Strict and Relaxed Sieving for Multi-Core Programming

Anton Lokhmotov[1⋆], Alastair Donaldson[2],
Alan Mycroft[1], and Colin Riley[2]

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

Abstract. In Codeplay's Sieve C++, the programmer can place code inside a "sieve block" thereby instructing the compiler to delay writes to global memory and apply them in order on exit from the block. The semantics of sieve blocks makes code more amenable to automatic parallelisation. However, strictly queueing writes until the end of a sieve block incurs overheads and is typically unnecessary. If the programmer can *assert* that code within a sieve block does not write to and then subsequently read from a global memory location, the sieve semantics can be relaxed: writes can be executed at any point from their originally scheduled time until the end of the block. We present experimental results demonstrating the benefits of relaxed sieving on an x86 multi-core system.

1 Introduction

We are living in a time of change, where commodity computer systems are becoming increasingly parallel and heterogeneous. General-purpose processor vendors, such as Intel and AMD, have shifted their efforts away from boosting the clock frequency and architectural complexity of single-core processors, concentrating instead on producing processors consisting of multiple simpler cores.

Another growing trend is to supplement general-purpose "host" processors with special-purpose co-processors, or *accelerators*. Co-processors can either be located on the same chip as the host, or on a different chip (often on a separate plug-in board). Examples include the Synergistic Processing Unit (SPU) of the Cell processor, as well as graphics, physics and scientific computing boards. Accelerators which are comprised of *tens* or *hundreds* of cores can be dubbed *deca*- and *hecto*-core respectively, to distinguish them from the currently offered dual- and quad-core general-purpose processors.

Parallel and heterogeneous systems are fast and efficient in theory but are hard to program in practice. Unsurprisingly, efficient automatic parallelisation has been a programmer's sweet dream for many decades. Ideally, the programmer would like to write clear and concise code in a familiar, mainstream programming language assuming a single processor and uniform memory; concentrate on computation rather than on communication; and then sit back and relax, while the compiler automatically distributes the program across the target system in an efficient and error-free manner.

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

But the dream is but a dream. Difficulties abound.

Compilers excel at doing mechanical tasks, that are either unaccessible to programmers in high-level languages (such as register allocation or instruction scheduling) or too tedious (such as common subexpression elimination or strength reduction). Compilers cannot in general convert a sequential algorithm into a parallel one. Human ingenuity is required to invent a new parallel algorithm which can then be presented to the compiler for optimisation. Expressing an explicitly parallel algorithm in a sequential language, however, may not be natural.

These problems aside, semantics-preserving re-ordering of a sequential program requires accurate dependence analysis which is difficult in practice. Mainstream programming languages, particularly object-oriented ones, derive from the C/C++ model in which objects are manipulated *by reference*, for example, by using pointers. While such languages allow for efficient implementation on sequential computers, the possibility of *aliasing* between references complicates dependence analysis. Sadly, *alias analysis* is undecidable in theory and intractable in practice for large programs. This often precludes parallelisation, even when the programmer "knows" that computation can proceed in parallel.

The compiler's failure to notice opportunities for parallelisation which are obvious to the programmer provides a strong argument *against* the use of sequential languages. The programmer, however, can help the compiler by explicitly giving it more information about the sequential program than the compiler can extract itself. The grateful compiler can, in return, generate more efficient parallel code.

1.1 Restricted pointers in C99

In ANSI C99 [1], the programmer can declare a pointer with a **`restrict`** qualifier to *assert* that the data pointed to by the pointer in a given scope will not be accessed via any other pointer in that scope. *Correct* use of **`restrict`** has no effect on code semantics, but may enable certain compiler optimisations which would otherwise be precluded by the possibility of aliasing; incorrect use causes undefined behaviour.

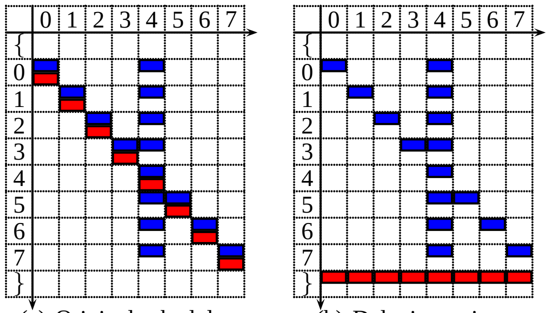
1.2 Sieve blocks in Sieve C++

In Codeplay's Sieve C++ [2–4], the programmer can place a code fragment inside a *sieve block* – a new lexical scope prefixed with the **`sieve`** keyword – 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.

