Auto-parallelisation of Sieve C++ programs

Alastair Donaldson[1], Colin Riley[1], Anton Lokhmotov[2]*, and Andrew Cook[1]

[1] Codeplay Software
45 York Place, Edinburgh, EH1 3HP, UK
[2] Computer Laboratory, University of Cambridge
15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK

Abstract. We describe an approach to automatic parallelisation of programs written in Sieve C++ (Codeplay's C++ extension), using the Sieve compiler and runtime system. In Sieve C++, the programmer encloses a performance-critical region of code in a *sieve* block, thereby instructing the compiler to delay side-effects until the end of the block. The Sieve system partitions code inside a sieve block into independent fragments and speculatively distributes them among multiple cores. We present implementation details and experimental results for the Sieve system on the Cell BE processor.

1 Introduction

Computer systems are increasingly parallel and heterogeneous, while programs are still largely written in sequential languages assuming a single processor connected to uniform memory. The obvious suggestion that the compiler should automatically distribute a sequential program across the system usually fails in practice because of the complexity of dependence analysis in the presence of *aliasing*.

In Codeplay's Sieve C++ [1–3], the programmer can place a code fragment inside a special *sieve* block, thereby instructing the compiler to *delay* writes to memory locations defined outside of the block (global memory) and apply them *in order* on exit from the block. For example, by writing:

```
float *pa, *pb; ...
sieve { // sieve block
  for(int i = 0; i < n; ++i) {
    pb[i] = pa[i] + 42;
  }
} // writes to pb[0:n-1] happen on exit from the block
```

the programmer requests to delay the writes to global memory locations referenced by `pb[0]`,..., `pb[n-1]` until the end of the block. In this example, we can also say that the programmer requests the semantics of the Fortran 90 vector notation

```
pb[0:n-1] = pa[0:n-1] + 42;
```

in which all the reads happen before all the writes [4]. (The vector notation semantics departs from the conventional one if vectors `pa[0:n-1]` and `pb[0:n-1]` overlap.)

* This author gratefully acknowledges the financial support provided by a TNK-BP Cambridge Kapitza Scholarship and by an Overseas Research Students Award.

It is easy to see that the sieve semantics is equivalent to the conventional semantics if code within a sieve block does *not* write to and then subsequently read from a global memory location (otherwise, delaying the write violates the true dependence). The compiler preserves the order of writes to global memory by recording the writes in a FIFO queue and applying the queue on exit from the sieve block.

The sieve construct is different from Software Transactional Memory: code within an atomic block can immediately read new values of modified free variables; code within a sieve block "commits" its side-effects without retrying [2].

Using sieve blocks is attractive for several reasons. First, in order to parallelise code in a sieve block, the compiler needs to conduct dependence analysis only on memory locations defined within the block (local memory). Second, global memory can be read on entry to the sieve block and written to on exit from the block. This maps well to a natural programming style for heterogeneous systems with hierarchical memory. Third, the compiler can distribute the computation across the system speculatively (for example, if the number of iterations is not known at compile time). Results from excessive computation can simply be discarded when committing the side-effect queue to global memory. Fourth, the sieve semantics is deterministic, hence program behaviour is predictable and repeatable.

In this paper, we describe the Sieve C++ iterator and accumulator classes (§2), speculation in the Sieve system (§3), and present implementation details and experimental results for the IBM/Sony/Toshiba Cell BE processor (§4).

2 Syntax

2.1 Sieve and immediate functions

A function called from inside a sieve block must be explicitly specified as either *sieve* or *immediate*. Sieve functions can *only* be called from inside sieve blocks or other sieve functions (sieve *scopes*), and have their writes to global memory delayed. Immediate functions can be called from both sieve and non-sieve scopes, and must *not* update global memory. The compiler enforces the correct usage of these function types.

2.2 Iterator classes

In C/C++, the induction variable and increment value for a loop can be changed within the loop body [4]. Sieve C++ defines special *iterator* classes to track changes to induction variables in order to facilitate speculation.

