Develop Once Deploy Anywhere

Achieving Adaptivity with a Runtime Linker/Loader Framework

Joy Mukherjee
660 McBryde Hall,
Dept. of Computer Science
Virginia Tech,
Blacksburg VA 24061
1- (540)-231-9431
jmukherj@vt.edu

Srinidhi Varadarajan
660 McBryde Hall,
Dept. of Computer Science
Virginia Tech
Blacksburg VA 24061
1- (540)-231-9431
srinidhi@vt.edu

ABSTRACT

This paper presents Load and Let Link – a framework for flexible runtime loading and linking of procedural native code components. LLL has several novel aspects. First, it provides componentization without requiring an object-oriented language. Second, LLL performs linking at runtime, providing arbitrary code expansion, contraction and substitution. This enables (a) adaptive applications that can rewire themselves in response to dynamic conditions, (b) code patching for mission critical systems and (c) automatic overlaying in memory constrained environments. LLL is language neutral and orthogonal to current software development methodologies, thus providing the substrate for the next generation of develop once deploy anywhere software. In this paper, we present the LLL framework, its implementation on 32 bit x86 architectures and two case studies that showcase the capabilities of the framework.

Categories and Subject Descriptors

D.4.9 [**Operating Systems**]: Systems Programs and Utilities; D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming.

General Terms

Performance, Design, Languages.

Keywords

Composition, runtime linking and loading, adaptive applications, software component architectures.

1. INTRODUCTION

With continuing trends in processor miniaturization, computing power is becoming more commonplace. Embedded processors in handhelds, cellular telephony, PDAs and consumer appliances account for the majority of the processor market. These domains have several characteristics. First, they are low power devices with relatively small memory footprints. In contrast, application software prevalent in these domains – user interface libraries, web browsers, email clients etc. – continue to increase in complexity. Second, the domain of these devices necessitates interactions with real-world dynamic inputs that may not be completely characterized a priori. This requires support for adaptive applications that can rewire themselves in response to changing conditions. Effectively, this flexibility entails one or more of extending, contracting, replacing, and restructuring the code-base [1, 3, 9, 14, 17, 27, 31]. This is reflected in the current trends of componentization and composition illustrated by object [29] and aspect-oriented [5] programming techniques and reflective and interface oriented designs [14, 39]. Finally, industry trends in software engineering [21, 31, 35], in particular verification/validation and quality assurance, demand maximal commonality in code bases, thus requiring applications to run in a wide variety of environments and resource constraints.

We argue that handling these requirements needs new compile and runtime support tools. These tools should be orthogonal to software development methodologies, enabling existing applications and middleware to easily use their functionality without significant changes to programming models.

This paper presents *Load and Let Link* (LLL) – a framework for agile and flexible runtime loading and composition of native code components – to enable adaptability for legacy procedural programs. LLL provides simple object-oriented (OO) componentization mechanisms without requiring an OO language. It treats encapsulated runtime images of native object files as modules and offers OO services such as inheritance and overloading. The enabling technology of LLL lies in its method for program composition. Just like object files are linked at compile-time to generate a program executable, LLL links modules at *runtime* to obtain a program image. In doing so, LLL maintains the reified structure inherent in an executable's constituent object files even after runtime composition. LLL is more lenient towards undefined references, which is conducive to systems involving arbitrary code-base expansion and contraction [30]. It also provides a default flow of control that may be used to monitor and asynchronously modify the application constitution at runtime. Whereas reflection, interface orientation, and adaptive programming techniques help architect application adaptability, an orthogonal low-level tool such as LLL complements them by providing the necessary runtime support.

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, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
RM '05, November 28 – December 2, 2005, Grenoble, France.
Copyright 2005 ACM 1-59593-270-4/05/11…$5.00.

Extra facilities for runtime flexibility offered by a binary loading and linking framework are afforded to all high-level languages, frameworks, and models. LLL supports (a) programs seeking to load a new component [14], (b) applications trying to reduce memory usage [30], and (c) interface oriented systems trying to switch between implementations [39]. In particular, traditional procedural programs stand to benefit the most. Current non-OO software development processes support a fair degree of modularity and interface standardization. For instance, procedural programs written in languages such as C and FORTRAN are compartmentalized into separate source files (`.c` and `.f` respectively) and are compiled into corresponding object files (`.o`). Once the binding interfaces (external references between these objects) are fixed, the internal implementations may be changed. The problem is that this is restricted to the pre-runtime domain. To create an executable, the objects are integrated into one file thus obfuscating the reified structure of the application at runtime (see Figure 1).

