APRES'08

Adaptive and Reconfigurable Embedded Systems

First International Workshop on Adaptive and Reconfigurable Embedded Systems

St. Louis, MO, USA
April 21st, 2008

Editors:
Luís Almeida
Sebastian Fischmeister
Insup Lee
Julián Proenza

Sponsored by:

artist
European Network of Excellence on Embedded Systems Design

First International Workshop on Adaptive and Reconfigurable Embedded Systems

(APRES'08)

St. Louis, MO, USA, April 21st, 2008

Organizers:

Luís Almeida, Univ. of Aveiro, Portugal
Sebastian Fischmeister, Univ. of Waterloo, Canada
Insup Lee, Univ. of Pennsylvania, USA
Julián Proenza, Univ. of the Balearic Islands, Spain

Program Committee:

Anton Cervin, Lund University, Sweden
Antonio Casimiro, University of Lisbon, Portugal
Arnaldo Oliveira, University of Aveiro, Portugal
Carlos Eduardo Pereira, UFRG, Brazil
Chang-Gun Lee, Seoul National University, Korea
Christoph Kirsch, University of Salzburg, Austria
Eric Rutten, INRIA Grenoble, France
Guillem Bernat, Rapita Systems, UK
Jane Liu, Academia Sinica, Taiwan
Jean-Dominique Decotignie, CSEM, Switzerland
Jörg Kaiser, University of Magdeburg, Germany
Joseph Sifakis, VERIMAG, Grenoble, France
Lucia Lo Bello, University of Catania, Italy
Marco Caccamo, University of Illinois UC, USA
Marga Marcos, University of the Basque Country, Spain
Marisol García-Valls, Univ. Carlos III in Madrid, Spain
MoonZoo Kim, KAIST, Korea
Neil Audsley, University of York, UK
Pau Martí, Technical University of Catalonia, Spain
Paulo Pedreiras, University of Aveiro, Portugal
Raj Rajkumar, Carnegie Mellon University, USA
Robert Trausmuth, Univ. of Applied Sciences WN, Austria
Roman Obermaisser, Technical University Vienna, Austria
Stefan Petters, NICTA, Australia
Thomas Nolte, Malardalen University, Sweden
Xue Liu, McGill University, Canada

Additional Reviewers:

Eduardo R. B. Marques
Elisabet Estévez
Emanuele Toscano
Federico Pérez
Insik Shin
Qixin Wang
Rainer Trummer
Reiner Perozzo
Robert Staudinger
Sina Meraji
Stanley Bak
Yunho Kim

Sponsors:

IST-004527 ARTIST2 Network of Excellence on Embedded Systems Design

Preface

Adaptive systems can respond to environmental changes including hardware/software defects, resource changes, and non-continual feature usage. As such, adaptive systems can extend the area of operations and improve efficiency in the use of system resources. However, adaptability also incurs overhead in terms of system complexity and resource requirements. For example, an adaptive system requires some means for reconfiguration. These means and their mechanisms introduce additional complexity to the design and the architecture, and they also require additional resources such as computation, power, and communication bandwidth. Consequently, adaptive systems must be diligently planned, designed, analyzed, and built to find the right tradeoffs between too much and too little flexibility.

The issue is how to provide the adaptability to the application, because it affects all aspects of the development process (e.g., capturing, methodologies, modeling, analysis, testing, and implementation), the chosen system technologies (e.g., computation and communication models, interfaces, component-based design, programming languages, dependability, and design patterns) and the system itself (e.g., operating system, middleware, network protocols, and application frameworks).

In many systems, flexibility and the resulting tradeoffs is usually ignored until a very late stage. Many try to retrofit existing prototypes, middleware, operating systems, and protocols with concepts and means for flexibility such as run-time system reconfiguration or reflexive diagnostics and steering methods. Such retrofitting typically leads to disproportionate overhead, unusual tradeoffs, and in general it leads to less satisfactory results.

The purpose of the workshop is to discuss new and on-going research that is centered on the idea of adaptability as first class citizen and consider the involved tradeoffs.

Among the 26 initial submissions, 16 papers have been selected and organized in 4 sessions, covering a wide spectrum of the subject of Adaptive and Reconfigurable Embedded Systems. It is our wish that the workshop provides an appropriate and relaxed environment to discuss these new ideas and approaches. In order to facilitate it each speaker will have 15 minutes for the presentation and each session will finish with a 30-minute panel discussion with the 4 speakers of that session. Moreover, we will have Prof. Karl-Erik Arzen from Lund University in Sweden as invited speaker. The title of his speech will be: Adaptivity in Embedded Systems, Why, What and How.

We would like to thank all the people that have made possible this event. First of all, thanks to the organizers of the Cyber Physical Systems Week for accepting our proposal of celebrating this workshop; second, to the authors that submitted their articles; third, to the members of the Program Committee and other reviewers for their fundamental contribution to the quality of the final program; and last but not least, to the ARTIST2 Network of Excellence on Embedded Systems Design for their financial support.

Luís Almeida, Univ. of Aveiro, Portugal
Sebastian Fischmeister, Univ. of Waterloo, Canada
Insup Lee, Univ. of Pennsylvania, USA
Julián Proenza, Univ. of the Balearic Islands, Spain

List of Papers

1. Systems

Semantics-Preserving and Incremental Runtime Patching of Real-Time Programs[†]

Christoph M. Kirsch
University of Salzburg
ck@cs.uni-salzburg.at

Luís Lopes
CRACS/University of Porto
lblopes@dcc.fc.up.pt

Eduardo R. B. Marques
University of Porto
edrdo@dcc.fc.up.pt

Abstract

We propose semantics-preserving and incremental runtime patching of real-time programs as a robust means for reconfiguring hard real-time systems at runtime. We consider programs that describe non-functional aspects of processes such as their timing properties and communication behavior, and give examples written in the Hierarchical Timing Language (HTL). Runtime patching is the process of replacing portions of such programs at runtime by new code. It is semantics-preserving if the switch to the resulting code and the code itself could have been compiled beforehand, had the patch been known. It is incremental if analyzing and generating the code only involves an effort proportional to the size of the patch, not the patched program. This can even be done with system-wide properties such as schedulability by exploiting HTL-specific features.

1. Introduction

Software has the great advantage of being flexible. In fact, for now, it probably remains the single most flexible concept for engineering even the most complex systems. The majority of IT industries exploit that flexibility and sometimes even use it as foundation for their business models. There are important exceptions though. Large portions of the real-time systems industry, in particular, the ones working on mission- and safety-critical applications essentially ignore software-related flexibility. There are good reasons after all. Getting large software systems and, in particular, real-time systems right is still extremely difficult. How can we then even think about modifying such systems while they are running? Clearly, adaptivity is not just a nice-to-have feature, especially in real-time systems where it may give rise to unforeseen application scenarios and software development methodologies. What is even more exciting though is that the essential, enabling technologies may currently be shaping up to make adaptivity of even hard real-time systems a reality.

We believe there are two key ingredients. Adaptivity needs a strong semantical foundation and non-trivial scalability. We need to know what reconfiguration means and how to do it fast, even on large systems. There is a growing research trend towards so-called semantics-preserving execution environments for real-time systems such as the real-time language Giotto [9] and its successors but also other work on synchronous reactive languages [2], which provide notions of composability that go beyond the typical schedulability guarantees of more traditional real-time languages and operating systems. For example, Giotto programs can be modified without changing the relevant properties of the unmodified portions as long as there are sufficient computational resources. Relevant properties are not just schedulability but also task functionality, intertask communication, and I/O times. Nevertheless, checking schedulability and other system-wide properties remains necessary but is often difficult and may limit scalability of reconfiguration attempts. Recent work, however, on incremental schedulability analysis of traditional task models [5] but also language-based models [7], in combination with stronger, semantical notions of composability, may lead to fast, scalable, and semantics-preserving reconfiguration of real-time systems.

In this paper, we propose semantics-preserving and incremental *runtime patching* of real-time programs as a robust means for reconfiguring even large systems at runtime, and give examples written in HTL [7], a Giotto successor. So far, we have only studied the idea conceptually and worked with examples. Our plan is to design and implement runtime patching support in our existing HTL infrastructure [1] and perform experiments with unmanned vehicles in Salzburg [4] and Porto [12]. In Section 2, we give an intuitive overview of our approach. In Section 3, key concepts of HTL are provided, followed by a presentation of an HTL-based runtime patching model in Section 4.

[†]C. M. Kirsch is supported by a 2007 IBM Faculty Award, the EU ArtistDesign Network of Excellence on Embedded Systems Design, and the Austrian Science Fund No. P18913-N15. L. Lopes is partially supported by project CALLAS from Fundação para a Ciência e Tecnologia (contract PTDC/EIA/71462/2006). E. R. B. Marques is supported by the SFRH/BD/29461/2006 grant from Fundação para a Ciência e Tecnologia.

We would like to thank João Sousa and Raja Sengupta for inspiration in this work and Sebastian Fischmeister for some relevant suggestions and comments.

2. Runtime Patching

We consider programs that describe non-functional aspects of processes such as their timing properties and communication behavior. The syntax tree of such a program P is depicted schematically in the left portion of Fig. 1. The program describes a set of processes as illustrated in the right portion of Fig. 1. We assume that there is a means to identify subprograms of a given program, for instance, by unique path names. The syntax tree in Fig. 1 shows such a path σ to a subprogram S of P. We also assume that there is a homomorphic relationship F between (syntactic) program and (semantical) process composition in the sense that a strict subprogram of a given program can only affect the relevant behavior of a strict subset of the processes described by the program, i.e., $F(P \setminus S + S) = F(P \setminus S) + F(S)$. Fig. 1 indicates that subprogram S only affects the relevant behavior of a strict subset of all processes described by program P. Examples of relevant behavior are process functionality, periodicity, and I/O times while resource consumption such as CPU usage is not. Processes described by S may share resources with processes described by other parts of P and may therefore affect their access to resources. We discuss how to check resource consumption in principle below.

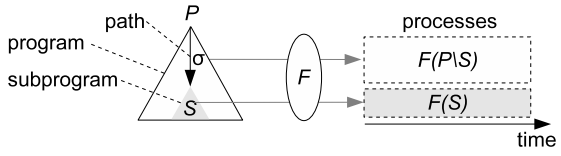


Figure 1. Syntax and semantics

By runtime patching we intuitively mean the process of identifying a subprogram O of a program P by a path σ and then replacing O in P by a new program N, logically instantaneous at runtime, i.e., during the execution of P, resulting in a program P', as shown in Fig. 2. The time instant when the patch takes effect is called the install instant I. Applying a runtime patch may take time and may therefore be started some time before I. However, a runtime patch should only take effect atomically at I, similar to, for example, atomic transactions in databases. Runtime patching enables software adaptivity because patches (σ and N) do not need to be known at compile time, and programs do not need to be stopped for patching. Patch operator implementations may require some form of dynamic loading and linking as well as possibly incremental compilation, unless programs are interpreted.

We do not intend runtime patching to extend the expressiveness of the language in which patched programs are written. In fact, we advocate runtime patching to preserve the exact original language semantics. In other words, there must be a program Q that is syntactically equivalent to P but with O replaced by a conditional expression choosing

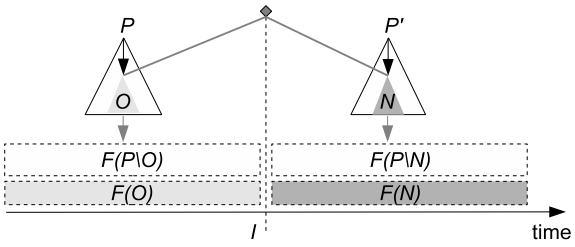


Figure 2. Patching

between O and N as illustrated in Fig. 3. During the execution of Q, the conditional expression, depicted by a box, mimicks runtime patching by switching from O to N exactly at I, i.e., when O in P is patched to become N. Runtime patching is thus a semantics-preserving means to modify programs at runtime in a way that could have been done at compile time, if the timing of the patch (I) and the patch itself (σ and N) had already been known.

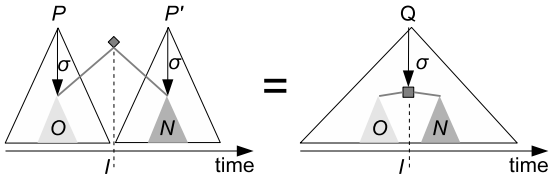


Figure 3. Preserving semantics

Runtime patching involves program analysis and code generation. If a patch requires re-checking program-wide properties such as overall resource consumption, or even full re-compilation, runtime patching may either be limited in scale or may take too long and make the application of the patch ineffective. However, incremental compilation, i.e., incremental program analysis and code generation, may enable fast and scalable runtime patching. With incremental program analysis, checking if the patched program P' is correct, given that the original program P is correct, should only involve an effort proportional to the size of the patch (σ and N) and some context C of the patch but independent of P, as shown in Fig. 4. The size of C should be determined by the size of the patch. Similarly, code generation should be proportional to the size of the new program N, and linking should only involve considering context C.

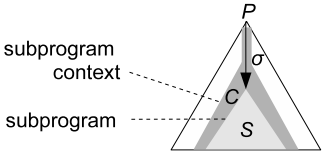


Figure 4. Scalability

Incrementally checking even global properties such as overall resource consumption may also be possible by taking advantage of language properties such as, if P is correct (e.g., resource-compliant) and program N is in some sense

compatible with context C, then P' is also correct. This property reduces checking global correctness to checking local compatibility. For example, C may contain an abstract specification implicitly describing a possibly infinite set of concrete programs for which the resulting patched program is guaranteed to be correct, i.e., without re-checking global correctness. Then, checking if a concrete program is compatible with the abstract specification is sufficient for global correctness (but not necessary since there might be concrete programs that are incompatible but for which the patched program is correct anyway).

3. HTL Overview

The Hierarchical Timing Language (HTL) [7] is a coordination language for distributed hard real-time applications. HTL programs specify timing properties and communication behavior of interacting real-time tasks that are potentially distributed across multiple hosts but not task functionality, which is assumed to be implemented in some other language than HTL. Prior to execution, HTL programs are compiled into E code [10], or HE code [8], which supports separate compilation. E and HE code are interpreted in real time by a virtual machine, which uses an EDF scheduler for executing the tasks. The ability to compile parts of HTL programs separately [8] and check their schedulability incrementally [7] is a prerequisite for incremental runtime patching of HTL programs.

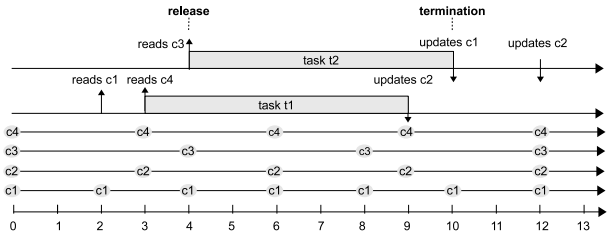


Figure 5. Tasks and communicators

Tasks and communicators. An HTL task is defined by a sequential code procedure with no internal synchronization, a set of input/output variables called *ports*, a period for execution, and a worst-case execution time (WCET). Tasks with different periods interact by exchanging port values through *communicators*, which are timed variables that can be read and written at logical time instants according to the communicators' own periods. This interaction is illustrated in Fig. 5. The periods of interacting tasks must be multiples of the involved communicator periods. Tasks with the same period may interact with other tasks through their ports as long as the tasks read from have completed and the reading tasks have not yet started executing, which gives rise to task precedence constraints.

An HTL task has a *logical execution time* (LET) given by its *release* and *termination* events, which are defined by communicator read and write actions: the release time is the latest time instant for a communicator read and the termination time is the earliest time instant for a communicator write. Fig. 5 illustrates this for tasks `t1` and `t2` and their interaction through communicators `c1` to `c4`. This task model is a generalization of the LET model in Giotto [9], to tasks with input and output ports interacting through communicators. The key advantage of the LET model is that the relevant behavior of LET programs (functionality but also exact I/O times) is preserved across different hardware platforms and software workloads as long as there are sufficient computational resources [9].

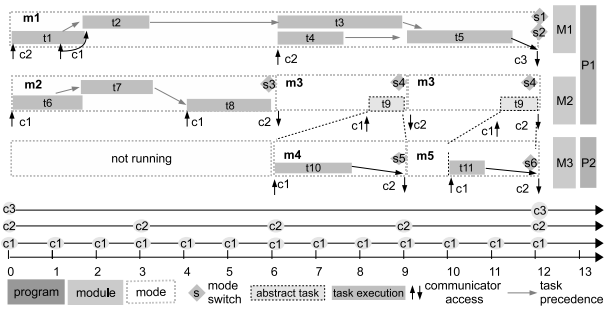


Figure 6. An HTL program

Program structure. Building up on the foundation of tasks and communicators, the other structuring concepts in HTL are modes, modules, programs, and hierarchical program refinement. Fig. 6 gives an overview of their assembly and execution.

A *mode* is a set of tasks with the same period (the mode's period) and an acyclic graph that expresses data flow among input and output ports of the tasks in the mode. A mode's execution equals the logical execution of all its tasks under communicator timing constraints (to interact with tasks external to the mode) but also under task precedence constraints. For example, in Fig. 6, `t2` in mode `m1` is only released after `t1` has completed, even though `t2`'s release time (the access to `c1`) is actually earlier. A *module* is set of modes and a set of boolean predicates called *mode switches* that are evaluated over communicators and ports. At any time, exactly one mode executes within a module, and, at the end of its period, execution is either switched to a different mode in the module or continued in the same mode, according to the evaluation of mode switches. For example, in Fig. 6, in module `M2`, the mode switch `s3` is evaluated at time instant 6 at the end of the execution of mode `m2` and changes execution to mode `m3`. A set of modules and a set of communicator declarations form a *program*, and a program's execution equals the parallel execution of all its modules with the tasks interacting through the program's communicator set. For example, in Fig. 6, program `P1` consists of modules `M1` and `M2` as well as communicators `c1`, `c2`, and `c3`.

Hierarchical program structure is expressed using *refinement* of a mode by an entire program, as shown in Fig. 6 for mode `m3` and program `P2`. A mode being refined, called the *parent mode*, may have declared *abstract tasks* that have no implementation and simply act as schedulability-conservative place-holders for *concrete tasks* in the refinement program conforming to a set of syntactic restrictions [7]. The refinement constraints preserve schedulability and simplify program analysis: if the parent mode is analyzed and asserted as schedulable, then the refinement program is also known to be schedulable. Checking refinement constraints is generally faster and more scalable than checking schedulability and can therefore be done incrementally. The former is linear in the size of the refinement program whereas the latter may be exponential in the size of the refined mode because of higher-level mode switching. We have also studied refinement constraints with so-called logical reliability of communicator updates instead of task schedulability [3] but have not yet considered it in runtime patching.

4. A Runtime Patching Model for HTL

Runtime patching for HTL requires a mechanism to load, analyze, and apply patches at runtime. We propose to use a *patch supervisor* process that monitors a running HTL program and allows its patching. The patch supervisor is not meant to be an HTL entity itself but one that operates on top of it in the sense of a program rewriting other programs in congruence with our principle that syntax and semantics of the patched programs are preserved. The patch supervisor should apply instrumentation in a way that at any time instant the running program is a proper instance of the original language in which it was written. Also, it should be executed at a lower priority than the patched program, so that the real-time performance requirements of the latter are not compromised, and only declare a patch as ready to take effect once all required time-consuming aspects of readying the patch are done. Its typical cycle will be: attend to program patch requests, perform program re-compilation, and apply patches logically instantaneous at time instants that ensure coherent atomic transitions between the original and patched program.

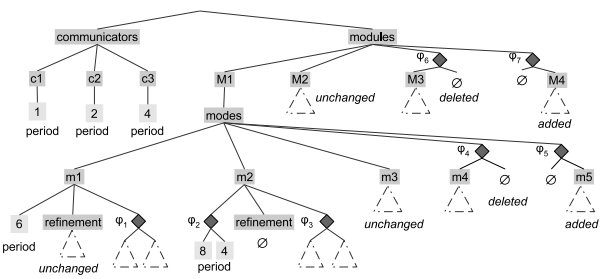


Figure 7. An HTL program patch

Patch specification. Fig. 7 displays an HTL program P in the form of a simplified syntax tree and a patch applied to it yielding program P'. A patch may consist of multiple program transformations, expressed at the syntactic, source-code level, through rewriting of the program's syntax tree. The diamond notation represents a program transformation through patching, with the original subprogram on the left and the new subprogram on the right. The patch shown consists of changes at the mode level for module `M1` (transformations φ_1 to φ_5) and at the module level for the top-level program (φ_6 and φ_7). Patching at the mode level within a module can change an existing mode (φ_1 to φ_3), delete a mode (φ_4), and add a new mode (φ_5). Patching an existing mode may change timing properties like a mode's period, as in φ_2, but also other aspects (e.g., within φ_1 and φ_3) like task precedences, communicator accesses and WCET estimate, as well as functional aspects like task and mode switch implementations. Patching at the module level may remove (φ_6) and add modules (φ_7). Program patching may also be recursive and apply to refinement programs. In constrast to [6], which describes a mechanism for semantics-preserving replacement of real-time program functionality for the Timing Definition Language (TDL), a subset of Giotto and thus of HTL, our approach generalizes to patching concurrent modules and, in particular, modes, besides considering scalability aspects for patching.

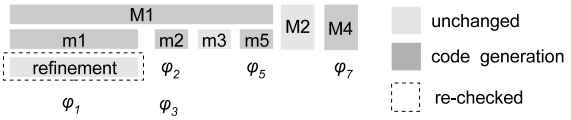


Figure 8. Runtime compilation

Runtime compilation. Compilation of a patched HTL program at runtime must validate the program syntactically to assert the program as valid, analyze the schedulability of the program depending on the type of transformations applied, and re-generate and link code for the changed program parts. Syntactic validation needs to consider only a context composed of the modified parts and their dependencies, which are induced by program refinement and communicator writes that may be performed by pre-existing modules (which could result in race conditions). Code generation for HTL may adopt a separate compilation strategy even down to the level of modes [8]. Thus it is possible to re-generate code only for the modified parts of a patched HTL program.

The subprogram context for syntactic validation and code generation for each transformation in our patching example is illustrated by Fig. 8. Syntactic validation and code generation is required for all changed and added functionality, as shown. Assuming there are no dependencies induced by communicator writes in the example, there is, however, a need to account for the dependencies of program parts

changed by program refinement: even though the refinement program for `m1` does not require re-compilation, it must nevertheless be re-checked with respect to syntactic refinement constraints to make sure that the patched program is still schedulable.

In general, schedulability analysis, however, may not be scalable if the patch targets top-level specifications. If timing behavior is patched, schedulability analysis is only incremental to changes if the patched program is a refinement but not a top-level program since only refinement constraints preserve schedulability. If a top-level program is in question, as in our example, then schedulability may be asserted through full program analysis but with exponential time complexity in the size of the program, or potentially faster through other incremental schedulability analysis techniques such as in [5], assuming they can be generalized to cover mode switching.

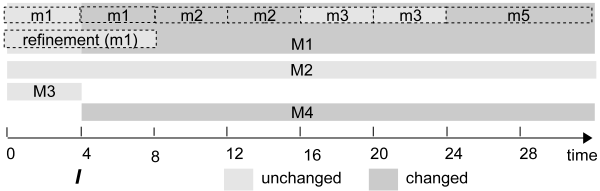


Figure 9. Patched execution

Patched execution. For the patch supervisor to instrument running HTL programs logically instantaneous in a semantics-preserving way, we consider the timing and integrity effects of patching. A runtime patch at the level of modules executing concurrently within an HTL program is constrained by the transformations it involves: (1) modules added by the patch must not be started before the time instant that marks the beginning of the least common multiple of all communicator periods in the module, so that the module has a coherent time origin, (2) modules removed by the patch must terminate execution as soon as the current mode ends execution (the outcome of mode switch evaluation will be ignored), and (3) modules changed by the patch must switch to the patched behavior when execution of the current mode ends, including the evaluation of mode switches which must yield a mode that is defined in the patched program, i.e., one that has not been removed.

The time instants to which patching is constrained by (1) to (3) determine the set of possible install instants for the patch. We assume that activating the patched program takes logically zero time. As discussed before, all time-consuming aspects of runtime compilation complete before the install instant. In the sense of the various aspects surveyed in [11], the runtime patching model we consider is therefore *synchronous*. If the patch involves more than one kind of transformation, the install instant must satisfy all of their timing constraints, i.e., be a valid synchronization point for all transformations. This condition may be relaxed for simultaneous module updates and removals to happen before additions. New modules could start after all module updates and removals have been completed. This mode of operation can be interesting for defining more flexible patching schemes. However, it would imply an interval of logical time for patching, rather than a logical time instant, and require additional schedulability analysis, as in *asynchronous* patching [11]. In any case, a patch supervisor has the flexibility of applying different transformations that may be part of a set of patches at different appropriate time instants, so the above constraints may not be too restrictive.

Fig. 9 illustrates patched execution for our example. The patch is applied logically at time instant 4 in line with the constraints stated above. Time instant 4 is the least common multiple of all communicator periods (1, 2, 4 for `c1`, `c2`, and `c3` in Fig. 7), so that `M4` can be started at that time assuming that mode execution for `M1` (modified) and `M3` (deleted) properly terminates. The patched execution within module `M1` shows changed components and sample mode switching behavior. When patching has completed, execution is switched from `m1` (according to the specification of old code for `m1`) to the patched version of itself. Any other switch would also be valid as long as the target mode is defined in the patched program (`m2`, `m3`, or `m5`). A mode switch to `m4` at time instant 4 would invalidate the patching operation since the patch specifies `m4` to be deleted.

References

[1] J. Auerbach, D. Bacon, D. Iercan, C. Kirsch, V. Rajan, H. Röck, and R. Trummer. Java takes flight: Time-portable real-time programming with Exotasks. In *Proc. LCTES*, 2007.

[2] P. Caspi, N. Scaife, C. Sofronis, and S. Tripakis. Semantics-preserving multitask implementation of synchronous programs. *ACM TECS*, February 2008.

[3] K. Chatterjee, A. Ghosal, D. Iercan, C. Kirsch, T. Henzinger, C. Pinello, and A. Sangiovanni-Vincentelli. Logical reliability of interacting real-time tasks. In *Proc. DATE*, 2008.

[4] S. Craciunas, C. Kirsch, H. Röck, and R. Trummer. The JAviator: A high-payload quadrotor UAV with high-level programming capabilities. In *Proc. AIAA GNC*, 2008.

[5] A. Easwaran, I. Shin, O. Sokolsky, and I. Lee. Incremental schedulability analysis of hierarchical real-time components. In *Proc. EMSOFT*, 2006.

[6] S. Fischmeister and K. Winkler. Non-blocking deterministic replacement of functionality, timing, and data-flow for hard real-time systems at runtime. In *Proc. ECRTS*, July 2005.

[7] A. Ghosal, T. Henzinger, D. Iercan, C. Kirsch, and A. Sangiovanni-Vincentelli. A hierarchical coordination language for interacting real-time tasks. In *Proc. EMSOFT*, 2006.

[8] A. Ghosal, D. Iercan, C. Kirsch, T. Henzinger, and A. Sangiovanni-Vincentelli. Separate compilation of hierarchical real-time programs into linear-bounded embedded machine code. In *Online Proc. APGES*, 2007.

[9] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: A time-triggered language for embedded programming. *Proc. of the IEEE*, January 2003.

[10] T. Henzinger and C. Kirsch. The Embedded Machine: predictable, portable real-time code. In *Proc. PLDI*, 2002.

[11] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *RTS*, Springer, 2004.

[12] Seascout LAUV. http://whale.fe.up.pt/seascout.

Limitations of Adaptable System Architectures for WCET Reduction

Jack Whitham and Neil Audsley
Real-Time Systems Group
Department of Computer Science
University of York, York, YO10 5DD, UK
jack@cs.york.ac.uk

Abstract

This paper identifies three major issues facing worst-case execution time (WCET) reduction algorithms on adaptable architectures based on research carried out for the MCGREP-2 CPU project. The issues are exposing more instruction level parallelism (ILP) in code, reducing loading costs for the memory and processing elements used to reduce WCET, and making use of application-specific hardware. Potential difficulties in each of these areas are identified and possible solutions are proposed.

1 Introduction

Embedded systems often include some real-time functionality, such as control of external machinery [1]. Real-time tasks must operate within known time bounds (*deadlines*) in order for the overall system to be *safe*, and this poses an additional requirement for software design. Computing the *worst-case execution time* (WCET) of real-time tasks is an important step towards assuring the safety of the overall *real-time system* (RTS) [18]. WCET *reduction* for a task is a closely related problem involving the allocation of some memory or computing resource in order to minimize the WCET (Figure 1).

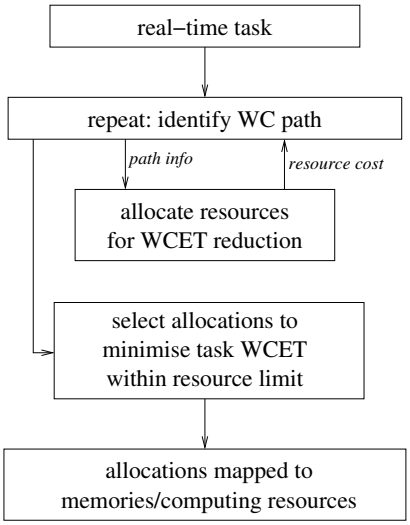


Figure 1. Generalized WCET reduction process.

General execution speed-up technologies such as cache memory, deep CPU pipelines and out-of-order superscalar issue units [15] are good for *average case execution time* (ACET) reduction, but the dynamic behavior of these components makes computing the WCET more difficult [10,24]. Therefore, even if the WCET of a task can be reduced by such techniques, the safety of the RTS cannot be easily assured. This motivates approaches that explicitly reduce the WCET of programs without introducing dynamic behavior, either automatically or with programmer assistance [28]. Automatic approaches have their roots in hardware/software co-design, i.e. partitioning tasks and subtasks between hardware and software in order to meet an optimization goal [2], e.g. ACET or WCET minimization. However, because of the relative difficulty of evaluating the resource consumption of candidate partitions (requiring hardware synthesis [8]) and because of the differences between hardware and software languages [7], current WCET reduction techniques avoid any need to generate hardware and instead operate by migrating subtasks into memory units that enable faster execution. These include instruction scratchpads [17] and lockable caches [3]. In these cases, the partitioning problem is relatively simple and can be solved by fast heuristics [21].

Some forms of adaptive system facilitate WCET reduction. The MCGREP-2 CPU [25–27] provides a *writable control store* (WCS) [20] that can store subtasks encoded as microinstructions (Figure 2). Our recent work [28] shows that subtasks can be selected from the *worst-case execution path* (WC path) of a program and translated automatically into microinstructions: this leads to greater WCET reductions than instruction scratchpad techniques because *instruction level parallelism* (ILP) can be exploited to execute the WC path in a shorter time period. MCGREP-2 is adaptive and reconfigurable in the sense that the control store can be updated at any time, allowing an unlimited number of tasks to benefit from WC path optimizations. Using microinstructions to implement subtasks is predictable in two senses: (1) execution timings are not data-dependent, and (2) resource consumption is easily computed [28]. However, the speed of each subtask is limited by the microarchitecture and the input program.

This paper explores the issues that limit WCET re-

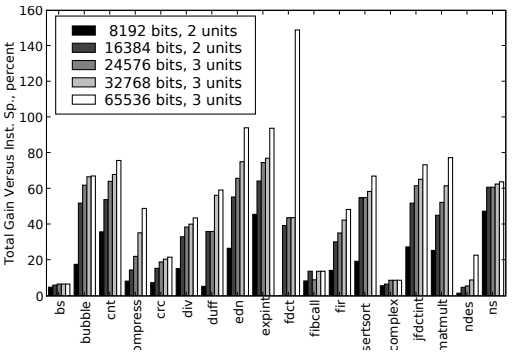


Figure 3. WCET reductions with various MCGREP-2 configurations using various benchmark programs.

duction within a general adaptive reconfigurable system, based on our research for the MCGREP-2 CPU project and its associated WCET reduction algorithms [25, 28]. We assume that WCET reduction begins with a task specified in a software language such as C, and then proceeds through a fully automatic process described in previous work [17,21,28] (such processes follow the general outline of Figure 1). We consider WCET reduction algorithms making use of scratchpads, locked caches, co-processors and run-time reconfigurable hardware [11], such as a *field programmable gate array* (FPGA).

Although WCET reductions can be achieved using the MCGREP-2 WCS, the execution time improvements that have been demonstrated are currently limited to about 50-150% [25] over an instruction scratchpad [17, 21] (Figure 3). Independent of the architecture actually used to implement the WCS [27], and independent of the technology used to apply WCET reductions [28], the magnitude of the possible reduction is limited by three major factors. These are: (1) the ILP available within the task, (2) the cost of loading the control store with the required information, and (3) the speed of the general-purpose microarchitecture that executes the microinstructions. These factors apply to any WCET reduction process, whether it is based on a scratchpad, locked cache, or some form of run-time reconfigurable hardware. The issues related to each are examined in sections 2, 3 and 4. Section 5 concludes.

2 The ILP Limitation

The ILP available within each task is influenced by both the source code and the compiler. WCET reduction approaches that operate by allocating instruction scratchpad or lockable cache space [3, 17, 21] do not consider ILP in code since conventional machine instructions are sequential. Trace scratchpad allocation approaches [28] do consider ILP, as machine instructions are converted into explicitly parallel code for storage in the WCS. This simplifies WCET analysis, but also implies that the degree of WCET reduction is limited by the ILP in the task.

In many programs, the degree of ILP is limited to two or three instructions within a single basic block, and around twice that number if basic block boundaries can be ignored through speculation [22]. This is a hard limit on WCET reduction for general software. For ACET reduction, the limit is approached by current superscalar CPU designs, and some of the same principles can be applied for WCET reduction [28]. Reaching this limit is an implementation challenge requiring the design of superscalar CPU pipelines that are also amenable to timing analysis.

Obtaining WCET reductions *beyond* the ILP limit is a language issue. Tasks that are *vectorisable* can be parallelised across a very large number of processing elements [22], because most subtasks are independent of each other. However, not all programming languages allow vectorisable code to be declared. A limited form of automatic vectorization is provided by *modulo scheduling* [4], but in general a specialist language is needed. The compiler needs additional information about data and control dependences in order to be able to arrange the subtasks for vector processing. Dataflow languages provide the required features, allowing both coarse-grained [5, 9] and fine-grained [30] reconfigurable arrays to be programmed. More conventional languages can also support extensions for vectorization, e.g. [12].

3 The Load Cost Limitation

Regardless of whether WCET reductions are provided by a scratchpad, locked cache or run-time reconfigurable hardware, a loading time cost is incurred whenever the configuration is updated. Some WCET reduction algorithms assume that loading takes place before task start-up [21, 28], but this is restrictive because it places a limit on the complexity of each task. This limit also applies to WCET reduction approaches that make use of fixed co-processors, since these are not run-time reconfigurable. The solution is to allow loading during execution and incorporate it into the WCET reduction process [17], after partitioning each task into regions with local memory maps [16]. The total loading time must be less than the total WCET reduction that is achieved.

Loading costs for scratchpads and locked caches are small, since burst-mode transfers can be used to rapidly move information from large external memory into smaller scratchpads. In a task with sufficient temporal locality on the WC path (e.g. many loops), loading time will be significantly smaller than the WCET for both instruction scratchpads [17] and a WCS [25]. However, the degree of temporal locality that is required is higher. If one instruction can be loaded into an instruction scratchpad in a single clock cycle, and then executed in a single clock cycle, then that instruction only needs to be executed twice to recover the cost of loading the scratchpad. But microinstructions are often larger than conventional instructions, and consequently more executions are required to recover

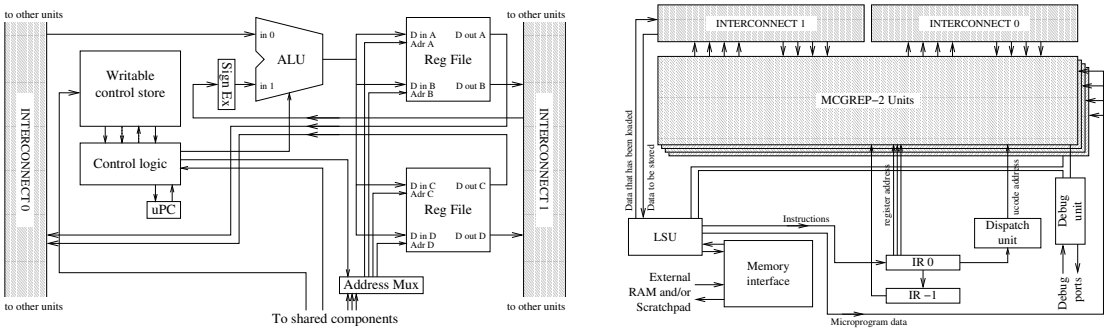


Figure 2. MCGREP-2 CPU: one array unit (left) and top level (right). MCGREP-2 is a simple form of coarse grained reconfigurable architecture (CGRA) in which each array unit is a small CPU capable of executing code from a writable control store, which can be used to reduce task WCETs.

the loading cost.

Loading can be carried out in parallel with task execution by introducing a *direct memory access* (DMA) controller to manage the copying process. A second task may be executed during the loading process, or the first task may continue execution as information is loaded into scratchpad for use in the near future. Both techniques have been previously explored by research into *overlaying* [14] and are implemented by modern CPU architectures in the form of *simultaneous multithreading* (SMT) and cache filling. Predictable forms of these dynamic operations could be used to eliminate effective loading costs in some cases.

We believe that loading costs are likely to become a significant problem for some systems. Run-time reconfigurable hardware loading costs can be very large: typical FPGA bitstream sizes are given by [30]. Run-time reconfigurable systems may include decompression modules to reduce the cost [11]. Consequently, a very high degree of temporal locality is required to reduce the overall WCET unless loading costs can be eliminated by parallel operation.

