Mainstream Parallel Array Programming on Cell

Paul Keir[1], Paul W. Cockshott[1] and Andrew Richards[2]

[1] School of Computing Science, University of Glasgow, UK,
[2] Codeplay Software Ltd., Edinburgh, UK,

Abstract. We present the E♯ compiler and runtime library for the 'F' subset of the Fortran 95 programming language. 'F' provides first-class support for arrays, allowing E♯ to implicitly evaluate array expressions in parallel using the SPU co-processors of the Cell Broadband Engine. We present performance results from four benchmarks that all demonstrate absolute speedups over equivalent 'C' or Fortran versions running on the PPU host processor. A significant benefit of this straightforward approach is that a serial implementation of any code is always available, providing code longevity, and a familiar development paradigm.

1 Introduction

Collection-oriented programming languages [1] are characterised by the provision of a built-in selection of operations to manipulate aggregate data structures in a holistic manner. Idiomatic code in these languages will commonly eshew the use of loop constructs. The potential to extract parallelism from this style of programming is consequently, and firstly, due to the *divisibility* of these aggregate data types; and secondly to the lack of side-effects in the expressions or constructs which stand in place of the imperative loops. Collection-oriented programming has often been applied to distributed parallel architectures, however it is just as relevant in the setting of heterogeneous multicore.

A perennial concern of performance-critical code structured around imperative loops appears within the context of implicit, or automatic, parallelism. An auto-parallelising compiler faced with a side-effecting loop which exhibits a sequential execution semantics, may overcome the challenge by a code transformation which introduces parallelism, along with locks or semaphores. The user of such a compiler is soon compelled to understand a new layer of diagnostic messages, which gradually cajole them towards an alternative, highly structured, coding style. The resulting code will often be data-parallel, and specify behaviour equivalent to that common to collection languages, though considerably more verbose.

In this paper we present a mainstream solution for scientific computing in the auto-parallelising array compiler, E♯, which targets the heterogeneous architecture of the Cell Broadband Engine. We also discuss the design decisions behind our implementation of four classic benchmarks[1] , before presenting an analysis of performance experiments.

1.1 Related Work

The seminal array language, originating in the 1960s, is Kenneth Iverson's APL. Subsequent decades brought a number of *parallel* array languages, for *distributed architec-*

[1]Available at `http://www.dcs.gla.ac.uk/people/personal/pkeir/hppc11code.7z`

tures: HPF, NESL, and ZPL being notable examples. Recent trends towards multicore systems have brought about a renaissance in the design of parallel array languages.

Single Assignment C (SAC) is a pioneering functional array research language based on the syntax and semantics of 'C'. SAC is distinguished by its first-class arrays, absent pointers and side-effects, and an advanced typing system capable of shape-polymorphic array function definitions. SAC has also recently targeted the heterogeneous GPU architecture via a CUDA backend [2]. The absence of a stack in CUDA however necessitates that no function calls are present within the SAC WITH-loops which provide the sites for parallelisation.

Cray's Chapel language, and IBM's X10, were the two finalists from the ten year DARPA High Productivity Computing Systems (HPCS) [3] programme. Both use a partitioned global address space (PGAS) model, and allow high-level, holistic, and parallelisable manipulation of arrays. The two standard implementations of these languages target distributed parallel architectures.

Microsoft's Accelerator project [4] targets homogeneous x86, and heterogeneous GPU architectures, using a data-parallel .NET array library which, by delaying the evaluation of array expressions, can minimise the creation of intermediate structures.

By virtue of the highly expressive Haskell typing system, the Repa [5] parallel array *library*, is refreshingly akin to an embedded array *language*. Absolute parallel performance comparable to serial 'C' derives from optimisations such as array fusion; and mandatory unboxed, strictly evaluated array elements. For the end user, performance can still depend on careful application of the force function; which replaces a *delayed* with a *manifest* array. Repa builds on the long-running Data-Parallel Haskell research strand, and for now targets only homogeneous multicore systems.

