Improving Distributed OS Performance by Flexible Incremental Linking

Markus Pizka
Technische Universität München, Institut für Informatik - I4
Boltzmannstr. 3, Germany - 85748 Garching

Abstract *Distributed Systems suffer from the enormous performance gap between local and remote operation. To bridge this gap a general purpose distributed operating system must provide a variety of techniques for a single purpose, such as both RPC and replication, for remote data access. Depending on the situation the OS has to select the most suitable technique dynamically. But the indirections needed for dynamic switching between different techniques must be as efficient as possible to avoid constant overhead. In this paper, we describe the design and implementation of the highly flexible linker FLink that allows the OS to switch between different techniques at runtime without constant overhead.*

Keywords: distributed systems, operating systems, high performance

1 Introduction

Achieving peak performance in parallel systems is challenging and even more difficult in the case of MIMD [1] like distributed systems [2], consisting of networks of workstations with no direct access to remote memory. Trying to build a general purpose distributed operating system (DOS) that delivers at least reasonable performance seems to push these difficulties to the extreme.

Since the late 1980s a huge amount of work has been invested into the design and implementation of general purpose DOS with systems such as Amoeba, Muse, Apertos, Mungi, Sprite, Chorus, MoDiS, and Spring [3, 4, 5, 6, 7, 8, 9, 10]. Without any doubts all of these projects contributed significantly to the advancement of the OS field. But as we take a look back on these initiatives 10 years later we have to admit that DOS have failed to live up to their expectations. Research interest is declining and there are still hardly any commercial DOS products although the demand seems to be increasing.

1.1 Flexibility and the 10^5 Gap

Obviously there are several technical as well as non-technical reasons for this, such as compatibility with existing applications and platforms, and the inherent complexity of distributed algorithms. But even in the presence of increasing popularity of interpreted languages like Java [11], *insufficient performance* still remains a major obstacle for further progress in the DOS field. This is because suboptimal resource management decisions in a DOS may not just cause some performance losses but orders of magnitudes.

latency (nanosec.)			**bandwidth** (MB/s)	
L1	mem	TCP	mem	TCP
12	273	2.5*10^6	150	44

Table 1: performance gap

The root of this phenomenon is the enormous gap between local (i. e. direct memory or even cache) and remote (i. e. network) access. Table 1 compares the latency and bandwidth of local versus remote accesses on a SUN V9 and 100 MBit/s FastEthernet. Particularly for small messages, the difference in speed between local and remote access may differ by a factor of up to 10^5.

Clearly a DOS must avoid this performance penalty as far as possible by keeping remote communication to a minimum while trying to maximize the utilization of distributed resources. Consequently, a DOS must carefully distinguish local and remote operations and treat them with completely different strategies. In [12] it was demonstrated how accesses to remote objects can be significantly optimized at the DOS level by dynamically switching between object replication, object migration, and remote invocation. The criteria for the dynamic selection of one of these mechanisms are the frequency of reads and writes. Obviously this is just one important example for the need of flexibility

and the results can be transfered to other tasks, such as to avoid false sharing in DSM systems [13], adjusting the granularity of transport units [14] and thread creation.

Thus, the 10^5 gap must be met with a great deal of flexibility by the DOS. Uniform resource management strategies will always deliver unacceptable performance in some situation.

1.2 Constant Overhead

Besides the lack of flexibility, constant overhead is the second major source for performance deficiencies. Comparisons of distributed systems with parallel multiprocessor systems or even conventional uniprocessors as in [15] expose this difficulty. While the DOS described in this study achieves perfect relative speed-up for a matrix multiplication example, one can also see that 5 nodes are needed to achieve the performance of a similarly powerful uniprocessor.

Obviously, multiplicative resource requirements and absolute slowdowns are in general not desirable. If we further keep in mind that realistic parallel algorithms contain a significant amount of sequential work and combine this with Amdahl's law [16], which gives an upper bound for the relative speedup dependent on the amount of sequential work, then it becomes evident that it is impossible to deliver satisfactory performance if there is significant constant overhead.