We illustrate the sieve semantics using the following code fragment:

```
float* a; ...
for(int i = 0; i < 8; ++i)
  a[i] = a[i] + a[4];
```



(a) Original schedule. (b) Delaying writes.

Fig. 1. The schedules of memory accesses for the example of §1.2.

Fig. 1(a) represents the logical schedule of memory accesses. The x-axis shows the offset from `a`; the y-axis shows the logical time. Blue boxes represent reads; red boxes represent writes. For example, the row marked ‘0’ represents the first iteration of the loop, during which `a[0]` and `a[4]` are read, and `a[0]` is written. The write happens after the reads, hence the red box is placed lower than the blue boxes.

Placing this code fragment inside a sieve block:

```
sieve {
  for(int i = 0; i < 8; ++i)
    a[i] = a[i] + a[4];
} // writes to a[0:7] happen on exit
```

changes the schedule to that in Fig. 1(b). The side-effects are collected at the end of the block, as if the code has been raked from the beginning to the end of the block using a *sieve*, which is pervious to reads and impervious to writes (hence the name).

Note that this new schedule results in different values being written to `a[5:7]` because the write to `a[4]` is delayed.

1.3 Declaring, not instructing

The use of sieve blocks eases parallelisation, because the compiler is free to reorder computation involving reads from global memory. (The order of writes to global memory can be preserved by recording such writes in a FIFO queue and applying the queue on exit from the sieve block.)

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. (We will also say that such code does not generate true dependences on global memory.) In other words, for any column in the memory access schedule, no blue box is located below a red box (unlike in Fig. 1).

In this paper, we advocate that the use of the **`sieve`** keyword on a block should be treated as an *assertion* that code inside the sieve block generates no true dependences on global memory, rather than the directive to delay side-effects (as in §1.2). As in the case of **`restrict`** in C99, *correct* use of **`sieve`** will have no effect on code semantics; incorrect use will cause undefined behaviour.

We further illustrate and contrast use of the **`restrict`** and **`sieve`** keywords (§2), and explain that the assertion that code generates no true dependences not only makes code easier to develop and maintain (§3) but also may reduce the overheads of side-effect queueing (§4). We conclude by presenting experimental results (§5).

2 Mary Hope and the Delayed Side-Effects

In a free interpretation given in [3], the programmer informs the compiler that code inside a sieve block generates no true dependences. This seems sensible: why would the programmer ever want to write a new value into a global memory location and then read from this location, knowing that the write will be delayed (by his/her own request) and the read will return the old value of this memory location?

In this section, we illustrate that the programmer may want to exploit the sieve semantics, but argue that he/she should be discouraged from so doing.

2.1 FIR filter design

Suppose Mary Hope, an expert in digital signal processing, has designed a one-dimensional mean filter, for which the output at time i is given by the formula:

$$y_i = \frac{1}{k}\sum_{j=0}^{k-1} x_{i+j-\lfloor k/2\rfloor}, \text{ for } i = \lfloor k/2\rfloor, \ldots, n-1-\lfloor k/2\rfloor,$$

where $\{x_i\}$ and $\{y_i\}$ are, respectively, the input and output sequences (signals), n is the number of input samples, and k is the number of input samples to compute the mean over. Since the outputs of this finite impulse response filter can be computed in parallel, Mary hopes that the compiler will exploit her multi-core computer to speed up the processing.

2.2 FIR filter implementations

Implementation in C Mary decides to implement the filter in a familiar, efficient and portable language, *i.e.* C. She represents the signals as arrays, and passes them into the filter function in Fig. 2(a). Sadly, arrays in C are passed as pointers. As in most signal-processing codes, if two arrays *look* different, they *are* different. This domain-specific knowledge, however, is unknown to the compiler, which sees that the function receives pointers to two memory regions that *may* overlap. That is, on one iteration an assignment *may* write into a memory location that will be read on a subsequent iteration. This creates a loop-carried data dependence, which prevents the compiler from running loop iterations in parallel [5]. The compiler has to conservatively preserve the order of iterations of the outer loop.

