Evolving Software Tools for New Distributed Computing Environments [*]

Markus Pizka, Claudia Eckert, Sascha Groh
Munich University of Technology
Department of Computer Science
80290 Munich (Germany)

Abstract *In future, parallel and distributed computing paradigms will replace nowadays predominant sequential and centralized ones. Facing the challenge to support the construction of complex but high-quality distributed applications, appropriate software environments have to be provided. Experience has shown, that these new paradigms require support by the resource management system, comprising compiler, linker and operating system to name only a few of the tools involved. As the implementation of new and high-quality tools demands tremendous efforts, modification but basically reusage of existing tools as far as possible is desirable.*

This paper presents an approach to develop a resource management system supporting parallel and distributed computing. As an example of our implementation strategy we elaborate on the utilization and customization of the well-known GNU compiler gcc *to compile the distributed and parallel programming language used in the approach.*

Keywords: distributed systems, languages, tools, compiler, gcc

1 Introduction

Currently, a broad spectrum of research activities is focusing on the transition from sequential and centralized processing to distributed, parallel and cooperative computing. To support the construction of high-quality and efficient complex distributed systems, appropriate software environments have to be provided. These environments have to fulfill at least two somehow contradictory goals. On the one hand, they should significantly ease distributed programming by hiding as many details of the distributed nature of the hardware configuration as possible. On the other hand, they have to enhance performance by providing adaptability and scalability without introducing an distinct management overhead.

New resource management systems, comprising software such as compiler, linker and operating system (OS) kernels are required to meet these requirements. We argue that the implementation of theses tools does not have to start from scratch. Existing software can be modified to meet the demands of distributed computing.

Developing appropriate software tools, we are facing two major problems. First, their implementation demands tremendous efforts. Second, the quality of the tools determines the success of the system. This paper demonstrates, that by modifying but basically reusing existing tools both aspects can be addressed to develop high-quality tools with acceptable effort. As an example for our implementation strategy, we will explain modifications of the well-known GNU compiler *gcc* to match the requirements of a new distributed computing environment.

The rest of the paper is organized as follows. Section 2 will briefly present the key principles of our approach. The object-model and structuring concepts provided by the language IN-

[*]This project is sponsored by the DFG (German Research Council) as part of the SFB #342 – *Tools and methods for the utilization of parallel architectures.*

SEL are introduced in section 2.1 and 2.2. In 2.3 we present our general model of an adaptive and scalable distributed resource management system. The implementation of this model requires a systematic construction of the software tools needed, as sketched in 3. As an example for our strategy, we will describe the adaption of the GNU compiler *gcc* to our requirements in section 4.

2 The MoDiS Approach

In the project MoDiS (Model oriented Distributed Systems) [1, 2] we follow a top-down driven, language-based approach to provide the desired comfortable programming environment as well as to ensure efficient execution. A single homogeneous and transparent set of concepts is given to construct operating system functionality as well as user level applications. Objects representing new functionality (especially applications) dynamically extend the system, forming a system where applications and operating system functionality are systematically and dynamically integrated.

The properties of the abstract concepts elaborated in the MoDiS project define requirements at a high level of abstraction. It is the responsibility of the resource management system to map these requirements to available distributed hardware resources. This mapping is achieved by top-down oriented stepwise refinement of the abstract requirements. Some of the transition steps have to be performed at compile-time while others have to be executed at runtime.

To implement the MoDiS model of a distributed system, we designed the programming language INSEL, that offers language concepts that are well-adapted to our abstract model.

2.1 Programming Language INSEL

INSEL [3] provides language concepts to program distributed applications without any knowledge about details of the underlying distributed hardware configuration. It is a high-level, type-safe, imperative and object-based programming language, supporting explicit task parallelism.

INSEL objects support encapsulation and can dynamically be created during program execution as instances of class describing objects, called *generators*. To prevent dangling pointers, objects are automatically deleted according to a conceptually defined life-time [4]. In contrast to class concepts known, as for instance in C++ [5], generators are integrated into the system in the same way as other objects and can be nested within other generators or instances and vice versa.