Let $t_s+t_p = 1$ be the ratio of sequential resp. parallel amount of work, n the number of processors, l the constant local overhead and y the additional amount of work solely needed for remote operations. S_v approximates the relative speedup by augmenting Amdahl's law with the influence of additional overhead.

$$S_v(l,y) = \frac{t_s + t_p}{t_n} = \frac{n}{(t_s * (n-y) + y) * l}$$

As one can see, minimizing the impact of constant overhead is crucial for achieving reasonable performance of the system.

1.3 Setting the Goal

Now, the requirement to bridge the 10^5 gap by means of increased flexibility on the one hand contradicts the constraint to minimize constant overhead on the other hand. Therefore we have to investigate innovative ways to achieve flexibility while preserving low overhead.

As synchronous runtime measures, such as the evaluation of conditionals in the path of the computation, inescapably introduce constant overhead our goal was to develop new techniques that are not necessarily bound to the execution of the system. Besides improving the compiler [17] and applying dynamic recompilation exploiting advanced link loader techniques seemed to be a promising new approach for more flexibility with controllable overhead.

Based on these observations, the goal of the work presented in this paper was to develop an incremental and dynamic link loader with the following properties:

1. enable dynamic binding and unbinding of individual components
2. provide a set of binding intensities representing different grades of flexibility
3. allow the simultaneous application of the different binding intensities on different parts of the system
4. allow dynamic switching between binding intensities

1.4 Outline

The remainder of this paper is organized as follows. In section 2 we will first discuss related work on advanced link loader technology before we discuss design principles of our flexible incremental link loader FLink and its integration into the DOS architecture in section 3.2. Section 4 describes the different binding intensities available in FLink and their pros and cons. Section 5 will give some performance numbers of FLink before we summarize the results of this work in section 6.

2 Related Work

Commercial products such as Solaris [18] and Linux [19], employ deferred and dynamic linking techniques to minimize start up times and memory consumption of applications. One interesting feature is lazy evaluation of open references. All references from application level components to a library component c are set to point to the entry $plt(c)$ within the *procedure linkage table*. Initially each *plt* entry is a code fragment that activates the linker. Thus, the first call of c results in a jump to $plt(c)$ which activates the linker, resolves the address of c and replaces the entry $plt(c)$ with a newly

generated instruction sequence that branches to c. This technique is highly efficient. The additional jump causes hardly any overhead while relocation is simplified and flexibility is increased, because now it is possible to replace an implementation $\hat{c}_1$ of c with a different implementation $\hat{c}_2$ by modifying $plt(c)$ at runtime. Our linker FLink uses a similar technique to overcome the difficulties in distributed system as described above.

The experimental linker Alto [20] extends the capabilities of the compiler by performing inter-module data flow analysis and corresponding optimizations. Performance measurements of sequential algorithms showed average gains of 25%. This indicates that there is significant potential for performance improvements in advanced linking techniques. OMOS [21] uses a combined link/load server process for the dynamic management of executables. E. g. the locality of instruction execution is enhanced by monitoring access profiles and performing code movement at runtime. Cowan [22] successfully uses linking techniques to achieve dynamic reconfigurability of an adaptive OS.

Clearly, linking techniques play a subordinate role relative to fields such as OS kernels and compilers. Nevertheless, linking techniques should not be neglected. The research projects mentioned above, demonstrate that advanced linking techniques can contribute to significant overall performance improvements; especially if new linker features are adequately combined with new compiler and low level OS facilities.

3 OS Architecture

We consider all software tools involved in resource management decisions as the means to implement a DOS [23, 24]. This includes the OS kernel, runtime libraries, loader, linker, inlined and runtime generated code, and the compiler. The effectiveness of the whole DOS depends on both, the services provided by each tool and their interplay. Projects like Orca [25] and TreadMarks [26] demonstrated the performance gains achievable by integrating these different tools.

3.1 Role of the Linker FLink

With the linker FLink we follow this path. As the linker can do little to increase the performance of a distributed system by itself its value arises from its role as a mediator between compile time and run time. Our idea was to enhance the capabilities of the linker to enable a tighter integration of the compiler with the runtime system and the kernel, which opens a variety of possibilities for significant performance improvements. So, our view of the role of FLink for the DOS is:

- If the compiler can not decide on how to optimize a certain demand let the compiler prepare a set of alternatives.
- The kernel and the runtime system monitor the behavior of the system during execution and select the most suitable alternative provided by the compiler, dynamically.
- The selection is enforced without constant overhead by (re-)linking parts of the system during execution with sophisticated linking techniques provided by FLink.