2 Implicit Parallelism using the 'F' Programming Language

Fortran was originally developed by John Backus and others at IBM in the 1950s. Like 'C', Fortran is a statically typed, imperative language. Fortran has historically differentiated itself from 'C' by its absent pointer arithmetic; longstanding support for complex numbers; argument passing by reference; and with *Fortran 90*, first class array types. The Fortran language is ISO standardised, and *Fortran 2008* has been approved. Of the mainstream programming languages, Fortran has distinguished itself within the field of computational science, due to its relatively high level, and excellent performance profile.

The 'F' programming language is a subset of *Fortran 95* designed with the intention of providing a lightweight version of Fortran, free of the requirement to support 40 years of language artifacts. The primary motivation of the language design was to create a Fortran-based language for education, however 'F' is a perfectly adequate general-purpose language. Furthermore, any Fortran compiler will compile a program conforming to the 'F' language standard, the g95 compiler also has a command line switch to enable error messages.

Having the requisite support for arrays, the 'F' programming language is therefore a suitable language to explore the use of array expressions as a mechanism to drive implicit parallelism for scientific computing on a heterogeneous architecture such as the Cell Broadband Engine.

2.1 A Language Primer

The following code excerpt demonstrates an entire ‘F’ program, equivalent to a ‘C’ *main* function. The assignment on line 3 will *pointwise* multiply the elements of the two arrays bound by `b` and `c`, before adding the result to a third array induced by the literal `3`. Once all operations on the right hand side of the assignment are completed, the result is copied to the array bound to `a`.

```
1 program p
2   real, dimension(2,3) :: a, b = 1, c = 2
3   a = b * c + 3
4   a = muladd(b,c,3)
5 end program p
```

Fig. 1: Assignments involving intrinsic and user defined elemental functions.

The dimension attribute specification on line 2 of Figure 1 declares three arrays, each with an explicit *shape* vector of 2;3. The length of an array’s shape vector provides another useful metric: its *rank*, and is therefore 2 in this case. The terms in an array expression must all have equal shape, and in doing so, the shapes are said to *conform*. Scalar values, such as the numeric literals `1`, `2` and `3`, are promoted, or lifted, to an array type of conforming shape, when their context within an expression requires it. The induced array is then populated by elements of the same value as the inducing scalar. In Figure 1, the expression `b * c + 3` is therefore an array expression with a rank of 2, and shape of 2;3. This is in fact true of all the expressions in Figure 1.

For the E♯ compiler, an array expression involving one or more functions, or operators, will be evaluated in parallel. The array expression `b * c + 3` from Figure 1 will therefore qualify, and so trigger the appropriate compiler transformations to ensure a parallel execution.

Scalar functions free of side-effects may also be applied to array arguments with conforming shapes. Such functions in Fortran are classified as *elemental*. The call to `muladd` on line 4 of Figure 1 represents a user defined `elemental` function producing the same result as the elemental arithmetic expression on line 3.

Unlike many auto-parallelising compilers, the E♯ user has the certainty that *all* array expressions will execute in parallel. Consequently, other iterative constructs of the ‘F’ language, such as `do` or `while` loops remain useful. Such constructs should be used where there is insufficient work to justify the small cost of thread adminstration and direct memory access (DMA) operations.

3 The E♯ compiler

E♯ is a source to source compiler, translating from the ‘F’ subset of Fortran 95 to Offload C++ [6]: a C++ language extension utilising pointer locality. The compiler targets heterogeneous multicore architectures, and in particular the CBE. The ‘F’ language has a

```
1 offloadThread_t tid = offload {
2   int outer *po = &g;
3   int i = *po;
4   int inner *pi = &i;
5   *pi = *pi+1;
6   *po = i;
7 };
8
9 offloadThreadJoin(tid);
```

Fig. 2: A simple asynchronous offload block expression

large standard library, and this is made available to both the PPU and the SPU using the GNU Fortran runtime libraries. A C++ template class has also been developed which both abstracts over the multifarious internal array representations of essentially all Fortran compilers; and is also compatible with the dual memory address hierarchy exposed by Offload C++. The E♯ compiler is written in the pure functional programming language Haskell. Haskell's Parsec parsing library allowed the structure of the published 'F' grammar to be followed exactly, while the Scrap Your Boilerplate package was used to perform the crucial transformations of the abstract syntax trees.