Determined by the associated generator, INSEL objects may either be active, called *actors* or passive ones. An actor defines a separate flow of control and executes concurrently to its creator.

INSEL objects may communicate directly in a client-server style (message passing paradigm) as well as indirectly by accessing shared passive objects (shared memory paradigm). All requests to objects are served synchronously.

2.2 Structural Dependencies

Based on the specific properties of the language concepts, especially nesting of instances and generators, dependencies between the objects of an INSEL system are established. This kind of structuring information has several benefits. First, it is important to notice, that structural dependencies [4] reflect application-level properties and can therefore be exploited to enforce automated application-specific resource management. Second, they are implicitly determined by the programmer by employing our language concepts. He is not burdened with having to specify hints to the resource management system in addition to writing his application. And third, since most of these dependencies are based on class properties, they are easy to predetermine by software tools such as the compiler. Most important of these system structures is the termination dependency, which defines a partial order on the termination and deletion of objects. The lifetime of each INSEL-object depends conceptually on exactly one other object in a way that ensures that no object is deleted as long as it is accessible.

2.3 Resource Management Model

To enforce transparent, scalable and adaptable distributed resource management, we developed a model of a reflective management architecture [6, 7]. Based on the termination dependency, INSEL objects are clustered to actor-contexts (ACs) forming essential units of resource management. An AC comprises exactly one actor and all its termination dependent passive objects. With each AC, exactly one abstract manager is associated, which is responsible for performing AC-specific resource management, that is to fulfill all requirements of the actor-context. Besides fundamental tasks such as allocating memory for the stack, heap and code of the objects within the AC the manager might also have to provide facilities to maintain consistency of replicated objects, enforce access restrictions or perform load balancing. Conflicts, such as stack overflows, arising from different managers performing their tasks in parallel are solved by communication and cooperation between managers according to application-level structural dependencies between the ACs. This management scheme is scalable as it does not have a potential central bottleneck and is adaptable since resource management is performed based on the requirements of application-level objects. For instance, the resource management system implements actors in a non-uniform manner. There is no single mapping of actors to for example UNIX processes or threads with a fixed size stack portion.

3 Systematic Construction of Software Tools

Software tools are the means to implement the model presented in section 2.3, resulting in a scalable and adaptive resource management system. Furthermore, they have to perform the transition steps mentioned in section 2. The model of associated abstract managers has to be implemented by systematically incorporating management functionalities into the software tools involved. We do not employ a uniform manager implementation. An implemented manager might solely consist of stack managing code produced by the compiler or it may itself be a complex object comprising further activities. The functionality and granularity of the manager is tailored to the requirements of its AC. Figure 1 illustrates basically different alternatives to implement management facilities as well as it emphasizes the tight integration of all transition steps taken. Most important among these implementation alternatives are naturally the compiler and the OS kernel.

The goal of the resource management system is to improve the execution speed and to reduce the size of the target representation. Hence, the implementation strategy is to incorporate management functionalities into the compiler or the OS kernel instead of employing inlining techniques or runtime libraries.

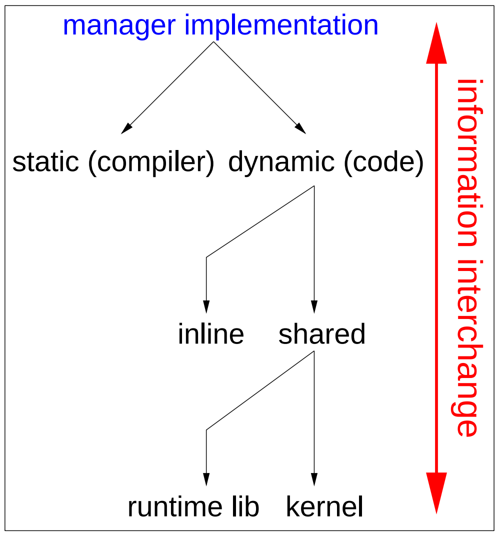

Figure 1: Implementation of AC Managers

A main issue of the MoDiS approach is to exploit information concerning overall system behavior as well as application-specific information gained from static and dynamic analysis to achieve an adaptive resource management. Information is systematically interchanged between all components involved in the management task.