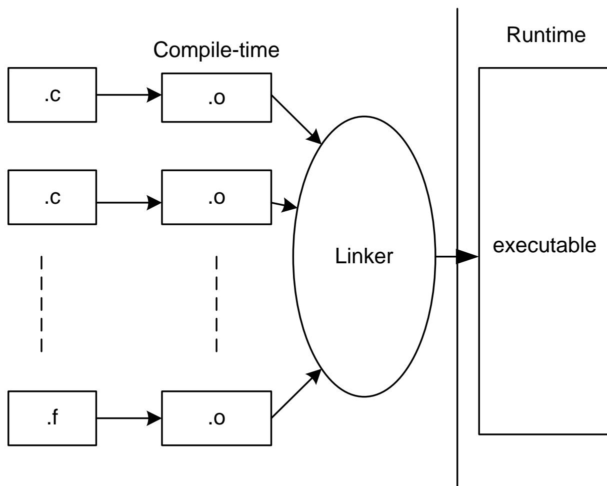

Figure 1. Traditional compile-time linking obfuscates the reified structure of an application.

In contrast, a framework that helps load constituent objects files into the runtime environment without affecting their distinct compartmentalization enables applications to exploit the pre-runtime native code based reification. LLL provides this functionality, which can be used by traditional procedural programs to achieve runtime decoupling, composition, and adaptation. Some aspects of LLL may be more productive when coupled with strategies for dynamic state and stack manipulation [11, 13].

The rest of the paper is organized as follows. Section 2 presents related work in this area. Section 3 describes the LLL framework for runtime composition. Section 4 presents the runtime elements of LLL that enable dynamic loading and linking. Section 5 briefly describes the LLL's prototype implementation. Section 6 demonstrates LLL's adaptive abilities as well as its handling of legacy codes through two case studies. Finally, Section 7 summarizes the paper and discusses ongoing work.

2. RELATED WORK

Substantial research on adaptive software has been directed at the level of frameworks [7], high-level techniques [8, 15], and programming methodologies [28]. These systems assume language in their use of object-oriented programming. This poses problems towards applying them to legacy procedural systems. There have also been attempts at decoupling legacy procedural codes at runtime by wrapping [16] and reference indirection (often through message passing schemes) [40]. These schemes, however, increase overheads while LLL maintains both the simplicity and efficiency of direct reference-definition bindings. Moreover, such schemes may still be used in conjunction with LLL where the architecture neutrality of message passing is desired. Runtime code patching [2] and dynamic code generation [23, 27] have also been used in projects to leverage adaptability. Besides the complexity inherent in their implementations, they do not directly address componentization of a program's binary at runtime.

Some existing approaches for on-the-fly software manipulation make use of runtime loading and linking of native code [12, 20, 22, 32, 34]. They illustrate mechanisms for dynamic updating (runtime replacement of components) and dynamic interposition or extension of default code bases. LLL deals with more comprehensive extents of adaptation including contraction and restructuring in addition to component replacement and dynamic extension. Furthermore, none of the referenced works address runtime modularity concerns from the viewpoint of traditional procedural programs. OMOS [19] is a powerful mechanism that uses a flexible native code based object-oriented framework. Several of LLL's operations are modeled after concepts illustrated by OMOS. However, OMOS does not make any attempt to apply these operations to achieve runtime modularity, adaptability, and dynamic composition in unmodified applications.

3. FRAMEWORK DESIGN

Like any other framework that targets adaptive programs, LLL requires that applications be composed from separate components providing specific services. However, LLL does not subscribe to any specific language-level constructs for componentization. Instead, it defines a *simple module* – the basic unit of encapsulation – as the runtime image of an object file compiled from source written in *any* language. Every module has a *Module Context Table* (**MCT**) that maps data and function references in the module's code to corresponding definitions. LLL offers runtime control over addition and modification of individual MCT entries, which is instrumental in letting applications dynamically add, prune, or modify their functionality, content, and structure. A *composite module* – the unit of inheritance – is defined as a base module (simple/composite) and an additional simple module bound together under a single ordinal identifier. It inherits all code and state from the base module, while the new module extends and/or overloads the inherited functionality. The composite module is, therefore, an ordered set of modules where the higher ordinals overload lower ordinals. This minimal definition of fundamental components allows LLL to flexibly work with most high-level frameworks and models for reification and reflection [4, 21, 33].