A simple iterator class has a private *state* variable and a method for updating this variable with the value it should have after a given number of loop iterations.[3] For convenience and efficiency, the class can also include a method for updating the state with the value it should have at the next iteration. All methods updating the state must be specified with the **`update`** keyword. All other methods must be specified as **`const`**, and other fields as private and immutable.

[3] The programmer is responsible for the correct behaviour of this method, as this is not checked by the compiler.

For example, an integer counter class can be defined as:

```
iteratorclass intitr {
  int cnt; // state variable
public:
  immediate intitr(const int cnt) { this->cnt = cnt; }
  immediate operator int() const { return cnt; }
  // update methods
  update void operator +=(const int x) { cnt += x; }
  update void operator ++() { ++cnt; }
}
```

and used in the vector addition example as follows:

```
for(intitr i(0); i < n; ++i) {
  pb[i] = pa[i] + 42;
}
```

Iterator classes are not confined to basic induction. Consider another example:

```
double opt[n]; const double up = 1.1;
for (int i = 0, double Si = 1000.0; i < n; ++i) {
  opt[i] = Si;
  Si *= up;
}
```

Here, the value of variable `Si` (in all iterations but the first) depends on its value at the previous iteration. The programmer can re-write this loop in Sieve C++:

```
sieve {
  powitr Si(1000.0, 1.1);
  for (intitr i(0); i < n; ++i) {
    opt[i] = Si;
    Si.mulUp();
  }
}
```

where `powitr` is defined as follows:

```
iteratorclass powitr {
  double val; // state variable
  const double up;
public:
  immediate powitr(const double val, const double up)
                { this->val = val; this->up = up; }
  immediate operator double() const { return val; }
  // update methods
  update void mulUp(const int x) { val *= pow(up, x); }
  update void mulUp() { val *= up; }
}
```

The parameterised `mulUp` method can be used to update the state variable `val` with the value it should have after `x` iterations.

2.3 Accumulator classes

Reduction is the process of obtaining a single element by combining the elements of a vector [4]. The following example computes the sum of the elements of pa:

```
float sum = 0.0;
for(int i = 0; i < n; ++i) {
  sum += pa[i];
}
```

If we assume that addition is associative, this reduction can be performed as a number of partial sums the results of which are summed to give the final result.

This computational pattern is supported in Sieve C++ with special *accumulator* classes, which have a distinguished parameterised method called the *merge rule* that is specified with the > symbol (*cf.* using ˜ for destructors in C++).

The floating point accumulator class:

```
accumulatorclass floatsum {
  float acc;
public:
  immediate floatsum() { this->acc = 0.0; }
  >floatsum(float * res, const floatsum **  resv,
                         const unsigned int resc) {
    *res = resv[0]->acc;
    for(int i = 1; i < resc; ++i) { *res += resv[i]->acc; }
  }
  immediate void operator += (float x) { acc += x; }
}
```

can be used to re-write the summation reduction in Sieve C++:

```
sieve {
  floatsum fsum() merges sum;
  for(intitr i(0); i < n; ++i) {
    fsum += pa[i];
  }
} // the merge rule '>' is implicitly called here
```

Each partial sum is accumulated into a (private to each core) variable acc via the += operator. On exit from the block, the sieve runtime calls the merge rule >floatsum to obtain the final result.

Accumulators can be defined for any associative operation. Practical examples include the sum, product, exclusive-or, min and max operators.

3 Speculative execution

3.1 Split points

The Sieve system uses the notion of a *split point* to parallelise C++ code within a sieve block. Split points are program points which delimit a sieve block into fragments which can be executed independently, and thus deployed across multiple cores. Split points can either be inserted implicitly by the compiler or explicitly by the programmer via the **splithere** keyword. Every sieve block has two implicit split points: at the start and end of the block.

Annotating the vector addition example with a split point inside the loop:

```
sieve { // implicit split point (1)
  for(intitr i(0); i < n; ++i) {
    splithere; // explicit split point (2)
    pb[i] = pa[i] + 42;
  }
} // implicit split point (3)
```

indicates to the compiler that it would be sensible to parallelise this loop.

