requested line. The appropriate slot in the directory entry is updated, and control returns to the point of the miss. The DMA operations in the miss handler take several hundreds of cycles, but this delay is roughly commensurate with L2 miss times on the PPE side of the CELL processor. The performance impact of the Software Cache is dominated by the cost of the cache probes, not by the miss cost.

As mentioned earlier, our current implementation supports two variants of the Software Cache. One variant supports only single-threaded code. In this version no attempt is made to keep track of which lines have been modified, and the miss handler perforce must write to system memory the whole of any evicted line, whether it has changed or not. It is probable that further compile-time analysis could ameliorate this situation, and we will be investigating this in the future. The second variant supports multi-threaded shared memory programs. In order to do this we must keep track in the cache directory on each SPE, which bytes have been written since the line was brought in from system memory. Then, when a line is to be evicted, either in support of a miss or to implement explicit flushes required for conformance with memory consistency rules, only the modified bytes are written by the DMA engine. Keeping track of dirty bytes requires that we insert additional code inline for each store operation, in addition to the lookup code discussed above. To accomplish this, the directory entry is extended by the addition of 4 quadwords, one for each set, which each contain 128 "dirty bits" allowing each byte in the line to be monitored.

Since the compiler must deal with the potential for aliases between cached data and data obtained by other means (for example via explicit prefetch in the tiler), the miss handler is required to take more care in choosing a line to be evicted than is normally the case with a hardware implementation.

Static Data Buffering

Of course, the naive approach of replacing loads and stores with the longer instruction sequences required for cache lookup has the effect of slowing program execution. We employ several techniques to mitigate this performance degradation. At the simplest level, we recognize that, in general, scalar data and small structured variables do not represent a large fraction of the space requirements of a program.These can often be directly allocated in the local stores, if it can be determined that they are not used to communicate between parallel tasks. In particular, the local, stack allocated variables of a function often fall into this class. Of course, difficulties arise when pointer usage causes aliasing between local and global variables, and it may also be problematic if large local arrays are used. It is also possible, although we have not yet implemented this, to explicitly allocate space in each local store for small shared variables, and additionally to allocate a home location in system memory. These variables can then be prefetched, and the home location be kept current by respecting the appropriate memory consistency model. In OpenMP, this would entail updating the home location at each explicit or implicit flush, and re-fetching the data at the appropriate acquire points.

For large arrays, if they are accessed in loops using affine expressions involving the loop induction variables, it is possible to use well known program restructuring techniques, [13, 14] such as loop blocking, to allow multiple elements to be explicitly fetched in a single operation. Further restructuring can effectively "software pipeline" the blocking loop, so that data movement and computation are overlapped, essentially prefetching the data. As is the case when compiling for more traditional memory hierarchies, for multi-dimensional loop nests the compiler can also use loop restructuring transformations such as blocking and interchange to increase locality [15, 16]. This is referred to as tiling; it requires legality checks based on the array dependence information and thus may not always be possible. Both prefetching and tiling in our environment cause the original variables to be replaced by references to much smaller temporary arrays, and this implies the rewriting of the indexing expressions also. Although tiling and prefetching techniques, as mentioned above, have been applied for cache memories previously, their use in the context of the SPE is somewhat different. Since explicit DMA operations are used, more care is required in the storage management. For many of the cases mentioned here, a further optimization is to try to combine data transfers using special DMA `list` commands, or in the case of multiple contiguous requests, by simply fusing several DMA operations into one. This has the effect of reducing the setup cost for DMA operations. To increase the opportunities for combining DMA operations, the compiler can reorder variables with respect to each other, including breaking up or combining structures, at link time. Our compiler already performs such transformations for other reasons, but we have not yet explored this additional use of that optimization.

For all of the foregoing techniques, when they can be used, the effect is to eliminate the need for cache lookup. Ultimately, it should only be necessary to use the cache to ensure correctness for references which are irregular, or unknown at compile time. Clearly to the extent that this can be accomplished, it requires substantial compile time analysis and the work is still in progress. We see very encouraging results with what we have implemented so far.

