# A Distributed Dynamic Aspect Machine for Scientific Software Development

Chanwit Kaewkasi
Centre for Novel Computing
School of Computer Science
University of Manchester
ckaewkasi@cs.man.ac.uk

John R. Gurd
Centre for Novel Computing
School of Computer Science
University of Manchester
jgurd@cs.man.ac.uk

## Abstract

This position paper proposes the use of an event-based dynamic AOP machine as an infrastructure for interactive development of high performance scientific software. Advice codes in the proposed approach are similar to mobile agents that execute on distributed computational nodes. The key ideas underlying this approach are multi-level separation of parallelisation concerns and event-driven dynamic join points. The primary aim of the research is to use the AOP paradigm to improve productivity for scientific software development; the dynamic AOP machine is also expected to be further developed as an interactive computational grid.

***Categories and Subject Descriptors*** D.3.4 [*Processors*]: Run-time environments

***General Terms*** Run-time environments, Languages

***Keywords*** Aspect-oriented Programming, Virtual Machine, High Performance Computing, Separation of Concerns

## 1. Introduction

Scientific software development often involves High Performance Computing (HPC) because many mathematical problems require considerable processing power. But performance is not the only requirement; productivity is becoming ever more important as scientific problems become more complex. In [13], there is a discussion of the characteristics of the High Productivity Computer System (HPCS) programme, under which it is sought to double productivity of Peta-flop systems every 18 months. The main target of the HPCS programme is to reduce the time-to-solution, rather than focusing only on hardware processing speed. The author suggests that the real cost for HPC codes consists of both the execution time and the development time. The execution time obviously relates to the performance of the executing code, including its parallelisation, while the development time may involve software engineering problems.

The work proposed in this paper focuses on applying an event-based dynamic aspect-oriented approach to reduce development time for HPC applications that run on distributed memory machines (DMMs). A DMM is a group of distributed computers that are connected together via a network in order to form a single computational resource. During the software development phase, interactivity is important for developers to gain productivity; fully compiling the whole program in every development cycle obviously impedes productivity.

This position paper proposes the concept and structure of a distributed dynamic aspect machine for scientific software to support interactive HPC software development with separation of concerns. The contributions of the proposed work are expected to be as follows. First, the use of an event-based dynamic approach [20] to aspect-oriented programming (AOP) [15] during the development phase may reduce the time to develop HPC programs. Second, the use of the proposed distributed advice code execution, which has the same characteristics as a mobile agent computing approach [6], may dramatically reduce communication overhead for data transfer, because the advice codes, in the form of mobile agents, will be mainly transferred among processors, rather than the processing data. It is also expected that the outcome of the proposed research, in the form of a distributed virtual machine, could be further developed to build an interactive computational grid for HPC software.

The remainder of the paper is organised as follows. Section 2 discusses related work dealing with parallelisation concerns and dynamic AOP, including the event-based approach. Section 3 introduces the concept of parallelisation sub-concerns, in particular Local Master Parallelisation and Remote Worker Parallelisation. Section 4 proposes the novel dynamic aspect machine for HPC applications. The paper ends in Section 5 with conclusions and ideas for future work.

## 2. Related Work

Two separate topics are pertinent, namely parallelisation concerns (and their limitations) and dynamic aspect-oriented programming.

### 2.1 Parallelisation Concerns

The proposed research follows on from the work of Harbulot and Gurd [7, 8], who have suggested that development of aspects for separating parallelisation concerns in HPC requires a new kind of join point model. They propose LoopsAJ, a join point model for loops, which has been implemented in `abc` [1] and is compatible with AspectJ's join point models. In their work, an analysis technique has been developed for finding loops in Java .class files. This technique detects back-edges to find appropriate weaving points for common kinds of advice, namely *before*, *after* and *around*.