We can view split points *statically* or *dynamically*: there are three static split points in the above example, as indicated in the comments. Dynamically there are $n + 2$ split points: at the start and end of the sieve block, plus a split point for each iteration of the loop. We can distinguish each dynamic split point in a loop nest by combining the associated static split point with an *iteration vector* (*IV*) [4], comprised of the values of iterators controlling the loop nest. Execution of a sieve block can then be organised as a chain of dynamic split points. For the above example we have the chain $1\langle\ \rangle$, $2\langle 0\rangle$, $2\langle 1\rangle$, . . ., $2\langle n-1\rangle$, $3\langle\ \rangle$ (where $\langle\ \rangle$ denotes an empty *IV*).

A *fragment* is any contiguous portion of a chain of dynamic split points. A sieve block can be efficiently executed by dividing its associated chain of dynamic split points into fragments and executing these fragments in parallel. Each fragment maintains a queue of side-effects which are applied to global memory in order on exit from the block. In addition, each fragment maintains a local accumulator variable for every accumulator declared within the sieve block. On exit from the block the values of these accumulators are merged into appropriate global data structures via the accumulator merge rules.

Note that a fragment typically spans multiple split points. The compiler and runtime system decide how large each fragment should be for the given code and parallel hardware.

3.2 Speculative execution

A fragment can be described by specifying a static split point, an *IV* giving the values of iterator variables at the start of the fragment, and an integer specifying how many split points should be traversed during the fragment. The result of executing a fragment can be described by: a queue of side-effects to global memory, a set of values for accumulator variables, and an *IV* giving the values of iterator variables at the end of the fragment.

Parallel execution of a loop in a sieve block can be achieved by *guessing* the *IV*s for a sequence of contiguous fragments. Since the first fragment always begins at the start of the sieve block, its *IV* is empty and thus trivial to guess. The runtime system assigns this fragment to one core for execution. The runtime then uses a strategy to *guess* the value which the *IV* will have at the end of this fragment. This guessed vector is used to generate a fragment for parallel execution by another core. If more cores are available, then this guessing process is extended so that each core executes a fragment starting with a guessed *IV*.

If the guessed *IV* for a fragment matches the actual final *IV* of the previous fragment, then the fragments are contiguous, and the guess is correct, or *valid*. Given a chain of correctly guessed fragments, applying the side-effects for each fragment to global memory in sequence has the same effect as executing the fragments in serial with the sieve semantics. If the *IV* for a fragment is incorrectly guessed then its side-effect queue (and the queues of its subsequent fragments) must be discarded. We refer to the execution of fragments with guessed *IV*s as *speculative execution*, since the execution results may have to be discarded if guessing turns out to be wrong.

3.3 Examples

We illustrate the idea of *IV* guessing and speculative execution using the vector addition example. Suppose that at runtime `n` = 20, so that there are 22 dynamic split points: $1\langle\ \rangle$, $2\langle 0\rangle$, $2\langle 1\rangle$,. . ., $2\langle 19\rangle$, $3\langle\ \rangle$. Suppose further that the arrays `pa` and `pb` both have size 20 and that before execution of the loop `pa` is set up so that `pa[`i`]=`i, for any $0 \leq i < 20$.

The following table shows a *perfect* guessing chain for execution of the loop on a quad-core machine:

core	**guessed *IV***	**guessed length**	**actual length**	**side-effects**	**final *IV***
1	$1\langle\ \rangle$	6	6	`pb[0:4]=[42..46]`	$2\langle 5\rangle$
2	$2\langle 5\rangle$	5	5	`pb[5:9]=[47..51]`	$2\langle 10\rangle$
3	$2\langle 10\rangle$	5	5	`pb[10:14]=[52..56]`	$2\langle 15\rangle$
4	$2\langle 15\rangle$	5	5	`pb[15:19]=[57..61]`	$3\langle\ \rangle$

The guessing chain is perfect because, for $i > 1$, the guessed *IV* for core i matches the final *IV* for core $i - 1$; the guessed length of each fragment matches the actual length of the fragment execution; computation is balanced as evenly as possible between the cores, and no unnecessary computation is performed.