Code Partitioning

We propose a code partitioning technique to reduce the impact of the local storage limitations of the SPE on the program code segment; the scheme we describe can be used in a stand-alone manner with the SPE compiler, or by users choosing to manually partition their applications. In the Single Source compilation context our code partitioning approach is integrated with the data software cache to allow for the execution of large outlined functions with large data to run seamlessly across multiple SPEs.

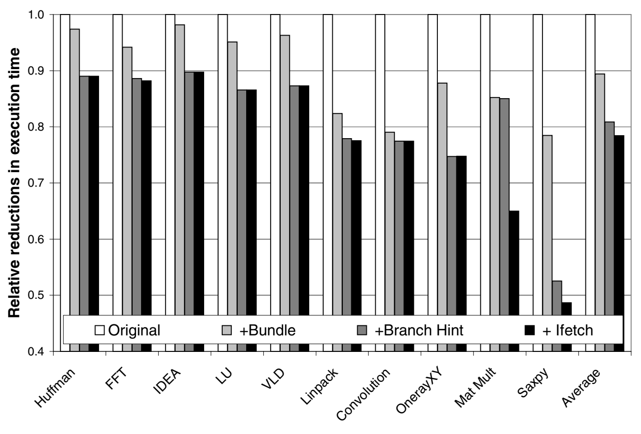

Figure 6: SPE optimizations.

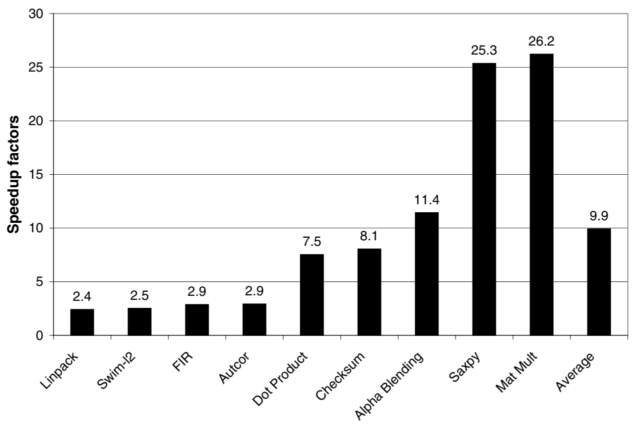

Figure 7: Simdization speedups.

In our code partitioning approach, the SPE program is divided into multiple partitions by the compiler. The home locations of code partitions, just as with data in our Software Cache approach, is system memory. When the compiler encounters such compilations it reserves a small portion of the SPE local storage for the code partition manager. The reserved memory is divided into two segments: one to hold the continuously resident partition manager, while the other holds the current active code partition. The partition manager is responsible for loading partitions from their home location in system memory into local storage when necessary, normally during an inter-partition function call or an inter-partition return. The compiler modifies the original SPE program to replace each inter-partition call with a call to the partition manager. Thus, the partition manager is able to take over control and handle the transition from the current partition to the target partition. The partition manager also makes sure an inter-partition return will return to the partition manager first.

Currently, the partitioning algorithm is a call graph based one, which means the basic unit of partitioning is a function. The compiler transforms the call graph into an affinity graph, with edge weight representing call edge frequency, and then applies a maximum spanning tree algorithm under a certain resource limit, typically the (adjustable) code buffer size.

In the code partitioning, currently we see respectable performance when executing partitioned functions on a single SPE relative to execution on the PPE. In the automotive suite of the EEMBC benchmark, across a single SPE, we see a slowdown of between 2 - 10 %. CJPEG, with code and data sizes of 1M, slows down 2.7 times, with both software cache and code partitioning enabled. Given the preliminary nature of this work, these results are encouraging.