4 The General Purpose CPU Architecture Limitation

Perhaps the most serious limitation of present WCET reduction approaches is the assumption that a general purpose architecture is used. A conventional general-purpose CPU is assumed by instruction scratchpad and locked cache allocation approaches [3, 17, 21]. Although the MCGREP-2 CPU is extensible with *application-specific instruction set processor* (ASIP) features [6], such as custom instructions to accelerate WC path execution, these must be declared and applied explicitly by the programmer. Automatic WCET reduction algorithms for scratchpads only make use of standard ALU features at present.

CGRA architectures [5, 9] provide arrays that are specialized for vectorisable code. Mapping subtasks to such architectures is one way to reduce WCET, but large CGRAs are not suitable for general programs because insufficient ILP is available. The same problem applies to fine-grained arrays such as FPGAs, but these can provide even greater reductions because the logic gates can be specialized to a particular task. Automatic ASIP custom instruction selection for WCET reduction has been explored [31], and it is known that large ACET [13] and WCET [23] reductions are possible by migrating software into FPGA hardware, even without vectorisable code. Since run-time reconfiguration can be used to load task-specific hardware, customized hardware could be used in a similar manner to an instruction scratchpad or WCS, but with greater potential WCET reductions than either approach.

However, the search algorithms used to find the best allocations for WCET reduction become far more complex, since it is not easy to calculate the resource consumption of each allocation decision. To get an exact answer, a complete FPGA or ASIC synthesis process must be executed with all chosen components in place, and this is computationally expensive. Estimation is commonly used instead [29, 31], but this lowers the accuracy of allocation decisions and is likely to lead to poor utilization of space, or backtracking in the event of an overestimate. Scratchpad allocation algorithms can make use of all available space [17, 28] because exact computation of resource usage is very fast, and backtracking is not necessary [21].

We believe that WCET reduction using custom hardware is a form of co-design problem, and therefore NP-hard [19]. However, with appropriate restrictions and assumptions, WCET reduction can nevertheless be applied effectively using custom hardware. For example, run-time reconfigurable modules (e.g. [11]) of fixed size could be generated to provide WCET reductions to specific subtasks: this would isolate the WCET reduction process from the considerations of resource consumption and on-chip communication.

5 Conclusion

This paper has explored three issues that affect the degree of WCET reduction available for tasks in an adaptable architecture. Major challenges exist: specifying vectorization in order to exploit greater ILP is important [22], as is minimizing loading time costs [17]. Finding a way to apply WCET reduction algorithms to custom hardware may be the most rewarding challenge, as large execution time reductions are possible [13,23] if the technical issues of efficiently searching for the best resource allocation can be solved. These problems have been given only partial consideration by existing work. Solutions would allow embedded real-time systems to carry out more operations per time unit by explicitly reducing the WCET of each task.

References

[1] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, 2001.

[2] R. Ernst, J. Henkel, and T. Benner. Hardware-software cosynthesis for microcontrollers. *IEEE Des. Test*, 10(4):64–75, 1993.

[3] H. Falk, S. Plazar, and H. Theiling. Compile-time decided instruction cache locking using worst-case execution paths. In *Proc. CODES+ISSS*, pages 143–148, New York, NY, USA, 2007. ACM Press.

[4] J. Fisher, P. Faraboschi, and C. Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2004.

[5] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor. PipeRench: A reconfigurable architecture and compiler. *Computer*, 33(4):70–77, 2000.

[6] R. E. Gonzalez. Xtensa — A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, 2000.

[7] B. Grattan, G. Stitt, and F. Vahid. Codesign-extended applications. In *Proc. 10th Int. Symp. Hardware/Software Codesign*, pages 1–6, 2002.

[8] R. K. Gupta and G. D. Micheli. Hardware-software cosynthesis for digital systems. *IEEE Des. Test*, 10(3):29–41, 1993.

[9] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger. KressArray Xplorer: a new CAD environment to optimize reconfigurable datapath array. In *Proc. ASP-DAC*, pages 163–168, New York, NY, USA, 2000. ACM Press.

[10] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proc. IEEE*, 91(7):1038–1054, 2003.

[11] M. Hubner and J. Becker. Exploiting dynamic and partial reconfiguration for FPGAs: toolflow, architecture and system integration. In *Proc. SBCCI*, pages 1–4, New York, NY, USA, 2006. ACM Press.

[12] Intel. Optimizing Applications with the Intel C++ and Fortran Compilers (accessed 26 April 07). `ftp://download.intel.com/software/products/compilers/techtopics/Compiler_Optimization_7_02.pdf`, 2004.

[13] R. Lysecky, G. Stitt, and F. Vahid. Warp processors. *ACM TODAES*, 11(3):659–681, 2006.

[14] R. J. Pankhurst. Operating systems: Program overlay techniques. *Commun. ACM*, 11(2):119–125, 1968.

[15] D. A. Patterson and J. L. Hennessy. *Computer organization & design: the hardware/software interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[16] I. Puaut and D. Hardy. Predictable paging in real-time systems: A compiler approach. In *Proc. ECRTS*, pages 169–178, Washington, DC, USA, 2007. IEEE Computer Society.

[17] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proc. DATE*, pages 1484–1489, San Jose, CA, USA, 2007. EDA Consortium.

[18] P. Puschner and A. Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Syst.*, 18(2-3):115–128, 2000.

[19] R. Niemann and P. Marwedel. Hardware/software partitioning using integer programming. In *Proceedings of the European Design and Test Conference (ED & TC)*, pages 473–480, Paris, France, 1996. IEEE Computer Society Press (Los Alamitos, California).

[20] R. F. Rosin, G. Frieder, and J. Richard H. Eckhouse. An environment for research in microprogramming and emulation. *Commun. ACM*, 15(8):748–760, 1972.

[21] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET Centric Data Allocation to Scratchpad Memory. In *Proc. RTSS*, pages 223–232, Washington, DC, USA, 2005. IEEE Computer Society.

[22] D. W. Wall. Limits of Instruction-Level Parallelism. Technical Report WRL-93-6, DEC Western Research Laboratory, 1995.

[23] M. Ward and N. Audsley. Hardware compilation of sequential Ada. In *Proc. CASES*, pages 99–107, New York, NY, USA, 2001. ACM Press.

[24] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Proc. Int. Conf. Quality Software*, Sep. 2005.

[25] J. Whitham. *Real-time Processor Architectures for Worst Case Execution Time Reduction*. PhD thesis, 2008.

[26] J. Whitham and N. Audsley. MCGREP - A Predictable Architecture for Embedded Real-time Systems. In *Proc. RTSS*, pages 13–24, 2006.

[27] J. Whitham and N. Audsley. A self-optimising simulator for a coarse-grained reconfigurable array. In *Proc. UK Embedded Forum*, pages 99–109. University of Newcastle, April 2007.

[28] J. Whitham and N. Audsley. Using trace scratchpads to reduce execution times in predictable real-time architectures. In *Proc. RTAS (to appear)*, 2008.

[29] Y. Xie and W. Wolf. Co-synthesis with custom asics. In *Proc. ASP-DAC*, pages 129–134, 2000.

[30] Xilinx. Virtex-4 Family Overview. Datasheet DS112, Xilinx Corporation, 2007.

[31] P. Yu and T. Mitra. Satisfying real-time constraints with custom instructions. In *Proc. CODES+ISSS*, pages 166–171, 2005.

Adaptive Framework for Efficient Resource Management in RTOS

Ameet Patil Neil Audsley
Real-Time Systems Group,
Department of Computer Science, University of York, York YO10 5DD, UK
Email:{appatil,neil}@cs.york.ac.uk

I. Introduction

Embedded systems, applications and the environment that they are deployed in have all become increasingly complex in recent years. Application demands for more resources and dynamic changes in the environment make resource management in Real-Time Operating Systems (RTOS) extremely challenging. Key approaches include specialisation or adaptation of the RTOSs resource management policies according to the dynamic requirements of the applications. This paper describes a reflection-based adaptive RTOS framework that allows dynamic application driven adaptation of RTOS resource management policies.

The context assumed by the paper is that of low-cost, limited-resource embedded systems with modern hardware ie. complex CPUs with support for virtual memory. Also, when developing a general purpose RTOS, there is limited knowledge of the potential applications that will use the RTOS. Thus, the RTOS is built for the general-case rather than according to application-specific requirements. Such an RTOS implements generic resource management policies. This paper discusses the issues related to providing application-specific resource management in a general purpose RTOS. An existing approach of defining an adaptive framework in the RTOS using reflection is described along with some results of its implementation in Linux (2.6.16 kernel). The paper also discusses the possible future directions to the approach.

The paper is organised as follows. The next section provides background and motivation for the approach adopted within the paper. Section III describes the reflection-based RTOS framework for adapting the resource management policies. Section IV presents some experimental results of the implementation of CASP [4] - an application-specific paging mechanism using the reflective framework. Section V discusses the future work to be taken up to provide better support and further improve the existing reflective approach. Finally, conclusions are presented in section VI.

II. Background

Specialising or customising the hardware and the RTOS resource management policies for every change in the application requirements is an expensive and time consuming process. Existing approaches use standard hardware along with a general purpose RTOS that implements generic resource management policies that can provide good average-case performance. As a result, the performance of applications with dynamic resource requirements is affected by the limited support provided by such generic policies. The RTOS is built for the generic-case rather than application-specific requirements.

Several different approaches have been proposed in the past to address this issue. Many approaches provide solution to only a part of the problem. For example: Rivas et. al. [16] proposed an ada-based API for application-defined scheduling in the RTOS, but no API for changing any other module like memory management; more recently Ruocco [17] proposed a user-level reflection-based approach for adaptive scheduling alone. The exokernel [10] approach also addresses the problem by allowing applications to use application-specific OS libraries. Due to the redundant OS library code attached to different applications, this approach is not suitable for realtime embedded systems with limited-resources.

Summarising, current approaches do not consider application based resource management across multiple resources. There is a need for a general approach encompassing all the system resources. Such an approach should be able to change or adapt the resource management policies to meet the dynamic application-specific requirements. The next subsection introduces reflection and related approaches.

A. Reflection

Reflection is a mechanism by which a program code or application becomes self-aware, checks its progress and can change itself or its behaviour dynamically at runtime or statically at compile time [6]. This change can occur by changing data structures, the program code itself, or sometimes even the semantics of the language its written in. To facilitate this, the application or program code has to have knowledge about the data structures, language semantics, etc. The process by which this information is provided to it is called *Reification*.

The Reflection model consists of a base-level and one or more meta-level forming a structure called *Reflective tower*. The code in the meta-level is responsible to analyse the reified information, intercept the necessary calls from or to the baselevel and affect any change if required. A protocol defined so as to establish a mechanism by which the meta-level entities introspect (analyse), intercede (Eg. by intercepting calls to or from base-level) and affect change to the base-level is called the *Meta-Object Protocol (MOP)* [15]. In reflection the meta-

level code can form a causal link (two objects are said to be causally linked to each other when a change initiated by one affects the other [6]) with the data structures in the base-level to affect a change directly.

The mechanism of Reflection has been widely used in object-oriented programming, object-oriented databases, middlewares, artificial intelligence, virtual machines and OSs [11], [13]. Reflective OSs such as ApertOS [18], Chameleon [7] and 2K [14] focus on the aspects of composition and configurability of the system as a whole and not application-specific resource management. With the use of three different custom designed languages: Spring-C [8], SDL [9] and FERT [1], the Spring OS [12] makes use of reflective information in the system to bring about certain changes in the system. More subtle fine-grained changes to the resource management policies in an RTOS can be brought about by exchanging resource related information between the applications and the RTOS. The framework described in the next section lays the foundation to such information exchange and adaptation of the RTOS resource management policies.

III. Reflection-based RTOS Framework

In the context of an RTOS, the process of reification and introspection help applications and resource management modules exchange valuable information amongst each other. This allows interception to be used to bring about fine-grained changes in the resource management modules. Within this paper, the framework consists of a base kernel core implementing support for reflection, in the form of an interface to system modules and applications to reify information, introspect and intercept the base-level [2], [3].

Unlike existing reflective approaches, the RTOS framework uses a non-traditional method of reification such that the control over reified information lies solely with the kernel. This allows the kernel to maintain a priority based list of valuable information and discard any unwanted (eg. old) information. Fig. 1 shows the movement of information within the framework.

Normally, in the traditional approach, the meta-level component receives all the information reified by its base-level. This means that whether or not the reified information is useful, the meta-level will receive it potentially adding unnecessary communication overhead.

In the framework, all the information that is reified, is passed to and stored in the kernel. Each reified information is assigned a relative importance-level depending on the source, destination and the time the information was reified. It is stored in the kernel until a meta-level component explicitly requests the information or until it gets too old to be useful anymore. The kernel moderates the flow of information between the various reflective entities in the system.

System modules and applications can choose not to be reflective. Furthermore, the framework allows reification of information not only by the base-level components but by any entity in the system. Also, the information reified by an entity can have multiple destinations. i.e. a meta-level component is able to obtain information pertaining to its resource not only from its base-level but from any entity in the system providing greater freedom and flexibility in the system.

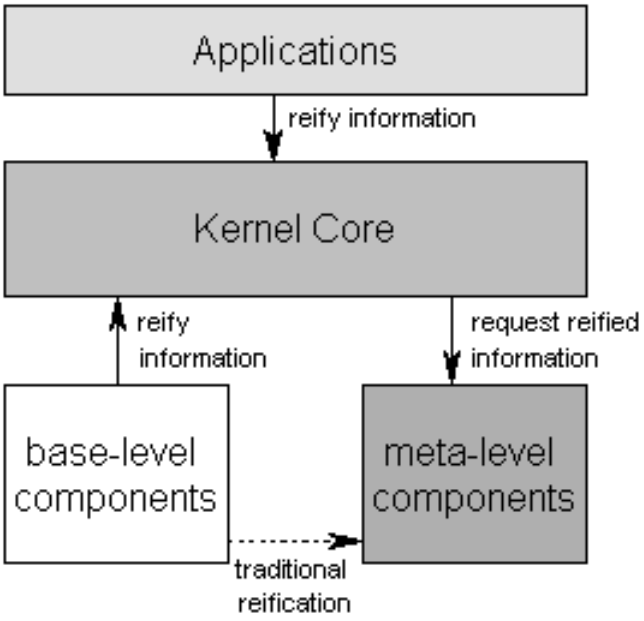


Fig. 1. Generic Reflective OS framework

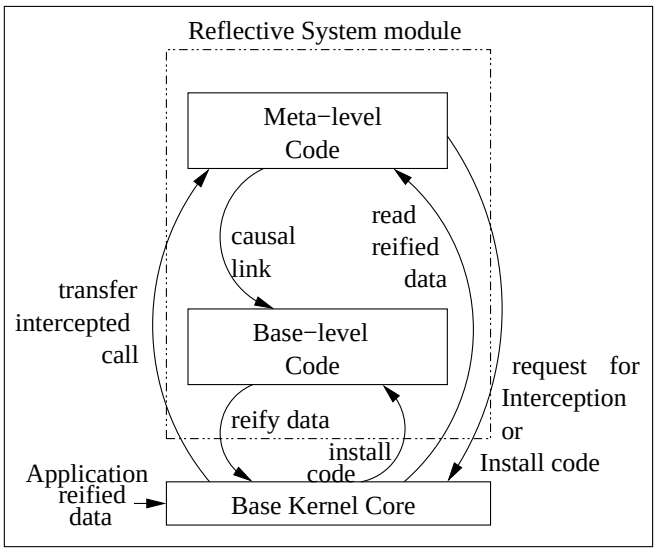


Fig. 2. Reflective System module

A reflective system module (eg. a reflective scheduler) makes use of the interface provided to access reified information stored in the kernel and take intuitive steps to intercept and change behaviour of the base-level module. On initialising, a reflective system module would implement a generic policy at its base-level (see fig. 2). For example: in case of a reflective scheduler, the base level could be a fixed priority scheduling policy. At runtime, depending on application requirements, the meta-level component of the scheduler can then change or

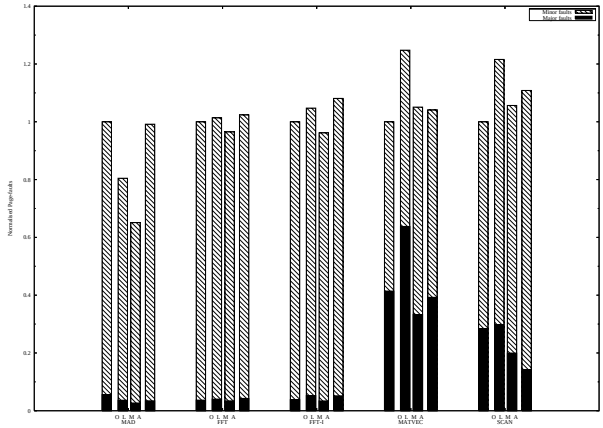


Fig. 3. Page-fault performance of benchmarks

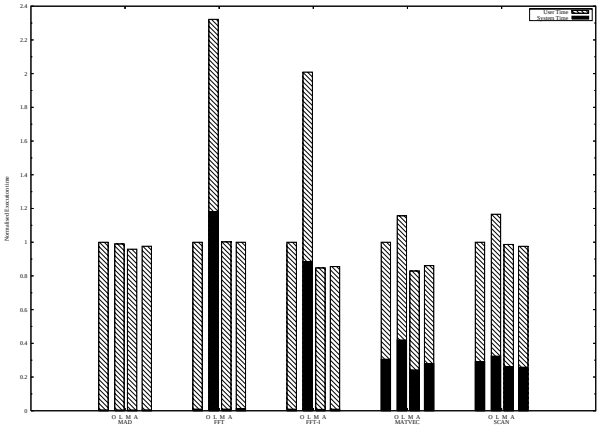


Fig. 4. Execution time performance of benchmarks

adapt this policy to meet the application-specific requirements.

The advantage of this approach is that the resource management policies can evolve and adapt to the changing environment and application requirements at runtime providing the best possible support. The next subsection describes CASP - an application-specific paging mechanism that uses the framework to control the existing page replacement policy.

A. CASP Mechanism

The CASP mechanism acts as a meta-level component to the existing page replacement policy in the OS. By accessing the information pertaining to applications memory accesses, CASP is able to lock and release pages from the global pool. The base-level page replacement policy operates as normal and has no knowledge of the pages that are being locked. CASP achieves this by using the page-isolation technique whereby it completely removes access to a page from any of the OS page-list(s) such that the underlying paging policy does not know about isolated pages and can never reclaim them. Thus, by looking at the applications memory access patterns CASP is able to dynamically lock and release pages into the global page pool such that pages that the application would access always remain in memory.

The reification calls notifying an applications memory usage and access patterns are inserted into the application source code either manually or automatically using the developed tool cloop. Two CASP-specific reification calls keep() and discard() suggest CASP to lock or release particular region in memory [4]. More detailed description of CASP can be found in the full paper [4]. The next section shows some important results of CASP implementation in the Linux 2.6.16 kernel.

IV. Experimental Results of CASP

The CASP mechanism including the framework was implemented in Linux 2.6.16 kernel. Five different benchmark applications: MAD an MPEG decoder, FFT fast fourier transforms, FFT-I inverse of FFT, MATVEC matrix vector multiplication and SCAN a benchmark to stress the virtual memory subsystem to its limits were used in the experiments. For each benchmark application, three versions of the same application were produced: (1) using CASP with manual insertion of reification calls, (2) using CASP with automatic insertion and (3) using manually inserted Linuxs mlock() [5] primitives. Version (3) is the same as (1) except that CASPs keep() and discard() are replaced by Linuxs mlock() and munlock() primitives.

The graphs in fig. 3 and fig. 4 show the number of page-faults and the execution times of all versions of the benchmark applications executed individually in Linux. Each benchmark result shows four bars: the original application (O), the application using Linuxs mlock() primitives (L), the application using CASP with manual hint insertion (M) and the application using CASP with automatic hint insertion (A). A bar is further divided into two parts for fig. 3: the top part shows the number of minor page-faults; the bottom part shows the number of major page-faults and for fig. 4: the top part shows the user-time; the bottom part shows the system-time.

CASP with manual insertion generated 22.3% less major page-faults; improving the execution time by 12.5% while CASP with automatic insertion generated 15.13% less major page-faults; improving execution time by 9% amongst all the benchmark applications when executed individually.

CASP was also tested by executing a combination of two and all benchmark applications. With manual insertion CASP generated 15.38% and 18.06% less major page-faults and improved performance by 28.03% and 12.30% for two and all benchmark applications respectively.

V. Limitations and Further Improvements

Although the RTOS framework allows adaptation of the resource management policies on-the-fly, there is much more work needed to improve the current status. Following lists a few limitations and possible improvements:

- categorisation of information: information pertaining to one resource could also be valuable for another. Thus, a standard approach to categorise and represent information in a generic way is necessary.

- extra memory: it requires additional memory to store reified information in the system. The kernel is burdened with efficient management of this information to keep the memory usage as low as possible adding additional code overhead as well.
- requires explicit reification: the application source needs to explicitly reify resource requirements to the RTOS. The accuracy of reified information is thus largely dependent on the application programmer or the automatic tools used.

Detailed analysis of the memory requirement for the CASP mechanism has shown to use up to 21% additional memory. The CASP mechanism itself helps reduce the resident memory set size of an application by almost 34%. However, the use of memory within the RTOS framework in general needs to be investigated.

Currently, we are looking at more robust analysis tools that would automatically insert reification calls into the applications either in source during compilation or in object code during linking. Embedding reification into the programming language by language extensions is also being looked as a possible solution. The Spring-C [8] language already supports similar constructs.

The *cloop* [4] tool currently developed analyses the application source identifying loop-based access to large memory regions and automatically insert reification calls. Extending the tool for common resource usage in general is a challenging problem. A combination of static and runtime analysis of the application can be used to determine its resource usage. Later, the insertion of corresponding reification calls in the application will help the adaptation of resource management policies at runtime.

VI. Conclusions

This paper described an adaptive reflection-based framework for embedded real-time operating systems to allow runtime adaptations of the resource management polices. The applications and the resource management modules are able to reflect on their performance at runtime, and change or adapt their behaviour dynamically to reflect the current state of the system. A paging mechanism - CASP that makes use of the reflective framework to adapt the paging policy according to the applications memory access patterns is described along with the experimental results. Finally, discussion on the existing limitations of the reflective approach, the current status of our work and the required improvements were presented.

References

[1] A. Bondavalli and J. Stankovic and L. Strigini. Adaptable Fault Tolerance for Real-Time Systems. In *Proceedings of the 3rd International Workshop on Responsive Computer Systems*, September 1993.

[2] Ameet Patil and Neil Audsley. An Application Adaptive Generic Module-based Reflective Framework for Real-time Operating Systems. In *Proceedings of the 25th IEEE Work in Progress session of Real-time Systems Symposium*, Lisbon, Portugal, December 2004.

[3] Ameet Patil and Neil Audsley. Implementing Application-Specific RTOS Policies using Reflection. In *Proceedings of the 11th IEEE Real-time and Embedded Technology and Applications Symposium*, pages 438–447, San Francisco, 2005.

[4] Ameet Patil and Neil Audsley. Efficient Page lock/release mechanism in OS for out-of-core Embedded Applications. In *Proceedings of the 13th IEEE Real-time and Embedded Computing Systems and Applications Symposium*, pages 81–88, Daegu, Korea, August 2007.

[5] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *Linux Kernel Internals*. Addison–Wesley, second edition, 1998.

[6] Brian Cantwell Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, Massachusetts Institute of Technology, January 1982.

[7] R. W. Bryce. *Chameleon, a dynamically extensible and configurable object-oriented operating system*. PhD thesis, Victoria, B.C., Canada, Canada, 2003. Adviser-G. C. Shoja.

[8] D. Niehaus. Program Representation and Translation for Predictable Real-Time Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 43–52, December 1991.

[9] D. Niehaus and J. Stankovic and K. Ramamritham. The Spring System Description Language. Technical Report UMASS TR-93-08, University of Massachusetts Amherst, 1993.

[10] Dawson R. Engler. *The Exokernel Operating System Architecture*. PhD thesis, Massachusetts Institute Of Technology, October 1998.

[11] John A. Stankovic. Reflective Real-Time Systems. Technical Report 93-56, Univeristy of Massachusetts, 1993.

[12] John A. Stankovic and Krithi Ramamritham. The Spring Kernel: a New Paradigm for Real-Time Operating Systems. *SIGOPS Oper. Syst. Rev.*, 23(3):54–71, 1989.

[13] F. Kon, F. Costa, G. Blair, and R. H. Campbell. The case for reflective middleware. *Communications ACM*, 45(6):33–38, 2002.

[14] F. Kon, A. Singhai, R. H. Campbell, D. Carvalho, R. Moore, and F. J. Ballesteros. 2K: A Reflective, Component-Based Operating System for Rapidly Changing Environments. In *ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, Brussels, Belgium, July 1998.

[15] J. Malenfant, M. Jaques, and F.-N. Demers. A Tutorial on Behavioral Reflection and its Implementation. In *Proceedings of the Reflection 96 Conference, Gregor Kiczales, editor, pp. 1-20, San Francisco, California, USA*, April 1996.

[16] M. A. Rivas and M. G. Harbour. Application-defined scheduling in Ada. In *IRTAW '03: Proceedings of the 12th International Workshop on Real-Time Ada*, pages 42–51. ACM Press, 2003.

[17] S. Ruocco. User-level fine-grained adaptive real-time scheduling via temporal reflection. In *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 246–256, Washington, DC, USA, 2006. IEEE Computer Society.

[18] Y. Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, pages 414–434. ACM Press, 1992.

Enhancing the Adaptivity for Multi-Core Embedded Systems with Dynamic Performance Scaling in FPGA

Yan Zhang Gang Quan
Dept. of CSE, Univsity of South Carolina
{zhangy,gquan}@engr.sc.edu

Abstract

Multi-core systems are usually designed to be capable of delivering high peak performance when necessary. However, they need to be adaptive for the sake of improving resource usage efficiency. This is particular critical for the embedded system due to its tightly constrained resources and highly dynamic nature. In this paper, our goal is to design and develop a multi-core embedded platform in FPGA with enhanced adaptivity by allowing the performance of each individual core be dynamically varied. This platform is general and flexible enough to be readily tailored for the purpose of theoretical research and practical design of multi-core type real-time embedded system.

1. Introduction

Recently, the computing industry is switching its gear from pursuing the super high performance single processor toward developing the multiple processor or multi-core architecture to keep up with the increasingly performance demands. The emphasis on multi-processor or multi-core structure in industry changes the computing landscape not only for the traditional high performance computing arenas such as scientific applications but also embedded applications as well [1], as embedded applications continuingly demand more computing power. The high performance computing system needs not only be able to deliver high performance when needed, but should also be adaptive and flexible enough to be capable of accommodating the run-time dynamics to use the resource more efficiently. This is particularly critical for embedded platforms, which usually have tight resource constraints, such as power/energy consumption, and highly dynamic workload.

For embedded systems designed in FPGA, their adaptivity can be enhanced by totally or partially reconfiguring the reconfigurable fabric or connections on the fly in response to the external environment. Today, new generations of FPGA chips such as Virtex 4 can integrate multiple built-in processors in one chip. Further, more software cores such as MicroBlaze [8] can be readily incorporated into the same FPGA chips. This presents an excellent opportunity to enhance the adaptive of multi-core based embedded systems in FPGA by dynamically varying the supply voltages/working frequencies, and thus performances, of these processor cores.

Dynamic voltage/frequency scaling (DVFS) technique has long been recognized as an effective means to improve the efficiency of resource usage, such as power/energy consumption. Traditionally, FGPA chips work under one constant supply voltage and working frequency. The new clock control module (DCM) in today's FPGA makes it very convenient to dynamically adjust the working frequency for functional blocks or processor cores in FPGA [13]. Even though current FGPA technology does not support dynamic voltage scaling (DVS) feature, i.e., a critical feature for saving the power/energy consumption more effectively, we notice that implementing such feature has been put into the agenda by the manufacturer for the development of new generations of FPGA chips [18].

Our goal is to develop a customizable and extensible multi-core platform with dynamically controllable performance in FPGA. One immediate use for this platform is to use it as a testbed to verify the theoretical research results on power-aware computing on multi-core embedded system design. While extensive theoretical research results have been published, these results are often based on idealized models and assumptions. They need further experimental work to validate and evaluate their applicability and effectiveness. While there are some commercial multi-core products available at present, for example, the Sony/Toshiba/IBM Cell Processor combines standard PowerPC core with eight SIMD cores [3] and Cisco is shipping a product with 188 cores on a single chip [16], these products are intended for commercial use rather than as an experimental platform. There are also some evaluation boards that support multiple DSP architecture, such as Tiger-PCI board [17], the scopes and ranges of experiments are limited due to factors such as the fixed architectures, dedicate software environment, and high development cost. We believe that the development of a general, flexible, low-cost, and readily available testbed based on FGPA can greatly encourage researchers to validate their theoretical research results in distributed embedded computing in a more rigorous and effective manner.

In this paper, we present our preliminary work on the design and implementation of a multi-core system with dynamic working frequencies on Virtex 4 using Xilinx FPGA developing tools. In Section 2, we present the general framework of our design, together with some design issues. Section 3 presents a proof-of-concept test case. We conclude the paper in section 4.

2. General Framework

The hardware architecture of our proposed platform is shown in Figure 1. All the design units can be fitted into a single FPGA chip except the external memory. Multiple Processing Elements (PE) are connected with interconnection network. Each PE consists a processor core (MicroBlaze or PowerPC), a small scratch-pad memory, a customized clock control IP, and a customized timer (for MB only since PPC has built-in timer already). The scratch-pad memory is small and fast local memory connected directly to the processor core. It is implemented with the Block RAM on FPGA, with size severely limited, i.e. a Virtex4 FX12 has maximal Block RAM as 72KB. To enable large test programs and data sizes, we divide the external memory into several private memory sections and one shared section. Each private memory section is associated to one processor and can only be accessed by this processor. The shared memory section, on the other hand, can be accessed by all processor cores.

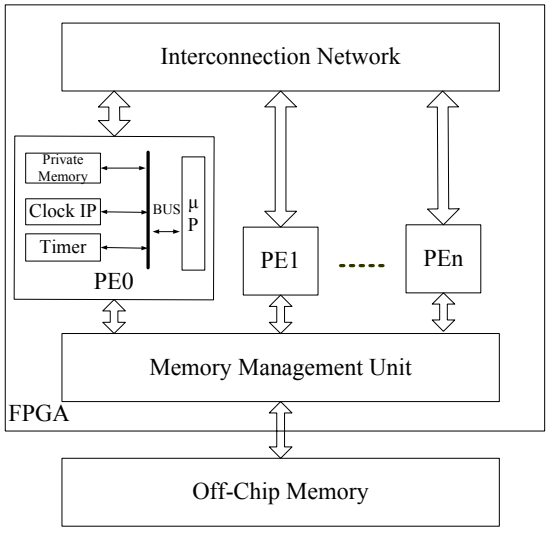


Figure 1. The hardware architecture

To dynamically control the performance of the processors, we built a customized IP (as shown in Figure 2) that can control the clock for each processor core on the fly. Each customized clock control IP consists of a Digital Clock Management (DCM) unit [15-16] and a configuration logic unit. The Xilinx's DCM of Virtex 4 and 5, is a multi-function clock management unit which supports dynamic configuration of clock frequencies ranging from 32MHz-210MHz. Two parameters, M and D, can be programmed into the DCM, and the output frequency is determined by $f_{out}= f_{in}$ *M/D. The valid value of M is 2~32, and 1~32 for D. By setting the input and maximal clock frequency for the PEs as 100MHz, we have approximately 221 different new frequencies available by choosing appropriate M and D values. Such a large number of different frequency scales, the target working frequency output from the theoretical model can be modeled reasonably accurate by finding appropriate M and D values. The dynamic clock can be varied continuously, but the cost --switching overhead --may be significant. The feedback approach is also possible as long as the feedback control takes the switching overhead into account. The bus interface of customized clock unit can be easily configured as OPB or PLB slave.

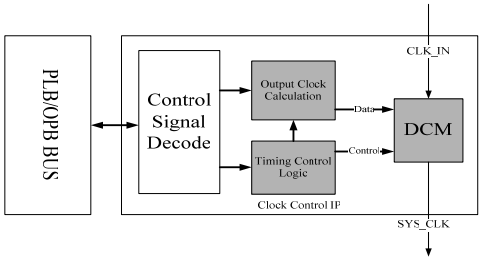


Figure 2. Clock Control Unit

There are number of choices for processor interconnections. The traditional bus connection (for example, using the On-chip Peripheral Bus (OPB)) is possible but less attractive due to the scalability concern. An alternative is to use the logic source in FPGA to create the Network-On-Chip (NOC) infrastructure. For example, Schelle and Grunwald [17] implemented a switching network as interconnection for general purpose processor in a Virtex II-pro device**.** One major disadvantage of this solution is the large amount of resources it requires. Based on our experiment, a 4x4 mesh with 32-bit data width would exceed the capacity of Virtex-II Pro 30 using VHDL-programmed NOC infrastructure.

In our design, we adopt the convenient point-to-point connection mechanism, i.e., the Fast Simplex Link (FSL) bus, provided by Xilinx. FSL is a FIFO-based connection and can be synchronous or asynchronous. An asynchronized communication scheme is particularly useful in our design with different processors running at different speeds. While Xilinx has not officially claimed to support DFS capability for MicroBlaze processor, we have tested FSLs with variable working frequencies. So far, there are no serious impacts on the system correctness. Further investigation on these impacts will be reported. In addition, each core from Xilinx supports multiple FSL buses. For example, a MicroBlaze has up to eight pairs of

FSL to connect up to eight different components for duplex communication, which makes it reasonably easy and effective to build popular multi-core topologies such as the tree, mesh, or torus structure.

3. The proof-of-concept experiment

We implemented the proposed hardware architecture in Figure 1 on Xilinx's test board ML403, which includes a Virtex 4 FX12 FPGA, with package number FF668 and the speed scale 10. Four MicroBlaze-based PEs (PE_0 – PE_3) were implemented on same the FPGA chip and connected with FSL into a simple tree structure, with PE0 as the root node and three others as leaf nodes. The MicroBlaze was configured to the basic, i.e. no float-point unit, cache, integer division, to save the resource usage. A UART is connected to PE_0 for debugging purpose. As the terminal usually fixes the baud rate, a fixed clock is necessary for PE_0. We therefore fix the clock for PE0 at the maximal working frequency. At this stage, we have not been able to make the external memory to function properly for multiple processor cores with different working frequencies. So the FSL is the only path that the data can be moved from one processor to another. The system resource utilization is reported in table II.

TABLE I
SYSTEM HARDWARE UTILIZATION SUMMARY

Device resource	Utilization on Device 4vfx12ff668-10		
	Used	Available	Percentage
Flip Flops	5968	10944	54%
4 input LUTs	7386	10944	67%
DCM	4	4	100%
DSP48s	12	32	37%
Global Clocks	3	32	9%
FIFO16/RAMB16s	32	36	88%

The matrix multiplication was used as our example application for its highly deterministic workload with very little profiling required. In this application, a large matrix is divided into four smaller sub-matrices. Each PE was used to compute one sub-matrix for the final matrix, with the PE_0 also in charge of initialization and result collection. We intentionally divided the matrices un-evenly. So we can deliberately vary the performance of individual PE without compromising the completion time when all PEs always run at their peak performance. In our case study, four different clock frequencies, i.e. 40MHz, 50MHz, 66.7MHz and 100MHz, are provided and we use the two least significant bits of the bus to select the desired frequency. It consumes less than 1% of the slices when synthesized in a Virtex 4vfx12 FPGA. More clock configurations will be implemented and studied in further research

We first investigated several interesting parameters with profiling. This includes the frequency transition timing overhead and the communication speed. The frequency switching overhead, i.e., the time required to change the processor frequency from one to another, was measured by inserting Xilinx's ChipScope Integrated Logic Analyzer (ILA) core into the design and sampling the IP internal signals via JTAG connection. The timing diagram is shown in Figure 3. The clk_in is the input and clk_out is system clock. The lock signal indicates the DCM working status: it goes low when reconfiguration starts, and goes high when stable output is available. Therefore, the interval when lock goes low represents the frequency switching overhead. After sampling the lock signal for all four different frequencies, we found that the switching overhead is around 56μs. Though it's not desirable, we did not find any impacts on the system correctness.

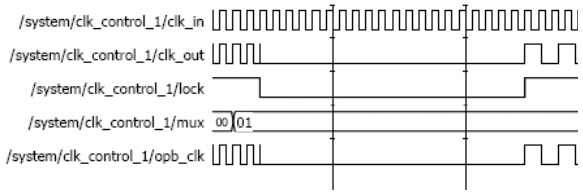


Figure 3. The Switching Overhead for Varying the Working Frequencies

For the communication speed between the PEs, the ideal bandwidth of FSL bus is one word per cycle. However, when we tested in the software environment, the communication cost is much greater. With our customized timer, we measured the interval between the start and end of data transfer and observed that it takes approximately 18.3 cycles to complete the transferring of one word (four byte). In addition, when running processors at different speeds, the cycle length is determined by the one running at the lower speed, which seems to be reasonable.

We tested an 8x8 integer matrix multiplication application on the developed platform. The matrix was divided into 4 sub-matrices with size 6x6, 6x2, 2x6, and 2x2. The profiled execution times (at the highest clock frequency) are shown in the following table.

TABLE II
RUNNING TIME FOR FIXED CLOCK

	PE0	PE1	PE2	PE3
Clock	100MHz	100MHz	100MHz	100MHz
Running time(ns)	75,150	26,360	20,950	8,540

With clock control unit, we can propose the following clocks to reduce the power dissipation and still complete the task in time.