3.2 Examples

Figure 1 illustrates two examples for the application of FLink within our DOS.

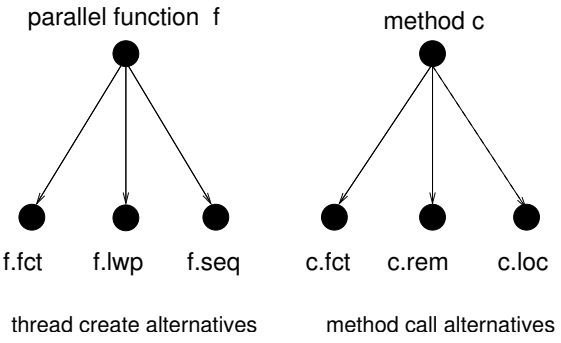

Figure 1: Production of alternatives

A major problem in parallel and distributed processing is the determination of an appropriate granularity of parallel computations; i. e. map a parallel entity on a local or remote thread or process, or execute it sequentially. Because of the absence of a solution to this problem, trivial parallel algorithms, such as

$$\textbf{for } \text{i:=}0 \textbf{ to } n \textbf{ do parallel } f(i);$$

can not be formulated directly but must be expanded with conditionals, synchronization primitives, etc. to arrive at an efficient implementation.

FLink aids in shifting this problem from the application level down to the OS level. For each parallel function `f` our compiler produces several alternative code modules. For example `f.fct` represents the body of the computation of `f`, `f.lwp` is the wrapper for `f` that calls `f.fct` from a newly

created thread, and `f.seq` is the sequential wrapper. The runtime system selects one of these possibilities according to the current load situation and monitoring information about `f`, and instructs the linker to relink references to `f`. Notice, without the linker, the decision had to be made for each call of `f` entailing high constant overhead.

We use the same concept to implement method invocation efficiently. Here, for each method `c` the compiler prepares different alternatives for remote invocation (`c.rem`) and local invocation (`c.loc`). Remote and local method invocation differ significantly and using just one uniform technique would lead to a severe performance degradation.

4 Binding Intensities

The ability to chose, e. g. between a remote method invocation mechanism or a light weight local procedure call raises a new difficult question. Under what circumstances is the effort profitable?

Invoking the linker at runtime is of course time consuming. Therefore, relinking is only reasonable if the anticipated benefit exceeds the costs of the link operation.

Obviously, the benefit or loss due to relinking at runtime strongly depends on the frequency of changes to the physical distribution and resource allocation. If for example, object migration occurs infrequently even high costs for relinking will be profitable. But if the migration frequency is high, even low extra costs will not be justifiable. Of course, object migration is just one aspect among many others, such as read/write ratios, load distribution, communication patterns, etc. There are two possible solutions to this dilemma between the needs for resp. costs of flexibility:

1. abolish the idea of dynamic relinking
2. divide the range of possible situations into a manageable set of classes and provide suitable options

Consequently adopting strategy 1 would mean to give up on solving difficult resource management problems. In fact, common OS tasks, such as paging, deal successfully with similar uncertainty by applying strategy 2.

We also aimed at solving this dilemma between flexibility and costs by equipping FLink with three different binding intensities. Each of these intensities has different characteristics concerning the costs for a) establishing a binding and b) using it[1].

[1]Bindings are "used" iff a method, function, or procedure is called.

caller / out refs	callee a	b	c	d	note
`a`/6	2	1	0	3	recursive
`b`/3	0	0	1	2	mutually recursive
`c`/3	0	1	0	2	
`d`/0	0	0	0	0	

Table 2: Module dependencies

The descriptions of the three different binding intensities in the following paragraphs use the example scenario shown in table 2. There a four code modules (e. g. compiled methods) `a` to `c` with references between them. Due to call dependencies, module `a` for example, contains a total of 6 references to the other modules; 2 to itself, 1 to `b` and 3 to `d`.