There are several opportunities which we are currently exploring to improve the overall performance of our code partitioning algorithm. Effectively the algorithm largely depends on the accuracy of the affinity (call edge frequency). To achieve the best results, profiling can be used instead of static estimation. Also, using the actual partition size rather than the size estimated in the compiler conservatively could improve the utilization of local code buffer significantly. Prefetching is of course the most promising optimization and has the potential to hide the latency incurred when fetching partitions from main memory. But prefetching requires multiple buffers implying a much smaller partition size limit. The net effect depends on the prefetching algorithm and the accuracy of the cost model applied.

7. MEASUREMENTS

We first evaluate the optimized SPE code generation techniques presented in Section 4 using a cycle-accurate simulator. Figure 6 presents the reduction in program execution time for each optimization, relative to the performance of the original compiler (standard optimizations at O3 level, scheduled for the SPE resource and latency model). We report an average reduction of 22%, ranging from 11 to 51%.

The benchmark programs used here are highly optimized, simdized kernels representative of typical workloads executing on the SPEs. Kernels include a variable length decoding (VLD) from MPEG decoding, a Huffman compression and decompression, an IDEA encryption, and a ray tracing (OnerayXY). Numerical kernels include an FFT, a 7x7 short integer convolution, a 64x64 float matrix multiply, a Saxpy, an LU decomposition, and a solver kernel of Linpack.

Bundling for dual issue results in an 11% average reduction in execution time, ranging from 2 to 22%. Large reduction percentages indicate benchmarks with large amounts of instruction-level parallelism and no lucky instruction alignment (where random instruction layout did not satisfy the dual-issue constraint).

Hinting predictable branches results in a further 9% average reduction in execution time, ranging from 0 to 26%. Large reduction percentages indicate predictable branches with a sufficient amount of work to hide the hint latency. Some of the small reduction percentages (such as 0% for matrix multiply) indicate such tight loops that hinting is not beneficial without jointly addressing the instruction starvation issue.

Generating explicit instruction fetches results in a further 2% average reduction in execution time, with peak impact for very tight loops such as the 20% reduction for matrix multiply.

We now evaluate the automatic simdization techniques presented in Section 5 targeting a single SPE, also using the SPE cycle-accurate simulator. Figure 7 presents the speedup factors achieved when automatically simdizing sequential code kernels. Comparisons are performed at the same optimization level, which includes high-level, interprocedural optimizations in addition to all of the SPE optimizations presented in Section 4. We report an average speedup factor of 9.9, ranging from 2.4 to 26.2.

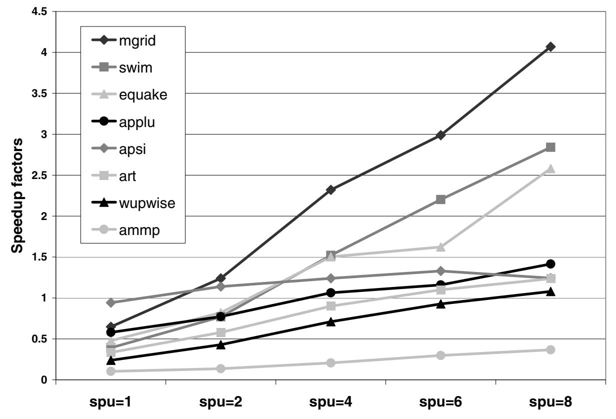

Figure 8: Speedups of parallelization.

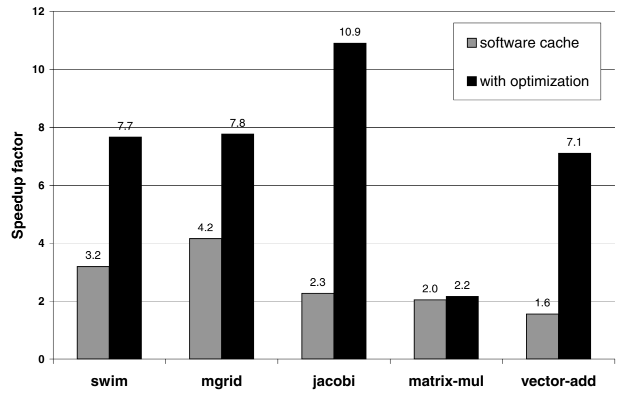

Figure 9: Speedups of parallelization with optimization.