TABLE III
RUNNING TIME FOR PROPOSED CLOCK

	PE0	PE1	PE2	PE3
Proposed clock	100MHz	40MHz	40MHz	40MHz
Running time(ns)	75,150	65,900	52,375	21,350

4. Conclusion and future work

The degree of adaptiveness, i.e., the adapativity, is critical for multi-core system-on-chip architecture, which has the great potential to meet the increasingly demanding performance requirements but also needs to satisfy the stringent resource constraints. In this paper, we discuss our work in the design and development of a multi-core embedded platform in FPGA with enhanced adaptivity by enabling the performance of each individual core be dynamically varied. This platform is general and flexible enough to be readily used for multi-core related research and practical design.

There are a number of directions for our future studies. First, we failed to use the existing memory control IP to access external memory by multiple processor cores running at different clocks. This limit the program and data size for the sample programs that can be validated with this platform. We are currently seeking help from Xilinx technical support for this problem. Second, we need to develop the real-time operating system (RTOS) support for the proposed platform. Currently the processes, communication, memory are managed in a ad hoc way, which will be extremely tedious and error-prone for large application. A reliable RTOS support can significantly facilitate the implementation and test of a large scale real-time multi-core system. Third, the power measurement is another problem that needs to be carefully addressed. Currently Xilinx provides several power measurement tools, such as XPower. It enables more accurate switching activity analysis by combining the register level switching activity file, i.e., the Value Change Dump (VCD) files and the system mapping file, i.e., the Native Circuit Description (NCD) file. It would be desirable that these switching activity analysis results can be used in a more accurate computation of power and energy consumptions. Another problem has to do with the size of VCD file. It takes about half an hour to run 2 ms of simulation and generate the VCD file about 4 GB. Some higher level power analysis tools may be needed to improve the measurement efficiency. Fourth, we are watching closely for the DVS features for the new generations FPGA product. While changing the frequencies helps to vary the performance, this has not transformed to its real benefit, i.e., more effective power/energy conservation.

Acknowledgment

This work is supported in part by NSF under Career Award CNS-0545913 and grant number DUE-0633641.

References

[1] K.Asanovic et al "The Landscape of parallel computing research: a view from Berkeley", http://www.eecs.berkeley.edu/Pubs/TechRpts/2005/EECS-2006-183.pdf

[2] J.Hennessy and D.Patterson, *Computer Architecture: A Quantitative Approach, 4 th edition*, Morgan Kauffman, San Francisco, 2007

[3] Chip Multi Processor Watch http://view.eecs.berkeley.edu/wiki/Chip_Multi_Processor_Watch

[4] M.LaPedus, "To save power, embedded tries multicore" *EE times*, Issue 1481, June 25, 2007 pp. 1-6T.

[5] J.Wawrzynek et al "RAMP: A Research Accelerator for Multiple Processors" http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-158.pdf

[6] J.Becker, M. Huebner, and M. Ullmann, "Power estimation and power measurement of Xilinx Virtex FPGAs: Trade-offs and Limitations" *Proc.16th Symposium on Integrated Circuits and Systems Design*, September 2003,pp. 283- 288

[7] J.Ou, V.Prasanna "Rapid energy estimation of computations on FPGA based Soft Processors" *Proc. IEEE International SOC Conference*, September 2004. pp.285-288.

[8] http://www.xilinx.com/products/design_resources/proc_central/index.htm

[9] http://www.xilinx.com/ise/design_tools/

[10] http://www.xilinx.com/ise/embedded/edk.htm

[11] Xilinx's ML403 Board User Guide http://direct.xilinx.com/bvdocs/userguides/ug080.pdf

[12] Xilinx's Virtex 4 User Guide, http://direct.xilinx.com/bvdocs/userguides/ug070.pdf

[13] R.Krueger, "Dynamic Reconfiguration of functional blocks" *XCell,* Xilinx, Issue 52, March 2005

[14] G.Schelle, D.Grunwald, "Onchip interconnect exploration for multicore processors utilizating FPGAs" *2nd Workshop on Architecture Research using FPGA Platforms*, 2006

[15] A. Wolfe, "Intel Clears Up Post-Tejas Confusion," VARBusiness, May 17, 2004. http://www.varbusiness.com/sections/news/breakingnews.jhtml?articleId=18842588

[16] W. Eatherton, "The Push of Network Processing to the Top of the Pyramid," keynote address at *Symposium on Architectures for Networking and Communications Systems*, Oct. 26–28, 2005.

[17] Tiger-PCI Quad ADSP-TS101 TigerSHARC PCI board http://www.bittware.com/products/boards/prod_desc.cfm?ProdShrtName=TSPC

[18] Nunez-Yanez et al "Dynamic Voltage Scaling in a FPGA-Based System-on-Chip" *FPL 2007*

2. Distributed Systems

Building Adaptive Embedded Systems by Monitoring and Dynamic Loading of Application Modules

Florian Kluge, Jörg Mische, Sascha Uhrig, Theo Ungerer
Department of Computer Science - University of Augsburg
86159 Augsburg, Germany
{kluge, mische, uhrig, ungerer}@informatik.uni-augsburg.de

***Abstract*— Networks of embedded systems usually lack flexibility due to the static configuration of their nodes. In this paper, we present a concept to adapt distributed embedded real-time environments at runtime using code relocation techniques. The main objectives of this work are to improve the availability of services within such environments and to increase the flexibility of the whole distributed system. The decision about reconfiguration is based on an extensive monitoring of the hardware and software system.**

I. Introduction

Today, the number of embedded systems is constantly growing and establishing distributed systems of *Embedded Control Units* (ECUs). For example, a car can contain over 70 ECUs fulfilling most different duties. Usually these ECUs are statically configured for specific tasks. This leads to the deployment of many different but highly specialised ECUs within such a system. Thus, the complete system has a high susceptibility for failures. Simultaneously, its adaptability is restricted to the abilities of the single ECUs.

With an intelligent management that supports the relocation of applications between processing nodes, it would be possible to decrease the number of ECUs in such a system, while the dependability of the whole system would be increased. Also, adaptability techniques were no longer restricted to the capabilities of single nodes, but could exploit the whole distributed system. Through the dynamic adaptability it would be possible to switch off single nodes to save energy. In cars this would directly decrease the CO_2 emissions.

Such a management uses the runtime information of the single nodes to adapt the whole system to the actual requirements. Applications that will probably fail operation in the near future due to resource constraints (e.g. low battery, low memory, high CPU load) should be moved to other nodes for proper operation. If hardware devices attached to a processing node fail operation, it might be possible to substitute their functionality by other devices. Two precondition exist for this approach:

1) an extensive monitoring of hard- and software parameters and
2) a mechanism to relocate applications between nodes.

In this paper we describe how we employ the system monitoring facilities of CAROS (**C**onnective **A**utonomic **R**eal-time **O**perating **S**ystem)[1] to detect or predict future bottlenecks of resources and possibilities for system optimization. With the help of a runtime linker the system is adapted to these new conditions and still able to keep maximum performance.

CAROS is part of the CAR-SoC (**C**onnective **A**utonomic **R**eal-time **S**ystem-**o**n-**C**hip) project [2]. The base of this project is the CarCore processor, which is binary compatible to the Infineon TriCore architecture, a well-known automotive processor. However, it is extended by the capability of simultaneous multithreading and implements a real-time capable scheduling technique [1] based on the Guaranteed Percentage scheduling [3]. Thereby, a thread gets a specific share of the processing time which can be guaranteed for real-time threads over a recurring period of time. The CarCore processor is currently available as a SystemC simulation model. The real-time operating system CAROS provides a comfortable interface for application development.

We draw special benefit from the scheduling technique of the CarCore processor which allows us to run so-called helper threads in parallel to real-time applications without disturbing their timing behaviour. “Helper thread” thereby is a term used for threads running in parallel to the actual application and supporting its operation, but they are not required for proper operation.

This paper is organized as follows. Section II presents related work. In section III we describe how applications and hardware core components are monitored and which adaptations are possible for specific system states. Our framework for the migration of applications is presented in section IV. In section V we present some performance measurements of our implementation. Section VI concludes this paper and gives a short overview of future prospects.

II. Related Work

The concept of code migration is well-known from the area of mobile agents. The MESSENGER project [4] provided a virtual machine for agents written in a special language. It was ported to the Java virtual machine as JMessengers [5], [6], now using common Java code for the framework and as well for

the agents. Another mobile agent system was developed in the ARA project [7]. Agents for this system are written in TCL or C/C++. This code is compiled into the MACE bytecode representation [8] which is interpreted at runtime.

The concept of mobile agents is similar to our aim of "mobile applications" with two main differences. Mobile agents are active software entities, which decide themselves if they want to migrate. Thus, the agents have an individual responsibility to fulfill their tasks. In our approach the runtime environment makes the decision about migration and also takes over the responsibility of the execution of the tasks. Furthermore, the presented systems use managed languages to implement the agents. In contrast, for reasons of efficiency we want to migrate native binary code.

AUTOSAR [9] is the upcoming industrial standard for automotive software. It aims to provide software developers with more flexibility in the development cycle. However, runtime reconfiguration is not yet integrated into the standard, so an AUTOSAR ECU is still statically configured. But there are several approaches to overcome this limitation [10], [11]. Perhaps future versions of AUTOSAR will also target the problems of reconfiguration and adaptability in more detail.

The dynamic loading and linking of code is also a common concept in modern operating systems (e.g. *Shared Objects* under Unix/Linux or *DLL*s under Windows). However, these operating systems run usually on high-performance machines whereas we target the area of embedded systems with all their restrictions. Also, the dynamic loading/linking in common desktop operating systems is targeting other objectives, i.e. the avoidance of code replications for commonly used libraries.

III. Application Monitoring

CAROS offers a wide range of monitoring points and also the convenience to add user- or hardware-defined monitoring points. Through the analysis of the monitoring data, the OS is able to detect upcoming nuisances. In such a case, apt countermeasures must be taken.

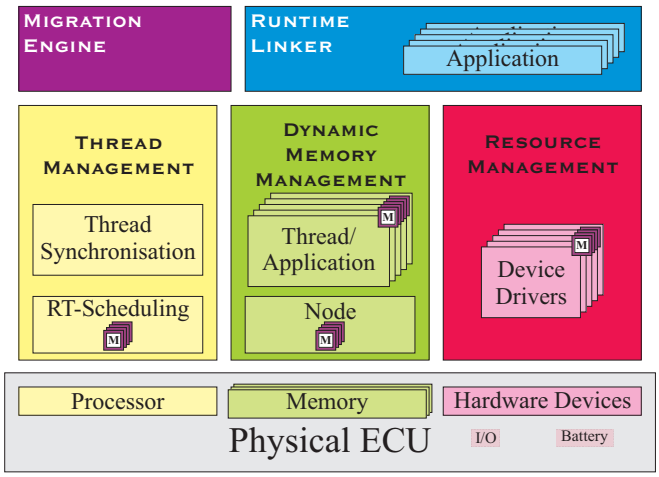


Fig. 1. Architecture of CAROS; note the monitoring points (M)

Figure 1 shows the architecture of CAROS. The OS core is built from the three parts *Thread Management*, *Memory Management* and *Resource Management*. Each of these parts is equipped with monitoring points (marked with *M*), which provide accurate information about the current system state. The *Migration Engine* and the *Runtime Linker* are located on top of these modules. The migration engine evaluates the monitoring data and decides about the relocation of applications. If it receives an application from another host, it uses the runtime linker to link the application into the running system.

In the following paragraphs, we will describe the monitors in more detail. The migration engine and the runtime linker will be detailed in the next section.

A. Processing Time

Due to the special scheduling technique of the CarCore, the OS only has to ensure that the sum of all real-time threads' processing time does not exceed 100%. The timing behaviour of the applications is guaranteed by the hardware scheduling technique. Thus, the load of the processor can be determined exactly at any time, supporting decisions about applications' relocations (sending and receiving). Especially techniques for *Dynamic Voltage and Frequency Scaling* (DVFS) are supported, as each scaling of frequency also needs an adaption of the application's share of processing time to preserve their timing behaviour.

B. Memory

The memory management of CAROS is split into two levels of allocation, the *Node* and the *Thread Level*. The Node Level is able to simultaneously manage several types of memory within one node. It allocates huge portions of memory to the threads. The Thread Level uses these portions to allocate memory to the applications with a predictable timing behaviour because of its exclusive access.

The memory types on the node level usually are distinguished by access time and energy consumption. Applications are attached to one type of memory. So, applications that need low latencies and fast response times would be put into a fast memory bank at the cost of a higher energy consumption. The memory management not only provides information about the overall memory consumption, but also about the each application's share of memory. If a memory bank impends to be fully occupied, the OS can detect the responsible application. Negotiations with other nodes can lead to the relocation of an application onto another node now more apt, i.e. having more available memory.

C. Energy - Battery

One of the most important monitors watches the battery. If the charge level drops below a certain threshold, it is necessary to trigger actions that do not only minimise the energy consumption of the node, but also help to ensure continuing proper operation of the whole system. This leads to close interaction with other monitors for memory and processing time. To save energy it is possible to

- switch off memory banks with high power consumption,
- decrease the processor's frequency and voltage,
- combine both of the above, or
- switch off the complete node, in the worst case.

On the other hand, a node is able to realize if its power supply will suffice for a long time. In such a case it can offer to take over the applications of failing nodes.

D. Other Hardware Resources

Most hardware resources are accessed through the resource management and device drivers. CAROS urges these drivers to provide at least minimum information about availability and state of the devices. The drivers may also provide more detailed information that can be used for relocation decisions. In fact, the energy monitoring for batteries is just another device driver that merely consists of the monitoring functions.

If a specific hardware device fails operation, a service depending on it may possibly be restarted on another node that can substitute the hardware functionality.

IV. The Migration Framework

In this section, we present the OS components that are related to the relocation of applications.

A. Application Development

We use the object-files (`.o`) generated by the Hightec `tricore-gcc` [12] as application images. Hence, we have developed a framework that ensures that these files have a certain layout. Additionally to the application itself, the developer has to provide the following:

- functions for un-/packing of the application's data: we give the application developer full control over the migration of the application data. The developer knows best, what data is necessary for a successful relocation and so also only the most necessary data is transmitted. Thus, the usually rather restricted communication network bandwidth is saved. The functions for un-/packing the application data are integrated into the object file, because they depend only on the application code.
- memory and processing time constraints of the application: these will be determined through a separate WCET analysis of the application. They are necessary for the relocation decision and to keep the timing behaviour of the application. Such information is kept separate from the object file, because it may vary for different system configurations and/or QoS levels.

Thus, a relocatable application consists of the "raw image" produced by the compiler and of the additional timing and memory information. The raw image cannot be executed directly, instead it must be linked into the running system by the *Runtime Linker*, which we describe next.

B. The Migration Engine and the Runtime Linker

In addition to the *Raw Image* and the *Application Constraints*, a relocatable applications also comprises the *Migration Data*. Here the application's runtime data is stored for migration.

Based on the monitoring data (section III), the migration engine decides about the acceptance respectively relocation of applications. The decision itself is made by a middleware which is not part of this work.

If a node accepts an application from another node, the Migration Engine receives a data packet from the application's source node. It disassembles the received data packet into the raw image, the application data, and the application constraints. The raw image is passed to the Runtime Linker, which creates the *Process Image*. If necessary, the memory constraints defined for the application are taken into account. Dependencies to symbols (variables and functions, e.g. calls to OS services, but also function calls within the application module) are resolved at this point ("code relocations"). Also, the application's data memory is prepared. After the linking process, the application data received along with the application image is passed to the application's `unpack` function. The scheduling parameters for the thread slot containing the application are set according to the given timing constraints. Now the application is ready to run. The raw image of the application is kept in the migration engine for later relocations to other nodes.

If a relocation is decided, the migration engine stops the application, its data is packed using the application's `pack` function and then it is bundled together with the raw image and the constraint data and sent to the target node. After it has been linked successfully, the local process image is destroyed and the corresponding thread slot is released.

C. Real-Time Considerations

The timing behaviour of the whole relocation process is subject to several constraints. The decision process and the transmission of the application to another node may be real-time capable, if the decision algorithm and the communication network are real-time capable. The timing behaviour of the un-/packing of the application's data is completely at the responsibility of the developer.

The timing of the linking process itself is a bit more difficult. Timing problems may arise from the necessary code relocation. In particular, we must distinguish the following code relocation types:

- **Internal**: The time to resolve symbols within an application module only depends on the application's code itself. It is constant over all nodes and is known in advance.
- **Kernel**: OS services usually are accessed through `system calls`, i.e. trap routines. These trap calls are resolved by the processor, so no code relocation is necessary. If OS services are accessed through normal function calls instead, code relocations are necessary. However, as each node runs the same OS, the access tables for the OS services are everywhere the same, and thus the time for

the code relocation is also the same for an application on every node.

- **External**: References to symbols defined in other dynamically loaded modules present a problem. The symbols usually are stored in dynamic data structures like hashtables, which cannot give timing guarantees.

Hence, if an application is restricted to the use of internal and kernel symbols, the migration time is predictable. Therefore, the linker should be run as a real-time helper-thread with a guaranteed share of processing time. If references to other modules exist, it is very hard to make any timing guarantees. This is the case especially if several modules are accessed through the same hashtable. Here the access times might depend on the order these modules were linked.

Also, it is possible to do the linking in the background on the target node while the application is still running on the source node. Just when the linking process is finished, the application is stopped and its data is migrated. Thus, the timing critical region is kept much shorter and more predictable.

V. Runtime Evaluation

We carried out some measurements regarding the timing behaviour of several parts of the migration process (re-/creation of the image, dispatching...). We used a simple counter program (CNT) and a PID controller (PID) as example programs. The measurement results are shown in table I.

TABLE I
RUNTIMES FOR APPLICATION MIGRATION (CLOCK CYCLES)

	CNT	PID
Create Image	98,647	180,008
Dispatch Application	22,305	52,996
Re-create Image	98,787	180,205
`pack_data`	36	176
Image Size (bytes)	1,576	3,700
Data Size (bytes)	4	60

We show the runtime measured in clock cycles, running the migration engine with maximum performance. The most expensive part is the (re-)creation of the process image. So, for the PID controller the performance cost of a migration is about 230,000 clock cycles. Assuming a clock frequency of 100 MHz, this would take 2.3 ms. However, usually the linker will be run only in the background and thus the linking will take some more time. Through our scheduling technique the user has full control over the linker's timing behaviour.

We did not take the time for the transmission of the application image into account, as it depends on the kind of communication network that is used.

VI. Conclusion

A. Contributions

With the presented relocation concepts, we improve the energy efficiency of distributed embedded systems. Through a uniform distribution of the applications over the nodes, the power consumption of each node is kept as low as possible. Simultaneously, the availability of services is increased, because they are not bound to one node, but can be moved between nodes.

Furthermore, we improve the flexibility of such embedded systems. Hardware units can be built more generic, saving development costs. The specialization of the units is done completely by the software at runtime by configuring the single ECUs adaptively towards the necessary tasks. Also, the reconfiguration of ECUs at runtime is possible allowing a more efficient use of the hardware resources.

Real-time operation is not influenced at all due to the used scheduling technique. On the contrary, the migration framework can be run under real-time requirements and thus will back the real-time behaviour of the complete system.

B. Future Work

Currently, applications themselves must take care of saving and restoring their state. In the future, we aim to extend our migration framework such that most of this work will be done by the migration engine. Thus, the application developer will be relieved from the work of de-/serialization of application data.

References

[1] F. Kluge, J. Mische, S. Metzlaff, S. Uhrig, and T. Ungerer, "Integration of Hard Real-Time and Organic Computing," in *ACACES 2007 Poster Abstracts*, (L'Aquila, Italy), Academia Press, Ghent (Belgium), Jul 2007.

[2] S. Uhrig, S. Maier, and T. Ungerer, "Toward a Processor Core for Real-time Capable Autonomic Systems," in *Proceedings of the 5th IEEE International Symposium on Signal Processing and Information Technology*, pp. 19–22, Dec. 2005.

[3] J. Kreuzinger, A. Schulz, M. Pfeffer, T. Ungerer, U. Brinkschulte, and C. Krakowski, "Real-time Scheduling on Multithreaded Processors," in *7th Int. Conference on Real-Time Computing Systems and Applications*, pp. 155–159, Dec. 2000.

[4] L. F. Bic, M. Fukuda, and M. B. Dillencourt, "Distributed Computing Using Autonomous Objects," in *IEEE Computer*, pp. 55–61, Aug 1996.

[5] M. Gmelin, J. Kreuzinger, M. Pfeffer, and T. Ungerer, "Agent-based Distributed Computing with JMessengers," in *I2CS Innovative Internet Computing Systems*, pp. 131–145, Feb. 2001.

[6] U. Wolf, J. Kreuzinger, and T. Ungerer, "Synchronisation im JMessengers Agentensystem," in *PARS-Workshop, München*, pp. 87–96, Okt. 2001.

[7] H. Peine and T. Stolpmann, "The Architecture of the Ara Platform for Mobile Agents," in *First International Workshop on Mobile Agents MA'97* (R. Popescu-Zeletin and K. Rothermel, eds.), vol. 1219 of *Lecture Notes in Computer Science*, (Berlin, Germany), pp. 50–61, Springer Verlag, Apr. 1997.

[8] T. Stolpmann, "MACE (Mobile Agent Code Environment) - Eine abstrakte Maschine als Basis mobiler Anwendungen," Diploma Thesis, Universität Kaiserslautern, Aug 1995.

[9] "AUTOSAR AUTomotive Open System ARchitecture." http://www.autosar.org/.

[10] R. Anthony, A. Rettberg, D.-J. Chen, I. Jahnich, G. de Boer, and C. Ekelin, "Towards a Dynamically Reconfigurable Automotive Control System Architecture," in *Embedded System Design: Topics, Techniques and Trends, IFIP TC10 Working Conference: International Embedded Systems Symposium (IESS), May 30 - June 1, 2007, Irvine, CA, USA* (A. Rettberg, M. C. Zanella, R. Dömer, A. Gerstlauer, and F.-J. Rammig, eds.), pp. 71–84, Springer, 2007.

[11] W. Trumler, M. Helbig, A. Pietzowski, B. Satzger, and T. Ungerer, "Self-Configuration and Self-Healing in AUTOSAR," in *14th Asia Pacific Automotive Engineering Conference*, (Hollywood, California, USA), SAE International, Aug 2007.

[12] HighTec EDV-Systeme GmbH, "Website." http://www.hightec-rt.com/.

A Programmable Arbitration Layer for Adaptive Real-Time Systems

Sebastian Fischmeister
Dep. of Electrical and Computer Engineering
University of Waterloo, Canada
sfischme@uwaterloo.ca

Robert Trausmuth
Department of Computer Science
University of Applied Sciences
Wiener Neustadt, Austria
trausmuth@fhwn.ac.at

***Abstract*— Adaptive real-time systems can respond to changes in the environment and thus allow for an extended range of operations and for improved efficiency in the use of system resources. Building such adaptive real-time systems requires flexibility at each layer in the system stack. In this paper, we introduce and discuss our on-going effort to build a programmable arbitration layer. It builds on the Network Code language and enables the developer to program application-specific arbitration mechanisms which optimize bandwidth or encodes specific properties such as data redundancy, collision-free communication, and temporal isolation.**

I. Introduction

Adaptive systems can respond to environmental changes including hardware/software defects, resource changes, and non-continual feature usage [1]. As such, adaptive systems can extend the area of operations and improve efficiency [15] in the use of system resources. However, adaptability also incurs overhead in terms of system complexity and resource requirements. Consequently, adaptive systems must be diligently planned, designed, analyzed, and built to find the right tradeoffs between too much and too little flexibility.

Building such adaptive real-time systems requires flexibility at each layer in the system stack. In this paper, we concentrate on the communication system and the arbitration layer. Our notion of flexibility in this context means that the developer can tailor the arbitration mechanism to the specific needs of the application.

Previous work [8] introduced a software layer, which allows developers to program application-specific arbitration mechanisms. This software layer can run inside the network-card driver, on top of an existing hardware arbitration algorithm such as the Controller Area Network [4], or on top of the operating system in the application software. However, wherever it runs, it requires expensive computational resources [12] and causes jitter (see Section III). So the question is: Can we build a programmable (i.e., flexible) arbitration layer with restricted capabilities so it provides comparable performance but with sufficient generality so it allows building adaptive systems?

This work breaks this question into several pieces and addresses them individually. First, Section II discusses what such a programmable arbitration layer would look like and builds on previous work [8]. It describes the model for managing message queues and access to the communication medium. Second, Section III shows a head-to-head comparison of the system, one implemented on top of the network card and one implemented in programmable hardware. This provides insight into the jitter introduced by a kernel, albeit the relative low level (i.e., inside the kernel and the network driver) of the software layer. Finally, Section IV examines the last part of the question and explains how the developer can use this programmable arbitration layer to encode different communication phases typically present in adaptive real-time systems.

II. System Overview

The objective of the proposed system is to provide a programmable arbitration layer for adaptive real-time systems. This arbitration layer must meet the following requirements:

- *Real-time Capabilities:* A communication system for a real-time system requires a real-time capable arbitration mechanism located in the data-link layer of the communication stack. The 'real-time capable' arbitration mechanism guarantees non-trivial upper bounds on the transmission delays of individual message requests. For example, randomized arbitration provides no such bounds while a simple round-robin method does.
- *Medium Access Control:* Access control grants write access for the medium to nodes. The arbitration layer must provide the single interface through which nodes access the network, otherwise, the systems cannot prevent interference and provide temporal guarantees.
- *Queue Management:* In complex systems, multiple tasks concurrently communicate with each other in multiple sessions. The arbitration layer must provide ports and queues to differentiate among sessions.
- *Quality of service (QoS):* Individual traffic has varying importance to the system. QoS functionality assures that the

developer can adjust the arbitration layer to the application demands.

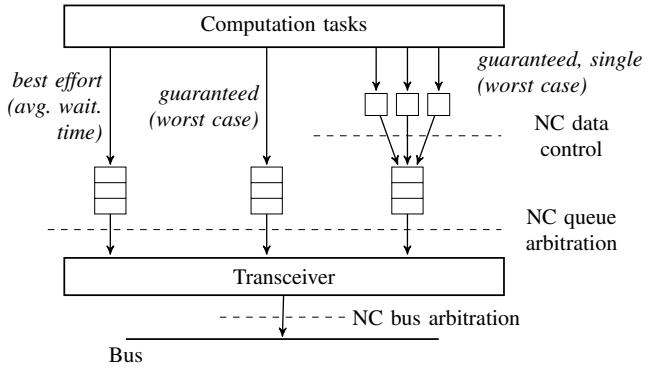


Fig. 1. Overview of the queues and controls.

For matters of programmability, we chose the Network Code language [8], [6] for the arbitration layer. Network Code represents a domain-specific language for programming communication schedules and arbitration mechanisms for real-time communication. Network Code programs of a certain structure remain verifiable [8], analyzeable [2], and composable [3].

Network Code provides two distinct types of QoS: best effort and guaranteed. Messages sent using the *best effort* quality class have no bounded communication delay, as the transmission can fail infinitely often for various reasons including getting blocked by guaranteed traffic or collisions. Messages sent using the *guaranteed* quality class have bounded communication delays. We can apply static verification [8] and analysis [2] to compute bounds as long as the traffic follows a predefined temporal pattern.

Network Code also provides data control functionality for buffers. This functionality allows the developer to create messages from these buffers and transmit them on the network. The developer can use this to replicate buffers across multiple nodes following a specific temporal pattern. For example, given that a specific buffer holds the sensor readings: The developer can write a Network Code program that transmits the sensor readings to all nodes every ten milliseconds. Replicated buffers can act as input for control-flow decisions in the program. The conditional branching instruction *if()* allows the developer to code alternatives. So for example, if the last sensor reading lies below a threshold, then the sensor will suspend the updates for some time.

Figure 1 shows an overview of the programmable arbitration layer used to implement Network Code, and how it interacts with the queues and the computation tasks. For further details, see the language specification [6] and the prototype software implementation [8].

III. Rationals and System Design

The previous section provided an overview of Network Code and the programmable arbitration layer. However, it leaves open the question whether we can build such a system with comparable performance to the original arbitration mechanism of the particular medium.

Previous work [8] demonstrates and measures a software implementation of this system using 100Mbit/s Ethernet. The measurements show that the software implementation provides less throughput by a factor of 4.23 compared to the theoretic limit and 3.52 compared to the empirically evaluated limit. Interrupts and variance in the execution times across the system stack causes jitter which leads to these throughput reductions. Recent jitter measurements support this reasoning. For example, Figure 2 shows a box plot for the execution time of the *send()* instruction. While the mean lies significantly below 500ns, sometimes, the execution time of the same instruction increases by two orders of magnitude. This forces developers into using very conservative estimates when calculating communication bounds.

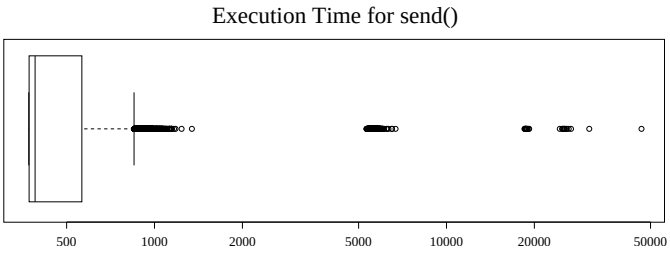


Fig. 2. Execution times for sending [ns].

To minimize jitter and system influences, we reimplemented the system using programmable hardware. Historically, initial work on the topic of real-time communication proposed customized hardware [5], [13], [14] that provided guarantees used for the analysis. At the time of that research, custom hardware cost too much to manufacture and to allow reasonable experiments. However, recent advances in FPGAs allow us now to create custom hardware at a cheap price. In our case, we built an application-specific processor that resides directly above the layer 1 of the networking stack. Figure 3 provides an overview of the FPGA architecture and shows how we implemented the individual language primitives with their control and data flow. For sake of brevity and focus, we skip details of the implementation.

We implemented the system on a XILINX Virtex IV FX12 running at 100 MHz. The clock resolution of the Network Code processor equals 10 microseconds. Given that transmitting merely the Ethernet packet with payload requires approximately 8 microseconds to transmit, we can recreate any raw Ethernet traffic with a timing resolution of 10 microseconds.

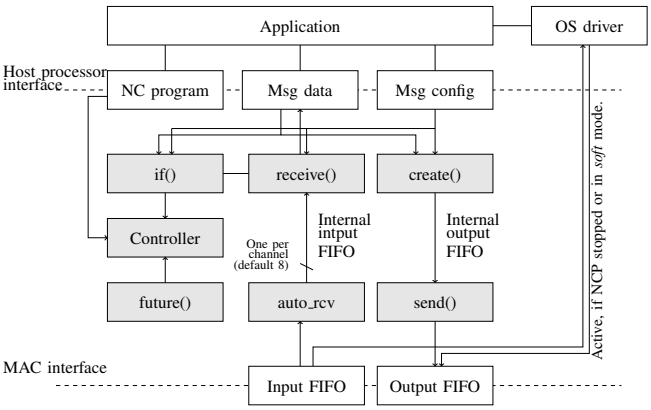


Fig. 3. Block diagram of the Network Code Processor.

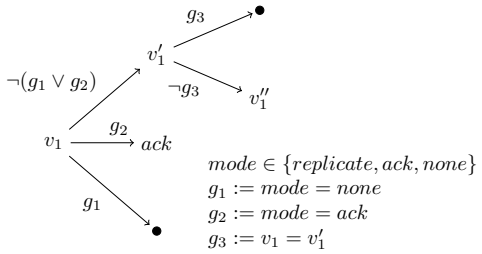


Fig. 4. Communication schedule with branches.

IV. Communication Phases

Real-time systems usually fulfill a specific purpose. This purpose comprises high-level goals such as maintaining stability of a plant in various circumstances. To achieve its goals, the system implements different operation modes, which provide functionality for solving problems within well-defined scopes.

Application modes form one class of modes which provides functionality for application-specific needs. One application can consists of multiple modes; For example, an airplane control system implements application modes for taking off, cruising, and landing (see [11]). In adaptive systems, each application mode may comprise several sub-modes of which each offers similar functionality with a varying level of quality in the result [1].

Service modes make another class of modes. Service modes provide no application-specific functionality, but offer basic services typically needed in a distributed real-time application. Examples of such modes include a membership service, a reconfiguration service, and debugging facilities.

In the following, we show how Network Code allows us to realize the communication behavior of these different modes. Although we realize only one particular approach for each, this exercise shows how Network Code can express the communication schedule of adaptive real-time systems.

A. Application Modes

Application modes realize application-specific functionality within well-defined scopes. In terms of communication behavior, these modes contain mixes of guaranteed and best-effort communication. Guaranteed communication usually follows a predefined temporal pattern (strictly periodic or sporadic) and its temporal behavior is subject to worst-case analysis. Best-effort communication usually shows aperiodic arrival patterns and augments the core real-time functionality of the system.

Network Code allows for both types of traffic, and encodes it in different system modes. Figure 1 already indicates, that the proposed system uses different queues for best-effort and guaranteed traffic. Network Code provides primitives for controlling, transmitting from, and receiving into these queues. Guaranteed traffic uses the *protected mode* (also called hard mode). In this mode, all transmissions originate from the guaranteed queues, which the computational tasks can only access using *send()* and *receive()* instructions. Best-effort traffic uses the *unprotected* mode (also called soft mode). This mode hands off control from Network Code-based arbitration to the native arbitration of the communication medium.

1) Guaranteed Traffic: Systems such as TTP[10], FTT-CAN [7], and TTCAN [9] demonstrate that table-based arbitration provides real-time capabilities and can deliver bounded communication delays. Network Code can express any arbitrary table-based arbitration specification and more. Network Code can unify multiple sub-modes encoding different QoS levels into one arbitration mechanism (as in contrast to providing multiple tables or recomputing the schedule on the fly). The conditional branching function provided in the language enables the developer to specify the communication behavior of different application modes in one program.

Consider the following example: in a distributed control system, one sensor periodically reports its sensor readings (variable v_1) on a shared medium. The system offers different levels of QoS for reliability. Figure 4 provides a visual representation of the resulting schedule for this example. Each vertex represents a transmission of a particular value. Edges specify possible transitions between vertexes, and guards on edges enable and disable these edges. Specifically in this example, the developer offers three levels of QoS: unreliable, receipt acknowledgement, and temporal replication. Each round, the sensor communicates its reading v_1. All further communication depends on the current selected QoS for reliability. Consider that the system in this run requires a high degree of reliability and uses temporal replication with triple modular redundancy. In this mode, the guard $\neg(g_1 \vee g_2)$ holds and enables the edge from v_1 to v_1'. Depending on whether the contents of v_1 equals v_1', the guard g_3 will enable or disable the edge connecting v_1' to v_1''. Let's assume that $v_1 = v_1'$: after transmitting v_1', the schedule proceeds to an empty vertex, which represents an empty slot.

2) Best-Effort Traffic: The developer can use the unprotected mode and best-effort traffic in two ways: exclusive medium access and shared access. During *exlcusive medium access,* at most one node uses the unprotected mode while the other nodes remain in the protected mode *and* schedule no transmission. This guarantees a single node exclusive access to the network and it can transmit without interference from other nodes. During *shared medium access,* more than one node uses the unprotected mode. So, nodes can interfere with each other as they try to transmit simultaneously. The communication delay analysis [2] differs for both methods.

Listings 1 and 2 demonstrate the use of these two concepts. Note, that we use one tick as safety gap between mode changes. For the first five ticks (0-5), Node 1 has exclusive access to the medium in the unprotected mode. For the next five ticks (5-10), Node 2 has exclusive access. Finally, for the last five ticks, both nodes run in the unprotected mode and access the network concurrently.

```
1 L0: mode(unprotected)
      wait(4)
3     mode(protected)
      wait(6)
5     mode(unprotected)
      future(5, L0)
7     halt()
```

Listing 1. Node 1.

```
1 L1: mode(protected)
      wait(5)
3     mode(unprotected)
      future(10, L1)
5     halt()
```

Listing 2. Node 2.

B. Membership Services

Dynamic and dependable real-time systems usually provide membership information to a central controller. This information includes data about present nodes and their status and allows the controller to attest the general state of the running application. A membership service provides such functionality and typically consists of three sub-services: a joining, a heartbeat, and a leaving mechanism.

The joining mechanism enables new nodes to announce their presence and join the application. The system start-up phase requires such a joining mechanism, because one node after the other will join the application as they become online. There exist different approaches to realizing such a mechanism and Listings 3 and 4 show an unsophisticated but working one. This example uses two distinct channel identifiers. The inviting node transmits an invite message on channel zero and new nodes reply to this message on channel one. Upon receiving the invite message, a new node has ten ticks to reply. As Listing 4 shows, a new node initially stays in a tight loop until it receives an invite packet on channel zero whereto it responds. The inviting node receives all replies into a buffer called *InviteReplies.* A computation task at the inviting node then processes the contents of this buffer and integrates the new node into the application.

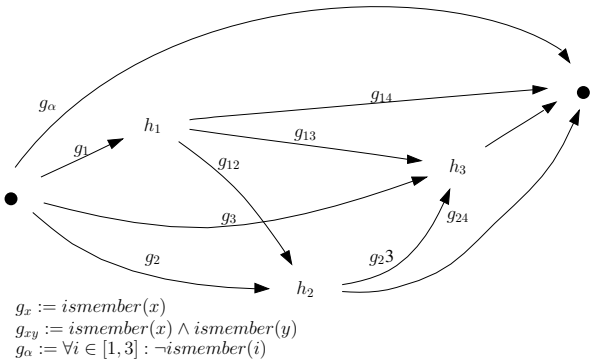


Fig. 5. Heartbeat with three registered nodes.

```
1 L0: mode(hard)
      xsend(0, 0, invite, 10)
3     wait(11)
  L1: if ((EMPTYCHANNEL 1), L2)
5     receive(1,InviteReplies)
  L2: nop()
```

Listing 3. Join request by the inviting node.

```
L5: if ((EMPTYCHANNEL 0), L5) // tight loop
2     xsend(1, 0, myID, 10)
      wait(10)
```

Listing 4. A node waiting for the invite.