4.1 Immediate Binding

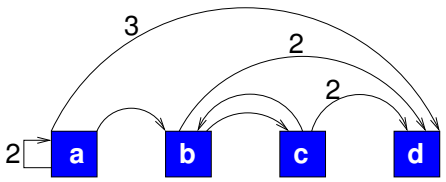

Figure 2: Immediate binding

A straight forward binding technique supported by FLink is immediate binding. References from a module x to a module y are replaced with the address[2] of y. The advantage of this binding is the efficiency of its use. The processor can immediately interpret the address of the callee when executing a method invocation therewith performing method invocations with lowest possible costs.

The disadvantage of this technique is its relative inflexibility. There are usually many direct references to a single module, such as to module `d` in figure 2. If we wanted to replace `d` with an alternative implementation `d'` we would have to first unbind 7 bindings and then establish 7 new ones. These expenses will rarely be profitable.

Due to its efficiency, common linkers usually use immediate binding. In the case of our DOS, immediate binding is used for modules that do not need increased flexibility; e. g. because of not being exposed to physical distribution.

[2]usually the virtual address

4.2 Mediated – Trampolining

A more flexible technique with low overhead is what we call *trampolining*. Similar to the procedure linkage table discussed in section 2, each mediately bound module y is preceded with its dedicated trampoline $t(y)$. All references to a mediately bound module y are no longer replaced with the address of y but with the address of $t(y)$ (see figure 3).

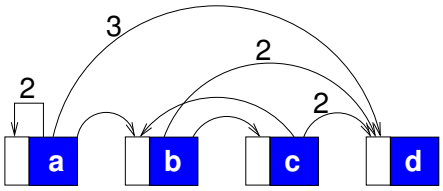

Figure 3: Mediated binding

The trampoline $t(y)$ is generated on demand at runtime. It consists of a sequence of instructions that loads the address of y and performs a jump to y (see figure 4)..

```
set   <APIB>, %g1
jmpl  %g1, %g0
nop
```

Figure 4: SPARC V9 Trampoline

The advantage of this technique is the bundling of dependencies. In our example, `d` can now easily be replaced with `d'` by generating a single new instruction within t(`d`). All references to t(`d`) are left untouched. The disadvantage of trampolining is some performance loss for executing of the trampoline code.

4.3 Weak Binding

The third technique, called *weak binding*, provides the greatest flexibility. Here, a search algorithm that we call *mediator* is used to interpret bindings. All references to a weakly bound module are replaced with the address of the mediator. Therefore, each call of weakly bound module y gets first redirected to a call of the mediator. The mediator then uses a linkage table to sub-call y. Note the simplicity of linkage. There is exactly one reference pointing to each weakly bound module and just one module being referenced, which is the mediator.

Weak binding is even more flexible than the mediated variant because it allows us to integrate arbitrary features and tasks into the mediator algorithm. For example, the mediator can be used to

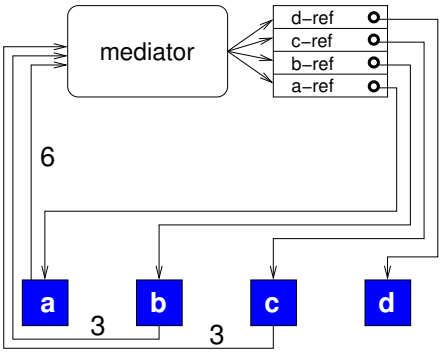

Figure 5: weak binding

enforce access restrictions or to keep track of the cumulative and current use of certain modules by registering module entries and exits. Of course, this extra flexibility comes with additional costs. Each call of a weakly bound module entails an extra call of the mediator and the execution of its algorithm.

4.4 Flexible Operation

It should be mentioned that FLink allows to use these different binding techniques simultaneously on different parts of the system and with some restrictions even on a single module. Modules can be linked and unlinked and it is also possible to change the binding technique used for a certain module during runtime. Because of this, it is possible that a reference is open, i. e. not bound to the target module with any of the techniques described above.

FLink handles such open references with a special form of weak binding, called *fail-safe* binding. All open references refer to the fail-safe entry in the mediator linkage table. Thus, executing an unbound reference entails activating the mediator resulting in a call of the fail-safe handler. We use this fail-safe binding to implement incremental dynamic extensibility [27]. The fail-safe handler performs a search for an executable target module according to the execution structure of the caller. If there is no such module it asks the user to provide it by activating the editor, compiles the new module, and links it dynamically to the running system.