3.1 Targeting Offload C++

E♯ translates from 'F' to Offload C++, a C++ language extension and runtime library [6] targeting heterogeneous architectures. The most prominent language feature of Offload C++ is the *offload block* which provides a traditional 'C' compound statement, prefixed with the keyword `offload`, to be executed asynchronously to the main thread. Running on the CBE, each new thread will be executed by the next available SPU. An offload block returns an integer thread identifier and, like Pthreads, performance parallelism is achieved through the launch of multiple threads; subsequently joined with a call to `offloadThreadJoin`. A related benefit of this approach, is *automatic call-graph duplication*: with little or no annotation, a function, or variable reference, defined once, may be used both outside and inside an offload block.

Equally significant is the extension of the C++ type system to allow statically assigned pointer locality. In Offload C++ a pointer is, either implicitly or explicitly, identified either with an *inner* or *outer* locality. Pointer arithmetic and assignment between those of differing localities is statically prohibited by the compiler. More proactively, the dereferencing of an *outer* pointer from within an offload block corresponds to a DMA transfer from main memory to SPU scratch memory; while assignment to an *outer* pointer results in a DMA transfer in the opposite direction. Figure 2 demonstrates the concept: assuming the variable `g` is defined at global scope, the resulting effect of the offload block is for `g` to be incremented by one. Note that the `inner` and `outer` pointer qualifiers on lines 2 and 4 in Figure 2 are optional and would be automatically inferred.

```
template <typename T, int Od>
struct PtrWrapper {
  T inout(Od) *m_p;
};
```

Fig. 3: Offload C++ template struct with pointer member

Flexibility in the locality of class or struct pointer members may be obtained using the static `inout` qualifier and an integer template parameter, as shown in Figure 3.

3.2 A C++ Template Interface to Fortran Runtime Libraries

Neither the 'F' nor Fortran language standards specify an application binary interface (ABI) for arrays. With over a dozen Fortran compilers it would be unfortunate to restrict the E$\sharp$ compiler to only one of the associated runtime libraries. The Chasm project [7] helps addresses this issue by providing a low-level 'C' API targeting the internal "dope vectors" used by each Fortran compiler. For E$\sharp$, two new C++ array template classes have been designed and implemented: `ArrayT`; and the statically sized `ArrayTN`. Each provides high-level support for Fortran array features such as sectioning; serialisation; de-serialisation; and fast indexing with optional non-`1` lower bound.

3.3 Parallel Operational Semantics

The E$\sharp$ compiler attempts a human-readable, one-to-one correspondence between 'F' input and C++ output language constructs. An exception occurs at a parallelised array expression. In this instance, the array expression is transformed into a nested `for` loop[1], with depth equal to an expression's rank. A team of threads is then launched, each assigned a statically allocated and contiguous chunk of the outermost iteration space. The precise number of threads is set on program startup using an environment variable, `ESHARP_NUM_THREADS`, and may range from 1 to 128. Each individual thread is given the full resources of an SPU, and sits in a notional FIFO queue until one is available. While it can be assumed that launching one thread for each SPU will incur the lowest thread administration costs, while maximising resource usage, a program with a large working set may need to be split into more than six pieces. For example, an array expression with a 6000KiB working set, will exceed the 256KiB local store of an SPU if partitioned across six threads. With 32 or more threads, the program should run.

4 The Benchmark Programs

The first two benchmark programs we will examine, BlackScholes and Swaptions, are financial simulations from Princeton Univerity's PARSEC benchmark suite, converted by hand to 'F' from original C and C++. Our Mandelbrot program allows us to look at DMA transfer bottlenecks. while exploring differing approaches to parallel decomposition. Finally, a simulation of the n-body problem is examined.

[1]The operation is also recursively applied to array subexpressions.

4.1 Blackscholes

Blackscholes is a financial simulation which prices a portfolio of options using a partial differential equation now known as the *Black-Scholes equation*. Scalability in performance is obtained using a chunked, fine-grained decomposition, and calculating multiple options in parallel. The original implementation of Blackscholes uses Threading Building Blocks (TBB) and Pthreads to facilitate parallelism, with both using an *array of structs* configuration. Beneath the requisite file IO and threading boilerplate, there are two functions within the call graph of the parallel region: `BlkSchlsEqEuroNoDiv`, and a "callee" function `CNDF`. The kernel is given 100 runs, each of which is launched by an application of the TBB `parallel_for` template. This invokes multiple calls to a user-defined worker class's overloaded function operator. The 'F' version requires only that we mark the function as `elemental`, and the kernel launch is then
`prices = BlkSchlsEqEuroNoDiv(dat)`

4.2 Swaptions

The Swaptions program prices a portfolio of interest-rate swap options by the Heath-Jarrow-Morton framework using Monte-Carlo simulation. The original program consists of around fifteen C++ source files, then converted to 'F'.