It is important to notice that at this point of view, the *operating system* is the management of the computing system, comprising tools and kernel. Hence, according to the approach followed for high level distributed processing all management units have to modified if not redesigned. The management splits up into two dimensions. First the architecture of AC specific managers and second, their components. Cooperation among all management units needed to achieve holistic distributed management is provided by information interchange as described above and explained in [8].

4 Compiler

As a base for the implementation of the INSEL compiler we chose the freely available GNU compiler *gcc*. Besides of its well-known properties such as portability [9] or abilities to perform extensive optimizations its modularized structure allows to add new languages by implementing a language-specific front-end that translates source code into *gcc*'s *AST* (abstract syntax tree) representation or to the lower *RTL* (register transfer language) level [10]. Examples for this technique are the GNU Ada compiler *gnat* [11] and *g77*, the front-end for Fortran.

By extending and modifying *gcc* to our requirements, we are able to circumvent the effort to reinvent an optimizing code-generator and avoid the necessity to implement layers of adaption and work-arounds, that would introduce run-time overhead. The tight integration of special aspects of distributed computing into the code-generator also allows us to control interactions with existing optimization techniques. As a profitable side-effect, adapting *gcc* also enables us to use the GNU debugger *gdb* for source-level debugging of the distributed system and other tools such as the GNU profiler.

4.1 Front-End vs. Back-End

The GNU compiler *gcc* is roughly structured into *front-ends* and the generic *back-end*. A front-end has to be written for each supported language. It is responsible for syntactical and semantical analysis and calls functions of the generic back-end for code generation. The code of each front-end is statically linked with the code of the back-end forming a complete source to assembler compiler. The front-end also has to provide some standard functions called by the back-end for language specific processing, such as handling of addressable objects. The back-end was originally developed for the language C but extended with support for C++, Pascal and others. Hence, implementing a language based on the *gcc* back-end delivers even a higher degree of freedom than mapping new language concepts to for example C.

4.2 Structure of the Compilation Process

Figure 2 illustrates the internal structure of the GNU INSEL compiler *gic*. The INSEL front-end parses the input file using common parser generator techniques. Instead of managing a symbol table and performing all static analysis on the parse tree, we use the tree transformation and attribute evaluation tool *MAX* [12] to transform the parse tree into an attributed *MAX* tree (MT) representation. The MT representation is decorated with attributes that represent compile-time as well as run-time properties of the application. It serves as an intermediate representation to enforce dynamic recompilation as well as to implement dynamic extensibility of the running system. For example, if run-time monitoring indicates the necessity to dynamically replace management functionality implemented by the compiler, the compiler is automatically called. It rereads the MT representation and fetches