The benchmark programs include video, numerical, and telecommunication applications. Kernels include a full Linpack solver, a short integer finite impulse response (FIR), an auto-correlation kernel, an integer dot-product, a TCP/IP checksum routine, and an alpha blending kernel. Two kernels are from the previous benchmarks, namely Saxpy and Matrix Multiply.

There are two tiers of benchmarks. The four leftmost kernels in Figure 7get respectable speedups (2.4 to 2.9) but below average. For Linpack, approximately 27% of the time is spent in a non-simdized part. The simdized part is analogous to Saxpy but has decreasing trip counts (from N to 1) that results in a higher loop overhead. For Swim, we believe that constant subexpression elimination does not currently address simdized references as well as scalar ones. For FIR and Autocor, we believe that there are data-type conversion issues that result in excessive overhead.

The rightmost 5 kernels get significant speedup (7.5 to 26.2). Both Dot Product and Checksum are performing a reduction which is not natively supported by the SPE's instruction set. This introduces some overhead which, in these two cases, can be efficiently hidden using partial sum reductions. Alpha Blending has data type conversions that are efficiently handled. Both Saxpy and Matrix Multiply exhibit peak performance due to perfect alignment and no data conversion overheads. Note that super-inear speedups are possible, *e.g.*, 26.2 on a single SPE, because the simdized version of the code not only computes multiple useful results per SIMD instructions, but also avoids all the overhead of running scalar code on SIMD units otherwise incurred on non-simdized code.

We now consider the parallelization techniques discussed in Section 6. The experiments described here were run on actual hardware. We first show, in Figure 8, results for parallel execution using only software cache. The experiment was conducted with SPEC OpenMP 2001 benchmark suite [17] with the three Fortran90 benchmarks excluded. The measurements are ratios of the execution times when running the sequential part on PPE and the parallel loops on various number of SPE processors, to those when running on the PPE processor alone. It may be more intuitive to use the execution time on one SPE as the baseline. However, these benchmarks are too large to be directly run on an SPE processor, without our code partitioning technique which does in itself incur some performance penalty. We reduce the data size of several kernels to get a rough idea of the relative speed of PPE and SPE processors. The measurements discussed do not consider the simdization of the parallelized benchmarks because the integration of these two optimizations is not yet complete. For the purpose of these measurement discussions, without simdization enabled, one SPE is approximately 2 times faster than the PPE processor.

In the eight OMP2001 benchmarks, Equake, Mgrid and Swim have notable speedup because they have high coverage of the parallel part and plenty of data reuse in the innermost loops. The long cache line used in the software cache can effectively capture such spatial reuse. Mgrid also has data reuse across the outer loops that can be exploited by the software cache. Other benchmarks suffer, to some extent, for the following reasons:

- Poor data locality. Data accesses are so scattered and discontinuous that numerous DMA operations are invoked
- Frequent cache flush. Some benchmarks, for instance, ammp and applu, require frequent cache flush. Cache flush is a very expensive operation in our implementation. Cache flush can be further optimized.
- Some optimizations turned off. In order to generate smaller code for the SPE processor, some optimizations, such as loop unrolling and procedure inlining, have to be turned off.
- Lower coverage of parallel parts. Since the currently available hardware has limited virtual memory space. Smaller data sets have to be used, reducing the coverage in some benchmarks.

The performance based purely on software cache can be enhanced by techniques discussed in Section 6. Figure 9 reports the improvement compared with the pure software cache approach. Eight SPE processors are used to execute the parallel loops. All but the matrix multiplication are dramatically improved. The code generation for DMA operations in cases of jumping data access needs further tuning.