The advantages of this technique are that it can be used for analysis of compiled Java programs and it is compiler-independent. However, in practice, parallelisation aspects actually occur at

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*Workshop VMIL '07* March 12-13, Vancouver, British Columbia, Canada
Copyright © 2007 ACM 1-59593-661-5/07/03. . . $5.00

*application-level*. Application-level aspects are defined as a set of aspects that are specific for their base code. This kind of aspect is similar to a hyper slice, as described in [19]. Unlike infrastructure-level aspects, such as logging, transactional behaviour or security, parallelisation aspects in the context of the proposed research are *strategies* for the algorithms that they are woven into. Characteristics of parallelism over loops in other programming models, such as OpenMP [3] and HPF [17], suggest that it is necessary to know the semantics of the variables that are accessed in each loop in order to convert the loop into the proper parallel form. If there is insufficient information about these variables, it may be difficult for the compiler to control the parallel blocks efficiently.

The authors themselves identify several problems that can limit their approach for processing loops [8]. First, a whole method analysis may be required in order to identify data dependencies among variables in loops to ensure that the loop can be made parallel; such analysis is obviously complex and time-consuming. A possible solution is to develop a compiler that is able to embed additional structural information (e.g. block markers), that can be retrieved via reflection or similar techniques, into the compiled program for further processing by the machine; this is the approach that will be taken in the proposed research.

Second, loops in scientific programs often involve a large number of local variables. The nature of block statements is different from that of methods, which often have defined fixed arguments; block statements, such as loops, have no mandatory list of arguments. The approach introduced in [8], that manages some local variables as the loop's arguments, may not be flexible enough for complex HPC programs. For example, Figure 1 shows a parallel code written using OpenMP directives.[1] One can see that three array variables are being used in the parallel loop; in a more complex HPC code, a much larger number of local variables might be expected. Rather than passing these as arguments to the loop, the contextual programming style introduced in [16] is more appropriate for accessing them. Contextual programming utilises Java 5 annotations by assigning values to annotated variables at runtime. This technique could help the advice code to be more readable because it is not necessary to declare the potentially large number of local variables in the argument list of the join point context.

Third, bytecode transformation may change the original semantics of a loop. This problem arises because the weaving process makes direct modifications to the loop block. The loop modification techniques that are used in [8] are, for example, moving invariants out of the loop (in order to expose them in the join point context) and insertion of advice codes in several places within the loop. It is difficult to prove that the woven code has the same loop semantics as the original. The technique proposed here to solve this problem is event-driven dynamic weaving [20], which is described below and will be implemented in the proposed work. It is not necessary for this kind of weaving to modify the base code.

### 2.2 Dynamic Aspect-Oriented Programming

An AOP system uses a weaving process to compose separated concerns into the base code. Unfortunately, a static and load-time weaving process could be a time-consuming task, as it is in AspectJ. This weaving time problem is raised by Bockisch et al. [2], who introduce an approach, called envelope-based weaving, to make static weaving faster by using *envelopes* to reduce the number of potential join point shadows. However, the dynamic aspect machine will tackle this problem using a different approach.

Dynamic weaving is a weaving process in AOP that allows the system to redefine aspects at runtime. Current implementations of

[1] This fragment of code is from the OpenMP tutorial at http://www.llnl.gov/computing/tutorials/openMP.

```
#include <omp.h>
#define N 1000

main () {
 int i;
 float a[N], b[N], c[N];

#pragma omp parallel shared(a,b,c) private(i)
 {
 #pragma omp sections nowait
  {

  #pragma omp section
  for (i=0; i < N/2; i++)
   c[i] = a[i] + b[i];

  #pragma omp section
  for (i=N/2; i < N; i++)
   c[i] = a[i] + b[i];
  }  /* end of sections */
 }  /* end of parallel section */
}
```

**Figure 1.** A simple OpenMP program.

| Where \ Characteristic | Weaving | |
|---|---|---|
| | Static | Dynamic |
| Compilation | Yes | No |
| Post-Compilation | Yes | No |
| Loading | Yes | No |
| Execution | No | Yes |
| **Techniques** | (1),(2) | (3),(4) |
| **Redeployment** | hard | easy |

(1) source transformation
(2) bytecode transformation
(3) runtime bytecode insertion
(4) dynamic join point dispatch

**Figure 2.** Summary of characteristics of different weaving processes.

dynamic weaving are, for example, PROSE [18] and Steamloom [10, 9]; both are AOP systems for the Java virtual machine (JVM) execution environment. There are several models for describing the behaviour of dynamic weaving [4]. The model which will be used in the proposed research is an event model. As explained in [20], events will be triggered at join points to invoke the weaver and execute advice codes that are matched by pointcut designators during the execution flow. PROSE [18] is known to support this join point execution model. This model allows straightforward implementation for block-level join points which are difficult to weave by a static weaving process [8]. Figure 2 gives a summary of the characteristics of different weaving processes.