The core capability of LLL lies in its method for program composition. Just as object files are linked at compile-time to generate a program executable, LLL links modules at *runtime* to obtain a program image. There are, however, several important differences. While compile-time linking results in blurring of boundaries between different objects, LLL maintains the reified structure even after composition (see Figure 2), which aids

component replacement. While the former rigorously requires resolution of all references for successful completion, the later is more lenient towards undefined or, more aptly, *dangling* references, which is conducive to systems involving arbitrary code-base expansion and contraction. Applications may specify their own runtime handlers for such references to tune dynamic composition to their needs. This is similar to late-binding mechanisms [20] except that handlers may be explicitly specified by the application, may be different for different references, or may default to `NULL`.

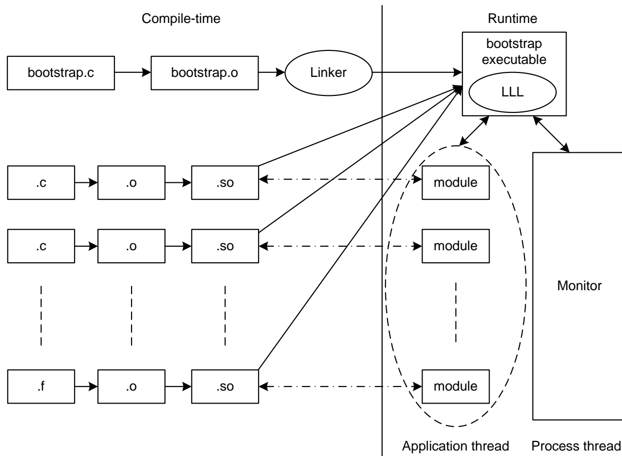

Figure 2. The LLL framework uses dynamic loading and linking to preserve the compile-time reified structure of an application at runtime. This enables on the fly contraction, expansion and substitution of the code-base.

Since LLL performs application composition at runtime, it provides a minimal bootstrapping module, which is the main process thread of every application. This bootstrapping module is primarily responsible for setting up the main application and loading/composing its constituent modules. Once composed, the target application runs on a lightweight intra-process thread spawned by the bootstrap module. LLL permits applications to customize the main process's functionality after it has started the application thread. Hence, the main process thread may later be used to monitor and asynchronously modify the application constitution at runtime. This architecture, coupled with the adaptability innate in the module definitions, aids extensive dynamic rewiring including, potentially, replacement of core components and changes to structural componentization.

4. LOAD LINK TOOL

Traditional binary loaders are typically monolithic systems-level libraries. The interfaces they expose and functionality they offer are not enough to support the flexibility demands of a framework such as LLL. For instance, they do not allow unconditional loading of incomplete modules with dangling references. (Contemporary late binding mechanisms for binaries tolerate dangling functional references only as long as they are not accessed. But when such accesses do happen in absence of compatible definitions, they result in anomalous behavior. Dangling references to data elements are not allowed). In order to maintain the modular structure at runtime, LLL requires that objects be loaded separately before application composition. This can result in some modules being loaded with several dangling references not just to functions but also to data elements. Again, typical loaders do not provide an explicit interface to connect a reference to an arbitrary definition. Hence, LLL provides its own tool for dynamic loading and linking of modules that offers extra control through simple interfaces (API). This section presents a brief description of the LLL loader and linker. For a more detailed discussion, we refer the reader to [18].

The LLL loader maps given object files on disk to corresponding simple modules in memory. Additionally, it also resolves all references that are defined within the simple module itself. The point to note here is that unlike traditional loaders, LLL does not try to resolve any external references at load-time. This is particularly important for dangling references, since any attempt to resolve them at this time would result in avoidable overhead. All cross-binding actions involving multiple modules are delegated to the LLL linker, which dynamically composes a program image given an ordered set of modules. The linker carries out typical binding actions trying to resolve the external references in each module to the rest in the set, subject to a few rules:

(a) Constituents of a composite module are first resolved within themselves (inheritance).