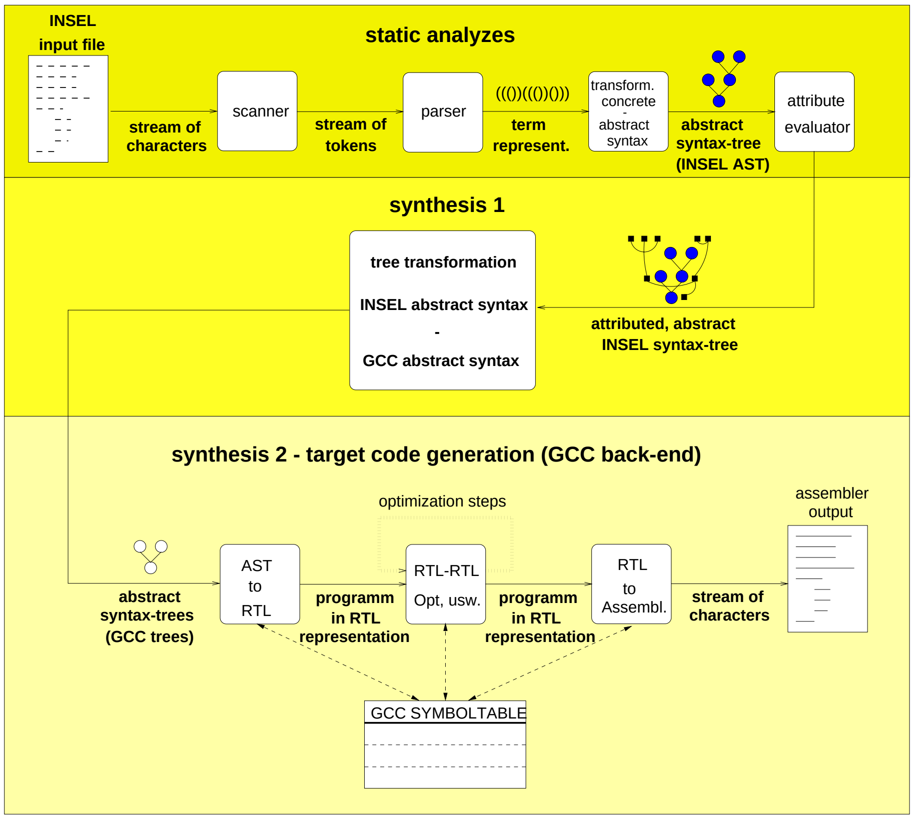
static analyzes
INSEL input file
scanner
parser
stream of characters
stream of tokens
((())((()())))
term represent.
transform. concrete - abstract syntax
abstract syntax-tree (INSEL AST)
attribute evaluator
synthesis 1
tree transformation
INSEL abstract syntax - GCC abstract syntax
attributed, abstract INSEL syntax-tree
synthesis 2 - target code generation (GCC back-end)
optimization steps
assembler output
abstract syntax-trees (GCC trees)
AST to RTL
programm in RTL representation
RTL-RTL Opt, usw.
programm in RTL representation
RTL to Assembl.
stream of characters
GCC SYMBOLTABLE

Figure 2: GIC Compilation Process

the current values of the run-time attributes, that in turn influence the synthesis step, leading to the desired new different target representation.

The final task of the INSEL front-end is to transform the MT into the GNU AST representation by traversing the MT and calling procedures of the generic back-end of *gcc*. The GNU back-end manages an own symbol table and performs several RTL to RTL transformations before producing assembler code.

4.3 Modifications to gcc's Back-End

The desired exploitation of *gcc*'s capabilities for a distributed computing environment requires modifications of *gcc*'s code generating facility, such as restricting some optimization techniques, introducing new ones and adapting memory management to meet the requirements of distributed and parallel computing. In the subsequent sections we will explain some of the interventions needed.

4.3.1 Access to non–local variables

gcc provides a simple but efficient way to access non–local data in languages that support nesting. Instead of employing displays [13] it simple uses a chain of static predecessors. On Sun Sparc, global register `%g2` is used to hold the address of the static predecessor. Assuming that levels of nesting are low and program execution is centralized, this scheme delivers sufficient performance and of course, was easy to implement.

Without support by the code-generator, nesting can be implemented by constructing compound objects consisting of non-locally accessibles and passing a pointer to this compounds to the callee. The callee then uses this pointer to access non-local data. Such an implementation is done for example in p2c (GNU Pascal to C translator). Comparing the performance of this implementation with *gcc*'s support for nesting we measured a 30% benefit for the *gcc* implementation, demonstrating that integration in the code-generator is superior.

In a distributed environment, a chain of static predecessors is unacceptable, since tracking the chain might necessitate communication over the network for each link. Hence, *gcc*'s code generator has to be enhanced by replacing chaining with the handling of displays.

4.3.2 Trampolining

For compatibility reasons, *gcc* implements pointers to nested sub-programs via a trampolining technique. If the address of a nested function g is taken within function f, a portion of code, that sets up information about static predecessors before branching to g is inserted in the stack frame of f and the address of the trampoline is used in place of the address of g. This technique allows to use existing libraries, such as pthreads [14] without modifications together with languages that support nesting.

Unfortunately, since trampoline code is statically produced by the compiler, this strategy hampers dynamic extensibility. Trampolines can not be dynamically placed on stack frames of existing functions at the time new functionality is to be integrated into the running system. To overcome this deficiency we replaced the trampolining mechanism with a customized addressing scheme for nested functions.