5 Performance Analysis

Concerning our goals stated in 1.3 we aimed at increased DOS performance by bridging the 10^5 gap. Our aim was to contribute to this goal by increasing the flexibility of the linker. As discussed in 4 we designed and implemented a repertoire of binding

intensities to adequately meet the different needs for more or less flexibility. Obviously, the actual performance benefits or losses for parallel and distributed systems depend on two factors:

1. The efficiency of the alternative parallel and distributed resource management mechanisms provided to the linker, e. .g object migration, replication, and remote method invocation.

2. The extra cost introduced by the flexible linking techniques themselves.

As 1 is subject to detailed studies of certain techniques, we concentrate on discussing 2, here.

```
function foo(i: int): int
⌈ return (i+1); ⌋
```

Figure 6: Sample function foo

Table 3 shows performance results of executing function `foo` (see figure 6) with different binding intensities using our FLink implementation on Sun SPARC V9. Clearly, immediate binding outperforms mediated and weak binding with the cost of hardly any flexibility. Surprisingly, the more flexible mediated binding entails only a loss of $\approx 50\%$. Considering that more realistic functions will contain longer statement lists and therefore provide a better ratio between computation and calls, mediated binding can be considered a highly efficient alternative to immediate binding.

The costs for weak binding are high. Unfortunately, similar table driven techniques have been used intensively in the field of extensible OS kernels, such as Mach and Chorus [28, 8]. Presumably, the performance degradation due to weak binding was a major reason for the lack of acceptance of these systems. Weak binding should only be used if there is confidence in the frequent need for flexibility.

	time for n calls of `foo` in μsec			
n	imm.	mediate	weak	fail-safe
100	25	38	2380	4400
1000	230	360	24000	44100
10000	2400	3500	242000	440000

Table 3: Performance data

6 Conclusion

So far, general purpose distributed and parallel operating systems have failed to live up to their expectations. In this paper we argued that substantial performance problems are a major reason for this. We identified the 10^5 performance discrepancy between local and remote operation and constant overhead as the roots of these problems. We claim, that these problems are inherent to distributed computing and can not be solved satisfactorily with improved hardware, such as faster networks.

Instead, great efforts at the OS level are necessary to cope with the 10^5 gap while keeping the additional overhead low. The new flexible incremental and dynamic linker FLink addresses both aspects. Its dynamic binding and unbinding features open a new perspective for flexibility at the OS level because it allows the distributed OS to dynamically adapt its mechanisms to the needs of the application level. Furthermore, the three different binding intensities provided by FLink allow the OS to keep flexibility in balance with their costs. FLink has been implemented and its performance was measured. The results of this measurement help to better understand the consequences of different binding intensities.

Clearly, the linker FLink can not improve parallel and distributed performance by itself but it is the enabling technology for a whole range of new strategies because it provides for a close integration of compile-time with runtime. Consequently, our future work will be to exploit these new linker features by providing the OS with sets of alternative mechanisms for distributed resource management tasks.

Acknowledgment It is pleasure to thank Christian Rehn for his continuous support of this work. He contributed significantly to the idea of flexible incremental linking and implemented the first FLink prototype in his master thesis [29].

References

[1] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, 1972.

[2] Philip H. Enslow Jr. What is a "distributed" data processing system? *Computer*, 1978(1):13–21, January 1978.

[3] S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer Magazine*, pages 44–53, May 1990.

[4] Y. Yokote, A. Mitsuzawa, N. Fujinami, and M. Tokoro. Reflective object management in the

muse operating system. In *Proceedings of the 1991 International Workshop on Object Orientation in Operating Systems*, pages 16 – 23. IEEE Computer Society Press, 1991.

[5] Y. Yokote. The apertos reflective operating system – the concept and its implementation. In *Proceedings of the Conference on Object-Orientated Programming Systems, Languages and Applications (OOPSLA)*, pages 414 – 434. ACM Press, 1992.

[6] G. Heiser, K. Elphinstone, S. Russell, and J. Vochteloo. Mungi: A distributed single address-space operating system. Technical Report SCS &E Report 9314, School of Computer Science and Engineering, University of New South Wales, November 1993.