(b) While resolving *to* a composite module, the constituents are dealt with in order of descending ordinal number (overloading).

(c) References with multiple potential definitions are resolved to the first occurrence in the ordered set and notifications are generated for every other definition. An application may trap these notifications to select among alternatives via individual MCT entry redirections.

(d) Undefined references are reported for application-specific handling, or are directed to a default `NULL` handler.

In summary, the LLL linker builds a minimal functional program from the provided set of modules, while retaining the ability for future modification. Note that the individuality of the modules and the reified structure of the application are maintained even after object composition.

5. IMPLEMENTATION

LLL's current prototype implementation works on 32-bit x86 architectures running GNU/Linux. Being an object-level framework, it relies heavily on the Executable and Linkable File Format (**ELF**) [36] used by most UNIX systems. It recognizes shared object (`.so`) files as loadable modules, and the global offset table therein (`.got` section) as the MCT. Shared objects can be easily created from most relocatable objects (`.o`) compiled with position independent options (`-fPIC` for `gcc`). The rest of the implementation follows from the basic design and is extracted from GNU `ld` version 2.2.5 [10]. It fixes cross references between shared objects in the same manner as in the regular use of dynamic libraries. Using shared objects with position independent code can result in some extra instructions vis-à-vis normal compilation. However, the associated performance and memory overheads are as low as in regular dynamic libraries. LLL can be customized to use either POSIX threads (`pthreads`) [26] or GNU's user-level threads package, Pth [25]. Lastly, the implementation is fairly portable since both ELF and POSIX are

widely used standards. A port to the x86_64 architecture is currently underway. The philosophy behind LLL is generic to all systems and native code formats that support dynamic libraries. However, the complexity of a binary loader implementation makes a port to radically different operating systems such as Windows a time taking task.

6. CASE STUDIES

To demonstrate LLL's capabilities, we present two case studies. The first, and simpler, study illustrates dynamic module replacement, where a program component is asynchronously replaced by a better implementation. In this example, an application uses a component to sort integers. The program design is such that the sort function is called through a standardized interface –

```
void sort (int *array, int size);
```

This study experiment tries to show that if a better sort implementation is available later, LLL can help the application asynchronously switch to it. The application comprises a main routine that continually generates a random set of numbers and then calls a routine to sort the set. The sources for the `main` and the `sort` functions are programmed in `C` in two files – `main.c` and `sort.c` respectively – with an external reference (`sort`) from the former to the latter such that an executable `sorter` could be generated as –

```
gcc –o sorter main.c sort.c
```

For LLL modules, we compile the unmodified source files into shared objects `main.so` and `sort.so`. We load and link the modules in the bootstrapping segment of our process thread, and start a lightweight thread at the `main` function in `main.so`. The main process thread then waits for user commands to perform requested modifications. Our default sort routine is `bubblesort`. Midway through execution, without stopping the application thread, we ask the process thread to load an implementation of `mergesort` as another module (`mergesort.so`) and dynamically redirect the reference to `sort` in `main.so` to the definition `mergesort` in the new module. Once the last invocation of the default sort routine has returned, the module may be unloaded to save memory. We then retrace these steps with an implementation of `quicksort`. The output of the program shows a seamless handover from `bubblesort` to `mergesort` and then to `quicksort`. Details of the experiment including the programs and output may be accessed at –

http://blandings.cs.vt.edu/~joy/LLL/sort

The second case study showcases LLL's capabilities in leveraging legacy codes by (a) dynamically expanding or contracting the code-base and (b) dynamically changing the component schema of an application. Our target application is a minimal web-browser called Dillo [6] available for free under the GNU Public License. We choose Dillo for 3 reasons. (1) Its source-code is freely available. (2) Its lightweight and memory efficient minimalism is apt for mobile and limited-resource environments. (3) Its source is in traditional procedural C. Even though programmed in C, the code for Dillo is fairly modularized with different components being implemented through different source and header files that are compiled into distinct object files. This provides a minimal degree of reification. Maintaining best possible granularity without source modification requires us to compile each source file into one LLL module. However, for simplicity, we make an arbitrary cut and split the application into 4 modules (`m1.so`, `m2.so`, `m3.so`, `m4.so`) while maintaining, as far as possible, the linking sequence as decreed by the application's default building mechanism. Of these `m2.so` is of main interest. It comprises implementations for text find and link select functionalities (`findtext.o` and `selection.o`). All object files that precede the ones in `m2.so` in the default build for the application are bound into `m1.so` while all that follow are clubbed into `m3.so`. `m4.so` contains implementations for input/output, which are kept separate from the main browser objects as per the original implementation.