Parallel decomposition on both TBB and Pthread implementations was, like Blackscholes, static and course-grained, though distinguished by a significantly larger working set. An *array of structs* configuration was again present in the C++ code, and the kernel was again dominated by a single 16-parameter function, `HJM_Swaption_Blocking`, applied in parallel to chunks from an one-dimensional iteration space.

The `HJM_Swaption_Blocking` function was ultimately a suitable target for `elemental` status, however the element type of two of its arguments are pointers to 1D and 2D arrays. An 'F' `elemental` function cannot accept arrays as a "scalar" element type, so necessitating the definition of two array wrapper types. With the 1D *pdYield* array, this amounts to the type shown in Figure 4.

```
1 type, public :: yieldT
2   real(kind=ki), dimension(m_iN) :: y
3 end type yieldT
```

Fig. 4: A scalar 'F' datatype wrapping an array

The C++0x code generated by E♯ from Figure 4 is shown in Figure 5. Notice that the `struct` has a template parameter, used to specify the *locality* of the data accessed via the `ArrayTN` member at line 18. That this is an `ArrayTN`, rather than an `ArrayT`, is an automatic optimisation due to the `m_iN` from line 2 of Figure 4 being a compile-time constant; the integer template argument `11` specifies the statically-allocated data size.

```
template <int Od>
struct yieldT {
  inline yieldT () {};
  inline yieldT (const ArrayTN<__compiler,float,1,11,Od> &&y)
    : y(y) {};
  inline friend ostream & operator << (ostream &o,
                                       const yieldT<Od> &t) {
    o << t.y; return o;
  };
  inline friend istream & operator >> (istream &i,
                                       yieldT<Od> &t) {
    i >> t.y; return i;
  };
  template <int Od2>
  inline yieldT &operator= (const yieldT<Od2> &rhs) {
    y = rhs.y; return *this;
  };
  ArrayTN<__compiler,float,1,11,Od> y;
};
```

Fig. 5: The C++ struct generated from Figure 4 by E♯

4.3 Mandelbrot

Estimation of the Mandelbrot set requires iteration of the complex function $z_{n+1} = z_n^2 + c$. Of the two Mandelbrot benchmarks we have developed, the first is more straightforward. An array of the same size as the 8-bit output image is initialised with positive integer coordinate pairs within the appropriate range, leaving the `elemental` function to create the complex value upon which it iterates. A second, blocked, version of the program partitions the coordinate array into squares. A user defined type is used for the squares, and is the scalar type upon which the requisite `elemental` function is defined.

4.4 The n-Body problem

From earlier work [8] we were aware that an $O(n^2)$ "all-pairs" n-body simulation on CBE can exhibit good scaling at the expense of wall clock time, and so a tiled decomposition of the problem, inspired by research at Nvidia [9], was developed.

The kernel of our n-body algorithm performs the $O(n^2)$ force calculation in parallel while the remaining leapfrog-Verlet integration updates the positions and velocities, and is run in serial by the host processor. This choice seems reasonable as having only linear complexity, the percentage of runtime expended on the remaining integration stage becomes insignificant with larger body counts. A square shaped tile of the pairwise body interactions, maximises the number of calculations that can be performed per body. That is to say, a DMA transfer of $2p$ body positions and masses, will provide p^2 components of force for the integrator.