8. RELATED WORK

We are not aware of any related work on generating scalar codes exclusively on SIMD units such as the ones found on the SPEs. But underlying techniques that minimize the performance impact of scalar codes on SIMD units are well known, such as data padding for predictable alignment [18] and aggressive register allocation to eliminate local variables in memory.

Compiler optimizations that reduce the number of taken-branches have been intensively researched such as code transformations to co-locate consecutively executed basic blocks [19], if-conversion, or combination of both [20]. Building on prior work, we focus on eliminating taken-branches and disregard the locality enhancing aspect of prior work since the SPEs have no cache. Similarly, our if-conversion scheme currently focuses on eliminating small `if-then-else` branch structures since our SPE's hardware support is limited to select instructions.

Compiler assisted branch hints and instruction fetch mechanisms have also been actively researched [21], especially the interaction of instruction fetches and cache pollution [22]. Since SPEs have no cache and have only very limited number of instruction fetch buffers, we insert hints very conservatively compared to prior work. Branch hints on SPEs are also different than those on the Itanium 2 [23], as our hints both prefetch instructions and impact the branch prediction policy. SPEs' hint and instruction fetch timing is also tighter than the timing on a cache-based system, as we are dealing here with the contention between data and instructions through a single memory port.

The bundling of instructions has also been studied in the literature [24], *e.g.*, for Itanium [23]. On the SPE, however, bundling is strictly a performance issue since the hardware is fully capable of detecting parallel instructions. Unlike in Itanium, there are no stop bit; thus dual-issue bundling rules are enforced by adding `nop` instructions. As a result, the main issue faced by the SPE compiler is the interaction between code size increases (due to additional `nop`) and instruction fetch issues.

Prior work on automatic SIMD code generation falls generally into two categories, a loop-based approach [25, 26, 27] and an unroll-and-pack approach [8, 28]. Our approach [9, 10, 5] uses a combination of both.

In terms of alignment handling, one technique is to peel the loop until all accesses are fully aligned or fall into a known, compile-time pattern [18, 26]. Our approach is to peel a full simdized loop iteration and conditionally execute it, on a per statement-basis. This technique is general, as it effectively support distinct compile-time/ runtime alignments for each statement. It is also faster on SPEs because even the peeled iterations are executed in SIMD mode.

The direct handling of misaligned references in loops has also been discussed in prior work [25, 26, 27]. To our knowledge, prior work shifts all relatively misaligned data to slot zero (*i.e.*, the zero-shift policy). Our approach attempts to systematically minimize the number of shifts by lazily shifting relatively misaligned input streams directly to the alignment of the output stream (*i.e.*, the lazy-shift policy). We do so with compile-time and runtime alignment, with multiple misaligned statements, unknown loop bounds, and in presence of data conversion, reductions, induction, and private variables.

Prior work on supporting parallel shared-memory applications on multiple processors includes work related to UPC, OpenMP, HPF, and distributed shared memory systems [29, 30, 31, 32]. They are similar to ours in that they distribute work among multiple processors, determine memory accesses in parallel regions that are shared by multiple processors, and explicitly transfer data for these accesses and attempt to optimize them. However, our work is different in that it considers heterogeneous cores with different processing capabilities and restrictions on what the SPEs can execute. Also, in our case the processor cores are on the same chip, and even though the SPEs rely on DMA transfers, all cores use the same virtual memory space to address shared data. In effect, the cost model for data access and synchronization is different in the CELL processor context.

There has been substantial work on providing caching in software, including [33, 34, 35, 36]. Most of the previous work focuses on caching data and providing coherence for a software implementation of distributed shared memory in a network of processors. In our implementation, software caching is done within a CELL chip and is designed to work efficiently using architectural features specific to the CELL. Also, our software cache needs to be aware of compiler optimizations that buffer data in SPE local stores, and it works in tandem with such optimizations.