4.3.3 Parallel and Distributed Memory Management

Due to its well-known advantages concerning persistency and mobility of objects we employ a single virtual 64bit address space for our system. A major problem in such parallel computing environments with fine-grain parallelism is adequate management of multiple activities within the single non-segmented address space. First the management has to be performed decentralized to avoid bottlenecks and second, the stack size required for a parallel activity can not be statically predicted. A mechanism is needed that automatically handles stack growths, collisions and overflows.

For decentralized and adaptive virtual memory management the address space is partitioned into *regions* that are continuous ranges of pages and are assigned to Actor–Context (AC) specific Managers. Regions are units of cooperation between managers. If a new AC and manager is generated, the manager of the caller splits his own regions and provides an initial region to the callee for autonomous management. Only if the assigned regions are not sufficient, the manager cooperates according to application level dependencies with other managers to fetch or return regions. Within the assigned regions, each manager organizes four logical segments for stack, heap, code and free ranges of addresses of its AC. Each logical segment is realized by subdividing the assigned regions into real stack, heap, code and free segments, being continuous ranges of pages. Most interesting concerning compilation is the management of the logical stack segment.

In fact, hardware should provide advanced means to monitor stack evolution of multiple threads and the OS has to be prepared to expand and shrink stack sizes transparently. Since hardware support is not available, we have to integrate stack checks into the compiler. Whenever stack space is (de-)allocated, the frame-pointer has to be checked against upper and lower bounds of the current real stack segment. If these limits are exceeded, the runtime manager has to (de-)allocate real stack segments by splitting or merging the logical free segment. To avoid expensive reorganizations of the stack space, the newly allocated real stack segment does not have to be continuous with the existing ones, establishing a fragmented stack organization. According to the fragmentation of stack space the addressing scheme of the *gcc* back-end has to be changed. For example on a SUN Sparc arguments are addressed via a constant offset from the frame-pointer (`%fp`). We modified the addressing scheme to use local register `%l0` as an explicit argument pointer.

This example illustrates on the one hand the importance of adapting and seamlessly integrating all management units to the requirements of distributed and parallel computing and that on the other hand, efficient implementation is often possible by simple adaption of existing tools.

4.3.4 Optimization

A general problem using existing unmodified compilers for parallel or even distributed processing is, that some optimization steps are not applicable and hard or even impossible to turn off selectively. But compiling without any optimization usually leads to significant performance degradations of factor 3 to 5 that can hardly be compensated with distributed processing techniques. Distributed Performance measurements made on such a foundation must be considered biased. One such case is moving of loop invariant computations. For example implementing busy waiting via distributed shared memory (DSM) [15] might lead to endless loops if optimization is turned on, since the check of the condition is considered as being invariant. Declaring variables used for busy waiting as `volatile` in C solves this problem but is no general solution, neither for distributed programming nor for optimization in general. Developing the INSEL-compiler on top of the *gcc* back-end allows us to selectively activate and modify such optimization steps as well as adding new ones.

4.4 Extended Role of the Compiler

Besides generating code, the task of the compiler as part of the integrated management is to analyze the source code and the structural dependencies within the application. It has to analyze, prepare and forward information about the application to the runtime management. Information that is not collected by the compiler or not forwarded limits the success of runtime management. The collection of information out of the application source code is essential for the success of parallel and distributed computing with system-integrated management. Following a two examples of usage of this information.

4.4.1 Preparation for Object Distribution

A well known problem in distributed shared memories is false sharing [16]. A shared page includes several different objects. But caused by locality not all objects of a page are used by an activity. Untouched or unused objects are transported via the network without any need. This leads to a tremendous performance drawback in large systems with many activities spread over several workstations. The problem can be solved or to be more precise reduced with the help of the compiler. Data analysis during compile time can point out critical clustering of objects on pages. Afterwards the compiler is able to reorganize the object placement pattern.

Another important point is prefetching of needed objects. A usual DSM only fetches an object if it is accessed. The tight coupling of a DSM with the compiler allows to prefetch objects in advance transparently to the application-level. Not only the object currently needed is ordered by the DSM but also objects needed in the near future. Such an integration allows the distributed system to minimize data and page faults.