Implementation in C99 Mary is aware of this problem and annotates pointers `x` and `y` with the C99 **`restrict`** qualifier as in Fig. 2(b), indicating to the compiler that the input and output memory regions do *not* overlap. Thus, the compiler can deduce that the outer loop is free of loop-carried dependences, and hence run its iterations in parallel.

```
void mean1d(float * x,
             float * y,
             int n, int k)
{
  for(int i = k/2; i < n-k/2; ++i) {
    float sum = 0.0f;
    for(int j = -k/2; j < k-k/2; ++j)
      sum += x[i+j];
    // write to disjoint memory?
    y[i] = sum / (float)k;
  }
}
```

(a) Implementation in C.

```
void mean1d(float * restrict x,
             float * restrict y,
             int n, int k)
{
  for(int i = k/2; i < n-k/2; ++i) {
    float sum = 0.0f;
    for(int j = -k/2; j < k-k/2; ++j)
      sum += x[i+j];
    // write to disjoint memory!
    y[i] = sum / (float)k;
  }
}
```

(b) Implementation in C99.

```
void mean1d(float * x, float * y,
             int n, int k)
{
 sieve {
  for(intitr i(k/2); i < n-k/2; ++i){
    float sum = 0.0f;
    for(int j = -k/2; j < k-k/2; ++j)
      sum += x[i+j];
    // delay writes to disjoint memory
    y[i] = sum / (float)k;
  }
 }
}
```

(c) Implementation in Sieve C++.

```
void mean1d(float * x,
             int n, int k)
{
 sieve {
  for(intitr i(k/2); i < n-k/2; ++i){
   float sum = 0.0f;
   for(int j = -k/2; j < k-k/2; ++j)
    sum += x[i+j];
   // delay writes to same memory
   x[i] = sum / (float)k;
  }
 }
}
```

(d) Controversial implementation in Sieve C++.

Fig. 2. Implementations of a one-dimensional mean filter in C, C99 and Sieve C++.

Implementation in Sieve C++ To Mary's disappointment, her favourite compiler lags behind the latest hardware trends and only generates single-threaded code, which effectively exploits instruction and subword-level parallelism, but underutilises Mary's multi-core computer. Mary sets out to exploit alternatives and finds a paper on Sieve C++ ([4] or [3]). Mary is pleased to learn that she only needs to enclose the function body in a sieve block and use an instance of the Sieve C++ *iterator class*[3] `intitr` to control the outer loop as in Fig. 2(c).

Controversial implementation in Sieve C++ After pondering a bit more on the sieve semantics, Mary modifies her code: she is happy to discard the inputs after the results are computed, so she assigns the results to the inputs as in Fig. 2(d). Since within the sieve block the writes to global memory are delayed, the computation produces exactly the same results as before, but – as Mary *thinks* – requires less memory.

In the next section, we will explain this is *not* the case, and discourage Mary Hope from writing code in this way.

3 Strict and relaxed sieving

3.1 Going from sequential to parallel

We may interpret the sieve construct as a means to add parallel semantics to a sequential language. We draw an analogy with different semantics of vector assignment statements in early PL/I and Fortran 90. Consider the statement: `a[0:7] = a[0:7] + a[4];`

[3] Iterator classes are described in [4].

One interpretation of this statement is the following loop (cf. PL/I before the ANSI standard of 1976 [6]):

```
for(int i = 0; i < 8; ++i)
  a[i] = a[i] + a[4];
```

which may look as if it adds a scalar to a vector, but does not quite: the value of `a[4]` changes after five iterations, so one value (the original value of `a[4]`, say `t`) is added to the first five elements of the vector and another value (`2t`) to the rest. So iterations need to be executed in the given order to ensure correctness.

Another interpretation is that no writes occur until all reads have completed (cf. Fortran 90 [5]), as in the sieve construct. This can be expressed as:

```
float t = a[4];
for(int i = 0; i < 8; ++i)
  a[i] = a[i] + t;
```

This loop indeed adds to vector `a[0:7]` a scalar `t` – the original value of its fifth element. Moreover, any order of iterations produces the same result. For this reason, we believe this interpretation is more natural in the age of parallelism.