The concept of memory overlays is well-known in the context of operating systems [37]. We use the same basic idea when partitioning large SPE code sections. However, we perform code partitioning entirely in software, using the compiler to optimize the choice of what codes to be placed in a partition.

In many embedded or reconfigurable systems, the hardware/software co-design process iteratively refines a series of hardware designs, trading off a set of design parameters such as space, power, and performance. This iterative process is driven mostly by synthesis tools [38, 39, 40] and/or optimizing compilers [41, 42]. Like other embedded designs, the SPEs have reduced hardware supports, *e.g.*, strict memory alignment, no branch prediction, simple instruction fetch, for higher clock rate, lower power consumption, and smaller floor plan. The SPE compiler is optimized to provide higher level functionality and performance using the simpler primitives (compared to general-purpose processors) provided by the SPE.

9. CONCLUSIONS

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. In this paper, we have presented our compiler approach to support the heterogeneous parallelism found in the CELL architecture.

Our CELL compiler implements SPE-specific optimizations including support for compiler assisted memory realignment, branch prediction, and instruction fetch. It addresses fine-grained SIMD parallelization as well as more general OpenMP task-level parallelization, presenting the user with a single shared memory image through compiler mediated partitioning of code and data and the automatic orchestration of the data movement implied by this partitioning.

Using benchmarks suitable to this platform, we demonstrate average speedup factors of 1.3 for the SPE-specific optimizations, 9.9 for the simdization, and 7.1 for the task-level parallelization.

We are working on integrating and refining current techniques and further exploiting opportunities available on the CELL architecture for our target workloads.

10. REFERENCES

[1] Dac Pham et al. The Design and Implementation of a First-Generation CELL Processor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, 2005.

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

[3] 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.

[4] Michael Gschwind, Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. A Novel SIMD Architecture for the CELL Heterogeneous Chip-Multiprocessor. In *Hot Chips 17*, 2005.

[5] Peng Wu, Alexandre E. Eichenberger, Amy Wang, and Peng Zhao. An Integrated Simdization Framework Using Virtual Vectors. In *Proceedings of the International Conference on Supercomputing*, 2005.

[6] John Randal Allen and Ken Kennedy. Automatic Translation of Fortran Programs to Vector Form. *ACM Transactions on Programming Languages and Systems*, (4):491–542, October 1987.

[7] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1990.

[8] Samuel Larsen and Saman Amarasinghe. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2000.

[9] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD Architectures with Alignment Constraints. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2004.

[10] Peng Wu, Alexandre E. Eichenberger, and Amy Wang. Efficient SIMD Code Generation for Runtime Alignment and Length Conversion. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2005.

[11] Official OpenMP specifications. http://www.openmp.org /specs/mp-documents/cspec20.pdf.

[12] Official OpenMP specifications. http://www.openmp.org /specs/mp-documents/fspec20.pdf.

[13] Todd C. Mowry. Tolerating Latency through Software Controlled Data Prefetching. PhD Thesis Stanford University, March 1994.

[14] Michael E. Wolf and Monica S. Lam. A Data Locality Optimization. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1991.

[15] Gabriel Rivera and Chau-Wen Tseng. Tiling Optimizations for 3D Scientific Computation. In *Proceedings of International Conference on Supercomputing*, 2000.

[16] Abdel-Hameed A. Badawy, Aneesh Aggarwal, Donald Yeung, and Chau-Wen Tseng. Evaluating the Impact of Memory System Performance on Software Prefetching and Locality Optimizations. In *Proceedings of the International Conference on Supercomputing*, 2001.

[17] SPEC OMP2001 Description. http://www.spec.org/omp/.

[18] Samuel Larsen, Emmett Witchel, and Saman Amarasinghe. Increasing and Detecting Memory Address Congruence. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2002.

[19] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the Conference on Programming language design and implementation*, 1990.