As before, we load and link the modules in the bootstrapping segment of our process thread, and start a lightweight thread at the `main` function of Dillo (now in `m3.so`). The process thread then waits for user commands to perform requested modifications. Suppose that at some point after browser invocation, the system runs short on memory. We simply ask the main process thread (monitor) to asynchronously unload `m2.so` thus reducing memory utilization at the expense of some functionality – find and select. The modular structure of the rest of the application is maintained along with all their existing state. The dangling functional references resulting from the unloading of `m2.so` are redirected to a single minimal function that returns `0` (or `NULL`). Dangling data references are directed to a `NULL` value. When the memory situation improves, `m2.so`, or a better implementation thereof, may be reloaded and recomposed with the remaining modules to reinstate full functionality. To change the componentization of the running application, we split find and select into separate objects - `findtext.so` and `selection.so` respectively (originally together in `m2.so`). We then dynamically load the two new modules and compose them with `m1.so`, `m3.so` and `m4.so` while unloading `m2.so`. The browser may now be contracted, expanded, or upgraded at a finer granularity i.e. one may unload and upgrade find while keeping select or vice-versa.

Our last case shows another aspect of automatic contraction enabled by LLL. It involves a scenario where the application is compiled and deployed on different, but architecture compatible (same instruction set), machines. We build the browser via its default scheme, which gives us the standard executable called `dillo`. We then build LLL modules from the same object files along with our bootstrapping code to load and link them at runtime. Both the original and the LLL versions of the browser run identically with identical facilities and system dependencies. We now move both the versions (executables and modules only) to a different, but compatible, machine that lacks a utility library for portable network graphics, `libpng [24]`. The original executable fails to run due to non availability of a dependency library. The LLL version, however, promptly starts up with all facilities except the graphics. It contracts the application automatically according to the available resources. Note that the LLL modules were also compiled on the former machine and contain dangling references to `libpng`, but being redirected to the minimal `NULL` returning function, they do not adversely affect the rest of the application. Details of the experiment including the programs and instructions may be accessed at –

http://blandings.cs.vt.edu/~joy/LLL/browser

These case studies, though simple, present the capabilities of LLL, which may be deployed in more complex application architectures. We believe that with better reification schemes, and advanced stack and state mapping techniques [11, 13], LLL can instrument reconfigurations at arbitrary granularities modifying not just peripheral modules but also those at the core of an application. State of the art high-level reflective designs, adaptation-aware programming methodologies and interface oriented codes can exploit LLLs compile and runtime support to create a new breed of *develop once deploy anywhere* applications.

7. ONGOING WORK

This paper presented *Load and Let Link* – a framework for agile and flexible runtime loading and composition of native code components – to enable adaptability for legacy procedural programs. It also presented two case studies that showcase the use of LLL to achieve (a) adaptivity and (b) automatic overlaying to support memory-constrained systems. LLL can thus provide the runtime substrate for reflection, interface orientation, and adaptive programming techniques to enable the next generation of adaptive applications.

Ongoing work on LLL focuses on two major areas. The first area uses the LLL runtime to create an adaptive problem solving environment (**PSE**) for high performance numerical analysis. The PSE consists of a toolbox of numerical solvers, an application that uses the solvers and a recommender system with domain knowledge. The application accesses the solvers through a procedural interface. The recommender acts as a feedback control system by monitoring the performance of the solver and using the LLL runtime to substitute solvers for either performance and/or correctness (optimistic solvers may produce incorrect results). Since solvers generally do not have the same type signature, the recommender system uses domain knowledge to perform the necessary data conversions or parameter derivation. The second area uses LLL to perform algorithm selection in multiple-resource constrained environments. Here, an application may have memory, performance and power constraints described by a time-varying profile. The LLL runtime can be used to perform appropriate algorithm selection – say for power or performance – to achieve overall performance goals.