The E♯ compiler parallelises only the outermost of the generated loops. To fully exploit the two-dimensional decomposition already outlined, a "flattened", *one-dimensional*, array is used to feed the requisite driving `elemental` function. User-defined scalar types, are once again required for the input and output elements. For input and output respectively the two types `pchunk2d` and `accel_chunk` are shown in Figure 6.

```
type, public :: pchunk2d
  type(vec4), pointer, dimension(:) :: ivec4, jvec4
end type pchunk2d

type, public :: accel_chunk
  type(vec3), dimension(CHUNK_SIZE) :: avec3
end type accel_chunk
```

Fig. 6: The n-body kernel input (`pchunk2d`) and output (`accel_chunk`) wrapper types

5 Experimental Evaluation

The following benchmark results were measured and averaged across five runs on a PlayStation 3 running Fedora Core 7. Single-precision was used throughout, due to the CBE's slow double-precision execution. In addition to the 4.1.1 versions of the GNU C, C++, and Fortran compilers provided with the installed IBM Cell SDK v3.0, version 4.6 of GCC is also installed. Where a speedup metric is presented, the fastest available PPU serial version is used, with selection based on source language; compiler; and the often powerful GCC switch: `-mcpu=cell`. The Offload C++ compiler version is 2.0.2, patched to use SPU GCC 4.6. All compilers use the `-O3` switch throughout.

BlackScholes This benchmark exceeds the memory limitations of the SPU at low thread counts. However, with 18 threads the E♯ version outperforms GCC after 4K options. With 64 threads, 256K options become possible, and provide a final speedup of over 11; shown in Figure 7. The surprisingly horizontal E♯ curves indicate that the problem is dominated by thread administration. The serial results also demonstrate that the 'F' version performs competitively with the independently constructed 'C' version.

Swaptions Though speedup values also increased, slightly, with input data size, Figure 8 shows only a maximum speedup of around 2.3x over GCC 4.6 with the "Large" data set. Lower thread counts for Swaptions are possible, and the graph demonstrates good scaling to 6 threads. While greater thread counts, up to 128, are shown as redundant in this configuration, it is encouraging to see that increasing thread administration overheads have no noticeable effect, as no fall in speedup is observed.

Mandelbrot As anticipated, the blocked version of Mandelbrot, using 64x64 squares, outperformed the naïve version, presumably due to reduced DMA traffic. The blocked version was also able to create a 2048x2048 image, and so achieve a speedup of almost 12x. Mandelbrot reaches a similar maximum speedup as Blackscholes; though a distinctive fall-off is also observed. See Figure 9.

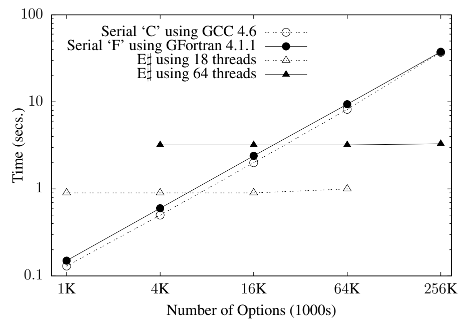

Fig. 7: Wallclock Blackscholes Kernel Timings for 100 iterations

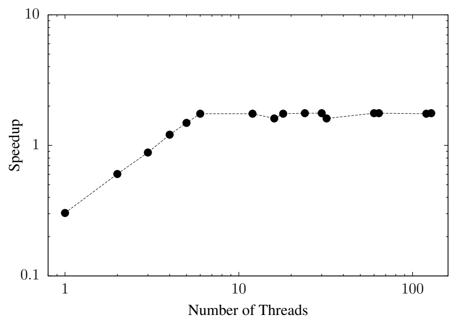

Fig. 8: E♯ Speedup with Swaptions and "Large" data set

The n-Body Problem Using 16x16 square tiles speedups increase gradually with data sizes, reaching a 3.4x speedup against the fastest 'C' configuration on PPU; which uses the older GCC 4.1.1 and the `-mcpu=cell` switch. In comparison to times obtained from the PPU only, using GFortran 4.1.1 `-mcpu=cell`, and the same 'F' code, a speedup of 4.9x is obtained with 16384 bodies; shown in Figure 10.