4.4.2 Optimizing Scheduling

The optimization of distributed scheduling is another important factor for the performance of the running system. The foundation for an optimal distribution is the cooperation analysis of the compiler. It analysis which activities cooperate via messages and shared datas. This information is evaluated by the load balancer to place cooperating activities on the same workstation.

5 Related Work

In approaches like Muse/Apertos [17] reflective management architectures are constructed similar to our manager approach. In this projects the reflective managers are objects for themselves. In contrast to this, our managers are abstract and implemented by tools, the system itself and the kernel. Other language-based approaches like ORCA [18] or Guide [19] also try to provide a homogeneous programming environment but often either lack the integration of activity into the languages hampering analyzes concerning parallelism or are not fully supported by a systematically developed resource management system. Often, for simplification compilers are constructed using C as a portable intermediate language, for example in Diamonds [20]. The compiler for Napier [21] goes steps beyond this translation scheme by exploiting extensions of GNU C, to for example place certain data in fixed hardware registers. Common to these approaches is, that overall management is not fully integrated, limiting either the success of static optimization or runtime management.

In the field of parallel programming, numerous new compilers with specific optimizations for parallel processing are developed, such as for the language psather [22]. We aim to incorporate these important experiences and techniques into our general approach to resource management. Data parallelism in addition to tasking parallelism will provide additional flexibility and increase performance.

The properties of single address space systems are investigated in projects like [23, 24, 25]. For time and space saving management of the single address space, we employ *guarded page tables* as introduced in [26].

6 Current State and Future Work

All implementations are based on a cluster of 14 SUN UltraSparc workstations connected via Fast Ethernet, running Solaris 2.5.1. Currently the INSEL-compiler *gic1* supports about 80% of the language concepts and includes simple semantic analysis as well as adaptions that allow for full optimization. At the same time, a new incremental linker and loader based on dynamic linking techniques known from *dynamic* and *shared* libraries is developed. The goal of the linker is to support dynamic extension of the running system and to be able to choose at runtime between different alternatives produced by the compiler. As described in [27] providing different implementation alternatives for classes of passive objects using compilation techniques, such as *replicable* or *migrable* implementations is capable of enhancing resource management. As soon as the compiler fully supports the language INSEL, we will add further alternatives for active objects, such as providing or omitting threads for abstract active objects. A major requirement for this extended flexibility is, that generatable alternatives are orthogonal. For example, the decision to implement an abstract actor with an own thread or not, must not influence the implementation of callers of the actor. This easy to state requirement must be fulfilled with systematic construction of all alternatives by avoiding the introduction of additional management overhead.

7 Conclusion

To support the development of complex and high-quality distributed programs, new programming environments are required. Within the MoDiS approach, we currently develop a new resource management system integrating compiler, linker and OS functionalities. The complete set of software tools involved in the transformation of a parallel program into an efficient distributed executable, is tailored to the requirements of distributed computing. The alternative to construct layers above existing unmodified tools and techniques introduces additional overhead and even conflicts, limiting the potential efficiency of distributed processing.

In this paper we demonstrated the benefits of using existing tools and techniques to reduce development overhead for software tools considerably. Existing and successful techniques integrated and implemented in available tools can and should be reused with modifications, reducing the development effort.

We presented the INSEL compiler *gic* based on the GNU *gcc* as an example to demonstrate our general implementation strategy. The tight integration of the modified *gcc* into our resource management system eliminated the expensive need to implement a new codegenerator. Furthermore, it gives us full flexibility for the source to target transformation and the opportunity to make use of a large collection of advanced compilation techniques, especially optimization steps.

References

[1] C. Eckert and H.-M. Windisch. A Top-down Driven, Object-based Approach to Application-specific Operating System Design. In *Proceedings of the IEEE International Workshop on Object-orientation in Operating Systems*, pages 153–156, Lund, Sweden, August 1995.

[2] C. Eckert and H.-M. Windisch. A new approach to match operating systems to application needs. In *Proceedings of the 7th IASTED - ISMM International Conference on Parallel and Distributed Computing and Systems*, Washington, DC, October 1995.