[7] J. K. Ousterhout, A. R. Cherenson, F. Douglis, and M. N. Nelson B. B. Nelson. The Sprite network operating system. *Computer*, 21(2):23–36, 1988.

[8] The Chorus Team. Overview of the Chorus distributed operating system. In *USENIX Workshop on Microkernels and other Kernel-Architectures*, Seattle, WA, 1992.

[9] C. Eckert and H.-M. Windisch. A new approach to match operating systems to application needs. In *Int. Conf. on Parallel and Distributed Computing and Systems*, pages 499 – 503, Washington, USA, Oktober 1995.

[10] Graham Hamilton and Panos Kougiouris. The Spring nucleus: A microkernel for objects. In *Proceedings of the USENIX Summer 1993 Technical Conference*, pages 147–160, Berkeley, CA, USA, June 1993. USENIX Association.

[11] Sun Microsystems, Mountain View, CA. *The Java Language Specification*, 1.0 beta edition, October 1995.

[12] H.-M. Windisch. Improving the efficiency of object invocations by dynamic object replication. In H. R. Arabnia, editor, *Proc. of PDPTA*, pages 115 – 131, November 1995.

[13] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Department of Computer Science, Yale University, New Haven, CT, October 1986.

[14] B. Buck and P. Keleher. Locality and performance of page- and object-based DSMs. In *Proc. of the First Merged Symp. IPPS/SPDP*, pages 687–693, Los Alamitos, March 30–April 3 1998. IEEE CS.

[15] W. G. Levelt, M. F. Kaashoek, H. E. Bal, and A. S. Tanenbaum. A comparison of two paradigms for distributed shared memory. *Software– Practice and Experience*, 22(11):985–1010, November 1992.

[16] G. Amdahl. The validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS conference proceedings, Spring Joint Computing Conference*, volume 30, pages 483–485, 1967.

[17] Markus Pizka. Design and implementation of the GNU INSEL-compiler gic. Technical Report TUM-I9713, Technische Universität München, Dept. of CS, 1997.

[18] SunSoft, Mountain View, CA. *Linker and Libraries Guide*, 1996.

[19] W. Wilson and R. A. Olsson. An approach to genuine dynamic linking. Technical report, Dept. of CS, University of Califonia, Davis, CA, Oktober 1990.

[20] Saumya K. Debray. The alto project homepage: Link-time code optimization. World Wide Web, April 1999. http://www.cs.arizona.edu/alto/.

[21] Douglas B. Orr, Robert W. Mecklenburg, Peter Hoogenboom, and Jay Lepreau. Dynamic program monitoring and transformation using the OMOS object server. Technical Report UUCS-92-034, University of Utah, January 2, 1998.

[22] Crispin Cowan, Tito Autrey, Charles Krasic, Carlton Pu, and Jonathan Walpole. Fast concurrent dynamic linking for an adaptive operating system. In *Proceedings of the International Conference on Configurable Distributed Systems*, Annapolis, 1996.

[23] M. Pizka, C. Eckert, and S. Groh. Evolving software tools for new distributed computing environments. In H. Arabnia, editor, *PDPTA'97*, pages 87–96, Las Vegas, NV, July 1997.

[24] Markus Pizka. *Integriertes Management erweiterbarer verteilter Systeme*. PhD thesis, Technische Universität München, June 1999.

[25] H. E. Bal and M. F. Kaashoek. Object distribution in orca using compile-time and run-time techniques. In *OOPSLA'93*, pages 162–177, September 1993.

[26] C. Amza, A. L. Cox, S. Dwarkadas, L-J. Jin, K. Raj amani, and W. Zwaenepoel. Adaptive protocols for software distributed shared memory. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):467–475, 1999.

[27] Markus Pizka. Sta – a conceptual model for system evolution. In *Intern. Conference on Software Maintenance*, pages 462 – 469, Montreal, Ca, October 2002. IEEE CS Press.

[28] Mike Accetta, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation For UNIX Development. Technical report, CS Department, Carnegie Mellon University, Pittsburgh, PA 15213, August 1986.

[29] Christian Rehn. Inkrementelles und dynamisches binden in einer verteilten umgebung. Master's thesis, Technische Universität München, Institut für Informatik, February 1998.