A heartbeat mechanism allows a controller to keep an up-to-date list of joined nodes and their status. In some systems, nodes piggyback the heartbeat on data messages or use frame packing, whereas others maintain a specific periodically-transmitted heartbeat message. Network Code can encode both scenarios, because the heartbeat message has equivalent properties as guaranteed application-specific traffic (see Section IV-A). Figure 5 shows a schedule that transmits the heartbeat messages h_x as necessary. For example, if node n_2 has not yet registered with the application, then g_2 and g_{12} do not hold and the schedule will not consider the heartbeat message h_2.

The leaving mechanism allows nodes to gracefully exit the current application. Similarly to the heartbeat message, the

leaving mechanism requires no special attention, because it has similar properties as the guaranteed traffic for registered nodes.

C. Reconfiguration Operation

Adaptive systems usually require means for reconfiguration [1]. System functionality such as admission control and schedule computation happen inside the computation layers and are out of the scope of this paper. Here, we only investigate the communication needs involved in a reconfiguration mechanism. Section IV-A already discussed one form of reconfiguration: how the developer can program different QoS levels for guaranteed traffic. This section demonstrates, how Network Code allows the developer to realize the communication part of a reconfiguration mechanism beyond QoS changes.

The event chain for such a reconfiguration mechanism comprises: the change request, the change reply, the new schedule distribution, and the new schedule implementation. The concrete mechanism depends on the requirements; our example uses best-effort for change requests and guaranteed traffic for the other elements. Listing 5 shows the Network Code program for the controller. Nodes transmit change requests during the best effort period in the first four ticks. A computational task processes these packets (either immediately or at some later point), and eventually sets the variable *NEWSCHEDULE* to one. After the best-effort section, the controller communicates the system status. This status includes the *NEWSCHEDULE* variable and updates its value on all nodes. If a new schedule becomes available, the controller continues at label *L1* and distributes the new schedule.

```
1  L0: mode(unprotected)
       wait(4)
3      mode(protected)
       wait(1)
5      xsend(0,0,STATUS, 2)
       wait(2)
7      if ( (= NEWSCHEDULE 1 ), L1)
       // proceed with normal operation
9      ...
       goto(L2)
11 L1: xsend(0,0,NEWSCHEDULEDATA,10)
       wait(10)
13 L2: nop()
```

Listing 5. Controller (reconfiguration)

V. Conclusions

Adaptive real-time systems can respond to changes in the environment and thus allow for an extended range of operations and for improved efficiency in the use of system resources. An important element of such an adaptive system is an expressive, freely programmable arbitration layer. Such a layer allows the developer to tailor the arbitration mechanism and its level of adaptivity to the application needs.

Network Code provides a programming model for stateful communication schedules and will enable a programmable arbitration layer for adaptive systems if: (1) the resulting system can provide sufficient throughput and (2) the language can express elements commonly used in adaptive distributed real-time systems. In this work, we demonstrate that we have achieved both goals. The implemented FPGA prototype provides a resolution of ten microseconds and provides comparable speed to raw 100MBit/s Ethernet. The presented programs show how we can use Network Code to express elements of an adaptive communication behavior.

Future work continues along two directions: On the systems work, we will create a template library for Network Code programs and improve the tool chain. On the experiments, we will investigate the impact of rouge traffic on real-time networks to estimate the necessity of real-time communication frameworks for modern wired Ethernet systems.

References

[1] L. Almeida, M. Anand, S. Fischmeister, and I. Lee. A Dynamic Scheduling Approach to Designing Flexible Safety-Critical Systems. In *Proc. of the 7th Annual ACM Conference on Embedded Software (EmSoft'07)*, Salzburg, Austria, October 2007.

[2] M. Anand, S. Fischmeister, and I. Lee. An Analysis Framework for Network-Code Programs. In *Proc. of the 6th Annual ACM Conference on Embedded Software (EmSoft'06)*, pages 122–131, Seoul, South Korea, October 2006.

[3] M. Anand, S. Fischmeister, and I. Lee. Composition Techniques for Tree Communication Schedules. In *Proc. of the 19th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 235–246, Pisa, Italy, July 2007.

[4] Bosch. *CAN Specification, Version 2*. Robert Bosch GmbH, September 1991.

[5] R. Court. Real-time ethernet. *Comput. Commun.*, 15(3):198–201, 1992.

[6] S. Fischmeister et al. Network Code Language Specification. Technical report, University of Pennsylvania, 2007.

[7] J. Ferreira, P. Pedreiras, L. Almeida, and J.A. Fonseca. The FTT-CAN protocol for flexibility in safety-critical systems. *IEEE Micro*, 22(4):46–55, July-Aug. 2002.

[8] S. Fischmeister, O. Sokolsky, and I. Lee. A Verifiable Language for Programming Communication Schedules. *IEEE Transactions on Computers*, 56(11):1505–1519, November 2007.

[9] T. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel, and M. Walther. Time Triggered Communications on CAN (Time Triggered CAN - TTCAN). In *Proceedings 7th International CAN Conference*, Amsterdam, Netherlands, 2000.

[10] H. Kopetz. *Real-time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[11] Jane Liu. *Real-Time Systems*. Prentice-Hall, New Jersey, 2000.

[12] J. Loeser and H. Härtig. Real Time on Ethernet using off-the-shelf Hardware. In *n Proc. of the 1st Intl Workshop on Real-Time LANs in the Internet Age (RTLIA 2002)*, 2002.

[13] Nicholas Malcolm and Wei Zhao. The timed-token protocol for real-time communications. *Computer*, 27(1):35–41, 1994.

[14] Kang Shin and Chao-Ju Hou. Analytic evaluation of contention protocols used in distributed real-time systems. *Real-Time Syst.*, 9(1):69–107, 1995.

[15] E.A. Strunk, E.A. Strunk, and J.C. Knight. Dependability through Assured Reconfiguration in Embedded System Software. *IEEE Transactions on Dependable and Secure Computing*, 3(3):172–187, 2006.

ViRe: Virtual Reconfiguration Framework for Embedded Processing in Distributed Image Sensors

Rahul Balani, Akhilesh Singhania, Chih-Chieh Han and Mani Srivastava
University of California at Los Angeles
Los Angeles, CA 90095, USA

***Abstract*—Image processing applications introduce new challenges to the design of sensor network systems via non-trivial in-network computation. As embedded processing becomes more complex, in-situ reconfiguration is seen as the key enabling technology to maintain and manage such systems. In dynamic event-driven heterogeneous sensor networks, reconfiguration also encompasses autonomous re-partitioning of applications across multiple tiers to provide a low-power responsive system by efficiently coping with variations in run-time resource usage and availability. Hence, we aim to provide an efficient low-power macro-programming environment that supports multi-dimensional software reconfiguration of heterogeneous imaging networks.**

Working towards this initiative, we present the ViRe framework for mote-class devices based on data-centric application composition and execution. Applications, modeled as dataflow graphs, are composed from a library of pre-defined and reusable image processing elements. Concise scripts capture the wiring information and are used to install applications in the network, while execution on the nodes is performed via processor native code to minimize overhead. A lean run-time engine tightly monitors application execution to provide an efficient, robust and scalable support for complex reconfigurable embedded image processing. Thus, the system is able to lower application re-partitioning overhead and minimize loss of work during software reconfiguration.

Index Terms **— Sensor Networks, Differential partitioning, Reconfiguration, Dataflow**

I. Motivation

Image sensors demonstrate vital importance in understanding and characterization of diverse environments. Applications involving these sensors pose new challenges to the design of sensor network systems, particularly in the context of recently developed low-power, but resource-constrained, image sensing platforms such as Cyclops [1] and AER Imager [2]. In-network sensor information processing takes on a particularly important role with image sensors, because to save communication bandwidth and energy, image sensor nodes must process the images *in-network* to extract relevant events or features before transmission. Prior work, comprising of mere filter chains, linear dataflow graphs or SQL aggregate functions, is not enough to affect adequate data reduction. Rather, significantly richer and complex recursive algorithms are needed to compress image frames, or to detect features or events of interest. Another reason for the recursive nature of the computation is that the high cost of image acquisition itself usually motivates application structures where a feedback control loop controls when to acquire the next image frame based on history. Thus, typical image processing algorithms require system software support for *efficient* and *reliable* functioning when embedded in resource-constrained sensor nodes.

Clearly, the ability to *reconfigure* application-specific embedded sensor processing on the nodes at run-time, for purposes of re-tasking and environment-specific tuning, without sacrificing the efficiency of processing, is important for effective operation and maintenance of complex sensor networks. This recognition has led to emergence of systems such as Contiki [3], SOS [4], TENET [5], VanGo [6], and Maté [7] where run-time retasking and reconfiguration of application-specific processing at the node are directly supported. However, these prior systems are either too low-level (e.g. Contiki and SOS) and thus do not provide support for higher level programming abstractions common to image sensing, or significantly restrict the complexity or efficiency of application-specific processing and flexibility of reconfiguration (e.g. TENET, VanGo, and Maté) and thus unable to support the requirements of reconfigurable embedded image processing.

However, in a dynamic event-driven multi-tier heterogeneous sensor network, software reconfiguration is not restricted to simple parameter and logic updates, but involves dynamic re-partitioning of the data processing pipeline across different tiers. It is required in dense high data-rate networks that exhibit a clear trade-off between local computation and transmission of data in terms of both latency and energy. For instance, processing data locally at a certain sensor node may reduce energy consumption by reducing data transmission across the network, but may increase overall time (latency) required to obtain the result at the base station due to lengthy computation. This trade-off is unique to each node as the network latency experienced at that node is not only affected by its proximity to the intermediate or final data sink, but also by status of its local transmission queue and network traffic at higher tiers. Thus, to support often conflicting goals of low latency and low power, it is necessary to monitor the above factors at run-time and migrate computation appropriately, resulting in different configurations of the same end-to-end pipeline at different nodes. Henceforth, this dynamic migration of components of a data processing pipeline, to support application re-partitioning, is referred to as *task migration* in this text.

Prior work [8][5] has essentially focused on providing an efficient, robust and scalable macro-programming environment for heterogeneous sensor networks, but consistently ignored

the complex dynamics of an event-driven network and software reconfiguration in the broader sense of run-time re-partitioning of applications. Hence, we propose to build an efficient, robust and reconfigurable macro-programming framework for heterogeneous sensor networks that autonomously monitors dynamic network context and supports application re-partitioning via task migration to achieve user-specified goals.

II. Introduction

As the first step in achieving our objective, we present the ViRe (Virtual Reconfiguration) framework for mote-class devices that constitute the lowest tier of a heterogeneous sensor network (figure 1). Motivated by results from our previous work [9], it explores a different point in the design space of retaskable and reconfigurable embedded sensor processing. It exposes a visual modular wiring diagram abstraction that is commonly used to express image processing algorithms. The modules encapsulate image processing functions, while the wiring is used to express dataflow which may be *non-linear* and with *feedback loops* (figure 3). ViRe permits wiring, module code, and module parameters to be incrementally reconfigured.

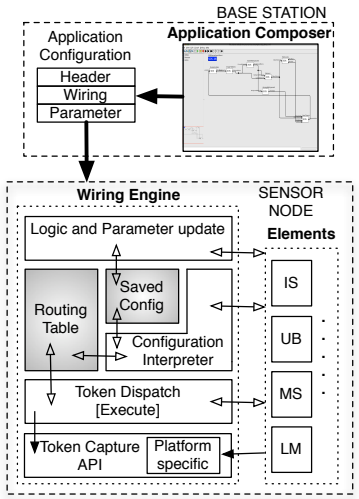


Fig. 1: ViRe Framework

Main idea behind this representation is to separate application instantiation on the nodes, via wiring information captured in concise *scripts*, from execution and data exchange performed in processor native code via dynamic linking at load-time. It has multiple advantages: (i) Execution overhead is kept low while update costs are reduced dramatically, (ii) Work done on prior data is conserved through unobtrusive application of incremental updates, referred to as *hot-swap*, and (iii) Migration overhead is reduced significantly as only a part of the wiring script is required to be transferred across the network during application re-partitioning.

Target sensor nodes interpret the *script*, generated by the *application composer*, to install or update the application through the resident run-time *wiring* engine. Basic communication and scheduling abstraction of the engine, a *token*, provides for efficient management of image data through cooperative memory sharing at run-time. ViRe optimizes the handling of image data since copying an entire sample (image frame), or worse a full block of multiple samples, between processing functions would be prohibitively expensive requiring multiple reads and writes of large, and typically off-chip, frame-buffer memory [1].

The *Token Dispatch* mechanism in the wiring engine ensures that execution follows correct data-flow semantics by coordinating the passage of tokens between elements. This coordination provides explicit control to the engine over application functioning and promotes concise element implementation, thus attributing to the reduced update cost. It is exploited to enable recovery from execution errors, protect against race conditions due to feedback, facilitate a *safe* hot-swap during graph update and minimize transfer of state during task migration.

Application execution begins when new input data is generated by the source element. An element is *fired* whenever its ready to accept new input on a port and a token is placed on that port. Depth-first traversal is followed where the constituting elements run to completion unless they yield, either waiting for inputs on multiple ports or for allowing the engine to schedule other tokens. Consequently, token queues are maintained on graph edges by the engine, instead of the elements, to promote concise module implementations. Thus, an application can be mapped to a pipeline architecture for analysis of its asynchronous behavior and various flow-control strategies at run-time.

The ViRe framework is currently implemented on top of SOS operating system [4] for the cyclops platform [1]. Cyclops is built as an imager sensor board for mote class devices like the Mica motes. Besides other components, it consists of a CMOS imager, an AVR AtMega 128L micro-controller, and 60 KB of external SRAM that is mainly used for buffering images.

III. Wiring Engine: Design

Supporting complex and reconfigurable data-flow representation on the sensor nodes introduces several challenges, including how to link the elements dynamically at run-time to support multiple fan-ins and fan-outs, and efficient exchange of data; how to avoid race conditions due to feedback in recursive image processing algorithms; and how to detect errors in the system to provide reliable functioning. In addition, handling high rate data requires efficient sample processing and memory utilization during execution. The ViRe run-time addresses all these challenges to provide a reliable system for reconfigurable embedded image processing with minimal memory and execution overhead.

A. Application Installation: Configuration Interpreter

Intelligent application installation is necessary to support fast execution with minimal memory overhead. The wiring engine uses a concise routing table, supported by a clear element

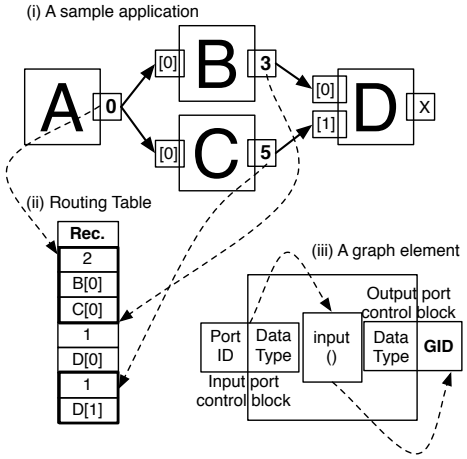


Fig. 2: A sample application graph, its corresponding routing table and element design showing its input and output control blocks.

design as shown in figure 2(iii), to represent the graph edges. The routing table is constructed from the configuration *script* and consists of a group of *output port records* representing the corresponding fan-out of each element port. A record is used whenever an element places a token on its port to facilitate data transfer. The ViRe engine accesses the record through an application-specific unique identifier (`GID`) that acts as a direct index into the routing table and is assigned to each output port at installation/load time. This is an inexpensive constant time access that helps achieve low overhead during data exchange as discussed in section II. Finally, to protect against run-time failures, the data types associated with input and output ports are used at application initialization to validate their dynamic linking.

B. Application Execution: Token Dispatch and Token Capture API

Execution of image processing algorithms in ViRe framework consists of *application specific computation* and *communication* between graph elements. Communication between elements involves transfer of data wrapped in a token structure [10]. It simplifies management of data by providing support for tracking its read-write permissions, ownership and platform specific type. This information is used by the **token capture API** to allow elements to create tokens and cooperatively share memory by releasing and capturing them.

The **token dispatch** mechanism coordinates the exchange of data between graph elements to facilitate efficient communication. Access to an input port of a destination element is enabled via a synchronous function call that provides the input token to the element and returns its current status to the engine. This implicit interaction enables the engine to tightly monitor application execution, maintain token queues for the elements when they are busy processing other tokens, and detect and recover from run-time errors. Thus, an *output port record* is constructed through dynamic linking [4] of multiple callee functions (input ports) to a single caller (output port) and storing the pointers for fast run-time access.

Synchronous data exchange is chosen over asynchronous message passing for execution and memory efficiency, and enabling the engine to control application execution. During execution, it results in a complete traversal of the sub-graph rooted at the output port, referred to as one *output port iteration*. Hence, to avoid race conditions due to feedback in the same iteration, the element is marked busy, and all input tokens destined towards it are queued.

C. Application Reconfiguration

The wiring engine supports logic and parameter reconfiguration on the embedded target, as well as task migration in a tiered sensor network. Logic reconfiguration involves (i) major updates that include significant modifications to the graph edges and elements such that the application, or its implementation, changes as a whole, or (ii) minor updates that include small changes to the graph like addition or removal of a wire or an element. Installing a major update removes the current application completely and initializes a new application as described in section III-A. The rest of this section focuses on the support for applying minor updates through *hot-swap* and reducing state transfer during task migration.

A mechanism supporting hot-swap should minimize loss of work on previously processed tokens and respect data dependencies between elements. Partially processed data, that may be relevant after update, should not be discarded. Data loss is only possible when an *active* element is replaced during the update leading to loss of token(s) being processed by the element at that time.

Typically, task migration across network involves code and state migration. Code size is already minimized through concise wiring representation as discussed earlier. The state of the element, which needs to be migrated across the network, should be minimized to reduce migration overhead. We can observe that local state of the element is minimum when it is not *active* as it does not contain information on partially processed token(s).

Thus, the engine hot-swaps updates or migrates tasks only when none of the involved elements are *active*. But, it is possible that the involved elements, though not *active*, may contain application pertinent state. Currently, the state of the old unused elements is discarded, i.e. not migrated to the new elements that may have replaced them. Therefore, hot-swap and task migration are suggested to include only the elements that never contain application pertinent state. However, in future versions, we propose to extend this mechanism to migrate old state across such incremental graph replacements and element migrations.

IV. Evaluation

This section provides a brief evaluation of the ViRe framework in terms of its update costs and execution overhead on a resource constrained micro-controller like AVR Atmega128.

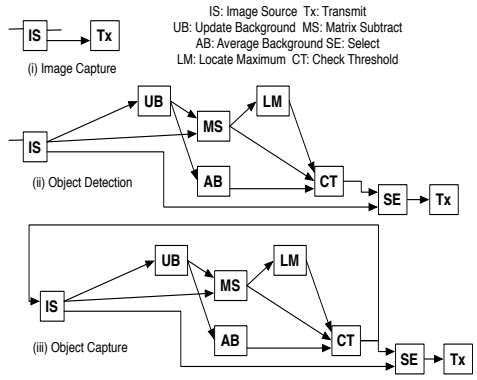


Fig. 3: Application Graphs

Applications	Exec. time *w/out ViRe*	ViRe Overhead	
		Mem. Opt.	Exec. Opt.
Image capture	82 μs	240 μs	80 μs
Object Detect	136 ms	5.6 ms	3.2 ms
Object Capture	136.2 ms	5.8 ms	3.3 ms

TABLE I: Total execution time
Memory optimized and Execution optimized versions of ViRe

Allocation Type	Component	**Memory** (bytes)
Static	Wiring Engine	36
Semi-Dynamic	Graph Elements (RAM)	41
	Routing Table	IC - 6
	(Mem. Opt. - Flash)	OD - 57
	(Exec. Opt. - RAM)	OC - 60

TABLE II: Memory Allocation
IC - Image Capture, OD - Object Detect, OC - Object Capture

Using the successive versions of surveillance application (figure 3) discussed in [10], we demonstrate that the system reduces logic reconfiguration cost of the application by at least an order of magnitude as compared to native implementation in SOS. This is attributed to concise wiring representation and aggressive reduction in the size of image processing modules by promoting simplicity in their design.

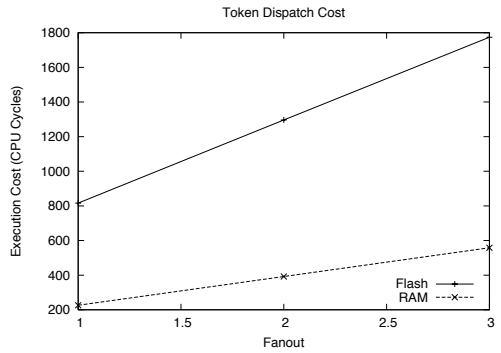


Fig. 4: Communication cost (Token dispatch) at an output port as a function of its fanout when the routing table is stored in RAM vs Flash memory

Moreover, this dramatic decrease in update cost incurs a low execution overhead (table I) ranging from mere 240 μs to 5.8 ms depending on the structure of the application graph. Table II shows the concise memory footprint of the ViRe framework along with installed applications. A thorough analysis of the execution framework [10] helped in identifying and removing communication bottlenecks, thus causing an average 50% reduction in the above overhead at a small expense of increased memory consumption (table I, column *Exec. Opt.*). Figure 4 establishes that the communication cost associated with the framework is closely tied to the graph topology i.e. the number of output ports and their fan-out. Our work in [10] further demonstrates that this cost is independent of the size of data exchanged between elements.

V. Future Plans

Finally, we believe that ViRe overhead will be negligible on processors like Intel/Marvell X-scale, popularly used in higher tiers of sensor network deployments, due to a richer instruction set, faster clock and larger memory. Next, we plan to extend this framework to a heterogeneous multi-tier network and implement distributed network monitoring to investigate the impact of network conditions on autonomous application partitioning that aims to balance low latency goals with minimal energy consumption. Preliminary results from a similar high data-rate network [11] show that the latency can be decreased by as much as 50% through intelligent application partitioning. Later, we will extend it to include the effect of energy availability in an energy-harvesting network.

References

[1] M. Rahimi, R. Baer, O. Iroezi, J. Garcia, J. Warrior, and M. Srivastava, "Cyclops: in situ image sensing and interpretation in wireless sensor networks," *EmNets*, pp. 192–204, 2005.
[2] T. Teixeira, E. Culurciello, J. Park, D. Lymberopoulos, A. Barton-Sweeney, and A. Savvides, "Address-event imagers for sensor networks: evaluation and modeling," *IPSN*, pp. 458–466, 2006.
[3] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki-a lightweight and flexible operating system for tiny networked sensors," *EMNETS*, 2004.
[4] C. Han, R. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava, "Sos: A dynamic operating system for sensor networks," *Mobisys*, 2005.
[5] O. Gnawali, K. Jang, J. Paek, M. Vieira, R. Govindan, B. Greenstein, A. Joki, D. Estrin, and E. Kohler, "The tenet architecture for tiered sensor networks," *Sensys*, pp. 153–166, 2006.
[6] B. Greenstein, C. Mar, A. Pesterev, S. Farshchi, E. Kohler, J. Judy, and D. Estrin, "Capturing high-frequency phenomena using a bandwidth-limited sensor network," *Sensys*, pp. 279–292, 2006.
[7] P. Levis and D. Culler, "Maté: a tiny virtual machine for sensor networks," *SIGOPS*, vol. 36, no. 5, pp. 85–95, 2002.
[8] A. Awan, S. Jagannathan, and A. Grama, "Macroprogramming heterogeneous sensor networks using cosmos," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, 2007.
[9] R. Balani, C. Han, R. Rengaswamy, I. Tsigkogiannis, and M. Srivastava, "Multi-level software reconfiguration for sensor networks," *EmSoft*, pp. 112–121, 2006.
[10] R. Balani, A. Singhania, C. Han, R. Rengaswamy, and M. Srivastava, "Vire: Virtual reconfiguration framework for embedded processing in distributed image sensors," NESL, UCLA, Tech. Rep., June – October 2007.
[11] M. Allen, L. Girod, R. Newton, D. Blumstein, and D. Estrin, "Voxnet: An Interactive, Rapidly-Deployable Acoustic Monitoring Platform," *SPOTS*, 2008.

Trade-off Analysis of Communication Protocols for Wireless Sensor Networks

Jérôme Rousselot, Amre El-Hoiydi, Jean-Dominique Decotignie
Centre Suisse d'Electronique et Microtechnique SA
Jaquet-Droz 1
2002 Neuchâtel
Switzerland

***Abstract*—Embedded systems must react and adapt to changes in their environment. Wireless sensor networks, which must guarantee network connectivity and offer low latency services while keeping energy consumption at a minimum in an unpredictable environment, are a typical example.**

Each layer of their communications stack offers some tunable parameters. Most sensor networks deployments define these parameters at design time, thereby forcing the network to remain always at the same operating point.

Instead, this work aims to identify a set of operating points between which the application could switch depending on its current state. Two protocol stacks are being implemented in a network simulator. The first uses existing state of the art ultra low power protocols and the second one approximates optimal latency and power consumption. Simulation runs allow the evaluation of two criteria: energy consumption and end-to-end latency.

Preliminary results are given for the optimal stack. It is shown that the transmit power of a popular ZigBee transceiver does not allow trading energy consumption against end-to-end latency in the considered configurations, albeit it might be possible with other transceivers. These results validate the methodology adopted in this work and allow to expect more practical results from the trade-off analysis of the ultra low power stack once its implementation is finished.

I. Introduction

Wireless sensor networks must often operate several years on battery. Their power consumption, generally dominated by the use of a radio transceiver, is a critical factor for correct operation. In the last few years, several low-power medium access control (MAC) protocols have been proposed. Energy-aware routing has also been studied extensively, and some transport protocols have been specifically developed for sensor networks.

These advances have made energy consumption less of an issue, and it is now realistic to envision wireless sensor networks reaching the end of their deployment with significant remaining energy. It is thus tempting to allow temporary more aggressive energy usage when the application requires it, e.g. when a fire detection network identifies an emergency situation. Many properties could benefit from an increase in power consumption. End-to-end latency, network connectivity and link quality are only a few examples. This work in progress aims to identify the potential trade-offs of a communications stack for wireless sensor networks.

The work presented in this paper was supported (in part) by the National Competence Center in Research on Mobile Information and Communication Systems NCCR-MICS, a center supported by the Swiss National Science Foundation under grant number 5005-67322.

A stack approximating optimal performance at low data rates has been defined and fully implemented in a network simulator. It uses a detailed model of an IEEE 802.15.4 radio transceiver, a carrier sense multiple access (CSMA) MAC protocol compatible with the non-beacon enabled mode of IEEE 802.15.4, and routes pre-computed offline with Dijkstra's algorithm for the all-shortest-paths-to-source problem. Another, more realistic stack, is composed of the same model of radio transceiver, of the ultra low-power WiseMac protocol and of the distributed routing protocol NSafeLinks. This paper presents intermediate results on the trade-off potential of the optimal stack.

This paper is organized as follows: section II gives an overview of communication protocols for wireless sensor networks. Section III presents the methodology adopted to find these trade-offs and explains the architecture of the two stacks. Section IV analyzes the results obtained thus far and section V concludes the paper.

II. Related Work

This section briefly describes the requirements of sensor networks deployments for the lowest layers of the communication protocols stack (Physical, Data Link and Network) and identifies for each of these layers the most common parameters.

A. Physical Layer

At the lowest layer, wireless communications pose the challenges of an unreliable communication channel and of variable connectivity. Low-power narrow band radios are widely available on the market, and many share a common set of characteristics as defined in the IEEE 802.15.4 [1] standard and are certified interoperable by the ZigBee Alliance [2]. They can often be configured to change the bit rate or the transmit power, influencing bit error rate, power consumption and network connectivity.

Other physical layers such as Ultra Wideband Impulse Radio [3] are in development and promise lower power consumption, robustness to multipath propagation and to multi user interference, and accurate ranging. Transmit power and bit rate can, here also, be configured.

B. Link Layer

Several medium access control protocols for sensor networks have been proposed during the last years. Their main objective is to reduce the power consumption caused by the radio transceiver. They are divided in two categories: scheduled access protocols and contention based protocols.

The first category regroups protocols which establish a Time Division Multiple Access schedule. This can be done using a central coordinator as in the IEEE 802.15.4 beacon-enabled mode or in a distributed fashion as in L-Mac [4], Flama [5] and Trama [6]. They usually define a superframe structure and communications slots whose durations are key parameters for transmission delay and reactivity to changes in network connectivity.

The second category allows nodes to access the medium using probabilistic mechanisms. Some such as S-Mac [7], T-Mac [8] and SCP-Mac [9] are based on a synchronous sleep mechanism. Others such as B-Mac [10], CSMA-PS [11], WiseMac [12], CSMA-MPS [13], X-Mac [14] and SyncWUF [15] are based on asynchronous sleep mechanism and wake-up preambles.

The performance of all contention based protocols, either synchronous or asynchronous, is heavily influenced by their wake-up interval. This parameter is the maximal time during which a node is inaccessible, and the choice of its value influences both latency and power consumption.

C. Network Layer

Wireless sensor networks pose significant challenges for routing [16]:

- the possibly large number of nodes makes the addressing scheme itself an issue ;
- specific traffic patterns such as convergecast must be considered while still allowing peer-to-peer communications ;
- fault tolerance must be built in the protocol because of the unreliability of the physical layer and because the large number of nodes and long deployment times increase the probability of a node failure ;
- load balancing between nodes to prevent early node failure due to an empty battery ;
- timeliness requirements may vary greatly from one application to the other.

Routing protocols for sensor networks can be regrouped in three categories : flat, hierarchical and location-based routing. In flat networks, all nodes are considered equal and are assigned the same functions (SPIN [17], Directed Diffusion [18], Energy Aware Routing [19], NSafeLinks [20]). Hierarchical networks select some of the nodes to act as gateways or cluster heads, rotating this function among several nodes to balance the power consumption (LEACH [21], PEGASIS [22]). Location based routing rely on positioning information to decide which route to use (GEAR [23]).

This work focuses on data collection applications. In this case, all traffic converges to a sink node (convergecast). Each protocol has often a long list of parameters. Size of neighbours table, timers, and metrics all influence performance.

III. Methodology

This work aims to evaluate the potential for trade-off in wireless sensor networks data collection applications by using the network simulator Omnet++ [24]. This section describes the simulation models used and explains the current state of the work.

Regarding the application, network nodes regularly generate data packets. The time intervals between the generation of two packets from the same node follows an exponential distribution of parameter λ: $P[t_i < x] = 1 - e^{-\lambda x}$, where t_i is the time between packets i and $i+1$, and the mean is equal to $\frac{1}{\lambda}$.

An ultra low power stack can be defined from the building blocks described in section II. At the physical layer, a ZigBee compatible chip like the Texas Instruments CC 2420 radio transceiver [25] is a popular choice. WiseMac is a high performance low power MAC protocol which has been tested in many deployments. NSafeLinks is a distributed routing protocol providing N redundant routes from each node to the sink, thereby offering load balancing and robustness to failures.

For benchmarking purposes, it is interesting to define a second stack to approximate optimal performance both in terms of latency and of power consumption. This can be done by using the same radio model to take into account hardware-induced delays, a CSMA/CA MAC protocol (compatible with the IEEE 802.15.4 non-beacon enabled mode) which allows almost immediate transmissions in low data rate conditions, and the use of routes pre-computed offline so that they minimize retransmissions, packet losses and the number of hops to the sink.

Routes are computed offline using a Matlab script. The problem is formulated as an all-shortest-paths-to-sink problem which can be solved using Dijkstra's algorithm. The weights of the paths between two nodes are computed as follows. Since the latency and energy cost of a link are related to the average number of retransmissions of a packet until successful reception and acknowledgement, the link cost is defined as

$$\overline{C_{DA}} = \frac{1}{P_{SD} P_{SA}}$$

where P_{SD} and P_{SA} are the probabilities of successful data and acknowledgement transmissions.

The probability of a successful transmission of a packet of n bits is $P_S = (1 - P_b)^n$ where P_b is the bit error rate. The bit error rate as a function of SNIR is given in [1]:

$$P_b = \frac{8}{15}\frac{1}{16}\sum_{k=2}^{16}(-1)^k \sum_{k}^{16} e^{20 SNIR \frac{1}{k} 1}$$

And the SNIR is obtained as follows:

$$SNIR = \frac{\frac{P_T}{L_P}}{N_0 W F}$$

where P_T is the transmitted power, L_P the pathloss, W the signal bandwidth, F the noise figure and $N_0 = 1.38 * 10^{-23} * 290$ the noise density. For a TI CC 2420 transceiver, $W$and F take respectively the values $2 * 10^6$ and 15.3 dB.

The end-to-end delay for each packet and the total number of packets sent by each node are recorded for each simulation

run. This allows to compute the average end-to-end delay and the total number of packets transmitted. Multiplying this last number with the energy needed to send and receive one packet gives the total consumed energy in the network. This energy is the duration of the 35 bytes packet (20 bytes payload) at 250 kbps multiplied with the power consumption of the TI CC240 radio chip at the selected transmit power (the maximum output power of the CC2420 chip is 0 dBm. A current consumption of 34 mA has been considered for a hypothetical 10 dBm output power). As idle listening and overhearing costs are excluded, this measure corresponds to the energy cost of a contention MAC protocol with an ideal wake-up scheme.

IV. Evaluation

The parameter of interest is the transmit power of an IEEE 802.15.4 compatible Texas Instruments CC 2420 radio transceiver. 21 values, uniformly distributed between -10 dBm and +10 dBm, are considered. The two considered metrics are the mean end-to-end delay between all nodes and the sink and the total energy consumption of the network.

Simulations have been run for five different random positions of 120 nodes distributed on a terrain of 300 x 300 meters. Each combination of random configuration and transmit power implies changes in network connectivity, and thus a specific routing tree. Each node generates 100 packets and the mean time between two packets is set to 60 seconds. The end-to-end delay for each packet is recorded as well as the total number of packets sent by each node. The average of the end-to-end delay is made over all packets and the total of the number of transmitted packets is made over all nodes. Thank to automatic repeat requests, all generated packets are finally received at the sink. The link level packet success rate varies between 95 % and 100% with varying transmit power over all simulations.

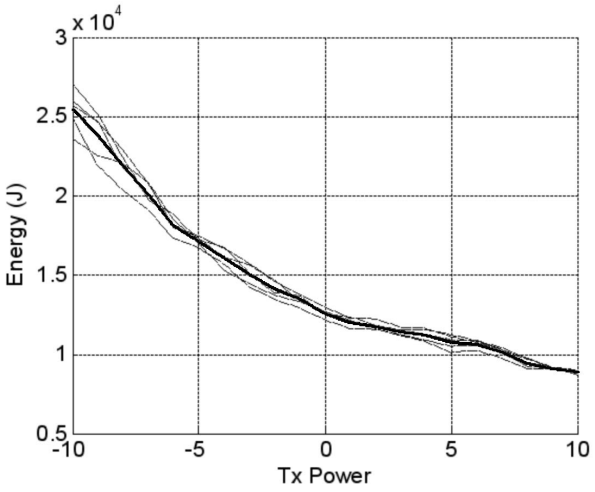


Figure 1. Total energy required for transmitting 100 packets from each node to the sink.

Figure 1 shows the total energy required to forward all generated packets to the sink and Figure 2 shows the average end-to-end latency experienced by those packets. Every different random position of the nodes results in different curves (thin grey lines). The thick black lines represent the average over all random positions.

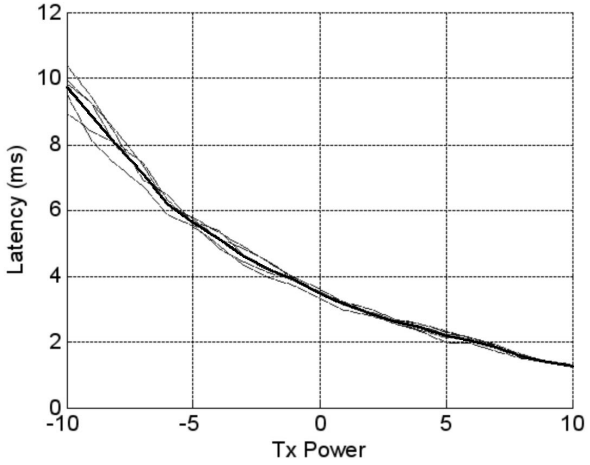


Figure 2. Average end-to-end latency between all nodes to the sink.

From the curves in Figures 1 and 2, it is clear that with the CC2420 chip, there is no trade-off to make (at least in low traffic situation) with the choice of the transmit power. The maximum output power provides the best results both in terms of consumed energy and in terms of latency. The potential advantage of using multiple hops to reduce the consumed energy is not present with the CC2420 because of its high base current consumption (8 mA in transmit mode and 18 mA in receive mode). If this radio would have a base current consumption of 2 mA both in receive and transmit mode, the latency versus energy curve would be as illustrated in Figure 3. In this case, latency and energy could be traded.