On the other hand, the following table illustrates a poor guessing chain for the same execution:

core	**guessed *IV***	**guessed length**	**actual length**	**side-effects**	**final *IV***
1	$1\langle\ \rangle$	12	12	`pb[0:10]=[42..52]`	$2\langle 11\rangle$
2	$2\langle 11\rangle$	12	9	`pb[11:19]=[53..61]`	$3\langle\ \rangle$
3	$2\langle 23\rangle$	12	1	`pb[23]=`$\perp$	$3\langle\ \rangle$
4	$2\langle 35\rangle$	12	1	`pb[35]=`$\perp$	$3\langle\ \rangle$

In this example core 1 performs most of the computation, and a correct *IV* guess allows core 2 to do the rest of the computation in parallel. The fact that the guessed fragment length for core 2 is too large does not affect correctness of execution: when the loop condition becomes false, this core reaches the end of the sieve block as expected. However, the guessed *IV*s for cores 3 and 4 are based on the expected fragment length for core 2. As a result, these cores attempt to read subscripts of `pa` and write to subscripts of `pb` which are beyond the bounds of these arrays. The resulting side-effects are marked grey in the above table, and the undefined values speculatively assigned to `pb[23]` and `pb[35]` are denoted $\perp$.

After this speculative execution the runtime assesses the correctness of its guessing effort. It determines that cores 1 and 2 have performed the required loop execution, and applies their side effects to global memory (filling the array `pb`). The runtime also detects that the guesses for cores 3 and 4 were incorrect and therefore does *not* apply their side-effect queues. As a result, although this poor guessing effort does not lead to optimal exploitation of parallel hardware, it still results in correct, deterministic execution of the sieve block.

3.4 Coping with invalid guesses

In the above example, attempting to read from `pb[23]` or `pb[35]` may result in an access exception. In more complicated examples, speculative execution could result in other exceptions (e.g. division by zero). These exceptions are caught by the runtime system and hidden until the runtime can determine whether the guess which caused a given exception is valid (i.e. whether this guess simulates serial behaviour). If a guess turns out to be invalid (as in the above example) then, in addition to discarding the side-effect queue, any exceptions arising from the speculated execution are also ignored. If the guess is valid, the runtime system will re-run the fragment and this time expose the exception so that the user can debug the program as usual.

3.5 Advanced techniques for guessing

As discussed in §2.2, the **`update`** methods provide a way to set up the state of iterator variables as if a given number of loop iterations had already been executed. Nevertheless, sometimes it is impossible to determine the number of iterations a given loop will execute: the loop may exit early via **`break`** statements; the loop bounds may change dynamically, *etc.* Thus, good guessing is a challenge.

A simple guesser can operate by running small fragments (e.g. with a length of one) to check for updates to iterator variables in the loop body. Once the pattern of these updates is discovered, the runtime can make larger guesses to ensure that the computation within each fragment is sufficient to outweigh the runtime overhead of managing the fragments.

More advanced speculation techniques could be employed by having the compiler communicate extra sieve block meta-data to the runtime. For example, the compiler could identify the split points which a given iterator spans, and mark split points across which no iterators are live, allowing speculation before and after these points to be independent. This would ease the task of checking guess-chain correctness, and increase the likelihood of valid guesses.

4 Implementation on the Cell BE

Experimental results showing the effectiveness of parallelisation via the Sieve system for multi-core x86 systems are presented in [2]. We focus here on implementation and experimental results for the Cell Broadband Engine (BE) processor [5]. Fig. 1 illustrates the Cell BE architecture, which consists of a “power processing element” (PPE) and eight[4] “synergistic processing elements” (SPEs). Each SPE has 256KB local memory, and accesses main memory using DMA transfers.