An AOP system that employs the event model for dynamic weaving has advantages over a static weaving system. A key characteristic of a flexible AOP system is that it allows re-definition and re-weaving of aspects at any time. The event-based dynamic weaving model offers a dynamic join point based on triggering of events during the flow of program execution. This means that the weaver

will be invoked by the interpreter using this triggering mechanism, and then the associated advice will be executed to modify the execution of the main program. In an implementation of this event-based model, the *static join point shadow* [11], a widely used concept in other weaving implementations [5, 12], is not needed because no transformations will be applied to the base code.

The machine proposed in this paper extends the event-based join point approach to encompass distributed computing. Events will trigger the execution of advice code on both local and remote machines, as will be described in Section 4.

## 3. Parallelisation Sub-concerns

Separation of concerns is a key idea to manage software complexity in software engineering. It forces programmers to deal with the software development concern-by-concern [19]. For example, in scientific computing, one typically starts by developing a mathematical model, next implementing the model as a computer algorithm, and then improving this to become a parallel code. Such development steps follow the concept of separation of concerns as they tackle the three concerns (mathematics, discretisation and parallelisation) one at-a-time.

Using parallelisation concerns that have been explicitly defined and separated out of the program as aspects is a good starting point for applying AOP to scientific software development [7, 8]. However, parallelisation might be more specific than other crosscutting concerns, such as logging or transaction behaviour, depending on the base algorithms and the intended machine environment(s). In other words, parallelisation is a parallelism strategy for this base algorithm to gain the best performance on a (set of) specific machine(s).

Parallelisation concerns can be further separated into sub-concerns. We call this kind of separation *multi-level separation* because our studies show that some kinds of concern, such as parallel 'task farming', contain related sub-concerns inside. The following subsections define two such parallelisation sub-concerns, namely Local Master Parallelisation (LMP) and Remote Worker Parallelisation (RWP).

### 3.1 Local Master Parallelisation

Local Master Parallelisation (LMP) is a sub-concern which exists only on a master machine in a DMM. The primary tasks of this sub-concern are as follows. Firstly, the code in LMP may pre-process data for worker machines. The pre-processing steps in LMP are, for example, dividing a large array to fit the number of processors, or initialising data or variables for remote machines. Secondly, the advice in LMP is responsible for distributing data to other machines. However, the mechanisms for distribution depend on the communication layers. For example, the advice may use an explicit API for sending data to the workers. Thirdly, the advice of LMP might call `proceed` to perform computing, or `parproceed` (parallel proceed) to perform the same task remotely. The definition of `proceed` and its parallel counterparts will be investigated later. Next, the advice receives or collects the computation results from remote machines. These steps also depend on the communication layer. Finally, post-processing steps might be performed on the received results; for example, all sub-results may be reduced or merged into a single array.

### 3.2 Remote Worker Parallelisation

Remote Worker Parallelisation (RWP) is a sub-concern that exists on worker machines. RWP is usually responsible for performing numerical computing tasks. An important characteristic of RWP is that it can enable nested parallelism, which allows another level of parallel computing inside the parallel code. For example, the second `sort@remote` in Figure 3 is for performing nested parallel computation. In a heterogeneous HPC environment, some machines have a single processor, while other machines may contain multiple processors or multi-core processors. This means that they need different RWPs to achieve their best performance. Without AOP, multiple RWP concerns might make the algorithm difficult to read. More importantly, RWP can be the right solution for scalable heterogeneous DMMs in that the new RWP code can be easier to develop for future computing nodes.

```
function find2nd(a) {
  x = sort(a)
  return x[2]
}
```

```
aspect find2ndParallel {
  sort@local(a) {
    aa[] = split(a, node)
    x = []
    aa.each { it ->
      x  += parproceed(it)
    }
    return proceed(x)
  }
  sort@remote(cond:{cpu==1}, aa) {
    return proceed(aa)[1..2]
  }
  sort@remote(cond:{cpu > 1}, aa) {
    aaa[] = split(aa, cpu)
    x = []
    aaa.forkEach { it ->
      r = proceed(it)
      x += r[1..2];
    }
    return proceed(x)[1..2]
  }
}
```