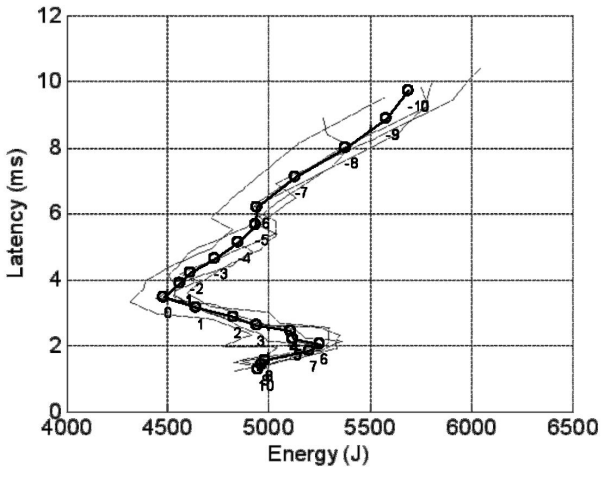


Figure 3. Trade-off analysis for an hypothetical radio.

The simulations used in this section to explore the latency versus energy trade-off with variable transmit power were all made with a low traffic. In high traffic conditions the use of a high transmit power increases the problem of interferences which can cause collisions, thereby increasing the need of

retransmissions and ultimately the power consumption. Even with the CC2420 radio chip, it can be of interest to choose a transmission power below the maximum permitted by the chip to limit interferences.

V. Conclusion

This work in progress studies the impact of a change in transmit power of a popular TI CC 2420 radio transceiver, using ideal low power MAC and routing protocols, on communications latency and network power consumption.

It is shown that there is no such trade-off in this particular case, and that by considering an hypothetical transceiver with slightly different characteristics, an interesting trade-off could be exploited. This result validates the methodology adopted by this work in progress.

The next step will be the study of a realistic ultra low power stack consisting of a TI CC 2420 radio transceiver, WiseMac and NSafeLinks. In addition to transmit power, the impact of the periodic wake-up interval of WiseMac, of the number of redundant routes of NSafeLinks, and of the maximum number of retransmissions will be evaluated. Other deployments scenarios will also be considered, in particular high density networks.

In addition to latency and power consumption, the percentage of traffic correctly reaching its destination (goodput) will also be considered.

Extending the scope of this work to more scenarios, more parameters and additional metrics can only increase the likelihood of identifying valuable trade-offs.

References

[1] *IEEE Std 802.15.4-2006, IEEE Standard for Information technology-Telecommunications and information exchange between systems-Local and metropolitan area networks-Specific requirements- Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)*.

[2] *ZigBee Alliance*, http://www.zigbee.org.

[3] M. Z. Win and R. A. Scholtz, "Impulse radio: How it works," *IEEE Communications Letters*, vol. 2, no. 2, pp. 36–37, February 1998.

[4] L. van Hoesel and P. Havinga, "A lightweight medium access protocol (lmac) for wireless sensor networks," in *Proceedings of the 1st International Workshop on Networked Sensing Systems (INSS 2004)*, 2003.

[5] V. Rajendran, J. J. Garcia-Luna-Aceves, and K. Obraczka, "Energy-efficient, application-aware medium access for sensor networks," in *Proceedings of the 2nd IEEE International Conference on Mobile Adhoc And Sensor Systems (MASS 2005)*, 2005.

[6] V. Rajendran, K. Obraczka, and J. J. Garcia-Luna-Aceves, "Energy-efficient, collision-free medium access control for wireless sensor networks," *Wireless Networks*, vol. 12, no. 1, pp. 63–78, 2006.

[7] W. Ye, J. Heidemann, and D. Estrin, "An energy-efficient mac protocol for wireless sensor networks," in *Proceedings of the Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, vol. 3, 2002, pp. 1567–1576.

[8] T. van Dam and K. Langendoen, "An adaptive energy-efficient mac protocol for wireless sensor networks," in *Proceedings of the 1st international conference on Embedded networked sensor systems*, 2003, pp. 171–180.

[9] W. Ye, F. Silva, and J. Heidemann, "Ultra-low duty cycle mac with scheduled channel polling," in *Proceedings of the 4th international conference on Embedded networked sensor systems*, 2006, pp. 321–334.

[10] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*, 2004, pp. 95–107.

[11] A. El-Hoiydi, "Spatial tdma and csma with preamble sampling for low power ad hoc wireless sensor networks," in *Proceedings of the Seventh IEEE International Symposium on Computers and Communications (ISCC 2002)*, 2002, pp. 685– 692.

[12] A. El-Hoiydi and J.-D. Decotignie, "Wisemac: An ultra low power mac protocol for multi-hop wireless sensor networks," in *Proceedings of the 1st International Workshop on Algorithmic Aspects of Wireless Sensor Networks*, 2004.

[13] S. Mahlknecht and M. Böck, "Csma-mps: a minimum preamble sampling mac protocol for low power wireless sensor networks," in *Proceedings of the 2004 IEEE International Workshop on Factory Communication Systems*, 2004, pp. 73–80.

[14] M. Buettner, G. Yee, E. Anderson, and R. Han, "X-mac: a short preamble mac protocol for duty-cycled wireless sensor networks," in *Proceedings of the 4th international conference on Embedded networked sensor systems*, 2006, pp. 307–320.

[15] X. Shi and G. Stromberg, "Syncwuf: An ultra low-power mac protocol for wireless sensor networks," *IEEE Transactions on Mobile Computing*, vol. 6, no. 1, pp. 115–125, 2007.

[16] J. N. Al-Karaki and A. E. Kamal, "Routing techniques in wireless sensor networks: a survey," *IEEE Wireless Communications*, vol. 11, no. 6, p. 6, 2004.

[17] J. Kulik and W. R. Heinzelman, "Negotiation-based protocols for disseminating information in wireless sensor networks," *Wireless Networks*, vol. 8, pp. 169–185, 2002.

[18] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: A scalable and robust communication paradigm for sensor networks," 2000.

[19] R. C. Shah and J. Rabaey, "Energy aware routing for low energy ad hoc sensor networks," in *IEEE WCNC*, 2002.

[20] T. Ikikardes, M. Hofbauer, A. Kaelin, and M. May, "A robust, responsive, distributed tree-based routing algorithm guaranteeing n valid links per node in wireless ad-hoc networks," in *Proceedings of the IEEE International Symposium on Computers and Communications (ISCC 2007)*, 2007.

[21] . B. H. Heinzelman W., Kulk J., "Energy-efficient communication protocols for wireless microsensor networks," January 2000, lEACH.

[22] S. Lindsey and C. S. Raghavendra, "Pegasis: Power-efficient gathering in sensor information systems," in *Proceedings of the IEEE Aerospace Conference*, vol. 3, no. 9-16, 2002, pp. 1125–30.

[23] Y. Yu, R. Govindan, and D. Estrin, "Geographical and energy aware routing: A recursive data dissemination protocol for wireless sensor networks," Tech. Rep., 2001.

[24] A. Varga, *Omnet++ Discrete Event Simulation System*, http://www.omnetpp.org.

[25] *Texas Instruments 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver*, 2007, http://www.ti.com.

3. Scheduling

A GA-Based Approach to Dynamic Reconfiguration of Real-Time Systems*

Marco A. C. Simões
Bahia State University
ACSO/UNEB, Brazil
msimoes@uneb.br

George Lima
Federal University of
Bahia - LaSiD/UFBA, Brazil
gmlima@ufba.br

Eduardo Camponogara
Federal University of
Santa Catarina (DAS/UFSC), Brazil
camponog@das.ufsc.br

Abstract

Modern real-time systems have become increasingly complex since they have been required to operate in modern architectures and environments with high level of temporal uncertainty, where the actual execution demand is determined only during execution and is valid only for a given period of time. Thus such systems require both temporal isolation and dynamic reconfiguration. In this paper we deal with both these requirements. By assuming that the system is structured such that there are several modes of operation to be chosen at runtime, we formulate a reconfiguration problem. Temporal isolation is ensured by CBS. A solution to the reconfiguration problem is based on Genetic Algorithms, which provide the graceful adaption of the system so that it copes with new demands of the environment. The proposed solution is evaluated by simulation and the results found show the suitability of the approach.

1 Introduction

The design of real-time systems has become increasingly complex. Modern hardware and software architectures and/or the support to open environments, for example, make it difficult or even impossible to estimate task worst-case execution times accurately [13]. A usual way of dealing with this problem is by providing *temporal isolation* in the system. Indeed, if a certain task runs more than what was expected, the effect of this timing fault should not propagate to other application tasks. There are several approaches to temporal isolation and for EDF-scheduled systems the Constant Bandwidth Server (CBS) has received special attention recently [1, 9, 5, 4].

Although temporal isolation is an important aspect, for certain kinds of modern applications, the system should also provide support for *dynamic reconfiguration*. Indeed, modern systems may be structured so that their tasks have one or more modes of operation. For example, a control task can execute one among several control functions and/or with one among several sampling periods previously set. In other words, a task that has more than one operation mode may have alternative pieces of code to execute or alternative release periods. Assuming that each operation mode gives a different benefit for the system, the problem is then to select, at runtime, the set of task modes that maximizes the system benefit subject to the system schedulability.

Applications such as autonomous mobile robots, which embed several complex functions into off-the-shelf complex hardware components, are among those that require support to both temporal isolation and dynamic reconfiguration. For example, the robot computer vision subsystem may experiment different operational modes due to environment changes. Light conditions, obstacles, vision angle and other unpredictable environmental characteristics may generate strong variation on execution times of vision subsystem tasks. Also, modifications in the robot goals during its lifetime may well require dynamic reconfiguration. Further timing faults in one of the system components, even during reconfiguration, should not jeopardize the system performance as a whole.

Some approaches to dynamic reconfiguration have been proposed. For example, Feedback Scheduling (FS) [11], based on control theory, treats the system scheduling as a control plant and uses a controller to adjust the scheduling parameters at runtime. This approach offers no temporal isolation. FS-CBS [14] uses FS. It is assumed that the system is composed of Model Predictive Control (MPC) tasks served by CBS. Although this approach provides temporal isolation, its applicability is restricted to MPC tasks. Further, FS-CBS does not deal with reconfiguration of server periods. Other approaches aim at providing server capacity sharing mechanisms [4, 5, 9]. However, they do not provide dynamic reconfiguration. If a system has no idle time, for example, there is no slack time to be shared.

In this paper we deal with dynamic reconfiguration in a new and challenging scenario, as will be seen in Section 2. Temporal isolation is ensured by the use of CBS. Differ-

*This work is funded by CNPq (grant number: 475851/2006-4) and CAPES/PROCAD (AST project)

ent server modes are previously defined, each one giving a benefit for the system. Dynamically reconfiguring the system is then seen as choosing the operation modes of each server so that the system benefit is maximized. Clearly, this is a reasonably complex optimization problem, which may require too much computational resources to be solved online. However, as will be explained in Section 3, instead of searching for the optimal solution, the approach provided here is capable of progressively reconfiguring the system toward the optimal solution. To do that, we use Genetic Algorithms (GA). The results of experiments, given in Section 4, highlight the suitability of the described approach for the kind of system we are considering. Final comments on this work and on the challenges it brings about are given in Section 5.

2 System Model and Problem Definition

We are considering a system with one processor supporting n Constant Bandwidth Servers (CBS) [1] $S = \{S_1, S_2, ..., S_n\}$ that are scheduled by the Earliest Deadline First algorithm (EDF) [10]. Each server $S_i \in S$ has $\kappa(i) \geq 1$ operation modes and we denote $K_i = \{1, 2, \ldots, \kappa(i)\}$. The k-th operation mode of S_i is denoted by the tuple (Q_{ik}, T_{ik}), where Q_{ik} represents the server mode capacity and T_{ik} is its period. Thus, each server S_i operating in mode k has a maximum processor utilization $U_{ik} = \frac{Q_{ik}}{T_{ik}}$. In other words, the system allocates, to each task served by S_i, a constant bandwidth defined by U_{ik} so that temporal isolation is ensured [1]. As the system is scheduled by EDF, it is possible to use 100% of processor resources. Clearly, by the CBS schedulability properties [1], if the parameters Q_{ik} and T_{ik} are appropriately set for each server, tasks meet their deadlines or have an acceptable lateness (in case of soft tasks) provided that there are no timing faults.

We assume that the system may require that predefined values of Q_{ik} and T_{ik} are assigned to each server S_i at runtime so that the system can adapt or switch itself to attend to new demands by means of dynamic reconfiguration. This is performed by a system call, say `reconfig`$(U_{1k_1}, v_1, U_{2k_2}, v_2, \ldots, U_{nk_n}, v_n)$, where U_{ik_i} is the new desired processor utilization for server S_i and v_i is the associated benefit to this new configuration. Thus, if the system can allocate at least U_{ik_i} for server S_i, there will be a benefit v_i for the system. Without loss of generality, we assume that the benefit function associated to S_i running in mode k is defined by

$$A_{ik}(U_{ik_i}, u_i, v_i) = \frac{min(u_i, U_{ik_i})}{U_{ik_i}} v_i \,, \qquad (1)$$

where u_i represents the processor utilization effectively allocated to S_i. Other benefit functions are possible and we do not impose any restriction on them. Interesting discussion on benefit functions can be found elsewhere [3] and is beyond the scope of this paper. The benefits v_i are defined by the application when it uses the `reconfig` system call.

The execution of `reconfig` solves the optimization problem R defined as follows:

$$R : f = Maximize \sum_{S_i \in S} \sum_{k \in K_i} A_{ik} x_{ik} \qquad (2a)$$

$$\sum_{S_i \in S} \sum_{k \in K_i} U_{ik} x_{ik} \leq 1 \qquad (2b)$$

$$U_{ik} = \frac{Q_{ik}}{T_{ik}} \qquad (2c)$$

$$\sum_{k \in K_i} x_{ik} = 1,\ S_i \in S \qquad (2d)$$

$$x_{ik} \in \{0, 1\},\ S_i \in S,\ k \in K_i \qquad (2e)$$

Equation (2a) defines the objective function, which is based on equation (1). Variable x_{ik}, defined by equation (2e), represents the choice of one of the $\kappa(i)$ server operation modes of S_i. Only one configuration must be selected by `reconfig`, which is ensured by equation (2d). Restriction (2b) guarantees the schedulability of S according to the EDF policy.

It is not difficult to see that the classical knapsack problem can be reduced to R and so it is a $NP-Hard$ problem. One can solve R by using some standard optimization techniques. In particular, we have used dynamic programming to get optimal results, which will be used to assess our GA-based reconfiguration approach, as will be seen in Section 4. There is a recursive formulation for R which leads to a solution via dynamic programming, which is not shown here for the sake of space.

3 GA-Based Dynamic Reconfiguration

In this section we explain how the reconfiguration procedure was set up. The parameters of the procedure were empirically adjusted using an example of 10 servers with 15 operation modes each. The fitness function is given by equation (2a). The individuals (represented by their chromosomes) that have the highest value returned by the fitness function represent an optimal solution for the problem. In the context of problem R, a solution can be defined by a list $(k_1, k_2, \ldots, k_n)$, where $k_i \in K_i$. Let $\kappa^* = \max_{i=1}^{n} \kappa(i)$. Thus, a binary encoding [12] can be used so that $\lceil n \log_2 \kappa^* \rceil$ bits are needed to represent a possible solution (an individual). For example, a system with 10 servers each of which with 15 operation modes would require a 40-bit chromosome.

At the start of the `reconfig` procedure, we set the initial population by randomly generating $5n$ individuals.

These initial parameters are based on the De Jong's test suite [8] with a few adjustments. At each iteration of the algorithm a new population is generated from the current population. To do that, the operations of selection, crossover, mutation and migration are performed:

Selection. First, the individuals are ordered according to their fitness values. Two of them (the best fit ones) are selected by elitism [8] and copied to the new population. Selection probability values are assigned to the other $n-2$ individuals by the stochastic uniform method [2] according to their fitness values.

Crossover. The crossover operation is carried out $\lfloor \frac{5n-2}{4} \rfloor$ times to generate 50% of remaining offsprings. Each operation combines two individuals of the current population, generating two new offsprings to be added in the new population.

Mutation. This operation is carried out for $\lfloor \frac{5n-2}{2} \rfloor$ selected individuals, following some usual recomendations [12]. Each selected individual can have its bits changed with a probability of 0.1 per bit. The new generated individuals after mutation are added to the new population.

Migration. We have divided our population into two subpopulations of $\frac{5n}{2}$ individuals each. Then the 20% best-fit individuals from each subpopulation are cloned and they replace the 20% worst-fit individuals of the other subpopulation. This migration operation [6] takes place every 10 generations.

4 Assessment

We implemented the GA-based reconfiguration using Matlab/Simulink and the TrueTime toolbox [7]. We simulated 100 systems and the results presented below correspond to the average values found for these systems. Each simulated system had 10 servers with 15 operation modes each. The server parameters were randomly generated as follows. First, the values of U_{ik} and Q_{ik} for all $S_i \in S$ were generated according to a uniform distribution in the interval $(0, 0.2]$ and $[10, 100]$, respectively. Time is measured in time units (tu).

The first evaluated parameter was the average performance ratio (APR) achieved by the proposed approach. APR is defined here as $\frac{1}{m}\sum_{j=1}^{m}\frac{V_j}{OPT_j}$, where $m = 100$ is the number of simulated systems, V_j is the value achieved by the proposed approach as for the simulated system j and OPT_j is its optimum value given by a dynamic programming based solution (as said before, not shown here). As can be seen from Table 1, the APR achieved by the proposed approach gives very good results, which lies very close to 80% to the optimum value. It is interesting to note that the performance ratio does not change significantly as a function of the number of generations used to reconfigure the system, indicating a rapid convergence of the reconfiguration mechanism. As can be observed, satisfatory solutions can be found using as few as 10 generations at the expense of about 73 tu.

Generations	APR	Std. Dev.	Av. Exec. Time (tu)
10	0.74	0.02	73.13
25	0.76	0.02	162.45
50	0.77	0.02	311.97
100	0.78	0.01	610.22
250	0.79	0.01	1505.70
500	0.79	0.01	3001.63
1000	0.79	0.01	6016.20

Table 1. Performance ratio.

We have set up a simulation experiment as follows. A specific server with a single mode, given by $(20, 100)$, was added to the system and reserved for carrying out the reconfiguration. This means that the other 10 servers have now 80% of processor resources available. During the simulation, the average performance ratio was measured for 100 simulated systems. The time interval of observation corresponds to the first 100 activations of the reconfiguration server after the time the `reconfig` system call is called. Each reconfiguration server instance manages to deal with 3.44 generations on average. The average values for all systems are shown in Figure 1. As can be seen, there is a fast convergence of the reconfiguration mechanism. More than 72% to the optimum is achieved after the third instance of the reconfiguration server while 76% is reached after its 15th instance. However, the reconfiguration procedure tends to converge to some local optimum. As we managed to set up the GA parameters so that the global optimum was reached for some specific simulated systems, we believe that the local optimum convergence is due to the fact that the GA parameters used are not suitable for all simulated systems, since they were generated at random. This suggests that a more systematic approach to setting up such parameters is needed.

It is important to emphasize the adaptive nature of the proposed approach. While each reconfiguration instance is able to manage only a few generations, it is producing partial results. For example, when the third instance of the server finishes executing, it gives 72% of APR. This partial solution can be immediatelly used and is progressively improved as long as the reconfiguration server is running. In the case of this simulation, 80 time units on average were needed to process the three first instances of the server. The 15th instance finishes after 1190 time units. It is also worth mentioning that we did not address the problem of mode

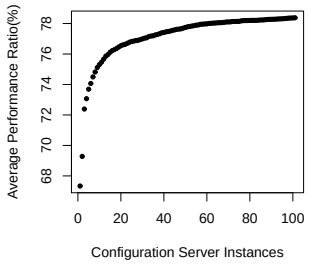


Figure 1. Average performance ratio growth.

change necessary to pass from one configuration to another.

5 Conclusions

We have described an approach to dynamic reconfiguration of real-time systems structured as a set of Constant Bandwidth Servers each of which can operate in one of the previously defined modes. Using a specific objective function, we have formulated a specific reconfiguration problem. We have shown that approximate solutions to this problem can be found in an effective way by using Genetic Algorithms. The described approach is evaluated by simulation and the results found have indicated the suitability of the described approach. For example, the proposed approach can be used for gradually adapting the system to environment changes. This is particularly interesting to deal with modern real-time systems operating in environments with high level of uncertainty.

Most parameters used in the reconfiguration approach were defined empirically. It would be interesting to derive mechanisms to adjust such parameters in a mechanized way by using, for instance, Neural Networks. More complex objective functions can also be investigated. For example, one may be interested in considering parameters such as lateness, deadline miss ratio, etc., which will give rise to multi-objective optimization problems. Another challenging goal would be to design self-adjustable scheduling policies so that the system itself decides when to carry out reconfigurations. Certainly, these and other research topics can use the proposed approach as a starting point.

References

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. *Real Time Systems Symposium(RTSS), The 19th IEEE*, pages 4–13, 1998.

[2] J. E. Baker. Reducing bias and inefficiency in the selection algorithm. In J. J. Grefenstette, editor, *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*. Erlbaum, 1987.

[3] A. Burns, D. Prasad, A. Bondavalli, F. Di Giandomenico, K. Ramamritham, J. Stankovic, and L. Stringini. The meaning and role of value in scheduling flexible real-time systems. *Journal of Systems Architecture*, 46:305–325, 2000.

[4] M. Caccamo, G. Buttazzo, and Lui Sha. Capacity sharing for overrun control. *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*, pages 295–304, 2000.

[5] M. Caccamo, G.C. Buttazzo, and D.C. Thomas. Efficient reclaiming in reservation-based real-time systems with variable execution times. *IEEE Transactions on Computers*, 54(2):198–213, Feb. 2005.

[6] E. Cantu-Paz. Migration policies, selection pressure, and parallel evolutionary algorithms. *J. Heurist*, 7:311–334, 1999.

[7] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K.-E. Arzen. How does control timing affect performance? analysis and simulation of timing using jitterbug and truetime. *Control Systems Magazine, IEEE*, 23(3):16–30, June 2003.

[8] K. A. De Jong. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. PhD thesis, University of Michigan, 1975.

[9] Caixue Lin and Scott A. Brandt. Improving soft real-time performance through better slack reclaiming. *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, 0:410–421, 2005.

[10] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[11] Chenyang Lu, John A. Stankovic, Sang H. Son, and Gang Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Syst.*, 23(1-2):85–126, 2002.

[12] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1998.

[13] Xiaorui Wang. *Adaptive Quality of Service Control in Distributed Real-Time Embedded Systems*. PhD thesis, Washington University, August 2006.

[14] Pingfang Zhou, Jianying Xie, and Xiaolong Deng. Optimal feedback scheduling of model predictive controllers. *Journal of Control Theory and Applications*, 4(2):175–180, Feb 2006.

CPU Utilization Control Based on Adaptive Critic Design

Jianguo Yao[1,2], Xue Liu[1]
jianguo@cs.mcgill.ca, xueliu@cs.mcgill.ca
[1]School of Computer Science, McGill University, Montreal, QC H3A2A7, CANADA
[2] School of Astronautics, Northwest Polytechnical University, Xi'an, Shaanxi 710072, P.R.CHINA

Most embedded systems have stringent real-time temporal requirements. Controlling the CPU utilizations under the schedulability bounds at individual nodes is an effective and efficient way to ensure real-time temporal guarantee. Recently, researchers have proposed solutions which employ model predictive control to maintain the desired CPU utilization. However, task execution times can vary greatly during run-time for most modern embedded applications. Although existing approaches can handle a limited amount of execution time uncertainties, the performance deteriorates when the variations are large. In this paper, we present an adaptive control approach to handle large model uncertainties. Our adaptive control approach is based on adaptive critic design. A comparative study via simulation is conducted to demonstrate the effectiveness of the new approach. Results show that the system based on adaptive critic design achieve better performance guarantee when large model uncertainties exist.

I. INTRODUCTION

Real-time embedded systems are becoming essential component of computing and they are playing an important role in people's everyday lives. Examples of such systems include avionics mission computing, autonomous aerial surveillance, and disaster recovery systems. These systems execute tasks and deliver services conforming to temporal constraints [1]. One of the most effective ways to ensure temporal guarantee in real-time embedded systems is to control the CPU utilization under the schedulability bounds on individual nodes. Traditionally, real-time embedded systems work in open-loop fashions. The scheduling decision relies on accurate characterizations of the platform and the workload (such as worst case execution times). However, in many practical applications, parameters such as task execution times may be highly dynamic. A key challenge faced by such applications is providing real-time guarantees while the workload cannot be accurately characterized a priori. In the past few years, researchers began to apply feedback control techniques to solve this challenge. In these control-theoretic approaches, system CPU utilizations are maintained under the CPU schedulability bounds through dynamic resource allocations in response to load variations [2], [4], [5]. CPU utilization control targets are usually set near the schedulability bounds so as to fully utilize the CPU resources and at the same time guaranteeing real-time deadlines to be met.

Most of the previous works on CPU utilization control in real-time embedded systems were focused on using Model Predictive Control (MPC) algorithms. Detailed discussions of how to apply MPC in utilization control for real-time embedded systems were presented in [2], [4], [5]. In these approaches, centralized or decentralized MPC controllers were designed based on approximated models of the underlying systems and the tasks running on top of them. It was shown that these approaches can handle successfully when limited workload uncertainties or variations are presented in the underlying systems. However, as we will show, when the uncertainties and variations are large, MPC-based approaches' performance deteriorates.

In order to better control the performance for these systems, adaptive feedback control design is needed. In this paper, we present a new CPU utilization control approach based on Adaptive Critic Design (ACD) to deal with large workload variations. It is worth noting that this pilot study only discusses centralized control problem. However, our approach can be extended to decentralized control in terms of composition method of distribute system as discussed in [5].

The remainder of this paper is organized as follows. Section II describes the problem statement on utilization control. Section III describes a general architecture of adaptive critic design (ACD). Section IV presents the detailed descriptions of the control system design based on ACD. Section V presents the simulation and evaluation results. Finally, we conclude the paper in Section VI.

II. PROBLEM STATEMENT

We employ CPU utilization model as discussed in [2]

$$x(k) = x(k-1) + Bu(k-1), \quad (1)$$

where $x \in \mathbb{R}^{n\times 1}$ represents the CPU utilization vector on the n nodes; $u \in \mathbb{R}^{m\times 1}$ represents the change to task rates for the m tasks running on the system; $B \in \mathbb{R}^{n\times m}$ is the unknown input matrix, which is related to parameters of estimated execution time and the corresponding gain on it. B is described as

$$B = GF, \quad (2)$$

where G is a gain matrix suggesting the ratio between the actual utilization and its estimation, which itself is unknown. F is the subtask allocation matrix, which is available. It represents which tasks are running on which nodes.

Suppose the goal of control system is to track the constant reference command x_d, where x_d represents the CPU utilization target vector. Let us denote the error between the measured CPU utilization and target utilization as $e(k) = x(k) - x_d(k)$. So the control goal is to minimize *e(k)*. Typically, we can choose the control law as follows

$$u(k-1) = (\beta - 1)B^{-1}e(k-1), \quad (3)$$

where β is a design matrix which ensures that the tracking error is bounded. Note we can not get matrix B accurately since we do not have the exact information of the gain matrix G due to the varying excursion time. As a result, applying the

model-based controller design method such as MPC, the control performance may deteriorate when large uncertainties exist. However, we can approximate the value of $(\beta-1)B^{-1}e(k-1)$ using adaptive learning design assuming it is a smooth function, hence we can get the control input. In the following, we will discuss the model-free adaptive critic design controller design.

III. A General Architecture of Adaptive Critic Design (ACD)

Adaptive Critic Design (ACD) is a powerful method to solve approximated optimal control problem of nonlinear and uncertain plant. It is based on the combination of Dynamic Programming (DP) and Reinforcement Learning (RL). In the ACD architecture, there are three modules: the utility function, the critic network and the action network, where the utility function indicates instant cost reward, the critic network approximates the cost-to-go function describing the performance of the system, and the action network gives optimal action by minimizing the output of critic network.

Fig. 1 shows a general architecture of ACD. Plant is a computing system with multiple inputs and multiple outputs (MIMO), which is influenced by many uncertainties, such as time delay, varying execution time and noise; ref is the reference command signal expected by the good system performance. The critic network tunes its parameters through reinforcement learning algorithm and the action network is adjusted by the output of critic network.

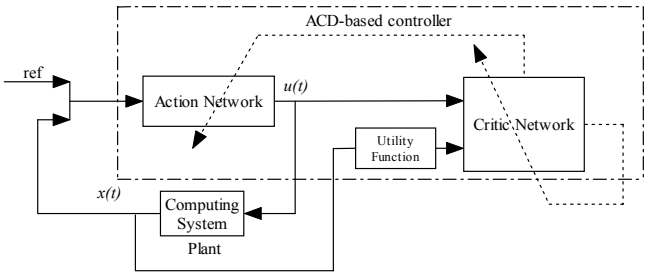


Fig. 1. A General Architecture of Adaptive Critic Design

IV. Online Adaptive Critic Design

In order to ensure that the system tracks the reference command with small error, we present an adaptive control algorithm based on the online ACD. The algorithm with one input is presented in [3], and in our research, we extend it to multiple inputs. The detailed ACD-based control algorithm is described in this section.

A. Utility Function

Utility function embodies the control objective. To achieve this goal, we represent the utility function by a step function, where "-1" suggests the trial has failed because the tracking error is larger than the threshold, and "0" suggests the trial has succeeded because the tracking error is smaller. It is defined as follows

$$U(k)=\begin{cases}0, & \text{for all } e_i(k)<Th\\ -1, & \text{else}\end{cases},\quad i=1,2,\cdots,n, \qquad (4)$$

where $e_i(k)=x_i(k)-x_{di}(k)$ is the tracking error of the ith element in $x(k)$, x_{di} represents reference of the ith element in the reference command vector $x_d(k)\in\mathbb{R}^n$, and Th is the threshold selected by the designer.

B. The Critic Network

The critic network approximates the cost-to-go function $J(k)$ indicating system performance. We define $J(k)$ as

$$J(k)=U(k+1)+\alpha U(k+2)+\cdots+\alpha^{N-1}U(k+N), \qquad (5)$$

where α is the discount factor, $0<\alpha<1$, and N is the final time step.

Dynamic Programming is employed to formulate optimal control problem. The Bellman Equation is given by

$$J(k)=U(k+1)+\alpha J(k+1). \qquad (6)$$

We define the error of critic network as

$$e_c(k)=\alpha J(k)-[J(k-1)-U(k)]. \qquad (7)$$

In the critic network, the goal is to minimize the following function

$$E_c(k)=\frac{1}{2}e_c^2(k). \qquad (8)$$

Then we apply neural network to approximate the cost-to-go function $J(k)$

$$J(k)=\hat{W}_c^{(2)}(k)\phi(\hat{W}_c^{(1)}(k)s(k)), \qquad (9)$$

where $s=[x^T,u^T]^T\in\mathbb{R}^{m+n}$ is the input vector of critic network, and $\hat{W}_c^{(1)}(k)\in\mathbb{R}^{N_h\times(m+n)}$ and $\hat{W}_c^{(2)}(k)\in\mathbb{R}^{N_h}$ are the weight matrices in the critic network; N_h is the number of hidden nodes in both critic network and action network; and the function $\phi(x)$ is defined as

$$\phi(x)=\frac{1-\exp^{-x}}{1+\exp^{-x}}. \qquad (10)$$

To minimize Equation (8), weight matrix $\hat{W}_c^{(1)}$ is updated as follows

$$\hat{W}_c^{(1)}(k+1)=\hat{W}_c^{(1)}(k)+\Delta\hat{W}_c^{(1)}(k), \qquad (11)$$

$$\Delta\hat{W}_c^{(1)}(k)=l_c(k)\left[-\frac{\partial E_c(k)}{\partial\Delta\hat{W}_c^{(1)}(k)}\right], \qquad (12)$$

where l_c is a positive learning rate of critic network.

Calculating the partial derivative in Equation (12), we get,

$$\hat{W}_c^{(1)}(k+1)=\hat{W}_c^{(1)}(k)-\alpha l_c e_c(k)\hat{W}_c^{(2)}\phi'(\hat{W}_c^{(1)}(k)s(k))s(k), \qquad (13)$$

where ϕ' is the derivative of function ϕ.

Similar to that of weight matrix $\hat{W}_c^{(1)}$, the weight matrix $\hat{W}_c^{(2)}$ is updated by

$$\hat{W}_c^{(2)}(k+1)=\hat{W}_c^{(2)}(k)+\Delta\hat{W}_c^{(2)}(k), \qquad (14)$$

$$\Delta\hat{W}_c^{(2)}(k)=l_c(k)\left[-\frac{\partial E_c(k)}{\partial\Delta\hat{W}_c^{(2)}(k)}\right]. \qquad (15)$$

Calculating the partial derivative in Equation (15), we get,

$$\hat{W}_c^{(2)}(k+1) = \hat{W}_c^{(1)}(k) - \alpha l_c e_c(k)\phi(\hat{W}_c^{(2)} s)\,. \quad (16)$$

C. The Action Network

Define the error of the action network as

$$e_a(k) = J(k) - U_c\,, \quad (17)$$

where U_c is set to 0 because the signal "0" represents "success" in the utilization function.

In the action network, the objective is to minimize the following function

$$E_a(k) = \frac{1}{2} e_a^2(k)\,. \quad (18)$$

We design the action output $u(k)$ using neural network. The equation is as follows

$$u(k) = u_{\max}\phi\left[\hat{W}_a^{(2)}(k)\phi(\hat{W}_a^{(1)}(k)x(k))\right], \quad (19)$$

where $u_{\max}$ is the maximum magnitude of input vector, $\hat{W}_a^{(1)}(k) \in \mathbb{R}^{N_h \times n}, \hat{W}_a^{(2)}(k) \in \mathbb{R}^{m \times N_h}$ are the weight matrices in the action network.

To minimize the Equation (18), the update law of weight matrix $\hat{W}_a^{(1)}$ is given by

$$\hat{W}_a^{(1)}(k+1) = \hat{W}_a^{(1)}(k) + \Delta\hat{W}_a^{(1)}(k)\,, \quad (20)$$

$$\Delta\hat{W}_a^{(1)}(k) = l_a(k)\left[-\frac{\partial E_a(k)}{\partial \Delta\hat{W}_a^{(1)}(k)}\right], \quad (21)$$

where l_a is a positive learning rate for action network.

Substitute Equation (19) into Equation (9) , we get

$$\begin{aligned} J(k) &= \hat{W}_c^{(2)}(k)\phi[\hat{W}_c^{(11)}(k)x(k) + \hat{W}_c^{(12)}(k)u(k)] \\ &= \hat{W}_c^{(2)}(k)\phi\{\hat{W}_c^{(11)}(k)x(k) \\ &\quad + \hat{W}_c^{(12)}(k)u_{\max}\phi\left[\hat{W}_a^{(2)}(k)\phi(\hat{W}_a^{(1)}(k)x(k))\right]\}, \end{aligned} \quad (22)$$

where $\hat{W}_c^{(1)}(k) = [\hat{W}_c^{(11)}(k), \hat{W}_c^{(12)}(k)]$; $\hat{W}_c^{(11)}(k) \in \mathbb{R}^{N_h \times n}$ and $\hat{W}_c^{(12)}(k) \in \mathbb{R}^{N_h \times m}$ are weight matrices corresponding to $x(k)$ and $u(k)$ respectively..

Calculating the partial derivative in Equation (21), we get

$$\begin{aligned} \hat{W}_a^{(1)}(k+1) &= \hat{W}_a^{(1)}(k) - l_a \frac{\partial E_a(k)}{\partial J(k)} \frac{\partial J(k)}{\partial \hat{W}_a^{(1)}(k)} \\ &= \hat{W}_a^{(1)}(k) - l_a e_a(k) \frac{\partial J(k)}{\partial \hat{W}_a^{(1)}(k)} \\ &= \hat{W}_a^{(1)}(k) - l_a e_a(k)\hat{W}_c^{(2)}(k)\cdot \\ &\quad \phi'[\hat{W}_c^{(1)}(k)s(k)]\hat{W}_c^{(12)}(k)u_{\max}\phi'\{\hat{W}_a^{(2)}(k)\cdot \\ &\quad \phi[\hat{W}_a^{(1)}(k)x(k)]\}\hat{W}_a^{(2)}(k)\phi'[\hat{W}_a^{(1)}(k)x(k)]x(k). \end{aligned} \quad (23)$$

Similar to the update law of weight matrix $\hat{W}_a^{(1)}$, the update law of $\hat{W}_a^{(2)}$ is given by

$$\hat{W}_a^{(2)}(k+1) = \hat{W}_a^{(2)}(k) + \Delta\hat{W}_a^{(2)}(k), \quad (24)$$

$$\Delta\hat{W}_a^{(2)}(k) = l_a(k)\left[-\frac{\partial E_a(k)}{\partial \Delta\hat{W}_a^{(2)}(k)}\right]. \quad (25)$$

Calculating the partial derivative in Equation (25), we get

$$\begin{aligned} \hat{W}_a^{(2)}(k+1) &= \hat{W}_a^{(2)}(k) - l_a \frac{\partial E_a(k)}{\partial J(k)} \frac{\partial J(k)}{\partial \hat{W}_a^{(2)}(k)} \\ &= \hat{W}_a^{(2)}(k) - l_a e_a(k) \frac{\partial J(k)}{\partial \hat{W}_a^{(2)}(k)} \\ &= \hat{W}_a^{(2)}(k) - l_a e_a(k)\hat{W}_c^{(2)}(k)\phi'[\hat{W}_c^{(1)}(k)s(k)]\cdot \\ &\quad \hat{W}_c^{(12)}(k)u_{\max}\phi'\{\hat{W}_a^{(2)}(k)\phi[\hat{W}_a^{(1)}(k)x(k)]\} \\ &\quad \phi[\hat{W}_a^{(1)}(k)x(k)]. \end{aligned} \quad (26)$$

D. Stability Analysis

The detailed stability analysis is presented in [3].