The second area targets a caching scheme to improve the performance of automatic overlaying in the LLL runtime. In a memory-constrained environment, LLL can be used to contract code by eliminating modules and automatically reloading them on demand. However, it may result in thrashing, when code that was eliminated to conserve memory is needed immediately. This can be solved by the use of a “victim” cache, with either LRU or LFU replacement policy. At equilibrium, using the cache to deduce the “core” set of necessary modules can minimize thrashing. Finally, we are working on providing facilities to aid safe replacement of components and automatic garbage collection of unused modules.

8. REFERENCES

[1] Aksit, M. and Choukair, Z. Dynamic, adaptive and reconfigurable systems overview and prospective vision. In *Proceedings of the 23rd international conference on Distributed computing systems workshops (ICDCSW ’03)* (Providence, Rhode Island, USA, May, 2003).

[2] Appavoo, J., Hui, K., Soules, C. A. N., Wisniewski, R. W., Silva, D. M. D., Krieger, O., Auslander, D. J. E. M. A., Gamsa, B., Ganger, G. R., McKenny, P., Ostrowski, M., Rosenburg, B., Stumm, M. and Xenidis, J. Enabling autonomic behavior in systems software with hot-swapping. *IBM Systems Journal, 42, 1* (2003).

[3] Capra, L., Blair, G. S., Mascolo, C., Emmerich, W. and Grace, P. Exploiting reflection in mobile computing middleware. *ACM SIGMOBILE Mobile Computing and Communications Review, 6, 4* (October, 2002), 34-44.

[4] Carpa, L., Emmerich, W. and Masolo, C. Middleware for mobile computing: awareness vs. transparency. In *Proceedings of the 8th workshop on Hot topics in operating systems (HOTOS ’01)* (Schloss Elmau, Germany, May, 2001).

[5] Chen, Y. *Aspect-Oriented Programming (AOP): Dynamic Weaving for C++*. Master thesis, Vrije Universiteit Brussel and Ecole des Mines de Nantes, 2003.

[6] Dillo: http://www.dillo.org/.

[7] Duran-Limon, H. and Blair, G. S. A resource management framework for adaptive middleware. In *Proceedings of the 3rd IEEE international symposium on Object-oriented real-time distributed computing (ISORC ’00)* (Newport Beach, California, USA, March, 2000).

[8] Flinn, J. and Satyanarayanan, M. Energy-aware adaptation for mobile applications. In *Proceedings of the 17th symposium on Operating systems principles (SOSP ’99)* (Charleston, South Carolina, USA, 1999).

[9] Gilani, W., Naqvi, N. H. and Spinczyk, O. On adaptable middleware product lines. In *Proceedings of the 3rd workshop on Adaptive and reflective middleware (ARM ’04)* (Toronto, Canada, October, 2004).

[10] GNU libc: http://www.gnu.org/software/libc/libc.html

[11] Heffner, A. M. *A Runtime Framework for Adaptive Compositional Modeling.* Masters Thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA, USA, 2004.

[12] Hicks, M., Moore, J. T. and Nettles, S. Dynamic software updating. In *Proceedings of the ACM SIGPLAN conference on Programming languages design and implementation (PLDI ’01)* (Snowbird, Utah, USA, May, 2001).

[13] Huang, G., Mei, H. and Wang, Q. Towards software architecture at runtime. *ACM SIGSOFT Software Engineering Notes, 28, 2* (March 2003) 8.

[14] Kon, F., Costa, F., Blair, G. and Campbell, R. H. The case for reflective middleware. *Communications of the ACM, 45, 6* (June 2002) 33-38.

[15] Liang, S. and Bracha, G.Dynamic class loading in the Java virtual machine. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages and applications (OOPSLA ’98)* (Vancouver, Canada, October, 1998).

[16] Mahmood, N., Deng, G. and Browne, J.C. Compositional development of parallel programs. In *Proceedings of the 16th workshop on Languages and compilers for parallel*

computing (LCPC '03) (College Station, TX, USA, October, 2003).