**Figure 3.** A sample algorithm and its aspect.

Further, separating parallelisation into these two sub-concerns provides better modularity and it becomes easier to execute their associated advice codes with different processors. A pseudo code for an algorithm and its parallelisation aspect, which wraps LMP (`sort@local`) and RWPs (`sort@remote`) together, is shown in Figure 3.

## 4. Dynamic Aspect Machine

The concept of a dynamic aspect machine is introduced next. A dynamic aspect machine is a virtual machine that contains special steps in its fetch-execute cycle for executing programs that use dynamic aspects. Normally, a virtual machine executes a program by fetching a 'current' instruction from the program's image, pointing the program counter at the 'next' instruction and then executing the 'current' instruction. To work with this normal machine, static weaving mechanisms have traditionally been employed to insert advice codes into the base program [14, 12, 18, 10]. However, it is not necessary for advice code to be woven into the base code. The advice can instead be stored in a separated address space and, as long as it is able to execute correctly from there, a physical weaving process can be avoided. The associated execution mechanism might use an external referencing table to jump between the address spaces of the base program and its advice codes when *events* occur. This kind of execution model for AOP has been described in

[20, 4]. However, a new kind of machine is needed to support this model by adding extra steps into the machine's execution cycle.

The main advantage of storing advice codes in a separate space is that it is not necessary to perform physical insertion of these advice codes into the base program. So, several steps of bytecode transformation, which are significant parts of the static weaving process, can be eliminated. The advice code references can be kept as an external table, which, for example, store offsets of join points in the base program together with their associated advice addresses. Context information for the advice is assumed to be available on the operand stack.

It is our view that high productivity, interactive development of complex systems, such as HPC codes, requires the separation of concerns techniques of AOP together with an event-based dynamic weaving model. This model has a number of important characteristics. First, it is a general join point model for sequences of instructions; its event-driven nature can be applied to all kinds of statement, including loops and conditions. Second, the base code does not need to be transformed. This technique solves the problem that bytecode insertion might change the original semantics of the base code, especially for fine-grained statements. Third, this model fully supports separate execution of the advice code on one or more different machines.

### 4.1 Sequential Behaviour

The dynamic aspect machine operates by means of interception at a block join point. The definition of block in this context is similar to that of Java's synchronisation block; interception occurs at the entry and exit points to and from the block. The block join points are defined using entries in the *transparent join point table*. This model differs from Java's synchronisation block in that our approach does not need specific 'marker' instructions (e.g. JVM's `MORNITOR_ENTER` and `MORNITOR_EXIT`) in the base program; this implies that no change will be made to the program. The main purpose of the block join point model is to generalise the LoopsAJ join point model for loops [8] by adapting it to be event-driven.

Figure 4 shows how the block join point works with *around* advice code. The two black bars indicate that an event will be triggered at these points when (a) entering and (b) exiting the current block. At (1), the virtual machine fetches an advice instruction whose offset has been stored in the transparent join point table. This triggers the start of the dynamic weaving process. The machine pushes the current program counter (PC) onto the call stack. The before code is then executed using the same evaluation context as the base code. At (2), the `proceed` instruction has been reached. The current advice is stored onto the advice stack. The machine then switches back to the base code; the base code's PC is popped from the call stack and the program continues. If there is a jump to outside the block's region (PC $<$ block entry or PC $>$ maximum block exit), the virtual machine raises a runtime error. At (3), the machine reaches the instruction at the block exit and the PC is again pushed onto the call stack. The advice is popped from the advice stack and the machine then switches again to perform the advice code in the after region, starting from the advice's PC. Finally, at (4), the machine reaches the end of the after region and then returns to the base program at the end of the block.