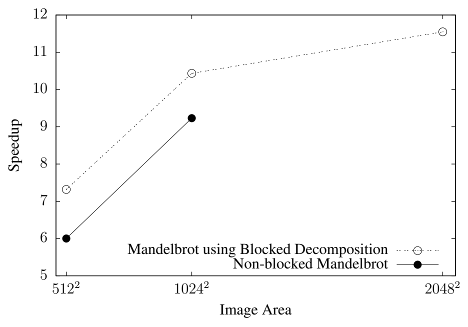

Fig. 9: Relative Mandelbrot Speedups against Image Area

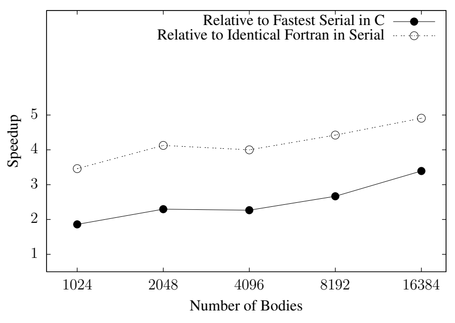

Fig. 10: Relative n-Body Speedups against Input Data Size

6 Conclusion

We have demonstrated the auto-parallelising array compiler, E♯, targeting the heterogeneous architecture of the Cell Broadband Engine. Encouraging performance results from four benchmarks are presented, and show speedups ranging from 2-11x over serial versions running on PPU only. The language employed, 'F', is a simple, useable, and standard dialect of modern Fortran, and is therefore well positioned for expected users from the scientific programming community. In addition, 'F' codes developed for use by E♯ are also valid Fortran; and shown to perform competitively in serial.

E♯ would benefit from the inclusion of streaming, rather than the current static partitioning of the iteration-space. This should allow access to a larger range of problem sizes, and hopefully more routine access to high performance. Also, as array expressions are free of side-effects, we can expose a finer level of granularity than currently offered by E♯, which presently partitions only the outermost rank. This should help load balancing on problems with small outer rank extents.

The techniques described here for the CBE could also be applied to new multicore processors such as Intel's Single-Chip Cloud Computer (SCC), or Knight's Corner. In the case of the SCC it would be possible to produce an E♯ compiler provided that a version of the Offload system were ported to the SCC. This is likely to result in somewhat lower performance than the Cell because of 3 factors: a) processors on the SCC can not initiate reads from host memory, b) inter-process communication on the SCC relies on a software library, RCCE, rather than the CBE's DMA hardware; c) the performance of the inidividual SCC processors is slower than the host Xeon, whereas the Cell SPUs are capable of higher throughput than the host PPC. For shared memory machines like Knight's Corner, we anticipate implementing E♯ by compiling to C++ with OpenMP pragmas. Some prototype work has already been done using this approach.

References

1. J. Sipelstein and G. E. Blelloch, "Collection-Oriented Languages," *Proceedings of the IEEE*, vol. 79, pp. 504–523, 1991.
2. J. Guo, J. Thiyagalingam, and S.-B. Scholz, "Breaking the GPU programming barrier with the auto-parallelising SAC compiler," in *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. ACM Press, 2011, pp. 15–24.
3. M. Weiland, "Chapel, Fortress and X10: novel languages for HPC," EPCC, The University of Edinburgh, Tech. Rep. HPCxTR0706, October 2007.
4. D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: Using Data Parallelism to Program GPUs for General-Purpose uses," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. ACM Press, 2006, pp. 325–335.
5. G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier, "Regular, shape-polymorphic, parallel arrays in Haskell," in *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*. ACM Press, 2010, pp. 261–272.
6. P. Cooper, U. Dolinsky, A. F. Donaldson, A. Richards, C. Riley, and G. Russell, "Offload - Automating Code Migration to Heterogeneous Multicore Systems," in *Proceedings of the 5th International Conference on High Performance and Embedded Architectures and Compilers*, vol. 5952. Springer, 2010, pp. 337–352.
7. C. E. Rasmussen, M. J. Sottile, S. S. Shende, and A. D. Malony, "Bridging the language gap in scientific computing: the Chasm approach," *Concurrency and Computation: Practice and Experience*, vol. 18, pp. 151–162, 2006.
8. A. F. Donaldson, P. Keir, and A. Lokhmotov, "Compile-time and Run-time Issues in an Auto-parallelisation System for the Cell BE Processor," in *Proceedings of the 2nd EuroPar Workshop on Highly Parallel Processing on a Chip*, vol. 5415. Springer, 2008, pp. 163–173.
9. M. H. Lars Nyland and J. Prins, "Fast N-Body Simulation with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley Professional, 2007, pp. 677–694.