[20] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective Compiler Support for Predicated Execution using the Hyperblock. In *Proceedings of the International Symposium on Microarchitecture*, 1992.

[21] Arthur Bright, Jason Fritts, and Michael Gschwind. A Decoupled Fetch-Execute Engine with Static Branch Prediction Support. IBM Research Report RC23261, 1999.

[22] Chi-Keung Luk and Todd C. Mowry. Cooperative Prefetching: Compiler and Hardware Support for Effective Instruction Prefetching in Modern Processors. In *Proceedings of the International Symposium on Microarchitecture*, 1998.

[23] Intel Itanium 2 Processor Reference Manual for Software Development and Optimization. In *Intel*, 2003.

[24] Daniel Kaestner and Sebastian Winkel. ILP-based Instruction Scheduling for IA-64. In *Proceedings of the Workshop on Languages, compilers and tools for embedded systems*, 2001.

[25] Gerald Cheong and Monica S. Lam. An Optimizer for Multimedia Instruction Sets. In *Second SUIF Compiler Workshop*, 1997.

[26] Aart Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic Intra-Register Vectorization for the Intel Architecture. *International Journal of Parallel Programming*, (2):65–98, April 2002.

[27] Crescent Bay Software. VAST-F/AltiVec: Automatic Fortran Vectorizer for PowerPC Vector Unit. `http://www.psrv.com/vast_altivec.html`, 2004.

[28] Jaewook Shin, Mary W. Hall, and Jacqueline Chame. Superword-Level Parallelism in the Presence of Control Flow. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2005.

[29] Wei-Yu Chen, Costin Iancu, and Katherine Yelick. Communication Optimizations for Fine-grained UPC Applications. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2005.

[30] Seung-Jai Min, Ayon Basumallik, and Rudolf Eigenmann. Optimizing OpenMP Programs on Software Distributed Shared Memory Systems. *International Journal of Parallel Programming*, 31(3):225–249, 2003.

[31] Vikram Adve, Guohua Jin, John Mellor-Crummey, and Qing Yi. High Performance Fortran Compilation Techniques for Parallelizing Scientific Codes. In *Proceedings of the International Conference on Supercomputing*, 1998.

[32] Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. An Integrated Compile-Time/Run-Time Software Distributed Shared Memory System. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, 1996.

[33] Zoran Radovic and Erik Hagersten. Removing the Overhead from Software-Based Shared Memory. In *Proceedings of International Conference on Supercomputing*, 2001.

[34] Sandhya Dwarkadas, Kourosh Gharachorloo, Leonidas Kontothanassis, Daniel J. Scales, Michael L. Scott, and Robert Stets. Comparative Evaluation of Fine- and Coarse-Grain Approaches for Software Distributed Shared Memory. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 1999.

[35] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *ACM Operating Systems Review*, 1995.

[36] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-Grain Access Control for Distributed Shared Memory. *ACM SIGPLAN Notices*, 29(11):297–306, 1994.

[37] R. J. Pankhurst. Operating Systems: Program Overlay Techniques. volume 11, pages 119–125, 1968.

[38] Mentor Graphics Inc. *Monet R44 User's Manual R44*, 2002.

[39] Synopsys Inc. Behavioral compiler user's guide, 1999. http://www.synopsys.com/products/beh_syn/ beh_syn_br.html.

[40] Synopsys Inc. Cocentric data sheet, 2002. http://www.synopsys.com/products/cocentric_studio/ cocentric_studio.html.

[41] Vinod Kathail, Shail Aditya, Robert Schreiber, B. Ramakrishna Rau, Darren C. Cronquist, and Mukund Sivaraman. PICO: Automatically Designing Custom Computers. In *IEEE Computer*, pages 39–47, September 2002.

[42] Byoungro So, Mary W. Hall, and Pedro C. Diniz. A Compiler Approach to Fast Hardware Design Space Exploration in FPGA-based Systems. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2002.