[3] H.-M. Windisch. The Distributed Programming Language INSEL - Concepts and Implementation. In *High-Level Programming Models and Supportive Environments HIPS'96*, 1996.

[4] M. Pizka and C. Eckert. A language-based approach to construct structured and efficient object-based distributed systems. In *Proc. of the 30th Hawaii Int. Conf. on System Sciences*, volume 1, pages 130–139, Maui, Hawai, January 1997. IEEE CS Press.

[5] Bjarne Stroustrup. *The C++ Programming Language.* Addison–Wesley, Reading, MA, 2nd edition, 1991.

[6] Sascha Groh. Designing an efficient resource management for parallel distributed systems by the use of a graph replacement system. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96)*, pages 215–225, August 1996.

[7] Sascha Groh and Markus Pizka. A different approach to resource management for distributed systems. In *Proc. of PDPTA'97 – International Conference on Parallel and Distributed Processing Techniques and Applications*, July 1997.

[8] Sascha Groh and Jürgen Rudolph. On the efficient distribution of a flexible resource management. In *Proc. of EuroPDS'97*, June 1997.

[9] Richard M. Stallman. *Using and Porting GNU CC.* Free Software Foundation, November 1995.

[10] Richard Kenner. Targetting and retargetting the GNU C compiler. slides, November 1995.

[11] C. Comar, F. Gasperoni, and E. Schonberg. The gnat project: A GNU-Ada9X compiler. Technical report, Courant Insitute of Mathematical Science, N.Y. University.

[12] A. Poetzsch-Heffter. Programming language specification and prototyping using the MAX system. Cornell University, NY.

[13] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilerbau (Teil 2).* Addison–Wesley Verlag (Deutschland) GmbH, 1988.

[14] OSF. *Introduction to OSF DCE.* Prentice Hall, Englewood Cliffs, NJ, 1992.

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

[16] W. J. Bolosky and Michael L. Scott. False sharing and its effect on shared memory performance. *Proc., Fourth Symp. on Experiences with Distributed and Multiprocessor Systems (SEDMS)*, September 1993.

[17] Y. Yokote, A. Mitsuzawa, N. Fujinami, and M. Tokoro. Reflective Object Management in the Muse Operating System. Technical Report SCSL-TR-91-009, Sony Computer Science Laboratory Inc., 3-14-13 Higashi-gotanda, Shinagawa-ku, Tokyo, 141 JAPAN, September 13 1991.

[18] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, 1992.

[19] M. Riveill. An overview of the Guide language. In *Second workshop on Objects in Large Distributed Applications*, Vancouver, 18.10. 1992.

[20] U. Bellur, G. Craig, K. Shank, and D. Lea. DIAMONDS: Principles and Philosophy. Technical Report CASE Center 9313, SONY Oswego, Dept. of Computer Science, June 1993.

[21] R. Morrison, M. P. Atkinson, and A. Dearle. Flexible incremental bindings in a persistent object store. Technical report, University of St. Andrews, St. Andrews, Scotland, June 1987.

[22] Stephan Murer, Jerome A. Feldman, Chu-Cheow Lim, and Martina-Maria Seidel. psather: Layered extensions to an object-oriented language for efficient parallel computation. Technical Report TR-93-028, International Computer Science Institute, Berkeley, CA, June 1993 November 1993.

[23] K. Murray, T. Wilkinson, P. Osmon, A. Saulsbury, T. Stiemerling, and P. Kelly. Design and implementation of an object-orientated 64-bit single address space microkernel. In *Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures*, pages 31–43, 1993.

[24] Jeff Chase, Hank Levy, Miche Baker-Harvey, and Ed Lazowska. Opal: A single address space system for 64-bit architectures. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 80–85, 1993.

[25] Kevin Elphinstone. Address space management issues in the Mungi operating system. Technical Report SCS&E Report 9312, University of New South Wales, Australia, November 1993.

[26] Jochen Liedtke. Page Table Structures for Fine-Grain Virtual Memory. Tech. Rep. 872, German National Research Center for Computer Science (GMD), October 1994.

[27] H.-M. Windisch. Improving the efficiency of object invocations by dynamic object replication. In *Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications — PDPTA*, pages 680–688, November 1995.