Similarly, the sieve construct, which performs writes to global memory only after all reads and computation have completed, provides a natural way to endorse parallel semantics over a block of statements.

3.2 Block-based structure of the sieve construct

Sieve blocks generalise single statement vector assignments in two ways:

– Sieve blocks can define local memory, writes to which are immediate.
– Sieve blocks can be nested.

Sieve blocks have a natural interpretation ("read in, compute, write out", e.g. via DMA) on heterogeneous systems having complex memory hierarchies [3]. For example, ClearSpeed's CSX [7] is a SIMD array processor consisting of a control unit (CU) core and 96 identical processing element (PE) cores operating in lock-step. The CSX processor is located on a plug-in board together with large on-board memory. Each PE core has its own local memory.

Thus, a host workstation equipped with a ClearSpeed board has main memory, on-board memory, and PE memory. This complexity can be abstracted away by using nested sieve blocks, as in the following example:

```
int a = 0; // host memory
sieve {
  int b = 0; // on-board memory
  sieve {
    int c = 0; // PE memory
    a++; b++; c++;
    print(a,b,c); // prints 0,0,1
  }
  print(a,b); // prints 0,1
}
print(a); // prints 1
```

For each hierarchy level, delayed writes to non-local memory are queued. In this example, writes to a are queued twice.

3.3 Disadvantages of strict sieving

Strictly following the original sieve semantics (§1.2) in general incurs space and time overheads. The runtime system needs to maintain a FIFO queue of side-effects to non-local memory (space overhead) and apply the queue on exit from a sieve block (time overhead).

Moreover, delayed side-effects may need to be copied more than once. First, this may happen in the case of nested sieve blocks. Second, this may happen if the queue does not fit into local memory, in which case the runtime system needs to "spill" the queue to the previous level of memory hierarchy. On exit from the block, the runtime will have to apply the queue from the spill location.

Hopefully, the overheads will be compensated for by parallel execution. The compiler, however, may have a cost model and decide that it is not worth parallelising a sieve block. Still, the sieve semantics must be preserved, so the program is likely to suffer an execution overhead.

All this is clearly not in Mary Hope's interests.

3.4 Benefits of relaxed sieving

We suggest that Mary should be able to assert *explicitly* that code she places inside a sieve block does *not* generate true dependences on global data.

First, the explicit assertion gives the desired equivalence with the conventional semantics. By analogy, this is as if Mary writes:

```
float t = a[4];
a[0:7] = a[0:7] + t;
```

without relying on any specific semantics of vector assignments. The benefit is that code is easier to reason about, hence write and maintain.

Second, writes to non-local memory can be applied at *any* moment from their original schedule to the end of the block. For example, if the compiler believes that it is not worth parallelising a sieve block, it can generate sequential code *without* the overhead of maintaining the side-effect queue. As we will show in §4, optimisation can reduce this overhead on a parallel system.

We will say that the assertion that code generates no true dependences allows *relaxed* sieving (by this we emphasise that under this assertion side-effects do *not* need to be *strictly* delayed until the end of the block).

3.5 Analogy in HPF

Interestingly, this distinction between strict and relaxed sieving has an analogy with the **INDEPENDENT** directive in High Performance Fortran, which says that the loop it is attached to is safe to execute in parallel [8]. This directive is essential for loops that cannot be analysed (for example, if array subscripts are not affine functions of the loop variables). If the loop has loop-carried dependences, however, running it in parallel may produce different results from sequential execution.

While this conflicts with the design goal that an HPF program must always produce the same results whether executing on a parallel system or on a sequential one (for which the HPF directives are ignored), the standard defines the **`INDEPENDENT`** directive as an *assertion*, and dictates that programs which violate this assertion do not conform to the standard.

Similarly, under the *assertion* that a sieve block generates no true dependences on global memory the **`sieve`** keyword can be ignored when compiling for a sequential system.