V. Simulation and Evaluation

We conducted simulation studies to evaluate the performance of ACD-based adaptive controller in CPU utilization control. In order to show the effectiveness of the proposed approach, we compared the control performance between ACD-based controller and MPC-based controller. The results are reported in this section.

A. Simulator and Experimental Setup

The simulation is conducted in Matlab. The simulator consists of four major modules. The first is the *Utility Function Module*. In this module, the utility function is obtained, and it suggests the action taken in the last sample whether is good or not. The second is the *Critic Network Module*. In this module, there are two functions: calculating the approximated cost-to-go function and updating the weight matrices in the critic network. The third is the *Action Network Module*, which comprises two parts. One is to calculate the action signal, and the other is to update the weight matrices in the action network. The last Module is the *Plant Module*. Its function is to conduct the action obtained from the Action Network Module and to get new state of the system.

Fig. 2 shows the architecture of control system applied to a simple computing system presented in [2]. In this system, we have two processors and three periodic tasks. Task 1 locates in Processor 1 (P_1) which is denoted by T_{11} and Task 3 denoted by T_{32} is in Processor 2 (P_2). Task 2 is divided into two subtasks T_{21} and T_{22}, which are in P_1 and P_2 respectively. T_{21} and T_{22} have the same task rate.

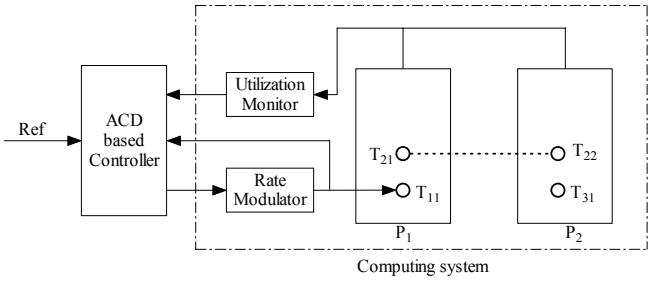


Fig. 2. Centralized Controller based on Adaptive Critic Design

Supposing $r_i(k)$ is the invocation rate of Task i in the *(k+1)*th sampling period., it has constraints as follows

$$R_{\min,i} \le r_i(k) \le R_{\max,i}, i = 1,2,3\,, \quad (27)$$

where $R_{\min,i}$ is the minimum task rate of task *i* and $R_{\max,i}$ is the maximum.

We choose the input vector

$$u(k-1) = \left[\Delta r_1(k-1) \quad \Delta r_2(k-1) \quad \Delta r_3(k-1)\right]^T, \quad (28)$$

where $\Delta r_i(k)$ is the change of the task rate $r_i(k)$, whose constraints are determined by the constraints of $r_i(k)$. According to the Inequation (27), we have input constraints as

$$\left|\Delta r_i(k)\right| \le R_{\max,i} - R_{\min,i}, i = 1,2,3. \quad (29)$$

B. Parameter Setting

The bound of CPU utilization is chosen based on the method suggested in [2]. Supposing $x_{\max,i}$ is the maximum value of utilization in CPU *i.* It can be calculated by the following equation

$$x_{\max,i} = m_i(2^{1/m_i} - 1), i = 1,2, \quad (30)$$

where m_i is the number of subtasks in CPU *i*, and in this experiment $m_1 = m_2 = 2$, so that we know that $x_{\max,i} = 0.828$, $i = 1,2$ and the utilization is subject to

$$0 < x_i(k) < x_{\max,i}. \quad (31)$$

Parameter matrix F in Equation (1) is suggested in [2], and we set the gain matrix G to imply whether or not there is model uncertainty. In MPC design, we assume gain matrix $G = [1,0;0,1]$, i.e. the actual CPU utilizations will be the same as the estimated ones and there is no model uncertainty. To test the system performance under model uncertainty, we choose $G = [0.5,0;0,0.5]$ in the simulation, i.e. the actual CPU utilizations are a half of the estimations.

Table 1. The parameters of ACD-based controller

parameter	α	Th	$l_c(0)$	$l_a(0)$
value	0.92	0.2	0.3	0.003
parameter	$l_c(f)$	$l_a(f)$	N_h	-
value	0.01	0.003	9	-

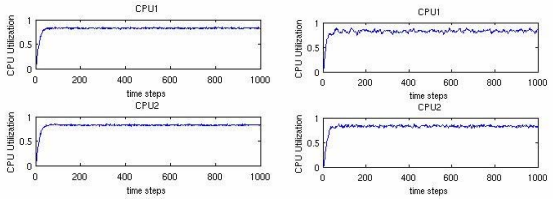
The parameters of ACD-based controller are shown in Table 1, where $l_c(0)$ is the initial learning rate of critic network; $l_a(0)$ is the initial learning rate of action network. $l_c(f)$ is the final learning rate for critic network; the middle learning rate $l_c(k)$ decreases every three samples until it reaches $l_c(f)$. $l_a(f)$ is the final learning rate of action network; the changing of middle learning rate $l_a(k)$ is similar to that of $l_c(k)$.

C. Evaluation Results

We compared the control performance of two approaches: MPC-based controller and ACD-based controller.

Fig. 3 compares the time responses of CPU utilization based on two controllers applied to the computing system with model uncertainty, where (a) and (b) are based on ACD-based controller and MPC-based controller respectively.

Using the aggregate of squared errors between the utilization target and the actual measured utilization on each CPU over the duration of the experiment in the steady state, we can compare the performance of the two schemes. The smaller the aggregate error, the better the control performance. Table 2 summarizes the results. In these results, the aggregate errors are calculated for the interval from time t=200 to t=1000.



(a) (b)

Fig. 3. CPU utilization control with noise and model uncertainty

Table 2. The aggregate errors under ACD and MPC

	ACD	MPC
CPU1	0.0069	0.0196
CPU2	0.0087	0.0156

Our experiment results demonstrate the advantage of ACD-based controller in guaranteeing the CPU utilization. Compared with MPC-based controller, the ACD-based controller can ensure better system performance in large model uncertainty.

VI. Conclusions

In this paper, we focus on the CPU utilization control problem to guarantee the performance of real-time embedded systems. In these systems, execution time may vary greatly, so an adaptive control approach is needed to adjust the CPU utilization to ensure real-time guarantee. Model-based controllers such as model predict control can handle small model uncertainty problems. To deal with the large model uncertainty problem, we employ adaptive critic design (ACD) based controller. A simulation based on a simple plant is conducted. The results suggest that the proposed ACD-based controller can achieve better performance than MPC-based controller for the CPU utilization with large execution time uncertainty.

Acknowledgment

This work was supported in part by an NSERC discovery grant and a National Study-Abroad Scholarship of P.R.China under Grant No. [2007] 3020.

References

[1] J. Liu, *Real-Time Systems*: Prentice Hall PTR 2000.

[2] C. Lu, X. Wang, and K. X., "Feedback utilization control in distributed real-time systems with end-to-end tasks," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 16, no. 6, pp. 550-561, 2005.

[3] J. Si and Y Wang, "Online learning control by association and reinforcement," *IEEE Transactions on Neural Networks*, vol. 12, no. 2, pp. 264-276, 2001.

[4] X. Wang, Y. Chen, C. Lu, and X. Koutsoukos, "FC-ORB: A robust distributed real-time embedded middleware with end-to-end utilization control," *Journal of Systems and Software*, vol. 80, no. 7, pp. 938-950, 2007.

[5] X. Wang, D. Jia, C. Lu, and X.A.K.X. Koutsoukos, "DEUCON: Decentralized End-to-End Utilization Control for Distributed Real-Time Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 7, pp. 996-1009, 2007.

A hierarchical approach for reconfigurable and adaptive embedded systems*

Moris Behnam, Thomas Nolte, Insik Shin
Mälardalen Real-Time Research Centre
Mälardalen University, Västerås, SWEDEN

Abstract

Adaptive and reconfigurable embedded systems have been gaining an increasing interest in the past year from both academics and industry. This paper presents our work on hierarchical scheduling frameworks (HSF) intended as a backbone architecture facilitating the implementation of operating system support for adaptability and reconfigurability.

1 Introduction

The work presented in this paper is motivated by the needs of adaptability and reconfigurability in multiple embedded systems domains. In this paper we target mainly the automotive domain; however the approach is also suitable to other application domains such as robotics. We present our work based on the hierarchical scheduling framework (HSF) [15, 16] as backbone architecture for applications with high requirements on adaptation and reconfiguration.

Automotive software systems are traditionally rather static in terms of their provided functionality, how they are configured, and where they are physically located. However, recent trends indicate an increased interest towards dynamic automotive systems [2]. Due to high requirements on safety, predesigned modes are created to cover for all possible usage and failure scenarios. At the same time, cost, weight and complexity motivated trends investigate the possibility of integrating more and more software on fewer electronic control units (ECU) [17], which, in turn, potentially increases the risk of single point of failure scenarios. Having fewer ECUs means that failure of one ECU has the potential to bring several functions down, functions that used to be distributed over multiple ECUs.

For safety critical systems, in the case of failure, certain functionality must always be provided. Firstly, if a failure occurs while driving the car it must be possible to bring the car to a safe state. A scenario of a car being uncontrollable due to a software failure would be unacceptable. Secondly, if possible, it is desirable if the car under failure provides a limited limp back home functionality, i.e., a set of functions allowing the driver to bring the car to repair, even if parts of the system have failed.

From an adaptability and configurability point of view, if one part of the system fails due to, e.g., an accident/crash with the car or due to some internal failure, functionality can be migrated to other nodes, bringing the car to a safe state. This can be provided by either static predesigned modes or by a more dynamic adaptation and reconfiguration functionality with the possibility of coping with more complex scenarios.

Robotics is another targeted domain, and we distinguish robotics used for automation from robotics used in the field.

Robotics used for automation is typically found along assembly lines, and they have high requirements on uptime. If an assembly line would be stopped, this can cause large costs to the manufacturer. In other words, changing, maintaining and adding functionality to the assembly line should not require the whole system to be stalled. The system should provide the possibility of online reconfiguration, tuning and monitoring, in order to minimize downtime.

Field robotics often relies on sensors in order to react on its (sometimes) dynamic environment. These sensors trigger different functionality and actions to be taken depending on the current situation.

Modes can be designed for specific purposes as a reaction to the robot's environment. For example, a tracing robot searching for a specific target can be in different modes depending on if it is lost or if it knows where it is going. Also, the trigger of specific sensors might trigger a more sensitive motion control, whereas normal behaviour would be less sensitive.

These modes can be offline designed, in the case when all possible modes can be predicted and managed beforehand. Dealing with more complex scenarios, the design can be more dynamic as a result of using online modes.

In summary, looking at the abovementioned application domains, there is a great potential for protocols, mechanism and architectures providing adaptability and reconfigurability as a first class citizen.

*The work in this paper is supported by the Swedish Foundation for Strategic Research (SSF), via the research programme PROGRESS.

In the following sections, the HSF is presented and how it provides operating system support for adaptability including policies and algorithms for resource reconfiguration. Finally, our ongoing work on admission control functions is presented, and the paper is concluded.

2 The hierarchical scheduling framework

2.1 What is HSF?

The hierarchical scheduling framework (HSF) has been introduced to support hierarchical resource sharing among applications under different scheduling services. The hierarchical scheduling framework can be generally represented as a tree of nodes, where each node represents an application with its own scheduler for scheduling internal workloads (e.g., threads), and resources are allocated from a parent node to its children nodes.

From a general point of view, it is desirable that a hierarchical scheduling framework can support the following properties; (1) *independency*: i.e., that the fulfilment of temporal requirements (schedulability) of a subsystem can be analyzed independently of other subsystems as there will be no unpredictable interference among subsystems. (2) *abstraction*: i.e., that a subsystem imposes minimal temporal requirements on its environments in order to guarantee functional and extra-functional correctness. (3) *universality*: i.e., that any scheduler can be used within a subsystem, allowing for the most appropriate scheduler to be used for a specific function. (4) *flexibility*: i.e., it should enable adaptation and reconfiguration of its subsystems, implementing operating system support for adaptability including policies and algorithms for resource reconfiguration.

2.2 What are the benefits?

The HSF has been constructed with *modularity* as a main criterion. Component based design has been widely accepted as a methodology for designing and developing complex systems through systematic abstraction and composition. The HSF provides means for decomposing a complex system into well-defined parts, called *subsystems*, and for *interfaces* specifying the relevant properties of these subsystems precisely, such that subsystems can be independently developed and assembled in different environments. The HSF provides a means for composing subsystems into a subsystem assembly, or *composite*, according to the properties specified by their interfaces, facilitating the reuse of subsystems. A challenging problem in composing subsystems is to support the principle of *composability* such that properties established at the subsystem level also hold at the composite level. The HSF can be effectively useful in supporting composability on timing properties in the design and analysis of real-time systems, since it allows the system-level timing property to be established by combining the subsystem-level timing properties (specified by individual subsystem interfaces).

The HSF can be used to support multiple applications while guaranteeing independent execution of those applications. This can be correctly achieved when the system provides *partitioning*, where the applications may be separated functionally for fault containment and for compositional verification, validation and certification. The HSF provides such a partitioning, preventing one partitioned function from causing a failure of another partitioned function in the time domain.

The HSF is particularly useful in the domain of open environments, where applications may be developed and validated independently in different environments. For example, the HSF allows an application to be developed with its own scheduling algorithm internal to the application and then later included in a system that has a different meta-level scheduler for scheduling applications.

2.3 Enabling adaptability and reconfigurability

The HSF is very useful when it comes to the implementation of operating system support for adaptability and reconfigurability needed in dynamic open systems, where applications (one or more subsystems) may be allowed to join and/or leave the system during runtime. In allowing such functionality, a proper *admission control* (AC) must be provided. Also, the HSF allows for a convenient implementation of quality of service management policies, allowing for a dynamic allocation of resources to subsystems.

2.4 Related work on HSFs

Over the years, there has been a growing attention to HSFs for real-time systems. Since Deng and Liu [6] proposed a two-level HSF for open systems, several studies have followed proposing its schedulability analysis [8, 9]. Various processor reource models, such as bounded-delay [12] and periodic [10, 15], have been proposed for multi-level HSFs, and schedulability analysis techniques have been developed for the proposed processor models [1, 5, 7, 10, 14]. The work in this paper extends [15, 16].

3 Admission control

The admission control (AC) applies one or more algorithms to determine if a new application (consisting of one or multiple subsystems) can be allowed to join the system and start execution (admission) without violating the requirements of the already existing applications (or the requirements of the whole system). The decision of the AC depends on the state of the system resources and the resources required by the new application asking for admission. If there are enough resources available in the system,

the application will be admitted; otherwise the application will be rejected.

In general, since the AC uses online algorithms the complexity and overhead of implementing these algorithms should be very low for several reasons, such as maintaining scalability of the AC and minimizing its interference on the system. Hence, one objective in designing the AC concerns keeping the input to these algorithms as simple as possible, e.g., the resource requirement for each individual task could be abstracted to the subsystem level. Another objective concerns minimizing interference between the AC and the system online, making it desirable to perform as much work as possible offline.

3.1 Resources

The *resources* considered by the AC may include, but are not limited to, *CPU* resources, *memory* resources, *network* resource and *energy* resources. Initially, we have been focusing on CPU and network resources, and are now also looking at memory resources.

CPU resources When using the HSF, traditional schedulability algorithms can be used in order to check the CPU resources, e.g., by using the global schedulability test in the HSF [15, 16]. This algorithm depends on the type of system level scheduler used, e.g., EDF, FPS, etc. The AC checks the schedulability condition of the system including the new subsystem. If the system is still schedulable, the new subsystem will pass this test; otherwise the new application will be rejected. In using this test, it is guaranteed that all hard real time requirements will be met. The input to the algorithm is the subsystem interface (subsystem budget and period) of each running subsystem together with the interface of the new subsystem. Note that these parameters are evaluated and determined during the development of the subsystem (offline).

Memory resources When allowing for a new application to enter the system, the AC should guarantee that there is sufficient memory space to be used by all subsystems. Otherwise, unexpected problems may happen during run time. In a similar way as for CPU resources, the maximum memory space required by each subsystem is evaluated during its development. In the AC test, a simple algorithm can be used to check if there is enough memory space available in the system, by checking if the summation of the maximum memory space for all subsystems is less than or equal to the memory space provided by the platform. Such an algorithm is very simple; however, the accuracy of the result is not high as all applications will not likely need their specified maximum memory space at the same time. Higher efficiency can be achieved by the usage of algorithms such as the approximated algorithm presented in [4].

Energy resources Most of the modern processors support changing the frequency and voltage of the CPU during runtime, in controlling the CPU's power consumption. The HSF can use this feature to select the lowest frequency/voltage that guarantees the hard real time requirements of the system. Decreasing the frequency of a processor will increase the worst-case execution time (WCET) of its tasks. In doing this, more CPU resources should be allocated to subsystems in order to ensure that all hard real time tasks will meet their deadlines. Looking at the HSF, if predefined levels of frequencies are used, we can find a subsystem interface for each frequency level for all subsystems. Then, during runtime, the AC will make sure that the processor is working with the lowest frequency keeping the schedulability of the current set of subsystems. When it is required to add a new subsystem, the AC will check the schedulability condition with the current processor frequency; if the system is deemed not schedulable, then the AC will try with higher frequencies. When a subsystem is removed from the system, the AC will try to reduce the frequency of the CPU in order to reduce its power consumption.

Network resources This type of resource is important in distributed systems where there typically exist communications between nodes in the network. The network resource is different from the other resources previously described in the sense that the network resource is shared by all nodes, while the other resources are local to each node. When the AC is faced with a request for adding a subsystem, it should check if the communications requirements will be met, i.e., check if all important messages will be delivered in proper time [13]. Selecting an algorithm that checks this resource is more complex as there are many different requirements, communication protocols, network types, etc. Covering all these aspects might not be necessary but as an illustration consider a simple algorithm which relies on the communication bandwidth. During the development of each subsystem, their maximum communication bandwidth requirements should be evaluated such that the AC can use it in order to check if the summation of required bandwidth for all subsystems is less than 100%.

3.2 Admission control in distributed systems

Implementing the AC in distributed systems is more complex than doing so for a single CPU. The main reason for this is that the information needed by the AC algorithms must be consistent. For example, when using the network resources, awareness of all network users must be maintained by the AC, and these users are typically located on many nodes throughout the distributed system. Commonly, information on the current state is kept at one place, managing the information needed by the AC. Also, when an application consists of more than one subsystem, and these subsystems are located at different nodes, all these subsystems should pass the AC tests before admitting the application.

In designing the AC we have identified 3 different approaches based on where the AC test will be implemented.

- A special **Master Node (MA)** will implement the AC tests of all resources in the system. Only the MA will have information about resources in the system. Hence, consistency is not a problem, it is easy to determine the order between AC requests, and the AC does not have to contact multiple nodes in getting the current system state as only a single AC request to the MA is needed. On the downside the MA is a single system level point of failure.
- **All Nodes (AN)** will implement the AC tests of all resources in the system. Each node should have the consistent information of all resources that are used by the system. Hence, in this fully distributed approach the AC test can be performed without having to communicate with other nodes. Also, the approach is tolerant to failures. On the downside, consistency must be maintained between all nodes, and ordering is more complex compared with MA. Also, more memory is needed in maintaining all resource state replicas.
- There will be **One Node (ON)** implementing the AC test of each resource in the system. Each node will maintain information about the resources that is responsible for. Hence, there will be no issues with respect to data consistencies between replicas of the same information, but a single AC request might have to communicate with a number of nodes in order to get a valid system state. Also, ordering among AC requests must be solved, and each resource owner will be a single resource level point of failure.

As a first step, we consider the MA approach that we believe possess strong properties in terms of flexibility, promoting the evaluation of multiple algorithms. Also, this approach fits very well with the HSF.

4 Summary

To summarise we are currently active in 3 areas related to the motivation of this paper. Firstly, we have done work in the area of synchronization algorithms for HSFs [3]. When multiple subsystems are sharing the same CPU it is likely that they also share logical resources that must be protected by the usage of a proper synchronization protocol. Secondly, we are implementing the HSF on a commercial operating system on an application given by one of our industrial partners. This implementation will have an important role in evaluating the efficiency of the HSF itself, as well as the AC approaches presented in this paper. Thirdly, we are designing the admission control and quality of service manager. These will also be implemented in the HSF implementation. In summary we believe that the result of these three areas will provide important knowledge towards adaptive and reconfigurable systems; results that have both high industrial relevance as well as academic relevance.

Acknowledgements

The authors wish to express their gratitude to the anonymous reviewers for their helpful comments.

References

[1] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *EMSOFT '04*, 2004.

[2] R. Anthony, A. Leonhardi, C. Ekelin, D. Chen, M. Törngren, G. de Boer, I. Jahnich, S. Burton, O. Redell, A. Weber, and V. Vollmer. A future dynamically reconfigurable automotive software system. In *Elektronik im Kraftfahrzeug*, June 2007.

[3] M. Behnam, I. Shin, T. Nolte, and M. Nolin. Sirap: A synchronization protocol for hierarchical resource sharing in real-time open systems. In *EMSOFT'07*, 2007.

[4] M. Bohlin, K. Hänninen, J. Mäki-Turja, J. Carlson, and M. Nolin. Safe shared stack bounds in systems with offsets and precedences. Technical Report, Mälardalen University, January 2008.

[5] R. I. Davis and A. Burns. Hierarchical fixed priority preemptive scheduling. In *RTSS*, 2005.

[6] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *RTSS '97*, 1997.

[7] X. A. Feng and A. K. Mok. A model of hierarchical real-time virtual resources. In *RTSS*, 2002.

[8] T.-W. Kuo and C.-H. Li. A fixed-priority-driven open environment for real-time applications. In *RTSS*, 1999.

[9] G. Lipari and S. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *RTAS '00*, 2000.

[10] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *ECRTS*, 2003.

[11] G. Lipari, J. Carpenter, and S. Baruah. A framework for achieving inter-application isolation in multiprogrammed hard-real-time environments. In *RTSS '00*, 2000.

[12] A. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *RTAS '01*, 2001.

[13] T. Nolte. *Share-Driven Scheduling of Embedded Networks*. PhD thesis, Department of Computer and Science and Electronics, Mälardalen University, Sweden, May 2006.

[14] S. Saewong, R. R. Rajkumar, J. P. Lehoczky, and M. H. Klein. Analysis of hierar hical fixed-priority scheduling. In *ECRTS '02*, 2002.

[15] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *RTSS '03*, 2003.

[16] I. Shin and I. Lee. Compositional real-time scheduling framework. In *RTSS '04*, 2004.

[17] G. Spiegelberg. The impact of new gateways and busses - are these the answers for furhter innovations?, Podiums Discussion at the Embedded Systems Week, Salzburg, Austria, October 2007.

Suitability of Dynamic Load Balancing in Resource-Constrained Embedded Systems: An Overview of Challenges and Limitations

Magnus Persson magnper@md.kth.se
Tahir Naseer Qureshi tnqu@md.kth.se
Martin Törngren martin@md.kth.se

Department of Machine Design, KTH (The Royal Institute of Technology), Stockholm, Sweden.

Abstract

In this paper, we discuss the challenges and limitations of applying load balancing to networked and resource constrained embedded systems. The paper proposes a problem formulation and a checklist for guiding design and implementation of load balancing in such systems.

Keywords

Load balancing, distributed systems, networked embedded systems

1 INTRODUCTION

With the advancement of technology, networked embedded systems are becoming increasingly common, more computationally and communication intensive and complex. Current design techniques are dominated by static and worst-case designs. This leads to underutilization of resources which causes an increase in product costs but also difficulties in exploiting software and computer systems flexibility. As a consequence, reconfiguration at runtime, i.e. *dynamic reconfiguration*, implying techniques to change the structural (e.g. allocation) and/or behavioral configuration (e.g. priorities), is increasingly considered in networked real-time embedded systems [1].

This trend is exemplified by the DySCAS[1] project [2], which develops an automotive middleware. In [3], the DySCAS consortium describes the project use cases, including load balancing, and in [4], the need for further investigation of the problem was identified.

Typically, many of the simpler *ECUs* (*electronic control unit*) in a car are only 8-bit, have kilobytes of memory, and communicate over a CAN bus which has a maximum bandwidth of 1 Mbit/s. In addition they run very simple real time operating systems (RTOS) or even more simplistic run time environments.

During the work of exploring approaches to dynamic reconfiguration for these systems, a number of questions regarding *dynamic load balancing* in resource-constrained embedded systems have been raised, such as the limitations of its use; implementation challenges; and which approaches are most suitable for embedded systems. In this paper, we explore some of these issues. A more extensive survey is given in [6].

[1] The abbreviation stands for *"Dynamically Self-Configuring Automotive Systems"*.

2 INTRODUCING LOAD BALANCING

Task allocation is the assignment of a set of tasks to a set of nodes in a networked system. *Load distribution* is a specialization of task allocation, where resource usage (loads) at the different nodes is taken into consideration. *Load balancing* is a further specialization, where the allocation is fully or partially optimized based on one or several resource usage metrics. These metrics convey some type of evenness or fairness of resource usage at different nodes. Even though this paper focuses on load balancing, its results are also relevant to load distribution in general.

There is a major difference between *static* and *dynamic* load balancing. Static load balancing is done at system design time (or possibly configuration time) and allocations are then kept fixed through-out the life time of the system. Dynamic load balancing is done during system startup or runtime, possibly taking different aspects of application behavior into consideration. When the task set or available resources in the network changes, dynamic load balancing is re-executed, potentially changing the task assignment during runtime.

Load balancing can be applied to several different types of resources. Without doubt, the most common one is CPU usage, but some systems also at least partly take other scarce resources into account; e.g. memory or network bandwidth.

The purposes of implementing load balancing in different systems are diverse. Generally, the main reason is often performance (average throughput) maximization. Parallel execution of a work task on several nodes, a common approach for performance improvement, by itself implies some scheme of load distribution.

Runtime load distribution is sometimes also implemented in systems with the main goal to provide e.g. flexibility, robustness or reliability. Common to these is that task assignment before deployment is problematic to implement. The main optimization goal in these systems is to provide certain level of service and maintain performance guarantees.

One final example objective for implementing load balancing is energy efficiency – by distributing processor loads equally, clock frequencies (and thus energy consumption) of processors can be minimized, or the number of nodes could even be reduced.

Load balancing is related to techniques for *quality of service* (QoS) where QoS usually refers to reservation and control mechanisms for different resources such as network traffic to provide a guaranteed level of data flow in a network. However, QoS and load balancing are rarely studied together [6]. We

consider load balancing as a technique that could be incorporated as one part of a QoS scheme.

In traditional computer engineering, a number of systems have been designed that deal with load balancing. Most of them were developed in an ad-hoc fashion. Examples include simple schemes, e.g. *round-robin*, of which a number of variations are described in [7]. In [8] the idea of *diffusion* is described. In [8] a hierarchical approach is also described. Finally, as done in Mosix [9], *local cost functions* can be used.

2.1 Load balancing viewed as a control problem

One way to understand dynamic load balancing is the viewpoint of control system engineering, as shown in figure 1, illustrating sensing, feedback, actuation and feedforward components. Note that all parts of the figure are not explicitly part of all load balancing approaches. For example, set-points are often only implicit, and many approaches do not have any feed-forward part, or the feedforward part only consists of simple scheme for *admission control*.

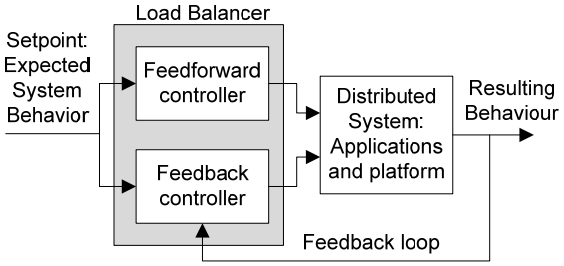


Figure 1: Load balancing from the control perspective

In feedback-based approaches, load balancing is performed in response to observations (sensing) of load changes on a particular network node or due to changes in average load on the entire network. Delays and disturbances in the feedback information lead to longer controller response times. This can result in poor efficiency and a non-robust controller, and can be eliminated to some extent by the use of feedforward control which uses an approximate system model to be able to predict the actual systems state prior to computing the control signals (in this case allocations). A specific case of feedforward control is admission control, where acceptance or denial of new tasks are the only possible outcomes. In traditional feed-forward control, some sort of prediction is made based on a model of the computer system in order to compute a suitable allocation [16, 18].

A challenge for control of modern computing systems is that they are hard to predict due to their hardware architecture, timing of external events, and component interdependencies, which makes mathematical modeling and analyzing of them non-trivial. This implies that feedback techniques generally would be more applicable since feed-forward requires system models.

Stability is another important issue in designing a controller. An example of an unstable load balancing algorithm would be e.g. one which makes a task repeatedly move between two nodes, because the algorithm sees the other allocation as preferable in both cases. There are several ad-hoc ways to stop such problems – one is to allow movement only after the application has run at its current node a minimum time. Another is to add hysteresis to the algorithm, preventing changes resulting in only minor improvements. Unlike traditional control approaches, stability of load balancing approaches is seldom proven formally.

There exist a few optimization based algorithms without explicit use of control systems theory. The difference between these approaches lies in the method to choose source and the receiver nodes [10,11]. Model predictive control has also been used where the concept of local and global controllers is employed for individual nodes and the complete system respectively. The usual target is the CPU utilization and deadline miss ratio [12] but a few techniques have also focused on end-to-end utilization [13]. Some of the approaches also focus on systems with I/O intensive applications [14] or with shared memory [15].

3 KEY DESIGN AND IMPLEMENTATION ISSUES

As described in section 2, there exist a lot of load balancing algorithms proposed for different systems. However, every algorithm has some issues and trade-offs involved which are discussed below.

3.1 Real-time requirements

Many networked embedded systems are also real-time systems. Unfortunately, most load balancing algorithms do not take timing properties of the algorithms into consideration; and the dynamic reallocation of tasks to nodes makes the problems of predicting task response times considerably harder. In essence, real-time properties are not typically a consideration of a dynamically load balancing systems, which makes them less suitable for systems or tasks with hard real-time requirements.

Reconfiguration caused by load balancing can be required on occurrence of an event, such as failure of some kind, increased or decreased load, or connection of a new node in network providing new potential services.

As shown in figure 2, the reaction delay τ_r for a reconfiguration is caused by time required for sensing τ_{se} which can be event triggered or time triggered, decision making or control mechanism τ_{de}, and finally reconfiguration or task migration τ_{re} which includes state and code transfer, followed by the startup at the new node.

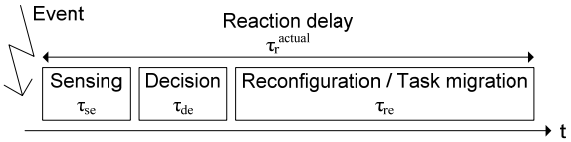


Figure 2: Timing properties from an event to final outcome of reconfiguration

If task migration due to load balancing is to be beneficial, it is self-evident that the following constraint needs to be true:

$$\tau_r^{\,actual} = \tau_{se} + \tau_{de} + \tau_{re} \leq \tau_r^{\,desired}$$

where $\tau_r^{desired}$ represents a system time constraint on the migration. Additionally, resource usage at the new node needs to be, in some manner, more efficient at the new node.

During design and implementation of load balancing, several additional issues have to be considered. The issues in the following sections have been identified as most important. In all of them, management of scarce resources is the common factor.

3.2 Conflicts with other mechanisms

The relation of approaches like QoS and QoC with load balancing is not well established. Even if QoS typically only assumes soft real-time requirements, it is usually not combined with load balancing. For some cases, admission control for new tasks is used, to ensure that the number of tasks to be load balanced is not excessive, preventing overload in the system as a whole.

If dynamic load balancing and quality of service are combined in the same system, there is a possibility for conflicts between them if not carefully coordinated. As an example, if overload is detected on one node in a system, this can potentially be solved in different ways – either by lowering QoS for one of the tasks running on the node, or by moving one task to another node through load balancing. In the case of load balancing, even if the resources on the new node are sufficient, performance may still be badly degraded (e.g. by communication delays if tight communication with the old node is needed). Perhaps performance would have been better, even with a degraded QoS, if the application would have been kept at the old node?

3.3 Design issues

One of the very basic design issues is the *problem definition* and *requirements formulation*. It is very easy to claim that load balancing is to be performed in a certain system without really defining what type of load balancing is to be performed. Precise formulation of requirements including the choice between static and dynamic load balancing, resources and overheads to be considered and the selection of rebalancing execution method is often neglected. Moreover, the purpose to perform load balancing is often ambiguous. It can be for performance improvement or to ensure some other property of the system. Many load balancing approaches improve only average performance and don't give any minimum performance guarantees, but it is seldom fully clear during requirements formulation if this is acceptable in a specific embedded system.

Similar issues show up due to implicit assumptions on the underlying hardware architecture. An algorithm intended to be used for multiprocessor computers with shared memory will probably not be suitable for network of computers and vice versa. Also, the performance of each part of the system differs a lot between 'small-scale' and larger systems.

3.3.1 Performance overheads

Adding dynamic load balancing to an existing system causes performance overheads. The types of resources usually considered are CPU utilization, memory and network. Causes for CPU overheads and memory usage are the computations required for load balancing, storage of software states and other relevant data, causing higher resource usage. Network overheads occur due to data movements and message passing between different processing nodes. Network overheads can be seen both in increased network traffic, and in delays caused by load balancing, e.g. when moving a task to another node.

Large overheads are undesirable – it is important to remember that load balancing can only improve system performance if there is a node with less resource usage that the task can be transferred to, and if none is available, load balancing is only an unnecessary overhead and will only make the system perform worse.

3.4 Detection and sensing issues

Dynamic load balancing is possible both through event-triggering and time-triggering. In the case of time-triggering, deciding on intervals for sampling, execution and actuation of load balancing is important. In event-triggered systems, the threshold for event detection is a similar parameter. Too frequent triggering will lead to bigger overheads, whereas too infrequent triggering will lead to bad performance of the load balancing algorithm.

3.4.1 Required information and parameters

Different load balancing algorithms require different input parameters. Common examples are execution time, queue length, task arrival and departure rates, delays in processing of a task, miss ratios, incoming requests and service time. Using a higher number of parameters makes it possible to build a better and more generic algorithm; there is however also a risk that a complex algorithm simply is too complex for systems with low processing power.

3.5 Decision and control issues

The decision and control issues are relatively well covered in the existing load balancing approaches. Basically, four questions as described in [19] need to be answered: *When* is task migration triggered, *which* task is to be migrated, to *where* will it migrate, and *who* makes the decision? As these questions are relatively well answered in the currently existing literature, we will not further investigate it in this paper.

3.6 Actuation issues

One well known practical problem from fault-tolerant computer systems related to dynamic load balancing is the *state transfer*. It can also be considered as a design issue. When the allocation of tasks to processors changes, the tasks reassigned to other processors need to be moved somehow. A pre-requisite is to identify the minimum system information required when switching/continuing processing on another node. If the code is not previously deployed on the new node, this includes the program code. In some load balancing approaches, especially where task life-time is typically short (e.g. web servers), this problem is avoided by only doing assignment of new tasks.

If the state transfer problem is not avoided, several types of data might have to be moved – application data (variables etc.), executable code, and implicit application state (program counter, processor registers, open files, and other metadata typically managed by the operating system) are possible candidates. The most powerful way to do it is process migration [17], this is however also the most complicated way.

Several other approaches also exist; remote invocation of tasks is one such example, or custom protocols can be used between the new and old instance of the application. In the context of resource-constrained systems, the overheads (e.g. amount of network traffic, time for transfer) caused by the state transfer are important to consider. Specifically, movement of code makes the problem harder, as many operating systems in small embedded systems only support static loading. It is possible to avoid this problem by deploying the code to additional nodes already at design time; however, this requires additional memory.

Related to the issue of task migration is also the size of the *distribution units*, or the granularity of the entities that are balanced. Common choices include processes, groups of processes, or incoming requests to the system. This granularity

will impact the result of the load balancing. However, choosing too small distribution units might impact system scalability.

4 CONCLUSIONS

For future implementations of load balancing in resource-constrained embedded systems, we propose the following checklist to be gone through at system design time to check if the suggested variant of load balancing is suitable:

- The design problem should be clearly defined., including:
  - The main purpose of using of load balancing.
  - The type of resources to be balanced.
  - Decision to allow runtime movement of tasks and the method to be used.
  - Size of distribution units.
- When choosing a specific load balancing algorithm, its performance overhead should be thoroughly evaluated.
- Real-time requirements on the system and real-time performance of the load balancing algorithm should be examined and compared in detail.

Due to real-time and dependability requirements, most of the current approaches to load balancing are not suitable in hard real-time systems. To deploy load balancing in such systems further research is needed. One approach is to develop load balancing algorithms that explicitly take hard real-time requirements into consideration e.g. through QoS or QoC mechanisms, or to build systems where hard real-time is ensured in the traditional way, and load balancing is only applied to other tasks.

The conflict and tradeoff between QoS and load balancing, as described in section 5.4, is not yet a well understood problem and needs further investigation, e.g. through simulation followed by analytical work. Integrating QoS and load balancing techniques requires a well defined architecture and design/analysis of the interactions. DySCAS [2] is studying this integration for automotive systems.

A further challenge in applying dynamic load balancing to embedded systems is that of performance. Some of the algorithms are so simple that they clearly will perform well even in embedded systems, but that is not all to it. In addition, a very simplistic algorithm may not be efficient enough to fulfill the performance requirements for the load balancing algorithm. The actuation of the rebalancing should also be efficiently performed. Clearly, some approaches to actuation, such as process migration, are only relevant to more powerful systems.