[4] Our implementation is for the Sony PlayStation 3 console, on which only six of the SPEs are available to the programmer.

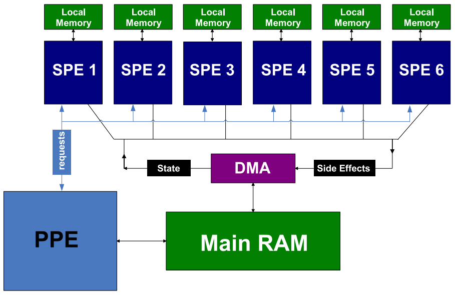

Fig. 1. Architecture overview of the Cell BE processor. (Only the six SPEs which are available to the PlayStation 3 programmer are shown.)

4.1 The Cell runtime

The Codeplay Sieve system is designed to be easily ported to new processor architectures. The Sieve compiler has an ANSI C backend which can be used to enable support for architectures where a C compiler already exists. In particular, the IBM Cell SDK 2.0 (which we used for our experiments) includes the GCC and xlC compilers.

In addition to C source files, the Sieve compiler also outputs details of which source files should be compiled for which processing elements. This allows auto-generation of a makefile for direct compilation of Sieve compiler's output to a parallelised binary.

A runtime for the Cell BE took only two weeks of time for one developer to create; this runtime incorporates simple loop-level speculation, support of iterator and accumulator classes, an SPE software-managed cache, side-effect queue management, and streaming DMA optimisations.

The runtime manages the SPEs as a pool. The SPEs boot into a tight runtime loop which checks for fragments which are waiting to be executed. The side-effect queue for each SPE is implemented using a ping-pong double-buffered streaming technique to achieve constant memory usage, yet to allow efficient use of SPE-initiated non-blocking DMA operations. Each SPE requires a certain amount of its 256 KB local store to be reserved for runtime use. Reserving too small an amount can lead to communication bottlenecks, as large side-effect queues must be streamed back in smaller chunks.

The PPE produces guesses, which are consumed by the SPEs, and writes SPE side-effect queues to main memory on exit from a sieve block. When an accumulator is used within a sieve block, the PPE is also responsible for collecting accumulated values from the SPEs and merging these values using the accumulator merge rule.

4.2 Experimental results

Fig. 2 shows the speedup (over a single SPE) for five example Sieve C++ applications: a cyclic redundancy check of a randomly generated 8MB message (CRC); generation of a ray-traced 500×500 image representing a 3D intersection of a 4D Julia set, with reflection and phong shaded lighting (Julia); a noise reduction filter over a 512×512 image, using a 20×20 neighbourhood per pixel (Noise); generation of a 1500×1500 fragment of the Mandelbrot set (Mandelbrot); a 4M-point Fast Fourier Transform (FFT).

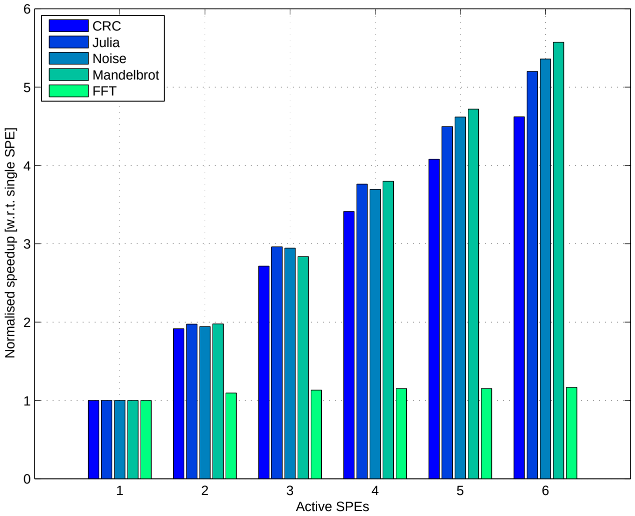

Fig. 2. Scalability results on Sony PlayStation 3.