### 4.2 Distributed Execution

As mentioned earlier, the advice can be executed in a distributed fashion. A distributed dynamic aspect machine consists of multiple advice execution units (AEUs) that are connected together using their own protocols. Each AEU is a processing unit which is responsible for executing advice codes that are triggered by dynamic join point events. The master machine contains a local AEU (LAEU), while worker machines contain remote AEUs (RAEUs).

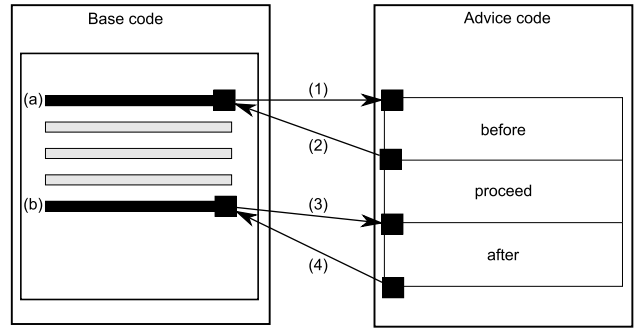


**Figure 4.** Behaviour of block join points.

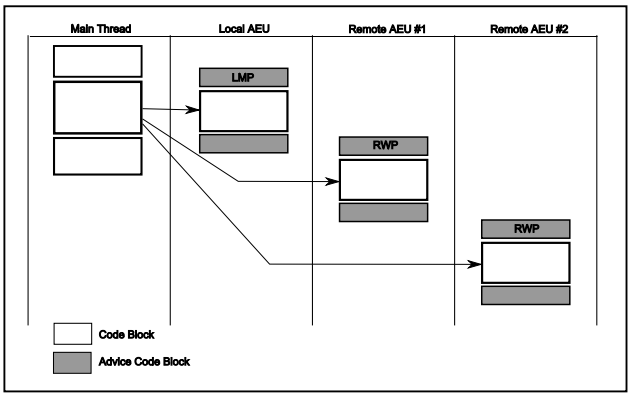


**Figure 5.** Sequence diagram showing the execution of a program in the distributed dynamic aspect machine.

An RAEU is a special case of an AEU which entails network communication. An RAEU receives the remote advice blocks, then it will be waiting for program fragments to-be-executed and associated data from the master machine.

For distributed execution, the basic execution steps from Section 4.1 need to be extended. The key actors are (1) the main thread, for executing the base program or algorithm, (2) the LAEU, for executing the advice codes for the LMP concerns, and (3) the RAEUs, for executing the advice codes for the RWP concerns. Figure 5 shows the execution steps in the proposed distributed machine. When the base program reaches a join point, it triggers the execution of all AEUs. At this point, the associated advice codes for each concern will be transferred to the appropriate AEUs. For example, from the pseudo code shown in Figure 3, the advice code `sort@local` will be sent to the LAEU, while the advice codes `sort@remote` will be sent to different RAEUs, according to their `cpu` pointcuts. After that, the appropriate base code will be sent to each AEU for use when the `proceed` instruction is encountered. Computational results can be returned from the RAEUs to be collected by the LAEU for a post-processing step. The final result will be returned to the main thread, which then continues by executing the next instruction.

Advice codes that are executed by RAEUs share some of the properties of mobile agents [6]. From the mobile agent point-of-view, each advice code will be serialized from the master machine in the form of a streaming JVM .class file. After receiving this advice code, an RAEU will instantiate and run it. When its task for this node finishes, the RAEU will serialize the advice code again with its current state and further migrate it to another AEU until the whole task is completed.