5 ACKNOWLEDGEMENT

This work was funded within the DYSCAS project part of the 6th framework program "Information Society Technologies" of the European Commission. Project number: FP6-IST-2006-034904.

6 REFERENCES

[1] K-E. Årzén, A. Cervin, T. Abdelzahler, H. Hjalmarsson, A. Robertsson, *Roadmap on Control of RealTime Computing System*, EU/IST FP6 ARTIST NoE, Control for Embedded Systems Cluster

[2] DySCAS project webpage, http://www.dyscas.org

[3] DySCAS Consortium, deliverable *D1.2 Scenario and System Requirements*, 2007

[4] I. Jahnich, A. Rettberg, *Towards Dynamic Load Balancing for Distributed Embedded Automotive Systems*, at International Embedded Systems Symposium, Irvine, CA, USA, May 29 – June 1 2007

[5] G. Buttazzo, M. Velasco, P. Marti, *Quality-of-Control Management in Overloaded Real-Time Systems*, IEEE Trans. on Computers, Vol. 56, Issue 2, pp, 253-266, Feb. 2007

[6] M. Persson, T. Naseer Qureshi: *Survey on Dynamic Load Balancing in Distributed Computer Systems*, Internal Technical Report, KTH, 2008

[7] V. Kumar and A. Grama: *Scalable Load Balancing Techniques for Parallel Computers*, in Journals of Parallel and distributed computers, vol. 22, p. 60, 1994

[8] M. Willebeek-LeMair and A. Reeves, *Strategies for Dynamic Load Balancing on Highly Parallel Computers.* IEEE Trans. on Parallel and Distributed Systems, vol 4, pp 979-993, 2004

[9] Mosix project webpage, http://www.mosix.org

[10] N. Widell, *Migration Algorithms for Automated Load Balancing*, Proceedings, Parallel and Distributed Computing and Systems, Boston, 2004

[11] T. Schenekenburger and G. Rackl. *Implementing dynamic load distribution strategies with Orbix*, Proceedings, IEEE Information Survivability Conference, SC, 2000

[12] S. Lin and G. Manimaran, *Douple-loop feedback-based scheduling approach for distributed real-time systems*, in Proc. Conference on High Performance Computing (HiPC), pp. 268-278, Hyderabad, India, Dec. 2003

[13] C. Lu, X. Wang, and X. Koutsoukos, *End-to-End Utilization Control in Distributed Real-Time Systems*, the 24th IEEE International Conference on Distributed Computing Systems (ICDCS 2004), Tokyo, Japan, March 2004.

[14] X. Qin, H. Jiang, Y. Zhu, and D. Swanson, *A Dynamic Load Balancing Scheme for I/O-Intensive Applications in Distributed Systems*, Proceedings of the 32nd International Conference on Parallel Processing Workshops (ICPP Workshop 2003), Oct. 6-9, 2003

[15] Y-T. Liu, T-Y. Liang, C-T. Huang, C-K. Shieh, *Memory Resource Considerations in the Load Balancing of Software DSM Systems*, Proc. of the 2003 ICPP Workshops on CRTPC, p. 71-78.

[16] K–M. Yu, S. J.-W. Wu, T-P. Hong, *A load balancing algorithm using prediction*, Proceedings of the 2nd AIZU International Symposium on Parallel Algorithms / Architecture Synthesis, p.159, March 17-21, 1997

[17] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, S. Zhou, *Process Migration*. In: ACM computing surveys, vol. 32, pp. 241-299, 2000.

[18] W-O. Yoon, Jin-Ha, S-B. Choi, *Dynamic Load Balancing Algorithm using Execution Time Prediction on Cluster Systems*, Proc. 2002 Int. Technical Conference on Circuits/Systems, Computers and Communications, vol. 1, pp. 176-179, July 2002.

[19] F. Douglis and J. Ousterhout, *Transparent Process Migration: Design Alternatives and the Sprite Implementation*, Software – Practice and Experience, vol. 21, pp. 757-785, 1991

4. Design and Modeling

Flexible User – Centric Automation and Assistive devices

J. W. S. Liu, C. S. Shih, T. W. Kuo, S. Y. Chang, Y. F. Lu and M. K. Ouyang

Abstract—**User-centered assistive and automation devices (UCAAD) must be flexible (i.e., configurable, customizable and adaptive). We are concerned with how to model, architecture and build flexible UCAAD and how to ensure such a device never causes any harm and works acceptably well as it adapts to user's condition and needs. This paper presents our approaches, progress to date and future plans**.

I. Introduction

This paper describes our work on technologies for building low-cost, high-quality user-centric automation and assistive devices and services, sometimes called UCAAD for short here. The devices are targeted for elderly individuals, as well as people who are chronically ill or functionally limited. Some are primarily devices of convenience designed to enhance the quality of life and self-reliance of their users. Many UCAAD can also serve as point-of-care tools when care becomes necessary. Examples include smart devices and assistive robots described in [1-6]. Aging global population has led increasing demands for these devices, and technology advances have made a broad spectrum of them viable.

By a UCAAD being user-centric, we mean specifically that its purpose is to compensate for the user's skills and weaknesses and that its service can adapt according to the user's condition and needs. As an example, a smart walker may initially be used to enhance of the user's physical dexterity but can adapt to provide the user with stability and mobility. The control exerted on the legs or hands of the user by a rehabilitation device designed to help the user relearn walking or grasping adapts as the user learns and improves and hence requires less help. While users of machine-centric devices (e.g., autopilot or factory automation tools) have essentially no choice but to rely on their devices and hence are willing to be rigorously trained, typical user-centric devices are for discretionary use [7]. It is not practical to require their users more than minimal (or any at all) training.

In addition to being adaptable, a UCAAD must be easily configurable and customizable, not only by technicians but also by users themselves. Here, we refer to configurability, customizability and adaptability collectively as *flexibility*. How to model, architecture and build the devices so that they are flexible is one of the important problems in building UCAAD. Another problem is how to make sure that the symbiotic system [8] consisting of the user and device works as desired. This problem arises from the fact that most of our devices are semi-automatic: Rather than doing everything automatically for the user, the device may rely on the user to perform some mission-critical functions, and some functions may migrate between the device and the user as the device adapts to changes in user's need and ability. We want to make sure that the system (user and device) stays *safe and sound*, meaning that it never does any harm and all unavoidable errors are either recoverable or tolerable. This problem is made more challenging by the fact that not only the users may be untrained but also their skills vary widely among the user population and for an individual user over time.

Our focus has been on these problems. A major thrust has been directed towards system architecture, components, platform and tool that support the workflow approach [9] to building flexible automation and assistive devices. Our work on safe and sound UCAAD focuses on an aspect that is unique for semiautomatic devices like UCAAD: One way to keep the system safe and sound is to instrument the device so it can effectively monitor user actions and prevent the user(s) from causing unsafe operations and intolerable faults. For this, we need techniques and tools.

Following this introduction, Sections II and III gives an overview of our work and thoughts on these problems: They present our approaches in the context of closely related work. Section 4 is a brief summary.

II. WORKFLOW APPROACH TO FLEXIBILITY

We have adopted the workflow paradigm [9] as a way to make flexibility a primary consideration in the model, architecture and design of UCAAD and the framework for their development and evaluation. The workflow approach has been widely used in enterprise computing systems, where automated business processes are defined in terms of workflow graphs. A *workflow graph* looks like a task graph [10]: Both types of graphs are directed. Each node represents an *activity*, called a job or task in real-time systems literature. Each directed edge represents a transition from one activity to another. Workflow graphs are executable. Sequencing and synchronization between activities are carried out by a *workflow engine* on behalf of the application process(s) which provides executables to implement the activities in the graphs. Today, there are standard process definition language and execution languages (e.g., [11, 12]), as well as matured engines and tools for defining, building and executing workflow applications (e.g., [13, 14]). They enable business process domain experts with little or no information technology expertise to tailor complex business processes to

J. W. S. Liu is with Institute of Information Science, Academia Sinica, 128 Academia Road, Section 2, Nankang, Taipei, Taiwan 115 (Phone: +886-919-36-4433; email: janeliu@iis.sinica.edu.tw)

C. S. Shih, T. W. Kuo, S. Y. Chang, Y. F. Lu and M. K. Ouyang are with Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan. (Email: {cshih, ktw, r94049, d93023 and d94032}@csie.ntu.edu.tw).

individual enterprises and across diverse enterprises worldwide.

A. *Workflow-Based Embedded System Architecture*

Modern engines can handle not only automated process but also activities triggered by external events and can treat manual activities by users and automated activities by hardware and software in an integrated way. Using such an engine, we can build embedded device families based on an architecture illustrated by Fig. 1(a): Only a small part of a device with this architecture is hardwired. Most of it is built from activities and workflows components. The embedded engine integrates the components by executing them and arbitrating their resource contentions as specified by the workflow graphs.

A workflow-based application can be easily configured and customized by changing the workflow graphs in it and by invoking different components for activities in the graphs. As an example, Fig. 2 shows a workflow-based implementation of two service robots: an automated vacuum cleaner in (a) and a medication transporter in (b). The table at the bottom lists basic activities, i.e., activities provided by the engine, including start, end, while, wait for event, and so on [14]. The dotted box on the left encircles components and user(s) that generate interrupts and asynchronous events. The box in the middle shows workflows triggered by events. They implement edge maneuver, contact maneuver, and track maneuver behaviors. Being relatively independent of robot hardware, the activities in this box may run on a computer. The dotted box on the right encircles robot hardware specific components, activities (behaviors) that generate commands to move the robot. Wait for the move in the middle box is the activity that coordinates the behaviors and moves the robot accordingly.

We note that the devices contain the same (configurable) activities. Their workflows, hence the graphs defining them, differ, however. We can also implement a Sumo toy using the workflow graphs in Fig. 2(a) but different activities (e.g., by using Back and hit instead of Back and random move for contact maneuver).

B. *Executable Operational Specification*

In addition to using workflows as components of UCAAD, we use workflows for two other purposes. Fig. 1(b) illustrates a common usage: using workflows for integration of embedded devices together and with support infrastructures. An example is the framework for integrating smart medication dispensers at the end-user level of the tool chain for medication use process with upper level tools [17].

Fig. 1(c) shows a simulation environment designed to support the design, development and evaluation of devices based on SISARL component model [16]. According to this model, a device has a resource (structural) view specification and an operational (behavior) view specification. The former tells us how the device is built. Existing component models typically support this view. SISARL component description language for this purpose is a variation of nesC language [18]. It allows us to specify components used to build the device and interconnections of the components. The preprocessor selects components and links them accordingly and translates the specification into standard C code.

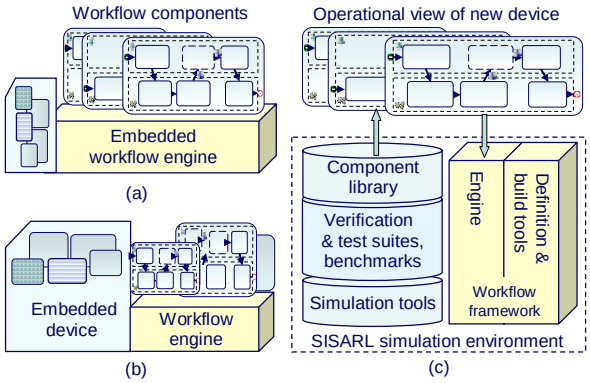
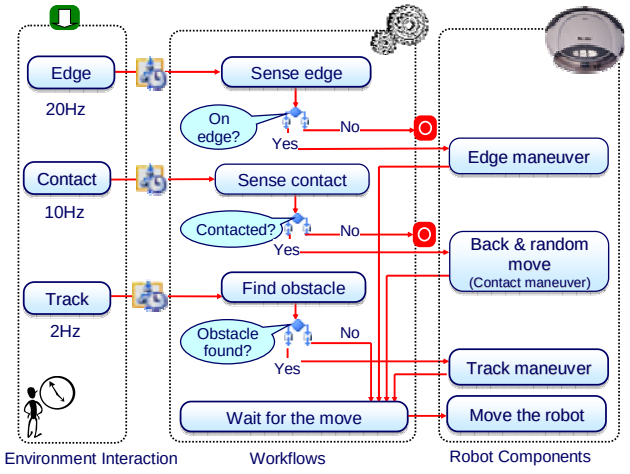
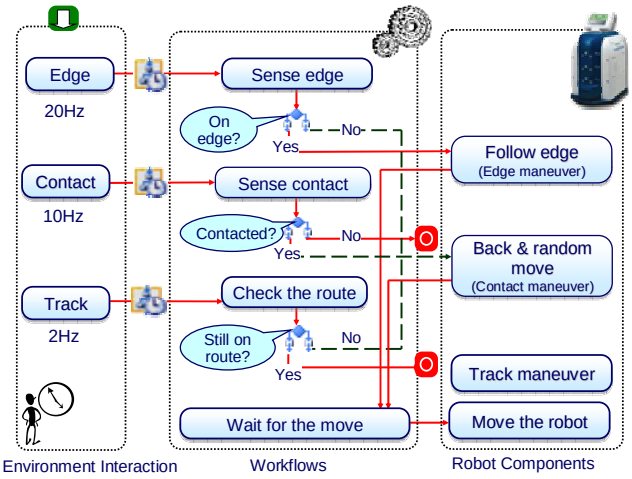


Fig.1. Workflow-based architectures and tools



(a)



(b)

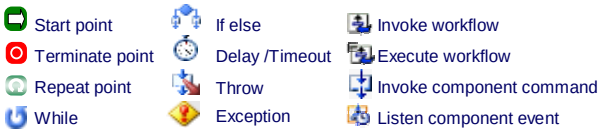


Fig.2. Workflow-based service robots (a) Automated vacuum cleaner, (b) Medication transporter

The operational (behavior) view specification of a device is written in terms of workflow graphs. It tells us how the device works and what actions it expects from its user(s). Specifically, the workflow graphs in the specification define the actions of the user, work by the device, and collaborations between the user and the device.

As an example, the workflow graph in Fig. 3 is a part of the operational specification of a personal medication dispenser [1, 3]. The device is used to manage medications of a single user and help the user administer his/her medications at home. The dispenser provides a number of sockets to hold containers of medications under its care and relies on the user to plug all containers into empty sockets during initialization. The location of each container is given by the number of the socket holding it. Each container is tagged with the RF ID of the medication in it. The dispenser acquires the id of the medication in the container by reading the tag.

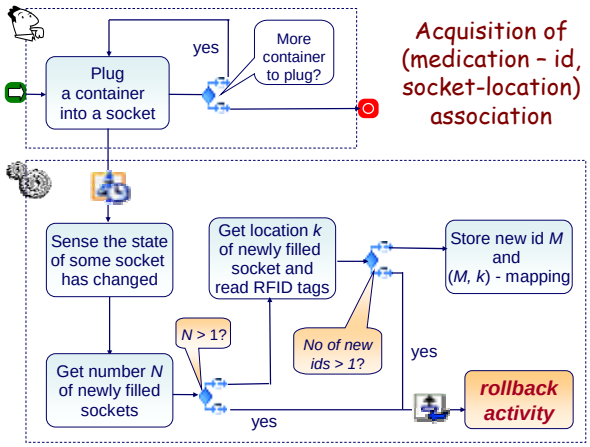


Fig. 3. Operational specification (initialize containers)

The workflow graph describes the collaboration between the user and the device in this task: The underlying assumption made by the dispenser is that the containers are plugged into empty sockets one at the time. When it senses that a socket (say socket *k*) just changed from being empty to non-empty, it records the socket number *k*, reads the RF id tags on all containers already plugged in, and discovers a new medication id (say *M*). It then concludes that the new id *M* is that of the medication in the container at location *k* and creates the new medication-id-socket-location association (*M*, *k*), which is maintained until the container is unplugged. Clearly, the dispenser must correctly associate the location and id of every medication. It can accomplish this if its assumption is valid. This is why whenever the dispenser finds that the states of more than one socket have changed from empty to non-empty when it services a socket state-change interrupt or reads more than one new tag, it prompts the user to carry out a rollback activity: The user unplugs the containers in the sockets involved and then plugs them back in one at a time.

By writing operational specifications of a new or revised device in terms of workflows, which are executable, we can simulate, emulate and experiment with its operations and its interaction with the user throughout the design, development and quality assurance process. The simulation environment shown in Fig. 1(c) is for this purpose.

C. *Light-Weight Engine and Embedded XPDL*

To date, few embedded devices and systems are built on workflow-based architecture. A major reason is the lack of engines suited for embedded systems such as UCAAD. Memory space and processor bandwidth of these devices are typically not scarce. Still even they cannot accommodate the resource demands of existing engines designed to run in J2EE or .Net environments. Another reason is that the standard language XPDL (XML Process Definition Language) [11] is too rich than needed for definitions of embedded workflows on one hand. On the other hand, it lacks many important elements. Examples include means for specifying timing constraints and basic activities (e.g., behavior coordination) for some applications (e.g., behavior-based robots). For flexible applications, the restriction to static workflow graphs forces us to provide *a priori* workflows for different configurations and adaptive behaviors. The ability to modify workflow graphs dynamically at runtime is also a needed feature named by the scientific computing community [19].

We are addressing these issues. The light-weight workflow engine (LIWWE) [15] developed for SISARL is a first step. We are enhancing the engine and plan to release it under the GPL license in the near future. SISARL-XPDL is a subset of XPDL. Scaling it down is straightforward. Expanding it to provide some of the features mentioned above while avoiding (or minimizing) incompatibility with a widely used standard is another matter.

Data obtained from a case study on LIWWE comparing memory footprint and runtime overheads of workflow implementation of a medication dispenser module with that of a customized, hardwired version indicate that runtime overhead is negligibly small. The engine added almost 50% extra footprint, however. We expect this result since the medication dispenser, like many other UCAAD, is not compute-intensive and its code is small compared with the engine. Obviously we need to do broader and deeper studies.

III. Monitoring Techniques and Tools

As stated earlier, we are concerned with how to keep the device and its user(s) safe and sound as parts of the symbiotic system changes. There have been extensive works on user models, formal verification methods, and runtime monitoring techniques [20], and so on for this purpose. A typical assumption is that the device is used as intended. This assumption is usually valid for machine-centric devices with their well trained users, but is not valid for UCAAD without help for their users. We are exploring the idea of having the device monitor user actions at runtime and help to prevent misuses that may have serious consequences.

The example in Fig. 3 illustrates this idea. In this case, the device can determine whether the underlying assumption for

its correct operation is valid by monitoring its own states. When it detects user actions that lead to violation of the assumption, it works with the user to correct the situation.

Fig. 4 shows a general runtime monitoring environment for this purpose. The device may have control mechanisms that do not interact with the user. They are included in the box labeled Plant. The symbiotic system is modeled as a feedback loop which takes user actions as input. User actions are partitioned into three types: The device guarantees to work as intended if the user actions are of the Assumed type. A device well designed for usability should have only few actions of the Disallowed type. A disallowed action may cause serious malfunctions. In the case of medication dispenser, actions involving plugging containers are either assumed or disallowed. An easy to use device may allow many Tolerable actions. The device works in a degraded manner in this case. An example is a semi-automatic smart pantry [4]. When the users act as assumed, it can order the right supplies to be delivered just in time. A tolerable action can cause the pantry to fail in a placing an order but is aware of the failure and can warn the user in time. A disallowed action causes wrong supplies to be delivered, incurring expenses and annoyance.

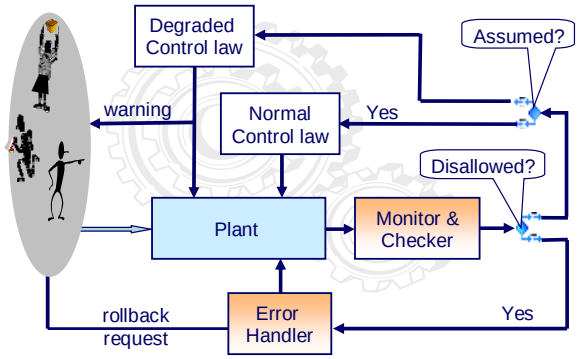


Fig. 4. Runtime monitoring of user actions

Needless to say that such a monitoring scheme works only if device state changes caused by user actions (or at least disallowed actions) are observable. To be effective, we also want the device to be able to rollback when a disallowed action is observed, and in this sense, to be controllable. This is indeed the case for the simple example in Fig. 3. In the case of smart pantry, the device is not always able to distinguish types of user actions. For such devices, runtime monitoring does not work.

Thus far, we have been looking at this problem in an ad hoc manner, one device at a time. General design principles such as disallowed actions should always lead to observable state changes cannot be easily translated into general design guidelines and rules and working methods and tools. Without them, we are forced to play it safe. This typically means that we restrict the adaptability and sacrifice usability of the device so we are sure it is always safe and often sound.

IV. Summary

Earlier sections described our approaches to building flexible UCAAD and ensuring their safe and sound operations. The work on applying workflow-based design and implementation to achieve flexibility is well underway. In contrast, the idea of having the device monitor user actions is yet to be developed into working techniques and tools. We are collaborating with Professors M. Kim and B. Y. Wang from the formal method community in this effort.

Acknowledgment

. This work is partially supported by the Taiwan Academia Sinica thematic project SISARL.

References

[1] J. W. S. Liu, C. S. Shih, P. H. Tsai, H. C. Yeh, P. C. Hsiu, C. Y. Yu, and W. H. Chang, "End-User Support for Error Free Medication Process," *Proceedings of High-Confidence Medication Device Software and Systems and UPnP Workshop*, IEEE Press, June 2007

[2] T. S. Chou and J. W. S. Liu, "Design and Implementation of RFID-Based Object Locator," *Proceedings of IEEE RFID Technologies*, March 2007

[3] P. H. Tsai, H. C. Yeh, C. Y. Yu, P. C. Hsiu, C. S. Shih and J. W. S. Liu, "Compliance Enforcement of Temporal and Dosage Constraints," *Proceedings of IEEE Real-Time Systems Symposium*, December 2006.

[4] C. F. Hsu, H. Y. M. Liao, P. C. Hsiu, Y. S. Lin, C. S. Shih, T. W. Kuo, and J. W. S. Liu, "Smart Pantries of Homes," *Proceedings of IEEE International Conf. on Systems, Man and Cybernetics*, October 2006.

[5] Y. Kaneshige, M. Nihei, and M. G. Fujie, "Development of new mobility assistive robot for elderly people with body functional control," *Proceedings of IEEE/RAS-EMBS International Conference on Biomedical Robotics and Biomechatronics*, February 2006.

[6] C. H. Lin, Y. Q. Wang and K. T. Song, "Personal assistant robot," *Proceedings of IEEE International Conf. on Mechatronics*, July 2005.

[7] J. Grudin, "Three faces of human-computer interaction," *IEEE Annals of the History of Computing*, Vol. 27, No. 4, 2005.

[8] S. Coradeschi and A. Saffiotti, "Symbiotic robotic systems: humans, robots and smart environments," *IEEE Intelligent Systems*, 2006.

[9] Workflow definition, http://en.wikipedia.org/wiki/Workflow

[10] J. W. S. Liu, *Real-Time Systems*, Chapter 3, Prentice Hall, 2000.

[11] XPDL (XML Process Definition Language) Document, http://www.wfmc.org/standards/docs/TC-1025_xpdl.2.2005-10-03.pdf, October 2005.

[12] BPEL (Business Process Execution Language), http://en.wikipedia.org/wiki/BPEL

[13] WfMC: Workflow Management Coalition, http://www.wfmc.org/,

[14] Windows Workflow Foundation, http://msdn2.microsoft.com/en-us/netframework/aa663328.aspx

[15] S.-Y. Chang, Y.-F. Lu, T. W. Kuo, and J. W. S. Liu, "The design of a light-weight workflow engine for embedded systems," *Proceedings of RTSS Workshop on Software and Systems for Medical Devices and Services*, December 2007.

[16] T.Y. Chen, P. H. Tsai, T. S. Chou, C. S. Shih, T. W. Kuo, and J. W. S. Liu, "Component model and architecture of smart devices for the elderly," to appear in *Proceedings of the 7th Working IEEE/IFIP Conference on Software Architecture*, February 2008.

[17] H. C. Yeh, C. S. Shih and J. W. S. Liu, "Integration framework for medication use process," *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, October 2007.

[18] P. Levis, TinyOS Programming, http://www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf

[19] Y. Gil, *et al*, "Examining the challenges of scientific workflow," *Computer*, December 2007.

[20] M. Viswanathan and M. Kim, "Foundations of the run-time monitoring of reactive systems," *Theoretical Aspect of Computing*, Vol. *3407/2005*, Springer

Towards an Integrated Planning and Adaptive Resource Management Architecture for Distributed Real-time Embedded Systems

Nishanth Shankaran[†], John S. Kinnebrew[†], Xenofon D. Koutsoukos[†],
Chenyang Lu[‡], Douglas C. Schmidt[†], and Gautam Biswas[†]
[†]Dept. of EECS Vanderbilt University, Nashville, TN
[‡]Dept. of Computer Science and Engineering, Washington University, St. Louis, MO

Abstract

Distributed real-time embedded (DRE) systems often operate in open environments where operating conditions, input workload, and resource availability cannot be accurately characterized a priori. Some DRE systems, such as NASA's Magnetospheric Multi-Scale (MMS) mission, perform sequences of heterogeneous data collection, manipulation, and coordination tasks to meet specified objectives/goals. These systems are also required to operate with a high degree of local autonomy and adaptivity as new data is acquired and analyzed, and as environmental conditions change. Key challenges in managing open DRE systems include effective planning and online management of system resources to accommodate for changing mission goals, environmental conditions, resource needs, and resource availability. This paper explores the benefits of an integrated planning and adaptive resource management architecture that combines decision-theoretic planning with adaptive resource management to control and ensure efficient functioning of open DRE systems.

1 Introduction

Emerging trends and challenges. *Distributed real-time and embedded* (DRE) systems form the core of many mission-critical applications that operate in dynamic and uncertain environments. An example of such an application is NASA's Magnetospheric Multi-Scale (MMS) mission [1]. As with many other DRE systems, the MMS mission system operates in *open* environments where operating conditions, input workload, and resource availability cannot be accurately characterized *a priori*. Moreover, overall mission goals and analysis methods may change as new data and information is obtained or scientists that operate the mission propose new goals/objectives.

Conventional resource management approaches, such as end-to-end task allocation and scheduling [2], are designed to manage system resources and maintain QoS in *closed* environments where the operating conditions, input workloads, and resource availability are known in advance and do not change much at runtime. These solutions, however, are insufficient for complex DRE systems such as the MMS mission system that operate in open environments since the autonomous operation of these systems require them to *adapt* to a combination of (1) changes in mission requirements and goals, (2) changes in operating conditions, (3) loss of resources, and (4) drifts and fluctuations in system resource utilization and application QoS at runtime.

Adaptation in such complex DRE systems can be performed at various levels, including at the (1) *system level* by deploying/removing end-to-end applications to/from the system, (2) *application structure level* by adding, modifying, and removing components or sets of components associated with one or more applications executing in the system, (3) *resource level* by reassigning resources to application components to ensure their timely completion, and (4) *application parameter level* by fine-tuning configurable parameters (if any) of application components. These adaptation decisions, however, are tightly coupled as they directly or indirectly impact the utilization of system resources and QoS of the system, which ultimately defines the success of the overall mission. It is therefore necessary that adaptations at various levels of the system are performed in a stable and *coordinated* fashion.

Solution approach → Integrated planning and adaptive resource management. To address the planning and adaptive resource management needs of open DRE systems, we developed the *Spreading Activation Partial Order Planner* (SA-POP) [3] and the *Resource Allocation and Control Engine* (RACE) [4], respectively. SA-POP combines decision-theoretic task planning with resource constraints for DRE systems operating in uncertain environments to produce effective executable component sequences that can achieve current mission goals, including desired QoS, with available system resources. RACE provides a customizable and configurable adaptive resource management framework that enables open DRE systems to adapt to fluctuations in utilization of system resources and QoS specifications.

Although SA-POP provides efficient re-planning capabilities in response to changes in the goals of applications, it cannot deal with uncertainties and fluctuations in task execution times and resource availability. To enhance the robustness of the system, RACE support adaptive resource management at two levels: (1) task reallocation in response to significant changes in component sequences caused by application re-planning, and (2) feedback-control-based

task rate adaptation. Note that the two adaptation strategies are complementary to each other. Task reallocation provides coarse-grained adaptation to application changes, while task rate adaptation provides fine-grained adaptation to resource fluctuations.

Our experience developing a MMS mission prototype [5] showed that although SA-POP and RACE performs effective adaptive resource management, neither SA-POP nor RACE, individually, have sufficient capabilities to efficiently manage and ensure proper functioning of such complex DRE system. To meet the challenge of such open DRE systems, we propose an integrated planning and adaptation resource management architecture.

2 An Integrated Planning and Resource Management Architecture

This section first describes SA-POP, RACE, and our integrated planning and adaptive resource management architecture. It then shows how we applied this integrated architecture to address the QoS needs of our MMS system prototype.

2.1 Overview of SA-POP

Open DRE systems can operate more efficiently and effectively by incorporating some degree of autonomy that allows them to adapt to changing mission goals and environmental conditions. SA-POP provides a planning approach for dynamic generation of component-based applications that operate with limited resources in uncertain environments. The architecture of SA-POP is shown in Figure 1.

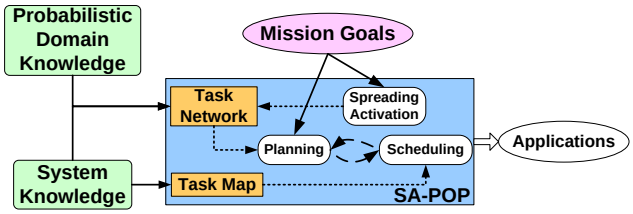


Figure 1. SA-POP Architecture

Given one or more goals specified by a software agent or system user, SA-POP takes into account current conditions to generate partial order plans with high expected utility [3]. Goals are specified as desired conditions with associated utility values. SA-POP uses a spreading activation mechanism [6] to generate expected utility values for individual tasks that contribute to achieving the specified goal conditions. Guided by these expected utility values, SA-POP's planning algorithm generates a set of task sequences that together achieve the goal while meeting all resource and time constraints.

For SA-POP to choose appropriate tasks to achieve a goal, it must know which preconditions must be satisfied for each task, its input/output data streams (if any), and the pertinent effects that result from its operation. Uncertainty as to whether tasks will produce the desired output or effects is captured via conditional probabilities associated with the preconditions and effects of a task. Together, these input/output definitions, preconditions/effects, and related conditional probabilities define the *functional signature* of the task.

To ensure applications and their scheduled executions do not violate resource and time constraints, SA-POP also requires knowledge of the expected resource consumption and execution time for each component/configuration that can implement a task, *i.e.*, its *resource signature*. The planning of SA-POP uses the task function signatures and associated component resource signatures to dynamically generate applications most suited to local conditions and resource availability. It also provides a schedule of acceptable time windows for the execution of each application component with any required before-after constraints on the execution components both within and between applications. Moreover, through re-planning, SA-POP can dynamically adapt deployed applications when environmental conditions and resource usage change unexpectedly.

2.2 Overview of RACE

RACE addresses two key challenges of adaptive resource management of open DRE systems: (1) efficient online resource allocation for applications, and (2) effective system adaptation in response to fluctuations in input workload, operating conditions, and resource availability. The RACE framework decouples adaptive resource management algorithms from the middleware implementation, thereby enabling the use of customized resource management algorithms without redeveloping significant portions of the middleware or applications. To enable the seamless integration of resource allocation and control algorithms into DRE systems, RACE configures and deploys feedback control loops.

Online resource allocation using RACE. As shown in

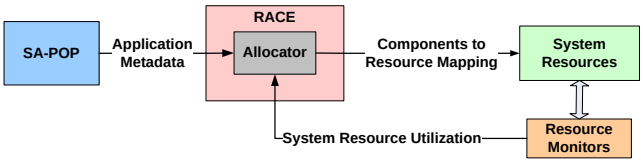


Figure 2. RACE: Online Resource Allocation

Figure 2, RACE features `Allocators` that implement resource allocation algorithms, such as multi-dimensional bin-packing algorithms [2], to allocate various domain resources (such as CPU, memory, and network bandwidth) to application components by determining the mapping of components onto nodes in the system domain. `Allocators` determine the component-to-node mapping at runtime based on *estimated* resource requirements of the components and current resource availability on the various nodes in the domain. As shown in Figure 2, input to

`Allocators` include the metadata of the application corresponding to the application generated by SA-POP and the current utilization of system resources.

Effective system adaptation. As shown in Figure 3,

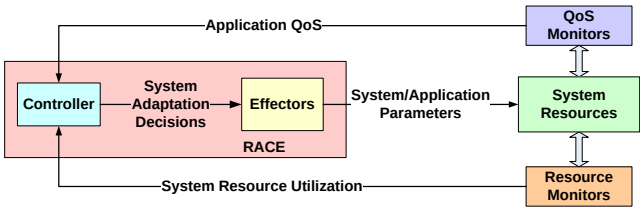


Figure 3. RACE: Online System Adaptation

RACE uses `Controllers` to implement control-theoretic adaptive resource management algorithms, such as EUCON [7], that enable DRE systems to adapt to changing operational context and variations in resource availability and/or demand. `Controllers` use the control algorithm they implement to compute system adaptation decisions to ensure that system performance and resource utilization requirements are met. Figure 3 also shows how these decisions serve as inputs to *Effectors* that modify system parameters (such as resources allocated to components, execution rates of applications, and OS/middleware/network QoS setting for components) to achieve the `Controller` recommended adaptation. RACE's `Controller` and `Effectors` work with resource monitors and QoS monitors to compensate for drifts/fluctuations in utilization of system resources and/or application QoS.

2.3 Overview of the Integrated Architecture

Our integrated planning and adaptive resource management architecture integrates SA-POP and RACE to address system management challenges of complex open DRE systems, such as the MMS mission system. Figure 4 shows the integrated SA-POP/RACE planning and adaptive resource management architecture. A set of QoS and resource monitors track system behavior and performance, and periodically update SA-POP and RACE with current resource utilization (*e.g.*, processor/memory utilization and power) and QoS values (*e.g.*, end-to-end latency and throughput). Although inputs to SA-POP and RACE include system behavior and performance metrics, SA-POP uses this information to monitor the evolution of the system with respect to its *long-term* plan/schedule for achieving given goals and to re-plan/re-schedule when necessary, whereas RACE uses this information to *fine-tune* application/system parameters in response to drifts/fluctuations in utilization of system resources and/or application QoS.

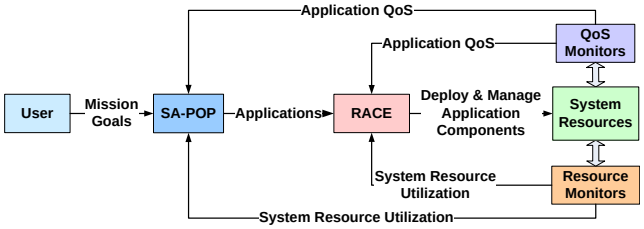


Figure 4. Integrated Architecture

Figure 4 also shows how the integrated SA-POP/RACE architecture is comprised of two hierarchical feedback loops: (1) the *inner feedback control loop* with resource and QoS monitors, RACE, and the DRE system, and (2) the *outer feedback control loop* that includes SA-POP, RACE, and the DRE system. The outer feedback loop enables a DRE system to adapt to new mission goals, major changes in resource availability (*e.g.,* loss of a satellite sensor or drastic deviations in resource consumption), and changes in environmental conditions, by performing *coarse-grained* system adaptation, such as adding/removing components deployed as part of an application and modifying the schedule for operation of components. The inner feedback loop computes *fine-grained* system adaptation decisions, such as fine-tuning application parameters (*e.g.*, execution rates) and system parameters (operating system and/or middleware QoS parameters), thereby compensating for drifts/fluctuations in utilization of system resources and/or application QoS.

2.4 Applying our Integrated Architecture to the MMS Mission System

As shown in Figure 5, our integrated planning and adaptive resource management architecture performs the following actions for the MMS mission system prototype:

1. Upon receiving a mission goal from the user, SA-POP employs integrated decision-theoretic planning to generate an application capable of achieving the provided goal, given current local conditions and resource availability.

2. After an appropriate application has been generated by SA-POP, RACE's `Allocator` allocates resources to application components and employs the underlying middleware to deploy and initialize the application.

3. RACE's `Controllers` and `Effectors` periodically compute system adaptation decisions and modify system parameters, respectively, to handle minor variations in system resource utilization and performance due to fluctuations in resource availability, input workload, and operational conditions.

4. RACE triggers re-planning by SA-POP if RACE's `Controllers` and `Effectors` are unable to adapt the system effectively to changes in resource availability, input workload, and operational conditions, *e.g.*, due to drastic changes in system's operating conditions, such as complete loss of resources. SA-POP performs iterative plan repairto modify the current application to achieve its goal even when it encounters unexpected conditions or resource availability.

Our integrated SA-POP/RACE architecture offers capabilities that: (1) efficiently handle uncertainty in plan-

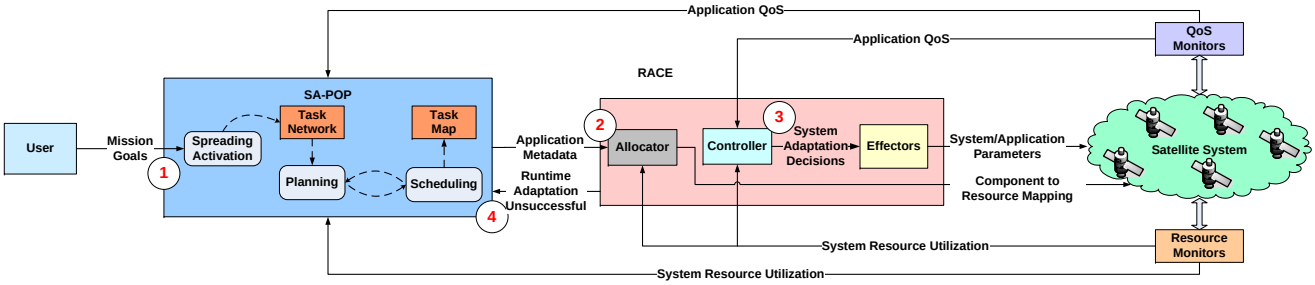


Figure 5. Applying the Integrated Architecture to the MMS Mission System

ning for online generation of applications, (2) planning with multiple interacting goals, (3) efficiently allocate system resources to application components, (4) avoid over-utilization of system resources, thereby ensuring system stability and application QoS requirements are met, even under high load conditions.