The results show that all Sieve C++ benchmarks, with the exception of FFT, scale well over multiple cores (similar to our results for multi-core x86 systems [2]), with 77%, 86.7%, 89.3% and 92.9% efficiency on 6 SPEs for the CRC, Julia, Noise, and Mandelbrot benchmarks, respectively. Of these scalable benchmarks, the CRC is least efficient because it uses accumulator variables which require a modest amount of serial computation on the PPE to execute the accumulator merge rule. The Julia, Noise and Mandelbrot do not use accumulators, and hence are more efficient. The experiments were performed using the delayed-write combining technique discussed next.

4.3 Combining writes

Managing the side-effect queue incurs space and time overheads. A side-effect queue element is written as a triple `(address,size,data)`, where `address` is the destination memory address and `size` is the data size (in bytes). In the current implementation, the combination of the address and size is 8 bytes long and data is padded to a minimum of 4 bytes. Thus, a delayed write of a single byte results in writing 12 bytes to the queue.

An easy way to reduce this space overhead is to combine a series of small consecutive writes into a single, larger write. This can be achieved by comparing each write with the last entry in the queue, merging the data and updating the data size information if the data items turn out to be contiguous in memory.

Consider the following sieve block which writes characters to an array:

```
char *p = 0xfeed;
sieve {
  p[0] = ’b’; p[1] = ’e’; p[2] = ’e’; p[3] = ’f’;
}
```

Without delayed-write combining, the side-effect queue grows significantly with each delayed write:

```
sieve {
  p[0] = 'b'; // queue: [(0xfeed,1,'b')]
  p[1] = 'e'; // queue: [(0xfeed,1,'b'), (0xfeee,1,'e')]
  p[2] = 'e'; // queue: [(0xfeed,1,'b'), (0xfeee,1,'e'),
              //         (0xfeef,1,'e')]
  p[3] = 'f'; // queue: [(0xfeed,1,'b'), (0xfeee,1,'e'),
              //         (0xfeef,1,'e'), (0xfef0,1,'f')]
}
```

Applying delayed-write combining results in a smaller queue:

```
sieve {
  p[0] = 'b'; // queue: [(0xfeed,1,'b')]
  p[1] = 'e'; // queue: [(0xfeed,2,"be")]
  p[2] = 'e'; // queue: [(0xfeed,3,"bee")]
  p[3] = 'f'; // queue: [(0xfeed,4,"beef")]
}
```

This optimisation is particularly beneficial for the Mandelbrot benchmark which writes pixels into a contiguous **`unsigned char`** array. Computing a 600×600 Mandelbrot image across 6 SPEs (working on 100 rows each) means that each fragment has 60,000 twelve-byte queue entries. Using the optimal transfer size of 16KB implies 44 DMA operations per fragment [5]. Applying delayed-write combining results in only 4 DMA operations per fragment (three 16KB transfers followed by a transfer of 10,872 bytes). The total number of DMA operations is thus reduced from 264 to 24. The benefit is in having less of a bus-bottleneck and less blocking whilst ping-ponging buffers (as the queues now take longer to fill). In our experiments, using delayed-write combining resulted in 21.4% faster execution time when computing a 1500×1500 pixel image.

5 Conclusion

We have presented the Sieve compiler and runtime system for auto-parallelising Sieve C++ programs. Our future work will focus on advanced implementation techniques of Sieve C++ programs for performance and scalability on the Cell BE and other multi-core architectures.

References

1. Codeplay: Portable high-performance compilers. `http://www.codeplay.com/`
2. Lokhmotov, A., Mycroft, A., Richards, A.: Delayed side-effects ease multi-core programming. In: Proc. of the 13th European Conference on Parallel and Distributed Computing (Euro-Par), pp. 629–638. Springer LNCS Vol. 4641 (2007)
3. Lindley, S.: Implementing deterministic declarative concurrency using sieves. In: Proc. of the ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming (DAMP), ACM Press, New York (2007)
4. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann, San Francisco (2002)
5. IBM/Sony/Toshiba: Cell Broadband Engine Programming Handbook Version 1.1 (2007)