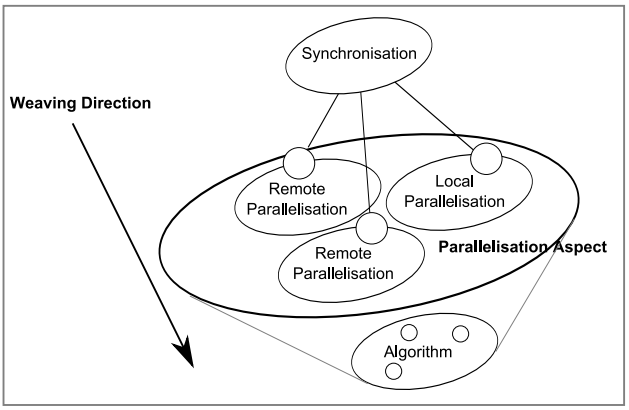


**Figure 6.** A possible way to further separate synchronisation concerns.

## 5. Conclusions and Future Work

This paper proposes a preliminary architecture for a distributed dynamic aspect machine that can aid high productivity, interactive, scientific software development. The machine has been specifically designed to cope with multi-level parallelisation concerns. The proposed machine is similar to the mobile agent computing model and is implemented using AEUs. Using this approach, it would also be possible to develop an interactive computational grid based on the proposed distributed dynamic aspect machine.

Figure 6 shows one possible way for further separating synchronisation concerns from an existing parallelisation aspect. After separating parallelisation concerns from the algorithm, one can see that synchronisation still cuts across a number of parallelisation concerns. An open question for multi-level separation of concerns in HPC and scientific software is whether synchronisation concerns should be separated out of the LMP and RWP sub-concerns. If yes, how should they be properly described? Is it possible to have another kind of structure to better describe synchronisation?

## References


[1] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhotak, Ondrej Lhotak, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM Press.

[2] Christoph Bockisch, Michael Haupt, Mira Mezini, and Ralf Mitschke. Envelope-based weaving for faster aspect compilers. In *NODe/GSEM*, pages 3–18, 2005.

[3] Rohit Chandra, Dave Kohr, Leonardo Dagum, Ramesh Menon, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.

[4] C. Dutchyn, G. Kiczales, and H. Masuhara. Aspect Sand Box. http://www.cs.ubc.ca/labs/spl/projects/asb.html, 2002.

[5] Eclipse.org. AspectJ project. http://www.eclipse.org/aspectj, 2006.

[6] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Trans. Softw. Eng.*, 24(5):342–361, 1998.

[7] Bruno Harbulot and John R. Gurd. Using AspectJ to separate concerns in parallel scientific Java code. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 122–131, New York, NY, USA, 2004. ACM Press.

[8] Bruno Harbulot and John R. Gurd. A join point for loops in AspectJ. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, 2006.

[9] M. Haupt. *Virtual Machine Support for Aspect-Oriented Programming Languages*. PhD thesis, Software Technology Group, Darmstadt University of Technology, 2006.

[10] M. Haupt, M. Mezini, C. Bockisch, T. Dinkelaker, M. Eichberg, and M. Krebs. An Execution Layer for Aspect-Oriented Programming Languages. In J. Vitek, editor, *Proceedings of the First International Conference on Virtual Execution Environments (VEE'05)*, pages 142–152, Chicago, USA, June 2005. ACM Press.

[11] Erik Hilsdale and Jim Hugunin. Advice weaving in aspectj. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, New York, NY, USA, 2004. ACM Press.

[12] JBoss.org. JBoss AOP. http://labs.jboss.org/portal/jbossaop, 2006.

[13] J. Kepner. HPC productivity: An overarching view. *International Journal of High Performance Computing Applications*, 18(4):393–398, 2004.

[14] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science, ECOOP 2001 - Object-Oriented Programming: 15th European Conference*, 2072:327, June 2001.

[15] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[16] Gavin King. JSR 299: Web Beans. http://www.jcp.org/en/jsr/detail?id=299, June 2006.

[17] David B. Loveman. High performance fortran. *IEEE Parallel & Distributed Technology: Systems & Technology*, 1(1):25–42, February 1993.

[18] Angela Nicoara and Gustavo Alonso. Dynamic AOP with PROSE. In *Proceedings of International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA 2005) in conjunction with the 17th Conference on Advanced Information Systems Engineering (CAISE 2005)*, 2005.

[19] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[20] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.