3 Concluding Remarks

The paper described how our integrated planning and adaptive resource management architecture helps address system management challenges of complex open DRE systems. By integrating SA-POP and RACE, our architecture enables open DRE systems to adapt to (1) high-level changes, such as new mission goals and drastic changes in operating conditions and (2) fluctuations in utilization of system resources and/or application QoS. Although SA-POP and RACE individually offer many features that enable the effective management of complex DRE systems, the benefits of our *integrated* architecture include:

Fast and efficient online planning and resource allocation. In our integrated architecture, SA-POP considers *coarse-grained* (system-wide) resource constraints during planning of applications. In contrast, RACE handles resource allocation optimization with *fine-grained* (individual processing node) resource constraints. The separation of concerns between SA-POP and RACE limits the search spaces in each during planning and resource allocation, thereby making the process of planning and online resource allocation faster, more effective, and more efficient.

Fast and efficient runtime system adaptation. Since planners such as SA-POP are designed to be domain independent, adaptation decisions computed by the planner are coarse-grained, such as adding/removing components deployed as part of an application and modifying the schedule for operation of some components. Moreover, system adaptation by SA-POP may involve significant computation for re-planning, as well as re-deployment of applications. System adaptation by planners are therefore more suitable for significant changes in system workload, resources, objectives, and/or operating conditions that occur at lower frequency. Conversely, RACE employs control-theoretic adaptive resource management algorithms that performs system adaptation and have minimal or low overhead. System adaptation by RACE is thus more suitable for more frequent fluctuations in workload and resources, such as changes in application resource utilization and minor disturbances from external sources.

As a part of our ongoing work, we are empirically validating and evaluating the stated advantages of integrating online planning (SA-POP) and an adaptive resource management framework (RACE).

References

[1] S. Sharma and S. Curtis, "Magnetospheric Multiscale Mission," in *Nonequilibrium Phenomena in Plasmas*. Springer Verlag, 2005, pp. 179–195.

[2] J. W. S. Liu, *Real-time Systems*. New Jersey: Prentice Hall, 2000.

[3] J. S. Kinnebrew, A. Gupta, N. Shankaran, G. Biswas, and D. C. Schmidt, "Decision-Theoretic Planner with Dynamic Component Reconfiguration for Distributed Real-time Applications," in *The 8th International Symposium on Autonomous Decentralized Systems (ISADS 2007)*, Sedona, Arizona, Mar. 2007.

[4] N. Shankaran, D. C. Schmidt, Y. Chen, X. Koutsoukous, and C. Lu, "The Design and Performance of Configurable Component Middleware for End-to-End Adaptation of Distributed Real-time Embedded Systems," in *Proc. of the 10th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC 2007)*, Santorini Island, Greece, May 2007.

[5] D. Suri, A. Howell, N. Shankaran, J. Kinnebrew, W. Otte, D. C. Schmidt, and G. Biswas, "Onboard Processing using the Adaptive Network Architecture," in *Proceedings of the Sixth Annual NASA Earth Science Technology Conference*, College Park, MD, Jun. 2006.

[6] S. Bagchi, G. Biswas, and K. Kawamura, "Task Planning under Uncertainty using a Spreading Activation Network," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 30, no. 6, pp. 639–650, Nov. 2000.

[7] C. Lu, X. Wang, and X. Koutsoukos, "Feedback Utilization Control in Distributed Real-time Systems with End-to-End Tasks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 6, pp. 550–561, 2005.

Designing Reconfigurable Component Systems with a Model Based Approach

Brahim Hamid
CEA LIST
email:brahim.hamid@cea.fr

Agnes Lanusse
CEA LIST
email:agnes.lanusse@cea.fr

Ansgar Radermacher
CEA LIST
email:ansgar.radermacher@cea.fr

Sébastien Gérard
CEA LIST
email:sebastien.gerard@cea.fr

Abstract

Dynamic reconfigurability of systems is of particular importance when considering dependability issues. In this paper we consider the utilization of such techniques in the context of fault-tolerance. Indeed, one way to achieve fault management is to use reconfiguration mechanisms. In this paper we propose a model driven approach to help specify and configure reconfigurability issues.Within this development process, component application and reconfiguration properties are declaratively specified at model level and are transparent for the component implementation. An application is described using UML and specialized extensions: QoS &FT, D&C profiles from the OMG. From this model, we generate descriptor files for a framework based on the CORBA component model and configure specific infrastructure components devoted to reconfiguration management. The approach is illustrated on a simple example.

Key words: *Distributed applications, CORBA Component Model, Connector, Fault tolerance, Profile, Reconfiguration, UML.*

1 Introduction

The "Inflexion" project [1] aims at providing a flexible infrastructure framework in the domain of real-time embedded applications. The framework promotes an infrastructure based on the CORBA Component model (CCM) [5] and its Component/Container paradigm. It extends this model with the concept of Connector that provides an abstraction for connections. The project provides also tools to configure and generate efficient containers – from a description of deployment specification at a model level. This work has already been successful in previous projects, namely the IST project Compare (*http://www.ist-compare.org*).

We present an ongoing work which extends this framework with support for the management of fault-tolerant architectures. It is based on the definition of a run-time framework and on code generation tools and an accompanying modeling process. The former offers transparent management of fault-tolerance (mainly fault-detection and redundancy mechanisms), the latter supports the application designer in the development process. In the scope of this paper, we focus on reconfiguration support at model level.

In the context of fault-tolerance, the failure of a node has to be properly dealt with. Whereas critical components are actively replicated and have to continue without interruption, other parts of an application may be *re-configured* in order to adapt to the node-failure. We study reconfiguration at multiple layers within a model driven engineering approach, including the model level and required runtime support from the execution platform.

A major reason for using CCM is its separation of business code from non-functional or service code within a container. Non-functional services support the reconfiguration process by blocking for instance the access to ports during reconfiguration in order to maintain a consistent state (avoid messages loss).

The paper is organized as follows: In the next section we present some background related to component model, the connector extension and the modeling support for adaptive quality of service design. In section 3, we present our approach to deal with reconfiguration at a model level. Section 4 describes briefly the proposed framework to implement reconfiguration in the context of fault-tolerance. In the following section, we propose our process to design reconfigurable applications. We review in Section 6 some related works. Finally, Section 7 concludes the paper with short

[1]This work has been performed in the context of the Usine Logicielle (software-factory) project of the System@tic Paris Région Cluster (http://www.usine-logicielle.org).

discussion about future works.

2 Background

In this section, we outline two different aspects: the component platform, namely the CORBA Component model and an extension of it and the modeling support for adaptive quality of service design.

2.1 CCM Connectors – non-standard interaction mechanisms

In the context of fault tolerance, a connection with a replicated component should perform the group communication. Whereas this could be done with standard CCM and a specific CORBA implementation supporting group communication, it would be impossible to configure and control it (in case for instance of node failures) from standard CCM.

This motivates the introduction of what we call *connectors* that offer the possibility to use a specific interaction semantic and multiple implementations of this semantic within CCM [10]. The basic idea of this extension is that the semantics of an interaction is defined by a certain port type and that one or more connectors can support this port type. The port types are already fixed at component design time, whereas the choice of a connector (and a specific implementation) is not fixed until application deployment.

A connector has certain similarities with a component. A main difference is that it is a fragmented entity: since the connection between a (part of a) component and its connector is always local, the connector needs to be split into a number of *fragments* which represent the connector in the address space of its users.

Since a connector fragment is responsible for incoming and outgoing messages, it is an ideal place for the integration of code that transparently interacts with a group of replicas. In addition, the connector can *intercept* messages and block these until a reconfiguration is done - thus avoiding message loss.

2.2 Modeling Support for Adaptive Quality of Service Design

Model Driven Engineering is becoming more and more successful in software engineering. Though, primarily used in information systems, it is particularly well suited to the domain of real-time embedded applications since it puts in action the separation of concerns promoted by the OMG standard Model-Driven Architecture™ (MDA). But until now, there was a lack of tools to support such methodologies and approaches in this domain.

In this context the "Inflexion" sub-project takes advantage of "Usine logicielle" environment and on recent advances in modeling non-functional issues offered mainly by QoS & FT [7] and more recently by MARTE(Modeling and Analysis of Real-Time Embedded systems) [8] OMG standards to support dependability modeling issues in the development of Component Based systems. Advances in AADL (Architecture Analysis and Design Language) and its recent extensions (behavior and error annex) devoted to dependability oriented modeling [3] have also been taken into account.

Adaptive systems implement generally decision strategies for dynamic reconfiguration depending on criteria defined during design. These can be expressed by means of constraints (defining contracts) on services offered or required by components, these constraints can be related to different quality levels and/or to different phases during the application life. Promoting a unified framework to specify declaratively such quality related concepts is thus a major issue. Several proposals have been issued until now :

- One of the most achieved work in this direction was the QoS&FT profile. In this profile, many aspects of Quality of Service have been investigated and modeling features have been proposed that are directly in relation with the modeling of adaptive systems. The profile helps modeling *QoSLevels*. These levels refer to states where a certain degree of quality can be achieved. A component remains in a *QoSLevel* as long as the constraints attached to this level are verified. When this is no longer the case, a *QoSTransition* is fired. This *QoSTransition* contains a set of *adaptationActions* that determine the reconfiguration process that must be achieved. This meta-model of adaptability is pretty much general and must be detailed in order to support adaptive systems development methodology. However, the framework fits well to most existing adaptive systems.

  However, the concept of *QosLevel* is not suitable for modeling operational *modes* of an application. An operational mode is a global application mode, in case of an aerospace application it may for instance correspond to a start and in-flight phase of a launch vehicle which have quite different demands. Modes and transitions between these are usually specified by means of some kind of automaton.

- MARTE profile provides a more general framework to specify non-functional properties (generalization of *QoSChracteristics*), libraries of predefined types devoted to RTE domain and specific sub-profiles have been defined to support Schedulability and Performance Analysis. These concepts can be used (but are not explicitly devoted) to express *QoSConstraints*. In MARTE profile, the notion of operational modes (close to AADL concept) has been introduced and is attached to *RT-Units* in a subprofile named RTEMoCC

which is devoted to the modeling of concurrent systems using high level abstractions. Operational modes is described by a UML state machine which describes operational modes of the application. We are currently developing a profile for dependability analysis compatible with MARTE that extends this profile for mode management (see next section).

- In AADL, mode management is explicitly supported. Modes represent alternative operational states of a system or component. Modes and mode transitions are represented thanks to state machines. Each mode corresponds to a particular system configuration.

3 Modes management modeling

In the meta-model we have developed to capture multi-mode management requirements, we consider modeling operational modes of an application at a coarse grain level. Modes can correspond to two types of preoccupations: 1) modeling multi-phase applications such as aerospace applications and 2) modeling degraded mode management in presence of partial failure of a system. The major features of this model are described in the next figure. The concepts supported are relatively limited. From this meta-model, a derived profile has been implemented and integrated within the PapyrusUML tool. The specifications entered in the tool are used to configure specific components from the infrastructure and to enrich code generation (see section 5).

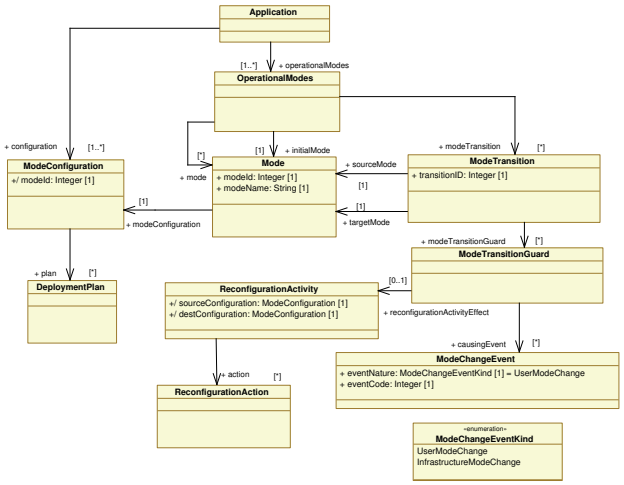


Figure 1: Our proposed MetaModel for Mode Management

The *OperationalModes* StateMachine describes the different states and transitions defined as valid for the application. In this approach we are considering statically predefined configurations. A configuration is given by a particular deployment plan that describes component instances, their configuration and allocation to a node along with the connections between instances. A transition from one operational mode to another will switch the system into the new target mode. A mode transition is thus described by its *sourceMode*, its *targetMode* and a *modeTransitionGuard*. Each mode is associated with a *modeConfiguration* that relates the mode to a deployment plan. A *modeTransitionGuard* relates a mode *ChangeEvent* and a *reconfigurationActivity*. The *reconfigurationActivity* describes the algorithm for switching from the current mode to the target one. It consists of terminating some services and starting new ones – both described in form of UML actions. The triggering conditions for mode change in this model can be raised either by the user or the infrastructure. A user event relates to explicit anticipated mode changes while infrastructure events relate generally to abnormal situation detected by the infrastructure.

4 Reconfiguration Framework

We propose a simple runtime framework to implement reconfiguration to deal with fault tolerance management in a system using replication to achieve fault-tolerance. Since these are realized as CCM components they are independent of an ORB, in particular the connector extensions allows for choosing different interaction implementations. The separation between components and containers in CCM allows to keep non-functional aspects out of the business code. Only the container and the associated connector fragments (which can be seen as part of the container) manage reconfiguration aspects. Non-functional services support the reconfiguration process by blocking for instance the access to ports during reconfiguration in order to maintain a consistent state. That is, the connector can *intercept* messages and block these until a reconfiguration is done - thus avoiding message loss.

The framework is composed of three kind of non-functional components: *Fault_Detector*, *FT_Manager* and *Replica_Manager*. We present in Figure 2 briefly the interfaces of the infrastructure components. An example of a simple scenario is: Fault detector executes *is_node_alive ()* to implement fault detector protocol. This method invokes *node_alive* on other nodes. When some fault occur, the method *manage_event* is invoked on the *FT_Manager* with the corresponding event. Then, the method *reconfigure()* is executed to transit to the specified mode.

5 Designing Reconfigurable Applications

Our laboratory *LISE* [2] has developed a tool, as shown in Figure 3, that supports UML modeling (Papyrus UML) based on the Eclipse environment. This tool suite provides

[2]Laboratory of Model Driven Engineering for Embedded Systems, which is part of the CEA LIST.

```
interface IFault_Detector
{
  void is_node_alive ();
  void node_alive ();
};
interface IFT_Manager
{
  void init(XMLFile Behavior);
  void manage_event(Event event);
  void reconfigure(Event event,
       Mode source, Mode target);
};
interface IReplica_Manager
{
  void manage_replica();
};
```

Figure 2: Interfaces of the components transition framework

a graphic UML modeling editor and code generation components (Java, C, C++). The tool supports also advanced UML profile management. We have developed additional plug-ins which generate CCM descriptor files from a model containing reconfiguration requirements.

These tools have been integrated in the Usine Logicielle platform, and interconnected with the microCCM chain tool developed in collaboration with Thales partner.

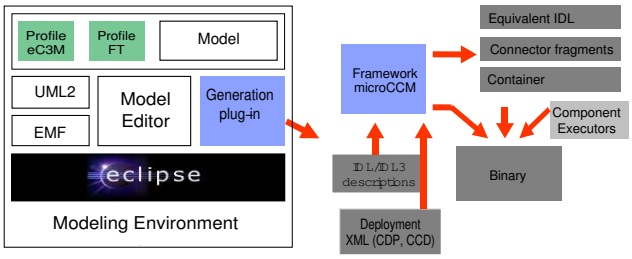


Figure 3: Chain tools

The profile capturing the component model is our *eC3M* proposal. It is based on the UML profile for deployment and configuration D&C [6], comprising also the assembly of an application from existing components. In general, the specification mechanisms are quite close to the composite structure mechanisms in UML. Once this model is defined, reconfiguration issues can be added. Reconfiguration transitions and the operational modes are modeled by using our profile for mode management. Modes and mode transitions are defined using UML statecharts. A simple example is shown in Figure 4, this application has two possible modes, a *NominalMode* and *DegradedMode*. Reconfiguration properties, for instance mode, mode transition and causing events are specified by means of stereotypes that are applied to the states and transitions of the statechart. More complex examples may of course involve a larger number of modes.

The application model is then parsed by a dedicated tool whose goal is build a configuration file for the reconfiguration framework itself so that specific components devoted to reconfiguration management can be populated according to application needs.

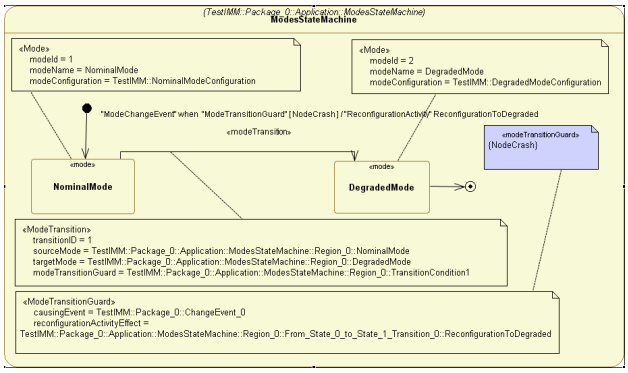


Figure 4: The mode transition behavior

The reconfiguration framework may be seen as a service offered by the execution infrastructure. The connection between application and the framework is done using an XML file generated from a statechart with the applied reconfiguration profile (see Figure 1). From this file, the reconfiguration framework generates adequate code that is used by the runtime framework to execute a certain transition. These mechanisms are currently only partly implemented.

6 Related Work

AQuA [11] provides tools to allow developers to specify the desired level of dependability, through the configuration of the system according to the availability of resources and the faults occurred.

The next two approaches are both based on component-oriented modeling: AFT-CCM [4] is based on CCM and treats fault-tolerance as a specific QoS requirement. The proposed framework enables the modification of QoS parameters at runtime. In [2], the authors designed and implemented Jade, a framework to build autonomic systems: it uses the Fractal component architecture to reconfigure applications according to observed events and using the knowledge of the application architecture. The representation of the environment is based on a component model.

[9] studies reconfiguration in the context of an operating system. In this work the reconfiguration is achieved during design time using a reflective component model and an associated architecture description language. The problem

of expanding all possible operational modes is addressed in [1]. The design flow proposed is devoted to flexible safety-critical systems.

The main difference between the reconfiguration CCM approaches above and our approach is the focus on a specification based on UML and a standardized profile (QoS+FT). Another difference is that we integrated the reconfiguration mechanism into a generic CCM extension. Regardless PIM/PSM (platform independent/specific models), our specification of reconfiguration aspects is already done at a platform independent model. The container (and connector) enables us to hide platform specific aspects at the model level.

7 Summary and Further Research

This work deals with application reconfiguration, i.e. the transition between operational modes. We present an approach to deal with reconfiguration at different levels within the development process of distributed applications. This is done within the larger scope of a component based design that relies on UML, a subset of the OMG profiles QoS&FT as well as Deployment and Configuration. From this model, we generate descriptor files for a framework based on the CORBA component model. Within this development process, reconfiguration properties are declaratively specified at model level and are transparent for the component implementation. While a premier support of fault-tolerance has been finished, the re-configuration support is still partial. The next steps are primarily a support for an automatic re-configuration of the application in a Fault tolerance context. The challenges of the integration include for instance the replication of the component performing the reconfiguration steps.

Acknowledgment: We thanks our partners in the Usine Logicielle project in particular Inflexion sub project(Thales, Astrium and Trialog) and anonymous reviewers of the APRES08 workshop for their comments.

References

[1] L. Almeida, M. Anand, S. Fischmeister, and I. Lee. A dynamic scheduling approach to designing flexible safety-critical systems. In *Proc. of the 7th Annual ACM Conference on Embedded Software (EmSoft'07)*, 2007.

[2] B. Claudel, N. De Palma, R. Lachaize, and D. Hagimont. Self-protection for distributed component-based applications. In *Conference on Security and Network Architectures (SAR), Seignosse, France*, volume 4280, pages 184–198. Springer, 2006.

[3] Peter Feiler and Ana Rugina. Dependability Modeling with the Architecture Analysis & Design Language (AADL). Technical report, 2007. CMU/SEI-2007-TN-043.

[4] J. Fraga, F. Siqueira, and F. Favarim. An adaptive fault-tolerant component model. *Object-Oriented Real-Time Dependable Systems, 2003. WORDS 2003 Fall. The Ninth IEEE International Workshop on*, pages 179–179, 2003.

[5] OMG. *CORBA Component Model Specification, Version 4.0*, 2006. document formal/2006-04-01.

[6] OMG. *Deployment and Configuration of Component Based Distributed Applications, v4.0*, 2006. document ptc/2006-04-02.

[7] OMG. *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms*, 2006. document formal/06-05-02.

[8] OMG. *UML Profile for MARTE*, 2007. document ptc/07-08-04.

[9] J. Polakovic, A. E. Ozcan, and J.B Stefani. Building reconfigurable component-based os with think. In *EUROMICRO '06: Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 178–185, Washington, DC, USA, 2006. IEEE Computer Society.

[10] Sylvain Robert, Ansgar Radermacher, Vincent Seignole, Sébastien Gérard, Virginie Watine, and François Terrier. Enhancing interaction support in the corba component model. In *From Specification to Embedded Systems Application*. Kluwer, 2005.

[11] W.H. Sanders Y. Ren, M. Cukier. An adaptive algorithm for tolerating value faults and crash failures. *IEEE transaction on parallel an distributed systems*, 2:173–192, 2001.

Enabling Extensibility of Sensing Systems through Automatic Composition over Physical Location

Maurice Chu and Juan Liu
Palo Alto Research Center
[mchu, jjliu]@parc.com

Abstract

Networked sensing systems are increasingly adopted in many applications, but today's systems are generally single purpose and hard to extend. This paper addresses the problem of enabling developers to develop extensible networked sensing systems. We propose a design methodology, which centers on a novel automatic composition service where the sensor processing software modules are parameterized by a physical location region. The automatic composer automatically configures the processing and communication occurring in a networked sensing system based on up-to-date sensing needs and sensor device availability. Our approach also enables adaptability and robustness against sensor failures.

1 Introduction

Adding functionality into existing sensing systems that were never designed to be extended can be such a complicated and costly task that deploying a separate system with its own set of sensors is often the simpler, more reliable, and economical choice. This short-sighted "reinvent versus reuse" design methodology results in multiple, isolated sensing systems that cannot be adapted or extended in a robust and reliable way.

For a concrete example, let us consider two video surveillance applications in high demand in retail stores today: security and marketing research. Commercial products for security include surveillance systems from ObjectVideo [1] and Vidient, and products for market research include those from Brickstream. Today, to get security and market research capabilities, separate systems must be purchased. Extending a security surveillance system to do market research (or vice versa) is difficult, which requires a thorough understanding of the processing flow and function call hierarchies. However, these two systems are similar in hardware usage and share a lot of functionality. A potentially more cost effective alternative is to design the systems to be extensible.

Compared to multiple isolated sensing systems, an integrated, extensible system has several advantages. Redundant deployment of sensors can be avoided, so that the upfront cost of deployment is amortized over multiple applications over the system lifetime. The availability of more sensors can improve performance through increased sensor or spatial diversity.

We are interested in a practical methodology that will encourage developers to design for extensibility. This methodology should be simple and impose minimal overhead cost for the benefit of extensibility. In this paper, we propose a design methodology that leverages the key role of *physical location* in sensing and an *automatic composition service*. The automatic composer monitors dynamically changing sensor availability and evolving sensing needs, and automatically instantiates, integrates, and reconfigures software components on devices as the system evolves.

The focus of this paper is on extensibility of sensors and sensing, not the entirety of application logic. Although handling heterogeneity in data types, protocols, interfaces, networks, and devices are important for a total solution, our focus will be on designing a methodology that minimizes developer effort to design extensible sensing systems.

Several existing middleware systems also perform automatic composition. For example, [2] composes an appropriate set of sensors and algorithms based on pre-defined roles and connections, but since it is designed for "tiny" nodes, it composes only at compile-time. GRATIS [3], on the other hand, performs dynamic reconfiguration but chooses among developer-specified alternatives of operations to meet quality-of-service goals. Neither of these focus on physical location as we do.

2 Practical Extensibility Requirements

Let us revisit the video surveillance applications: security and market research. Figure 1 shows a possible reuse scenario. In this example, the retail store is first deployed with a security surveillance system that captures video from cameras, detects people, and tracks them (CameraImager, PeopleDetector, and Tracker). With the initial deployment in place, store managers may wish to leverage this system to extract customer count information throughout a store to help them dispatch sales people accordingly, which can be

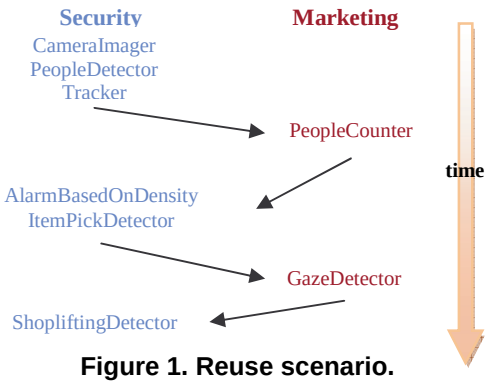


Figure 1. Reuse scenario.

accomplished by adding only a people counting algorithm (PeopleCounter) on top of the existing functionality. Security may then decide to add additional capabilities like detecting unusually dense crowds in restricted areas (AlarmBasedOnDensity). Furthermore, more complex human behavior analysis capabilities can be built on top, like detecting what items are picked up (ItemPickDetector), gaze (GazeDetector), and shoplifting (ShopliftingDetector). In this simple reuse scenario, both applications benefit from extensibility.

To derive requirements for a practical design methodology for extensibility, we consider two developer perspectives: that of the original developer and that of the extender, which refers to the developer who intends to reuse or extend existing functionality.

(i) Original developer's perspective.

For functionality to be reused or extended, the original developer must design for it, and the extra required effort must be nearly nothing.

Req.1a: *Packaging functionality into modular, independently invokable units must require minimum extra effort.*

Req.1b: *Developing generalized algorithms should require minimum extra effort.*

(ii) Extender's perspective.

To encourage the extender to reuse, we have the following requirements:

Req.2a: *Discovering what capabilities exist must be nearly effortless.* Strong developers know that the effort to reuse existing libraries will save time later. We would like to make this extremely easy, so that even weak developers are encouraged to reuse.

Req.2b*: The use of existing components should not require specialized expertise.* Developers have different domain expertise. For instance, security application developers may be computer vision specialists, while marketing application developers are sales experts. It is important that the relevant knowledge of how to use sensors, sensing algorithms, their capabilities, and their limitations are all packaged so that non-experts can use the outputs of these algorithms correctly

Req.2c*: Integrating components should be as simple as wiring to match inputs and outputs.* In our example, the PeopleDetector component takes images as input and outputs people detections in a specified region, while PeopleCounter takes people detection as input and outputs total counts of customers in a region. This simple example illustrates that the two components can be integrated by "wiring" matching inputs and outputs. In practice, wiring can become a massively complicated manual task when there are large numbers of modules and data types. With well-defined matching rules, integration could be automated, which is what our automatic composition service will do.

Req.2d*: Updating existing and adding new components should not break functionality.* Since one of the main benefits of an extensible system is that its functionality can be modified over time, it is critical that incremental updates be reliable without breaking existing functionalities.

3 Design Methodology

3.1 Naming data

For multiple applications to reuse each other's functionality, we need a common interface to refer to data. From our experience, the physical location and timing information of sensor data is the fundamental context information needed to interpret and extract high-level information. Thus, we will refer to data and extracted information by a name tagged with spatiotemporal context. For example, the software components in Figure 1 extract information like "images", "people locations", and "motion trajectories" in certain locations and times. Because the specification of arbitrary spatiotemporal regions can become quite complex, we will focus on the smaller class of specifying and delivering sensing information in arbitrary location regions at the present time only. Then, interests in sensing information can be represented by name-location pairs, with time being implicitly defined to the present.

$$< name, location >$$

Naming has issues like defining ontologies and dealing with non-uniqueness, which we defer to the literature.

3.2 Design Methodology

From the developer's perspective, to make a system extensible means that, the design needs to be modular and with light overhead (Reqs. 1a and 1b respectively). For modularity, we adopt a component-based model where all functionality must be packaged into

independently executable sensing software modules that can be linked together. As for Requirement 1b, since the physical location of sensors is critical context information to interpret sensor data, and it is not until an actual deployment of the system that device locations are established, our design methodology requires all software modules be *parameterized by location*. This is well-supported by our location-based naming scheme. For example, when developing the $PeopleDetector(R)$, it should be able to extract the detections of people within any specified physical region R. Developing location-parameterized modules requires extra effort, but this is a natural generalization for sensing tasks.

From the extender's perspective, discovering existing capabilities and components (Reqs. 2a and 2b) can be effortless with proper naming support, like a list of unique names that indicate the type of information that can be pulled out of the existing sensing system. This means developers do not even have to know about sensing modules because the data name provides a sufficient abstraction. Furthermore, a developer can assume that this information can be extracted from any location region by specifying a name-location pair, as long as there is sufficient sensor coverage in the region of interest.

Finally, the key idea to bring this all together is the inclusion of an *automatic composition service*, that automatically instantiates, sets location parameters of sensing modules, and "wires-up" sensing modules to realize the sensing needs of users and applications, which addresses Req. 2c and Req. 2d.

3.3 Automatic composition service details

The automatic composition service consists of four subservices in Figure 2. The Sensing Needs Monitor maintains information about the sensing needs of the application as a set of name-location pairs. A GUI can interface with this subservice to allow users to insert and delete name-location pairs. Developers can also programmatically interface with this subservice. The Sensing Module Registry maintains a registry of all sensing modules. It includes code and meta-data like the input and output names and the location relationship between inputs and output. The Device Database maintains the state of physical deployment of sensor devices on the network, including their physical location and sensor-specific calibration information, like lens distortion and view orientation in cameras.

These subservices maintain up-to-date information of dynamically changing state, which is then used by the Automatic Composer to decide what processing and communication should be occurring in the

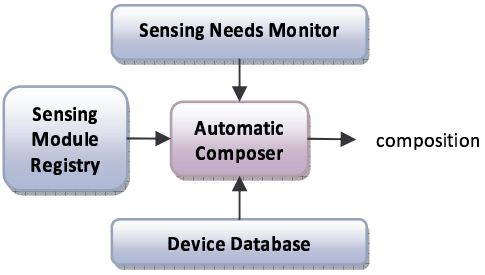


Figure 2. Automatic composition

networked sensing system by performing three tasks. First, it acts as a broker to figure out what software modules and hardware devices are needed to fulfill the sensing needs. Second, it acts as an architect to determine how many of each sensing module is needed, what location parameter to specify, how they should be wired up, and on which devices to instantiate them. Finally, it dispatches this plan, which we call a *composition*, to command the devices to execute it.

The composition algorithm starts with name-location pairs in the Sensing Needs Monitor, say people counts in a region R, $< peopleCount, R >$. Sensing modules that can provide the relevant outputs are instantiated, and their location parameters chosen, say $PeopleCounter(R)$. The inputs to these instantiated sensing modules produce a new set of name-location pairs, say $< image, R >$, which are used to recursively generate the composition all the way down to modules that pull out raw data from the deployed sensors of the system, say $CamImager(A)$ and $CamImager(B)$ if there are two cameras viewing regions A and B. Our algorithm is able to perform geometric unions and intersections with complex, polygonal spatial regions so that the algorithm would choose the appropriate location parameters, say $CamImager(A \cap R)$ and $CamImager(B \cap R - A)$. Figure 3 shows a simple example composition. Although not shown here, our algorithm can generate quite complicated graph topologies.

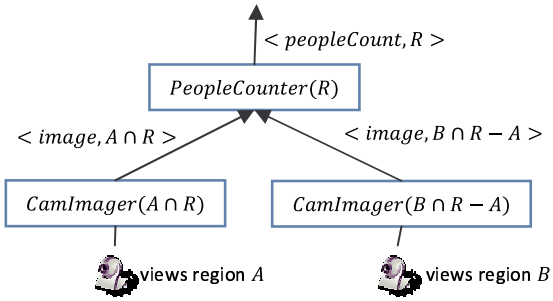


Figure 3. Example Composition

For a networked system, the final step is to determine on which devices to place each sensing module. We can use rules, like pushing as much functionality into the sensor nodes as possible, or various load balancing and distributed process migration techniques [4], which determine good placement under a variety of cost criteria.

By recomputing a new composition online in response to state changes of sensing needs and sensing modules, our design methodology produces an extensible and adaptable system that can reconfigure its processing and automatically incorporate new software without going offline. When new sensors are added or failures are detected, the automatic composer is able to respond by recomposing with the new availability and unavailability of sensors. This results in automatically loading appropriate software onto newly added sensors and robustness to failures since the system can recompose processing to not rely on unavailable sensors.

The architecture we envision for implementing a networked system with automatic composition over physical location is a two-tiered system with one or more "composer" nodes and several sensor nodes. This separation allows for sensor nodes to be mote-like devices rather than full-fledged computers.

4 Prototype system

We have built a prototype of the automatic composition service and a supporting runtime for sensor nodes to test out our ideas. We developed a security and personal advertisement application, much like that described in Section 2. The prototype consists of one composer node, which is a 1 GHz Linux box, and 15 sensor nodes, which are Linux OpenBrick computers with 300 MHz Geode processors. 14 of the sensor nodes, each with a camera, are mounted on the ceiling of our lab. The 15th sensor node is an RFID reader by the doorway. The composer and sensor nodes are networked using TCP/IP over wired ethernet.

Figure 4 shows the output of the ceiling camera images in our system. The red square is the result of an AlarmBasedOnDensity module detecting when there are more than two objects in the area. Figure 5 shows the GUI that interfaces with the Sensing Needs Monitor. Users can select the type of data they want from the system (color-coded) and specify a polygonal region to sense for that kind of data. The less saturated coloring indicates the lack of sensor coverage which is a side result from our automatic composition algorithm indicating insufficient sensor coverage.

Acknowledgments ---- We thank Fujitsu Limited for sponsoring this research and acknowledge James Reich, Teresa Lunt, Bo Begole, Kurt Partridge, Ignacio Solis, and Dan Larner of PARC for their contributions.

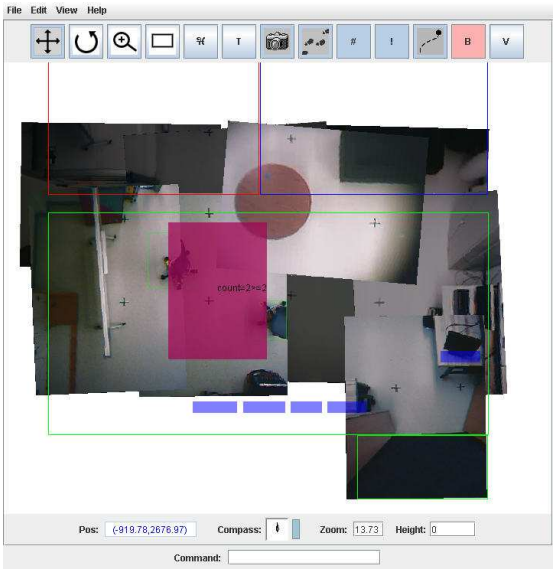


Figure 4. GUI screenshot

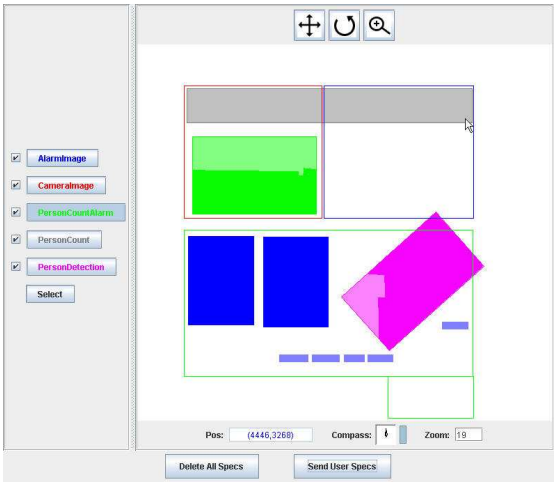


Figure 5. Sensing Needs Monitor GUI

5 References

1. Venetianer, Peter, et al., *Video Verification of Point of Sales Transactions.* London : s.n., 2007. Advanced Video and Signal based Surveillance.

2. Animesh Pathak, Luca Mottola, Amol Bakshi et. al., *A Compilation Framework for Macroprogramming Networked Sensors.* 2007. Proceedings of Distributed Computing in Sensor Systems (DCOSS). pp. 189-204.

3. Sachin Kogekar, Sandeep Neema, Brandon Eames, et. al., *Constraint-Guided Dynamic Reconfiguration in Sensor.* 2004. Proceedings of Information Processing in Sensor Networks (IPSN).

4. Milojicic, D., Douglis, F. and Wheeler, R. *Mobility: Processes, Computers, and Agents.* s.l. : ACM Addison-Wesley, Feb 1999.