3.6 Undefinedness of relaxed sieving

The ANSI C99 standard [1] specifies undefined behaviour as "behaviour, upon use of a nonportable or erroneous program construct or erroneous data, for which this International Standard imposes no requirements". An example is the use of expression `++i + ++i`, the result of which is compiler dependent.

Undefinedness is usually frowned upon, because it makes harder to write portable and reliable programs. Thus, introducing a new language construct with potentially undefined behaviour upon erroneous use may seem undesirable.

We note, however, that in the case of relaxed sieving the programmer's assertion (that code in a sieve block generates no true dependences on global memory) can be verified at run-time (and used for debugging) by additionally recording executed reads in the queue and checking that no read from a global memory location is preceded by a write to the same location. By contrast, it would be harder to verify at run-time an erroneous use of the **`restrict`** keyword.

4 Optimising relaxed sieving

4.1 Vectorisation

Suppose Mary Hope writes something like:

```
float * a;
sieve __attribute__((nodep(RAW)) {
  float t = a[5];
  for(intitr i(0); i < 8; ++i)
    a[i] = a[i+1] + t;
}
```

to inform the compiler that the enclosed code fragment does not generate a true dependence on global data and hence the compiler may relax sieving.

Fig. 3(a) represents the logical schedule of memory accesses. Since no write to a global memory location is followed by a read from the same location, the writes can be arbitrarily delayed (and the reads can be arbitrarily advanced) from their original schedule.

Assuming the architecture supports four-way vector instructions, the compiler may vectorise code producing:

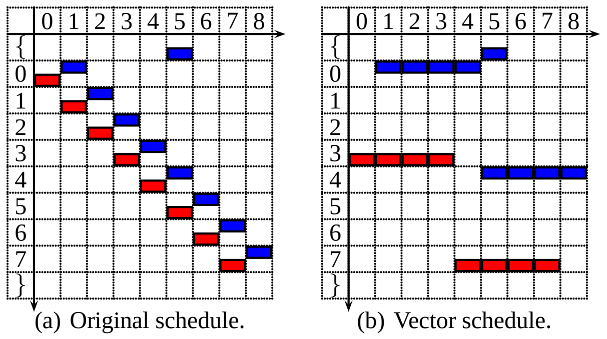


Fig. 3. The schedules of memory accesses for the example of §4.1.

```
float t = a[5];
for(int i = 0; i < 8; i +=4)
  a[i:i+3] = a[i+1:i+4] + t;
```

which has the memory access schedule as in Fig. 3(b), having *no* side-effect queueing overheads.

4.2 Speculation

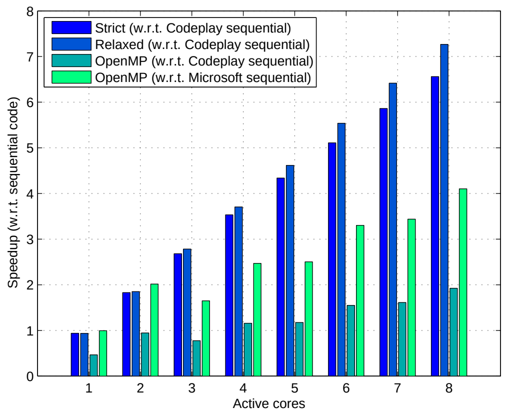
Note that if we speculatively distribute the first iteration of the vectorised loop above to one core and the second iteration to another, we cannot commit the side-effects of the second iteration (writes to `a[4:7]`) until the first iteration has read its input data `a[1:4]` (otherwise, the antidependence on `a[4]` is violated). In general, the side-effects of a fragment [4] need to be held until all its preceding fragments have completed and committed theirs, which also incurs overheads (although less than for strict sieving).

5 Experimental evaluation

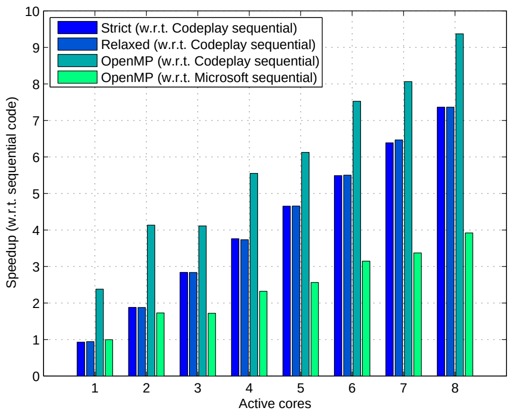
We present experimental data averaged over multiple runs on a homogeneous x86 multi-core system, with two 2GHz quad-core AMD Opteron (Barcelona) processors and 4GB RAM, running under Windows Server 2003.