[17] McKinley, P. K., Sadjadi, S. M., Kasten, E. P. and Cheng, B. H. C. Composing adaptive software. *IEEE Computer, 37, 7* (2004) 56-64.

[18] Mukherjee, J. and Varadarajan, S. Weaves: a framework for reconfigurable programming. *International Journal forParallel Programming, 33, 2* (June 2005) 279-305.

[19] Orr, D. B. and Mecklenburg, R. W. OMOS – an object server for program execution. In *Proceedings of the 2nd international workshop on Object orientation in operating systems (OOOS '92)* (Paris, France, September, 1992).

[20] Orr, D.B. Lepreau, J., Bonn, J. and Mecklenburg, R. Fast and flexible shared libraries, In *Proceedings of the Summer USENIX conference* (1993).

[21] Pierre-Charles, D. and Ledoux, T. Towards a framework for self-adaptive component-based applications. In *Proceedings of the 4th IFIP international conference on Distributed applications and interoperable systems (DAIS '03)* (Paris, France, November, 2003).

[22] Plugins: http://www.plugins.com/

[23] Poletto, M., Hsieh, W. C., Engler, D. R. and Kaashoek, M.F. C and tcc: a language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems, 21, 2* (March 1999) 324-369.

[24] PNG: http://www.libpng.org/pub/png/

[25] Pth: http://www.gnu.org/software/pth/

[26] Pthreads: http://www.llnl.gov/computing/tutorials/pthreads/

[27] Raatikainen, K., Christensen, H. B. and Nakajima, T. Application requirements for middleware for mobile and pervasive systems. *ACM SIGMOBILE Mobile Computing and Communications Review, 6, 4* (October, 2002), 16-24.

[28] Redmond, B. and Cahill, V. Supporting unanticipated dynamic adaptation of application behavior. In *Proceedings of the 16th European conference on Object-oriented programming (ECOOP '02)* (Malaga, Spain, June, 2002).

[29] Rentsch, T. Object oriented programming. *ACM SIGPLAN Notices, 17, 9* (September 1982), 51-57.

[30] Rigole, P., Berbers, Y., Holvoet, T. and Leuven, K. U. Mobile adaptive tasks guided by resource contracts. In *Proceedingsof the 2nd workshop on Middleware for pervasive and ad-hoc computing (MPAC '04)* (Toronto, Canada, October, 2004).

[31] Schmidt, D. C. Middleware for real-time and embedded systems. *Communications of the ACM, 45, 6* (June 2002) 43-48.

[32] Serra, A., Navarro, N., and Cortes, T. DITOOLS: Application level support for dynamic extension and flexible composition. In *Proceedings of the 2000 USENIX Annual technical conference (USENIX '00)* (San Diego, California, USA, June, 2000).

[33] Smith, B. C. *Reflection and semantics in a procedural programming language.* Ph.D. Thesis, Massachusetts Institute of Technology, Boston, MA, 1982.

[34] Stoyle, G., Hicks, M., Bierman, G., Sewell, P. and Neamtiu, J. Mutatis mutandis: safe and predictable dynamic software updating. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '05)* (Long Beach, California, USA, January, 2005).

[35] Szyperski, C. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

[36] Tools Interface Standards Committee, *Executable and Linkable Format (ELF) Specification.* May 1995.

[37] Unnikrishnan, P., Chen, G., Kandemir, M. and Mudgett, D. R. Dynamic compilation for energy adaptation. In *Proceedings of the international conference on Computer aided design(ICCAD '02)* (San Jose, California, USA, 2002).

[38] Varney, L. R. *Interface-Oriented Programming.* Technical Report TR-040016, Department of Computer Science, University of California, Los Angeles, CA, 2004.

[39] Varney, L. R. and Parker, D. S. Interface-oriented middleware and distributed service inference. In *Proceedings of the 3rd workshop on Adaptive and reflective middleware (ARM '04)* (Toronto, Canada, October, 2004).

[40] Yang, Z., Zhou, Z., Cheng, B. H. C. and McKinley, P. K. Enabling collaborative adaptation across legacy components. In *Proceedings of the 3rd workshop on Adaptive and reflective middleware (ARM '04)* (Toronto, Canada, October, 2004).