5.1 Implementation

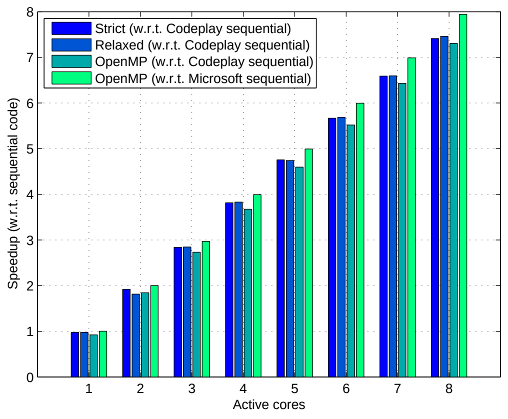
We have implemented a prototype extension to Codeplay's Sieve C++ compiler and runtime system that allows sieve blocks to be annotated using the syntax of §4 to indicate that writes can be arbitrarily delayed from their original schedule. Speculative execution [4] of these annotated sieve blocks results in side-effects that are committed, in order, by the run-time as soon as a speculated fragment is confirmed to simulate sequential execution.



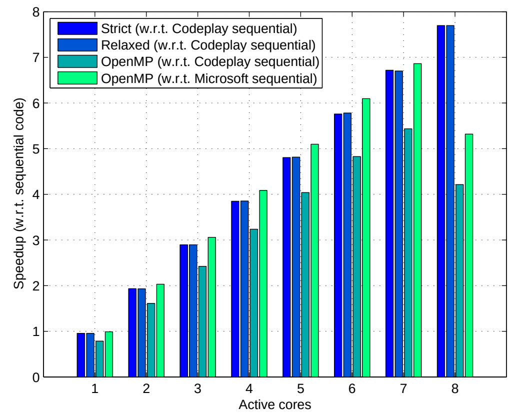
(a) Mandelbrot Set 8000×8000



(b) Julia Set 2000×2000



(c) Matrix Multiplication 2000×2000



(d) Cyclic Redundancy Check 32M

Fig. 4. Speedup w.r.t. sequential code.

5.2 Experimental setup

We use four benchmark programs. The matrix multiplication is performed for square 2000×2000 matrices. The cyclic redundancy check (CRC) is performed on a random 32M (1M = 2^{20}) word message. The Julia program ray traces a 2000×2000 3D slice of a 4D quaternion Julia set. The Mandelbrot program calculates a 8000×8000 fragment of the Mandelbrot set.

The graphs in Fig. 4 show (we used most aggressive compiler optimisation flags):

– the speedup of Sieve C++ programs over the original (sequential) C++ programs using strict (first bar) and relaxed (second bar) sieving (all code generated by Codeplay's Sieve C++ compiler);
– the speedup of C++ programs with OpenMP directives compiled by Microsoft's C++ compiler (version 14.00.50727.42, shipped with Visual Studio 2005) over the original C++ programs compiled by Codeplay's compiler (third bar) and Microsoft's compiler (fourth bar).

5.3 OpenMP vs. Sieve C++

The third bar shows the *performance* of OpenMP code relative to that of Sieve C++ code. For example, for the Mandelbrot benchmark [Fig. 4(a)] code generated by Microsoft's compiler to run on a single core is over two times slower than code generated by Codeplay's compiler. On the other hand, for the Julia benchmark [Fig. 4(b)] code generated by Microsoft's compiler is over two times faster than code generated by Codeplay's compiler.

The fourth bar shows the *scalability* of OpenMP code with the number of engaged cores. For the matrix multiplication [Fig. 4(c)] OpenMP code scales almost linearly and appreciably better than Sieve C++ code. Similar, for the CRC [Fig. 4(d)], with the exception of running on eight cores when the performance unexpectedly drops almost to the same level as running on five cores. OpenMP code for the Mandelbrot and Julia benchmarks scales sublinearly, running on eight cores only four times as fast as sequential code. In contrast, Sieve C++ code shows consistently good scalability.

Preliminary profiling using AMD's CodeAnalyst tool reveals no reasons for the OpenMP performance anomalies, but confirms that all cores are utilised thoughout execution, and shows no significant difference in L2 cache misses between configurations.

5.4 Strict sieving vs. relaxed sieving

Only the Mandelbrot benchmark [Fig. 4(a)] shows appreciably better performance improvement of relaxed sieving over strict sieving (up to 11% faster on eight cores), apparently because of the improved temporal locality. When working with large data sets, delaying side-effects until the end of a sieve block means that the side-effects of a speculated fragment get displaced from the cache by the side-effects of later fragments and are brought back to the cache when committing them to global memory. Committing the side-effects as soon as a speculated fragment is confirmed to simulate sequential execution reduces this overhead.

Note that Fig. 4 shows that Sieve C++ code implemented with relaxed sieving suffers a (small) performance overhead on a single core. As we have discussed in §3.4, the compiler might have chosen instead to output sequential code with no sieving, thereby incurring *no* overheads.

Fig. 5 shows memory overhead of sieving (memory overhead of OpenMP code is negligible). Note that the more cores are active, the more fragments can be speculated and their side-effects need to be held under relaxed sieving (for strict sieving overhead is invariant of the number of active cores). We report (maximum) memory overhead for eight active cores.

For the Mandelbrot benchmark relaxed sieving requires almost 50 times less memory overhead than strict sieving. The CRC benchmark performs a XOR reduction of a random message and generates a single value per fragment, hence creating small memory overheads in both strict and relaxed sieving.

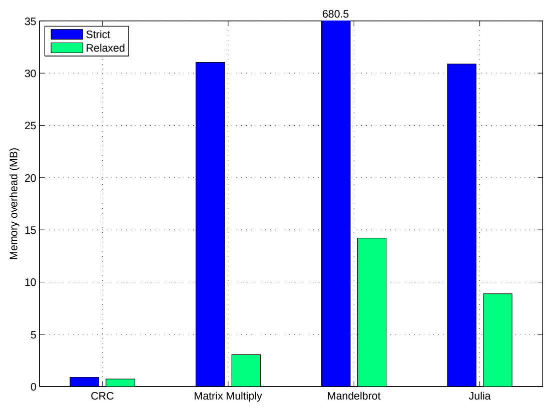


Fig. 5. Memory overhead.

6 Conclusion and future work

We have presented the concepts of strict and relaxed sieving. Automatic parallelisation based on sieving compares well to parallelisation via OpenMP pragmas. Relaxed sieving reduces memory overhead which can also result in performance improvement.

We are implementing relaxed sieving in the Sieve C++ backend for the Cell BE processor (previously reported in [4]) and will investigate further optimisations of relaxed sieving on heterogeneous multi-core platforms.

Acknowledgements

We thank Marcel Beemster for his constructive criticism that led to writing this paper and Paul Keir for his advice on implementing the benchmark examples in OpenMP.

References

1. American National Standards Institute: ANSI/ISO/IEC 9899-1999: Programming Languages – C. (1999)
2. Codeplay: Portable high-performance compilers. `http://www.codeplay.com/`
3. 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). Volume 4641 of Lecture Notes in Computer Science., Springer (2007) 641–650
4. Donaldson, A., Riley, C., Lokhmotov, A., Cook, A.: Auto-parallelisation of Sieve C++ programs. In: Proc. of the Workshop on Highly Parallel Processing on a Chip (HPPC). Volume 4854 of Lecture Notes in Computer Science., Springer (2007)
5. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann, San Francisco (2002)
6. Radin, G.: The early history and characteristics of PL/I. SIGPLAN Not. **13**(8) (1978) 227–241
7. ClearSpeed Technology: The CSX architecture. `http://www.clearspeed.com/`
8. Kennedy, K., Koelbel, C., Zima, H.P.: The rise and fall of High Performance Fortran: an historical object lesson. In: Proc. of the 3rd ACM SIGPLAN History of Programming Languages Conference (HOPL-III), ACM (2007) 1–22